Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Analyzing Program Termination and Complexity Automatically with AProVE

J. Giesl · C. Aschermann · M. Brockschmidt · F. Emmes · F. Frohn · C. Fuhs · J. Hensel · C. Otto · M. Plücker · P. Schneider-Kamp · T. Ströder · S. Swiderski · R. Thiemann

Abstract In this system description, we present the tool AProVE for automatic termination and complexity proofs of Java, C, Haskell, Prolog, and rewrite systems. In addition to classical term rewrite systems (TRSs), AProVE also supports rewrite systems containing built-in integers (int-TRSs). To analyze programs in high-level languages, AProVE automatically converts them to (int-)TRSs. Then, a wide range of techniques is employed to prove termination and to infer complexity bounds for the resulting rewrite systems. The generated proofs can be exported to check their correctness using automatic certifiers. To use AProVE in software construction, we present a corresponding plug-in for the popular Eclipse software development environment.

Keywords Termination Analysis · Complexity Analysis · Java/C/Haskell/Prolog Programs · Term Rewriting

1 Introduction

AProVE (Automated Program Verification Environment) is a tool for automatic termination and complexity analysis. While previous versions (described in [35,

Supported by the Deutsche Forschungsgemeinschaft (DFG) grant GI 274/6-1, the Air Force Research Laboratory (AFRL), the Austrian Science Fund (FWF) project Y757, and the Danish Council for Independent Research, Natural Sciences. Most of the research was done while the authors were at RWTH Aachen.

J. Giesl · C. Aschermann · F. Emmes · F. Frohn · J. Hensel · M. Plücker · T. Ströder
LuFG Informatik 2, RWTH Aachen University, Germany

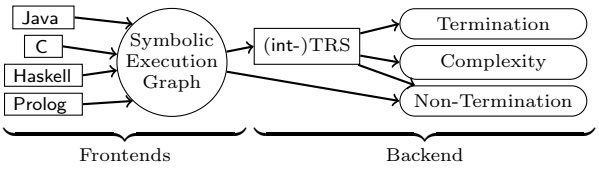
M. Brockschmidt
Microsoft Research Cambridge, UK

C. Fuhs
Dept. of Computer Science and Information Systems, Birkbeck, University of London, UK

P. Schneider-Kamp
Dept. of Mathematics and Computer Science, University of Southern Denmark, Denmark

R. Thiemann
Institute of Computer Science, University of Innsbruck, Austria

38]) analyzed only termination of term rewriting, the new version of AProVE also analyzes termination of Java, C, Haskell, and Prolog programs. Moreover, it also features techniques for automatic complexity analysis and permits the certification of automatically generated termination proofs. To analyze programs, AProVE uses an approach based on symbolic execution and abstraction [20] to transform the input program into a *symbolic execution graph*[1] that represents all possible computations of the input program. Language-specific features (like sharing effects of heap operations in Java, pointer arithmetic and memory safety in C, higher-order functions and lazy evaluation in Haskell, or extra-logical predicates in Prolog) are handled when generating this graph. Thus, the exact definition of the graph depends on the considered programming language. For termination or complexity analysis, the graph is transformed into a set of (int-)TRSs. The success of AProVE at the annual international *Termination Competition* [62] and the *International Competition on Software Verification (SV-COMP)* [60] at *TACAS* demonstrates that our transformational approach is well suited for termination analysis of real-world programming languages. A graphical overview of our approach is shown below.[2]



Technical details on the techniques for transforming programs to (int-)TRSs and for analyzing rewrite systems can be found in, e.g., [10, 11, 12, 14, 17, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 36, 37, 39, 40, 41, 43, 44, 51, 52, 57, 58]. Since the current paper is a system description, we focus on the implementation of these techniques in AProVE, which we have made available as a plug-in for the popular Eclipse software development environment [23]. In this way, AProVE can already be applied during program construction (e.g., by analyzing termination of single Java methods for user-specified classes of inputs). In addition to the full version of AProVE, we also have made AProVE's frontends for the different programming languages available as separate programs. Thus, they can be coupled with other external tools that operate on rewrite systems (e.g., on TRSs or integer transition systems) or on symbolic execution graphs. These external tools can then be used as alternative backends. Finally, AProVE can also be accessed directly via the command-line (with several possible flags) or via a web interface [4].

A very short description of AProVE's use for termination analysis of C programs was published in [59] and a preliminary version of parts of this paper was published in [42]. The present paper extends [42] substantially:

- We have updated AProVE's implementation and its description for the use on different programming languages and TRSs in Sect. 2 by covering new contributions (e.g., in addition to upper complexity bounds, AProVE now also infers *lower* complexity bounds for term rewriting [28]). Moreover, in addition to the features described in [42], we have developed a new exchange format for symbolic execution graphs, allowing to combine AProVE's frontends for programming languages with arbitrary external backends (Sect. 2.2).

[1] In our earlier papers, this was often called a *termination graph*.

[2] While termination can be analyzed for Java, C, Haskell, Prolog, TRSs, and int-TRSs, the current version of AProVE analyzes complexity only for Prolog and TRSs. In addition, complexity of integer transition systems (a restriction of int-TRSs) is analyzed by calling the tools KoAT and LoAT.

- To make AProVE applicable in practice, the techniques in the backend have to deal with large rewrite systems resulting from the original input program. To handle programs from real programming languages successfully, it is important to provide support for integers. Therefore, AProVE handles int-TRSs where integers are built-in. In Sect. 3, we present the techniques that AProVE uses to simplify such int-TRSs and to prove their termination afterwards. These techniques are especially tailored to the int-TRSs resulting from our frontends and we present experiments to evaluate their usefulness. (The proofs and further formal details on these techniques can be found in Appendix A.) This part is completely new compared with [42].
- AProVE solves most search problems occurring during the proofs by calling SAT or SMT solvers. We discuss the use of SAT and SMT solving for the several different techniques in AProVE in Sect. 4 (which is also new compared with [42]). In addition, AProVE can also be used as an SMT solver itself. To this end, we describe AProVE's SMT frontend for quantifier-free formulas over non-linear integer arithmetic QF_NIA.
- To increase the reliability of the generated proofs, AProVE supports their certification, cf. Sect. 5. Compared with [42], we now present different modes to control the "amount of certification" for automatically generated proofs. In this way, one can also use certification for proofs where not all proof techniques can be handled by the certifier yet.

2 AProVE and its Graphical User Interface in Eclipse

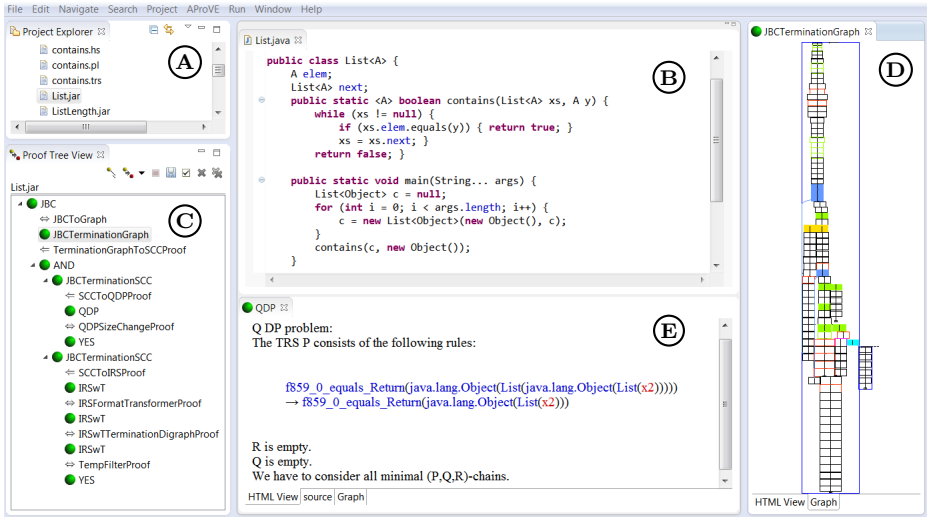
AProVE and its graphical user interface are available as an Eclipse plug-in at [4] under "Download". After the initial installation, "Check for Updates" in the "Help" menu of Eclipse also checks for updates of AProVE. As Eclipse and AProVE are written in Java, they can be used on most operating systems. We describe the integration of AProVE's frontends in Eclipse in Sect. 2.1. In Sect. 2.2 we explain how to use the frontends in a stand-alone way in order to couple them with external tools that operate on symbolic execution graphs or (int-)TRSs. Finally, Sect. 2.3 focuses on the backend of AProVE.

2.1 Analyzing Programming Languages

The screenshot on the next page shows the main features of our AProVE plug-in. Here, AProVE is applied on a Java (resp. Java Bytecode (JBC)) program in the file `List.jar` and tries to prove termination of the `main` method of the class `List`, which in turn calls the method `contains`. (The source code is shown in the editor window **(B)**.) Files in an Eclipse project can be analyzed by right-clicking on the file in Eclipse's Project Explorer **(A)** and selecting "Launch AProVE".[3]

When AProVE is launched, the proof (progress) can be inspected in the Proof Tree View **(C)**. Here, problems (e.g., programs, symbolic execution graphs, TRSs, ...) alternate with proof steps that modify problems, where "$\Leftarrow$" indicates sound and "$\Leftrightarrow$" indicates sound and complete steps. This information is used to propagate

[3] An initial "ExampleProject" with several examples in different programming languages can be created by clicking on the "AProVE" entry in Eclipse's menu bar.



information from child nodes to the parent node. A green (resp. red) bullet in front of a problem means that termination of the problem is proved (resp. disproved) and a yellow bullet denotes an unsuccessful (or unfinished) proof. Since the root of the proof tree is always the input problem, the color of its bullet indicates whether AProVE could show its termination resp. non-termination.

To handle Java-specific features, AProVE constructs a symbolic execution graph **(D)** from the program [10, 11, 12, 52]. From the cycles of this graph, (int-)TRSs are created whose termination implies termination of the original program.[4] Double-clicking on a problem or proof step in the proof tree shows detailed information on them. For example, the symbolic execution graph can be inspected by double-clicking the node JBCTerminationGraph and selecting the `Graph` tab in the Problem View **(D)**. This graph can be navigated with the mouse, allowing to zoom in on specific nodes or edges. Similarly, one of the generated TRSs is shown in the Problem View **(E)**. For *non*-termination proofs [11], witness executions are provided in the Problem View. In contrast to termination, non-termination proofs are directly guided by the symbolic execution graph (without using the (int-)TRS backend), from which one extracts a concrete (i.e., non-symbolic) non-terminating execution.

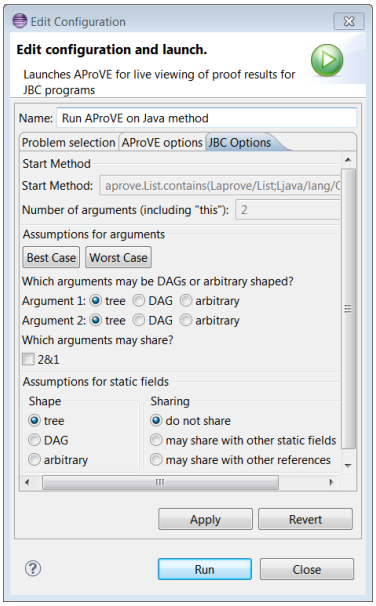
The buttons in the upper right part of the Proof Tree View **(C)** interact with AProVE (e.g., ■ aborts the analysis). When AProVE is launched, the termination proof is attempted with a time-out of 60 seconds. If it is aborted, one can right-click on a node in the proof tree and by selecting "Run", one can continue the proof at this node (here, one may also specify a new time-out).

For Java programs, there are two options to specify which parts of the program are analyzed. AProVE can be launched on a `jar` (Java archive) file. Then it tries to prove termination of the `main` method of the archive's "main class".[5] Al-

[4] In **(C)**, TRSs are listed as "QDP" (*dependency pair problems* [37]) and int-TRSs are shown as "IRSwT" ("Integer Rewrite Systems with Terms").

[5] See `http://www.termination-portal.org/wiki/Java_Bytecode` for the conventions of the *Termination Competition*, which also contain a detailed discussion of the limitations imposed on analyzed Java programs.

ternatively, to use AProVE during software development, single Java methods can be analyzed. Eclipse's Outline View (reachable via "Window" and "Show View") shows the methods of a class opened by a double-click in Eclipse's Project Explorer. An initial "JavaProject" with a class List can be created via the "AProVE" entry in Eclipse's menu bar. Right-clicking on a method in the Outline View and choosing "Launch AProVE" leads to the configuration dialog on the side. It can be used to specify the sharing and shape of the method's input values. Each argument can be tree-shaped, DAG-shaped, or arbitrary (i.e., possibly cyclic) [12]. Furthermore, one can specify which arguments may be sharing. Similarly, one can provide assumptions about the contents of static fields. There are also two short-cut buttons which lead to the best- and worst-case assumption. Moreover, under "AProVE options", one can adjust the desired time-out for the termination proof and under "Problem selection", one may replace AProVE's default strategy with alternative user-defined strategies (a general change of AProVE's strategy is possible via the "AProVE" entry in Eclipse's main menu).



C [58], Haskell [40], and Prolog [41] are handled similarly. The function, start terms, or queries to be analyzed can be specified in the input file (as in the *Termination Competition*). Otherwise the user is prompted when the analysis starts. For Prolog, AProVE can also infer asymptotic upper bounds on the number of evaluation steps (i.e., unification attempts) and prove determinacy (i.e., that there is at most one solution). Similarly to many other termination provers, AProVE treats built-in data types like `int` in Java as unbounded integers $\mathbb{Z}$. Thus, a termination proof is valid only under the assumption that no overflows occur which influence the control flow. However for C programs, we recently extended AProVE by an approach to handle fixed-width bitvector integers as well [43].

All our programming language frontends first construct symbolic execution graphs. From these graphs, AProVE generates rewrite systems which express the information that is relevant for termination. Thus, analyzing implementations of the same algorithm in different languages leads to very similar rewrite systems, as AProVE identifies that the reason for termination is always the same. For example, implementations of a `contains` algorithm in different languages all terminate for the same reason on (finite acyclic) lists, since the length of the list decreases in each recursive call or iteration.

2.2 Using AProVE as a Frontend

We have separated AProVE's programming languages frontends such that other tools can be used in place of the existing AProVE backend. As the symbolic execution graphs computed by AProVE represent over-approximations of the original program's behavior, many analyses can be performed on these graphs. For in-

stance, the absence of memory-safety violations in C programs, determinacy of queries in Prolog programs, or information-flow properties in all supported languages can be proved by analyzing the corresponding symbolic execution graphs. In general, analyses with the goal of proving universal properties can be executed on such graphs directly (e.g., to prove that *all* computations are finite). Moreover, together with a reachability analysis, existential properties can be analyzed as well. An example of such an analysis is our non-termination analysis of Java programs (i.e., that there *exists* an infinite computation). Here, we first detect infinite loops in the symbolic execution graph, and then check their reachability.

AProVE can export its symbolic execution graphs in the JSON format [9], which is both human-readable as well as suitable for automated processing. Such an export is produced by the command-line version of AProVE when running

`java -ea -cp aprove.jar aprove.CommandLineInterface.<X>FrontendMain <E> -j yes -o <D>`,

where `<X>` has to be replaced by `JBC`, `C`, `LLVM`,[6] `Haskell`, or `Prolog`, `<E>` is the input program, and `<D>` specifies the path to the directory where the output graph is written. This allows other tools to use AProVE for symbolic execution and to implement their own analysis on the graph obtained by AProVE. We refer to [4] for a detailed explanation of the command-line flags available for AProVE.

Instead of symbolic execution graphs, AProVE can also export the resulting rewrite systems in the formats used at the *Termination Competition* [62]. As these systems are generated with the goal of termination or complexity analysis, they only "over-approximate" the program's termination and complexity. So in contrast to the symbolic execution graphs, these rewrite systems are no general over-approximations for the original program's behavior. Still, other termination and complexity provers can make use of these systems and obtain analyses for programming languages by implementing only their own backend techniques and re-using our frontends. The command to export the resulting rewrite systems is the same as the one to generate the symbolic execution graph but without `-j yes`.

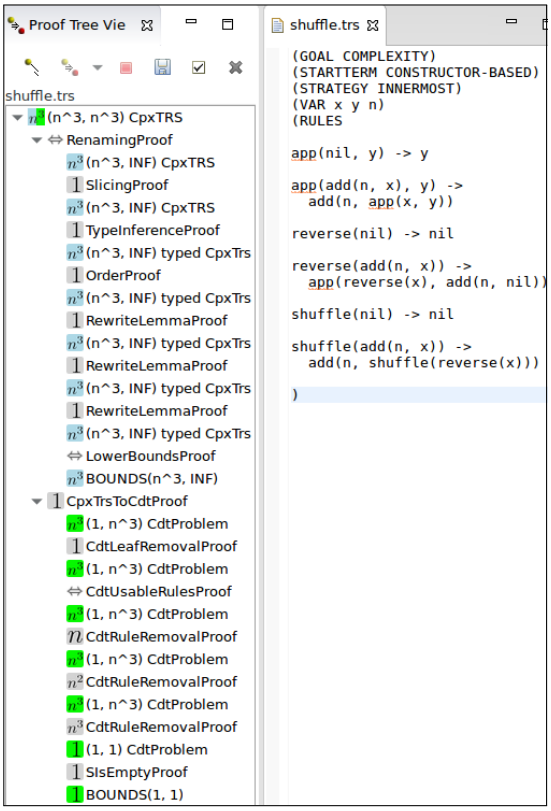
2.3 Analyzing Term Rewrite Systems

To prove termination of TRSs, AProVE implements a combination of numerous techniques within the dependency pair framework [37]. Non-termination of TRSs is detected by suitable adaptions of narrowing [25, 36]. The frontends for Java and C programs generate *int-TRSs*, a variation of standard term rewriting that has built-in support for integer values and operations. The advantage of built-in integers is that this simplifies termination proofs for algorithms on integers tremendously [33]. Note that most other tools for termination analysis of imperative programs abstract the program to *integer transition systems* (ITSs). Compared with ITSs, int-TRSs have the advantage that they allow a more precise representation of complex data structures by terms. We give an overview of the termination techniques used for this formalism in Sect. 3. The frontend for Haskell programs directly produces classical TRSs (as dependency pair problems [37]), while the frontend for Prolog offers techniques to produce definite logic programs, dependency triples (a

[6] Here, `LLVM` stands for the intermediate representation of the LLVM compilation framework [46]. To analyze C programs, they are first compiled to LLVM and analyzed afterwards. This is similar to our approach for Java where we consider Java Bytecode instead of Java source code.

variant of dependency pairs for definite logic programs) [57], and TRSs.

For complexity analysis of TRSs, AProVE infers bounds on the *runtime complexity*. As initial terms, here one only considers *basic terms* $f(t_1, \ldots, t_m)$ where $t_1, \ldots, t_m$ represent data (thus, $t_1, \ldots, t_m$ are already in normal form). This corresponds to the setting in program analysis where one wants to infer asymptotic symbolic bounds on the number of evaluation steps that the program can perform. While upper bounds are computed for innermost rewriting, lower bounds are inferred for innermost as well as full rewriting. The focus on upper bounds for innermost rewriting is motivated by the fact that the transformations from Sect. 2.1 yield rewrite systems where it suffices to consider innermost rewriting in the backend. (Polynomial) upper bounds on the runtime complexity are inferred by an adaption of dependency pairs for complexity analysis [51]. To solve the resulting search problems, AProVE re-uses the techniques from termination analysis to generate suitable well-founded orders.



To infer polynomial or exponential *lower* bounds, infinite families of reductions are speculated by narrowing. Afterwards, their validity is proved via induction and term rewriting. Then this proof gives rise to a lower complexity bound [28]. In addition, AProVE applies a narrowing-based technique to prove that the (innermost) runtime complexity of a TRS is infinite, i.e., that there is an infinite reduction sequence starting with a basic term. Blue icons like n^3 indicate lower bounds, while green icons like n^3 are used for upper bounds, cf. the screenshot above. Icons like n^3 represent tight bounds where the inferred lower and upper bound coincide. As the screenshot shows, AProVE easily infers that the above TRS has cubic asymptotic complexity. More precisely, the icon n^3 (resp. the result $(\mathsf{n}^3, \mathsf{n}^3)$) at the root node of the proof tree means that the longest rewrite sequences from initial terms $f(t_1, \ldots, t_m)$ of size n are of length $\Theta(n^3)$.[7] Moreover, AProVE also analyzes the complexity of integer transition systems with initial states by calling the tools KoAT [14] (for upper bounds) and LoAT [29] (for lower bounds).

[7] In the Proof Tree View, we do not only have complexity icons like n^3 or n^3 for problems, but proof steps also result in complexities (e.g., 1 or n). More precisely, in each proof step, a problem P is transformed into a new problem P' and a complexity c from the rewrite rule(s) whose contribution to P's complexity is accounted for in this step. Then the complexity of P is bounded by the maximum (asymptotically equivalent to the sum) of P''s complexity and c.

3 Termination Analysis with Integers

To handle standard arithmetic operations on integers, AProVE supports int-TRSs, i.e., term rewrite systems with built-in integer arithmetic. In Sect. 3.1 we introduce the notion of int-TRSs used by AProVE and present simplifications which substantially reduce the size of int-TRSs and therefore ease the search for termination arguments. These simplifications are needed to handle those int-TRSs efficiently that result from the transformation of programming languages. In Sect. 3.2 we then introduce the techniques that AProVE uses for termination analysis of int-TRSs. Finally, Sect. 3.3 contains an experimental evaluation of the presented techniques. We refer to Appendix A for full formal definitions and proofs.

3.1 Definition and Simplification of int-TRSs

To denote integers and standard pre-defined arithmetic operations, we use the signature $\Sigma_{\mathsf{pre}} = \mathbb{Z} \cup \{+, -, *, /, \%\}$. The terms in $\mathcal{T}(\Sigma_{\mathsf{pre}}, \mathcal{V})$ (i.e., terms constructed from Σ_{pre} and variables $\mathcal{V}$) are called *int-terms*. *Atomic int-constraints* have the form $s \circ t$ where s and t are int-terms and $\circ \in \{<, \leq, =, \neq, \geq, >\}$. An *int-constraint* is a Boolean combination of such atomic constraints.

To represent user-defined data structures, we use a signature Σ_c of data constructors. Moreover, we use a signature Σ_d of defined symbols. For imperative programs, the defined symbols represent program positions, and their arguments correspond to the values of the program variables. We require that Σ_{pre}, Σ_c, and Σ_d are pairwise disjoint. An *int-TRS* is a set of int-rules which are used to over-approximate the effect of program evaluation, operating on term representations of the data. An *int-rule* has the following form, where $f \in \Sigma_d$, $g \in \Sigma_d \cup \Sigma_c$, $s_1, \ldots, s_n, t_1, \ldots, t_m \in \mathcal{T}(\mathbb{Z} \cup \Sigma_c, \mathcal{V})$, and φ is an int-constraint.

$$f(s_1, \ldots, s_n) \to g(t_1, \ldots, t_m) \,[\![\varphi]\!]$$

The rewrite relation of an int-TRS is simple top-rewriting (i.e., rules may only be applied at the root of a term),[8] where we only consider data substitutions $\sigma : \mathcal{V} \to \mathcal{T}(\mathbb{Z} \cup \Sigma_c, \mathcal{V})$. As usual, $t\sigma$ results from replacing all variables of t according to σ. We call σ *applicable* to an int-constraint φ if $\sigma(x) \in \mathbb{Z}$ for all variables x in φ, and $\varphi\sigma$ is *valid* if it holds w.r.t. the usual integer arithmetic. Then, s *rewrites* to t (denoted "$s \hookrightarrow t$") if the int-TRS contains a rule $\ell \to r \,[\![\varphi]\!]$ and a data substitution σ that is applicable to φ such that $\ell\sigma = s$, $r\sigma = t$, and $\varphi\sigma$ is valid. In contrast to standard rewriting, we allow rules $\ell \to r \,[\![\varphi]\!]$ where the right-hand side r or the constraint φ may contain fresh variables that do not occur in the left-hand side ℓ. This does not necessarily imply non-termination, as variables may not be instantiated by terms with defined symbols.

Example 1 A program counting the number of occurrences of the number c in a list could be represented as follows. Here, we use the constructors N and C to represent lists, where N stands for the empty list and $\mathsf{C}(v, xs)$ results from inserting the number v in front of the list xs.

[8] So our int-TRSs extend int-*based TRSs* from [27] by constructors and restrict *integer TRSs* (ITRSs) from [33] by not allowing nested defined symbols.

$$\mathsf{count}(\mathsf{C}(v, xs), c, res) \to \mathsf{count}(xs, c, res') \qquad [\![v = c \wedge res' = res + 1]\!] \qquad (1)$$
$$\mathsf{count}(\mathsf{C}(v, xs), c, res) \to \mathsf{count}(xs, c, res) \qquad [\![v \neq c]\!] \qquad (2)$$

Applying this program to count how often the number 4 occurs in the list $[1, 4, 7]$ yields the rewrite steps

$$\begin{array}{ll} & \mathsf{count}(\mathsf{C}(1, \mathsf{C}(4, \mathsf{C}(7, \mathsf{N}))), 4, 0) \\ \hookrightarrow_{(2)} & \mathsf{count}(\mathsf{C}(4, \mathsf{C}(7, \mathsf{N})), 4, 0) \\ \hookrightarrow_{(1)} & \mathsf{count}(\mathsf{C}(7, \mathsf{N}), 4, 1) \\ \hookrightarrow_{(2)} & \mathsf{count}(\mathsf{N}, 4, 1) \end{array}$$

In practice, the int-TRSs automatically generated from programming languages by AProVE differ considerably from the manually crafted TRSs that are often discussed in the literature, both in the number of rules (of which there are usually hundreds to thousands) and in the arity of function symbols (which is sometimes exceeding 50). To tackle these int-TRSs, AProVE uses a number of simplifications that are similar to techniques employed in classic program analysis. The most important simplifications are *slicing* (the removal of needless arguments of function symbols) and *chaining* (the static combination of several rewrite rules into one).

3.1.1 Slicing

To reduce the number of arguments of the function symbols in our int-TRSs, we remove all arguments except those that are possibly *needed* for termination, i.e., that may influence the applicability of a rule to a term. While a related concept for the restricted formalism of `int`-based TRSs was informally mentioned in [27], the following formal criteria identify a substantially larger set of *needless* positions, and can handle data structures represented as terms, unlike earlier work [1]. A related approach for TRSs without built-in integers was presented in [3]. Here the goal was to find arguments whose elimination does not change the semantics, while our goal is only to keep the termination behavior unchanged.

We identify "needed" arguments by considering how the application of a rule can fail. In int-TRSs, this can either be because there is no data substitution that matches the left-hand side of a rule to a term, or due to an unsatisfied rule condition. Thus, if the left-hand side of a rule has the form $f(c(\ldots), \ldots)$ and there exist subterms $f(t, \ldots)$ in right-hand sides where t does not have the form $c(\ldots)$, then the first argument of f is *needed* since it determines whether *matching* succeeds. Similarly, if a left-hand side is *non-linear* (i.e., if a variable occurs several times in a left-hand side), then the corresponding arguments are also needed, as they influence whether a suitable data substitution exists.

Moreover, if the left-hand side has the form $f(x, \ldots)$ and the variable x occurs in some term t in the *constraint* φ of the rule, then this variable may determine the rule's applicability and f's argument is needed. Here, it suffices to consider just the *assignment-free* version of φ, in which we disregard simple assignments. In general, for a rule $\ell \to r\,[\![\varphi]\!]$ we call atoms "$y = t$" of φ *assignments* if y is a variable that neither occurs in ℓ nor t nor in any other atom of φ. The *assignment-free* constraint $\widehat{\varphi}$ is obtained by removing all assignments from φ. For instance, the assignment-free variant of the constraint of Rule (1) is just $v = c$ (i.e., the assignment $res' = res + 1$ is disregarded since res' does not occur anywhere else in

the constraint or the left-hand side of the rule). Finally, if a variable x is needed on the right-hand side r (or if φ contains an assignment $y = (\dots x \dots)$ where y is needed), then all occurrences of x in the left-hand side are also needed (i.e., then neededness is *propagated*). The following definition summarizes these ideas. Here, Σ stands for $\Sigma_c \cup \Sigma_d$.

Definition 2 (Needed Argument Positions) The *needed argument positions* $N \subseteq (\Sigma \times \mathbb{N})$ for an int-TRS $\mathcal{R}$ are the smallest set with $(f, i) \in N$ if $1 \leq i \leq \mathit{arity}(f)$ and one of the following holds for some rule $\ell \to r \,[\![\varphi]\!]$ of $\mathcal{R}$:

- *Matching*: ℓ contains a subterm $f(t_1, \dots, t_n)$, the right-hand side of some rule in $\mathcal{R}$ contains $f(u_1, \dots, u_n)$, and there is no matcher σ with $t_i\sigma = u_i$.
- *Non-linearity*: ℓ contains $f(t_1, \dots, t_n)$ and a variable $x \in \mathcal{V}(t_i)$ occurs more than once in ℓ.
- *Constraint*: ℓ contains a subterm $f(t_1, \dots, t_n)$ and $\mathcal{V}(t_i) \cap \mathcal{V}(\widehat{\varphi}) \neq \varnothing$ for the assignment-free constraint $\widehat{\varphi}$.
- *Propagation*: ℓ contains a subterm $f(t_1, \dots, t_n)$ such that a needed variable x occurs in t_i. A variable x is *needed* if one of the following holds:
  - x occurs at a *needed position* of r. (Here, x occurs at a *needed position* of a term t iff $t = x$ or $t = g(t_1, \dots, t_m)$, $(g, j) \in N$, and x occurs at a needed position of t_j.)
  - φ contains an assignment "$y = t$" where y is needed and $x \in \mathcal{V}(t)$

Thm. 3 states that the simplification of removing all argument positions that are not needed (by a corresponding argument filter [37]) is not only sound, but also complete. So the *needed* positions according to Def. 2 are indeed "exhaustive", i.e., they include all argument positions that influence the termination behavior.

Theorem 3 (Soundness of Slicing) *Let $\mathcal{R}$ be an int-TRS and let $\mathcal{R}'$ result from filtering away all argument positions that are not needed according to Def. 2. Then $\mathcal{R}$ terminates iff $\mathcal{R}'$ terminates.*

For the int-TRS $\{(1), (2)\}$, the third argument of count is not needed since its argument only occurs in an assignment $\mathit{res}' = \mathit{res} + 1$ of the constraint of Rule (1). Removing this needless argument position simplifies the int-TRS to

$$\mathsf{count}(\mathsf{C}(v, xs), c) \to \mathsf{count}(xs, c) \,[\![v = c]\!] \quad \mathsf{count}(\mathsf{C}(v, xs), c) \to \mathsf{count}(xs, c) \,[\![v \neq c]\!]$$

Here, $\mathit{res}' = \mathit{res} + 1$ was removed from the constraint of the first rule since res and res' do not occur in its left- or right-hand side anymore after filtering.

In examples from real programs, this technique greatly simplifies the proof search. As a typical example, in AProVE's analysis of the `addAll` method of the `java.util.LinkedList` data structure in Oracle's Java distribution, the average arity of function symbols in the resulting int-TRS is reduced from 12.4 to 3.4.

3.1.2 Chaining

As our frontends are based on the operational small-step semantics of the respective languages, every evaluation step of the program results in a separate rewrite rule. Thus, the generated int-TRSs contain many rules corresponding to intermediate program positions. Since the number of rules directly influences the size of the

resulting search problems when trying to synthesize well-founded orders, we perform *chaining* to merge rewrite rules that have to be applied after each other [27]. More precisely, we use *narrowing* of right-hand sides of rules to obtain new rules which simulate the effect of applying two rules directly after each other. In this way, we eliminate superfluous defined symbols (i.e., those generated for states on a long, linear path in a symbolic execution graph) and obtain simpler rewrite systems. In the analysis of JBC programs via constrained logic programming, similar *unfolding* techniques have been discussed [55].

Example 4 To illustrate the idea, consider the following int-TRS.

$$\mathsf{f}(x_1) \to \mathsf{g}(x_1') \;\; [\![x_1' = x_1 + 1]\!] \quad (3) \qquad \mathsf{f}(x_3) \to \mathsf{h}(x_3') \;\; [\![x_3' = x_3 - 1]\!] \quad (5)$$
$$\mathsf{g}(x_2) \to \mathsf{f}(x_2) \;\; [\![x_2 < 0]\!] \quad (4) \qquad \mathsf{h}(x_4) \to \mathsf{f}(x_4) \;\; [\![x_4 > 0]\!] \quad (6)$$

After applying rule (3), rule (4) must be used in the next rewrite step. Therefore, we can combine (3) and (4) using the unifier $\mu = [x_1'/x_2]$ of (3)'s right-hand side and (4)'s left-hand side (i.e., we *narrow* (3) with the rule (4)). The conditions of both rules are combined, too. Similarly, we can combine (5) and (6). Thus, we obtain the following int-TRS where the defined symbols g and h are removed.

$$\mathsf{f}(x_1) \to \mathsf{f}(x_2) \;\; [\![x_2 = x_1 + 1 \wedge x_2 < 0]\!] \;\; (7) \qquad \mathsf{f}(x_3) \to \mathsf{f}(x_4) \;\; [\![x_4 = x_3 - 1 \wedge x_4 > 0]\!] \;\; (8)$$

To perform the desired simplification, we define two helpful sets. For an int-TRS $\mathcal{R}$ and a defined symbol $f \in \Sigma_d$, let $\mathcal{R}_{\to f}$ consist of all rules where f is the root symbol of the right-hand side and $\mathcal{R}_{f\to}$ contains all rules where f is the root of the left-hand side. If $\mathcal{R}$ has no directly recursive f-rules (i.e., if $\mathcal{R}_{\to f} \cap \mathcal{R}_{f\to} = \varnothing$), then we can eliminate f by applying the rules from $\mathcal{R}_{f\to}$ to the right-hand sides of the rules in $\mathcal{R}_{\to f}$. More precisely, let $\mathcal{R}^{-f}$ denote the int-TRS which results from narrowing each rule from $\mathcal{R}_{\to f}$ with all rules from $\mathcal{R}_{f\to}$. Then $\mathcal{R}^{-f}$ does not contain the symbol f anymore. So for the TRS $\mathcal{R}$ from Ex. 4, we have $\mathcal{R}^{-\mathsf{g}} = \{(5), (6), (7)\}$ and $(\mathcal{R}^{-\mathsf{g}})^{-\mathsf{h}} = \{(7), (8)\}$.

Theorem 5 (Soundness of Chaining) *Let $\mathcal{R}$ be an int-TRS such that there are no directly recursive rules for $f \in \Sigma_d$. Then $\mathcal{R}$ terminates iff $\mathcal{R}^{-f}$ terminates.*

For example, in the analysis of the `java.utils.LinkedList.addAll` example discussed above, this technique reduces the number of rules in the resulting int-TRS from 206 to 6, using just one defined symbol instead of 204 symbols.

3.2 Proving Termination of int-TRSs

Originally, we used AProVE's support for integer term rewrite systems [33] as backend for our programming language frontends. However, experiments showed that the time spent in this backend dominated the time spent on the overall proof, even though in most cases, only simple termination arguments were required. This performance bottleneck was introduced by proof techniques that combined the handling of termination arguments based on term structure and arithmetic termination arguments. Therefore, AProVE now uses (specialized) existing techniques for termination of integer transition systems (*ITS*s, which do not contain

data constructors) and for termination of ordinary TRSs (which do not contain integers). More precisely, if the int-TRS does not contain symbols from Σ_{pre}, then we use the standard dependency pair framework for termination of TRSs [37]. To analyze the termination behavior of int-TRSs without data constructors from Σ_c, we repeatedly synthesize integer ranking functions. For this, we use a variation of the constraint-based procedure of [8, 53] that uses the technique of [2] to find separate (and possibly different) linear ranking functions for every defined symbol. Additionally, we use the calculus presented in [27] to synthesize non-linear ranking functions for such int-TRSs.

To handle int-TRSs that contain both integers and terms, AProVE uses two methods. In one approach (Sect. 3.2.1), we consider projections of the analyzed int-TRS $\mathcal{R}$ either to ordinary term rewriting (by removing all integers) or to ordinary integer transition systems (by removing all constructor terms) and use specialized standard techniques. These projections affect only a single step in the termination proof (i.e., the information that was removed by the projection is again available in the next step of the termination proof). Thus, for the next step a possibly different projection can be used.

In the second approach (Sect. 3.2.2), we use a fixed term abstraction to integers to obtain an integer transition system. The used abstraction is similar to the "path-length" abstraction of [56], but it is employed at a later point of the analysis. Hence, it allows us to make use of more precise information during the symbolic execution in our frontends. In our implementation, we use both techniques in parallel instead of applying heuristics to choose the right technique.

3.2.1 Termination Proving by Projection

We show how to filter away all "integer arguments" resp. all "term arguments" of an int-TRS and how to lift termination proofs on a filtered rewrite system back to the original system. To distinguish between integer and term arguments, we first define a rudimentary type system on int-TRSs. To identify all integer arguments of function symbols, we mark every argument containing an integer constant or a variable that occurs in the constraint of the rule. Then, we propagate this information through the rewrite system. Similarly, we mark arguments that contain a constructor symbol from Σ_c as term arguments and propagate this information.

Definition 6 (Integer and Term Arguments) The set of *integer arguments* $\mathcal{IA} \subseteq (\Sigma \times \mathbb{N})$ of an int-TRS $\mathcal{R}$ is the smallest set with $(f, i) \in \mathcal{IA}$ if $1 \leq i \leq \mathit{arity}(f)$ and one of the following holds for some rule $\ell \to r \,[\![\varphi]\!]$ of $\mathcal{R}$.

- ℓ or r contain a subterm $f(t_1, \ldots, t_n)$ and $t_i \in \mathbb{Z} \cup \mathcal{V}(\varphi)$.
- ℓ or r contain a subterm $f(t_1, \ldots, t_n)$, $t_i \in \mathcal{V}$, a subterm $g(s_1, \ldots, s_m)$ occurs in the left- or right-hand side of some rule of $\mathcal{R}$, $t_i = s_j$, and $(g, j) \in \mathcal{IA}$.

The set of *term arguments* $\mathcal{TA} \subseteq (\Sigma \times \mathbb{N})$ of an int-TRS $\mathcal{R}$ is the smallest set with $(f, i) \in \mathcal{TA}$ if $1 \leq i \leq \mathit{arity}(f)$ and one of the following holds for some rule $\ell \to r \,[\![\varphi]\!]$ of $\mathcal{R}$.

- ℓ or r contain a subterm $f(t_1, \ldots, t_n)$ and $t_i = g(\ldots)$ for some $g \in \Sigma_c$.
- ℓ or r contain a subterm $f(t_1, \ldots, t_n)$, $t_i \in \mathcal{V}$, a subterm $g(s_1, \ldots, s_m)$ occurs in the left- or right-hand side of some rule of $\mathcal{R}$, $t_i = s_j$, and $(g, j) \in \mathcal{TA}$.

Example 7 Consider the following int-TRS $\mathcal{R}$.

$$\mathsf{f}(\mathsf{C}(v, xs), v) \to \mathsf{f}(\mathsf{C}(v', xs), v') \qquad [\![v > 0 \wedge v' = v - 1]\!] \tag{9}$$

$$\mathsf{f}(\mathsf{C}(v, xs), v) \to \mathsf{f}(xs, v') \qquad [\![v \leq 0]\!] \tag{10}$$

Here, we have $\mathcal{IA} = \{(\mathsf{f}, 2), (\mathsf{C}, 1)\}$ and $\mathcal{TA} = \{(\mathsf{f}, 1), (\mathsf{C}, 2)\}$.

There exist int-TRSs that are not well typed, i.e., where $\mathcal{IA} \cap \mathcal{TA} \neq \varnothing$. However, we never automatically generate such int-TRSs in our programming language front-ends, and thus will ignore such int-TRSs from now on.

For any term t, let $\mathcal{IA}(t)$ result from t by removing all term arguments, i.e., the ith argument of f is removed iff $(f, i) \notin \mathcal{IA}$. We define $\mathcal{TA}(t)$ analogously. Then for any int-TRS $\mathcal{R}$, we can define its integer and term projections.

$$\begin{aligned} \mathcal{IA}(\mathcal{R}) &= \{\mathcal{IA}(\ell) \to \mathcal{IA}(r) \,[\![\varphi]\!] \mid \ell \to r \,[\![\varphi]\!] \in \mathcal{R}\} \quad \text{and} \\ \mathcal{TA}(\mathcal{R}) &= \{\mathcal{TA}(\ell) \to \mathcal{TA}(r) \mid \ell \to r \,[\![\varphi]\!] \in \mathcal{R}\} \end{aligned}$$

By construction, $\mathcal{IA}(\mathcal{R})$ is an int-TRS without user-defined data structures (i.e., an integer transition system) and $\mathcal{TA}(\mathcal{R})$ is an ordinary TRS without built-in integers. Now we can use standard techniques to obtain *reduction pairs* $(\gtrsim, \succ)$ for the respective filtered systems [37]. A reduction pair for $\mathcal{R}$ partitions $\mathcal{R}$ into a set of rules $\mathcal{R}_\succ$ that decreases w.r.t. some well-founded order $\succ$ with every use, and a set $\mathcal{R}_\gtrsim \setminus \mathcal{R}_\succ$ that does not increase w.r.t. this order. Consequently, existence of a reduction pair $(\gtrsim, \succ)$ for $\mathcal{R}$ proves that the rules $\mathcal{R}_\succ$ cannot be used infinitely often. The crucial observation is that if we find a reduction pair for a projection of our int-TRS, then we can lift it to a reduction pair on the original system.

Theorem 8 (Reduction Pairs From Projections) *Let $\mathcal{R}$ be an int-TRS. If $(\gtrsim_{\mathcal{IA}}, \succ_{\mathcal{IA}})$ is a reduction pair for $\mathcal{IA}(\mathcal{R})$, then $(\gtrsim, \succ)$ is a reduction pair for $\mathcal{R}$ where $t_1 \succ t_2$ holds iff $\mathcal{IA}(t_1) \succ_{\mathcal{IA}} \mathcal{IA}(t_2)$ and $t_1 \gtrsim t_2$ holds iff $\mathcal{IA}(t_1) \gtrsim_{\mathcal{IA}} \mathcal{IA}(t_2)$. The same holds for the restriction $\mathcal{TA}$.*

To illustrate the approach, let us prove termination of the int-TRS in Ex. 7. In the first step, we project $\mathcal{R}$ to its term arguments and obtain $\mathcal{TA}(\mathcal{R})$.

$$\mathsf{f}(\mathsf{C}(xs)) \to \mathsf{f}(\mathsf{C}(xs)) \tag{11}$$

$$\mathsf{f}(\mathsf{C}(xs)) \to \mathsf{f}(xs) \tag{12}$$

Here, (9) and (11) correspond to each other, as do (10) and (12). Using standard techniques for term rewriting, we obtain a reduction pair $(\gtrsim_{\mathcal{TA}}, \succ_{\mathcal{TA}})$ with $(\mathcal{TA}(\mathcal{R}))_{\succ_{\mathcal{TA}}} = \{(12)\}$ and $(\mathcal{TA}(\mathcal{R}))_{\gtrsim_{\mathcal{TA}}} = \{(11)\}$ (e.g., by using the embedding order). We can lift this to a reduction pair on $\mathcal{R}$, and obtain $(\gtrsim, \succ)$ with $\mathcal{R}_\succ = \{(10)\}$ and $\mathcal{R}_\gtrsim = \{(9)\}$. Hence, we have proved termination of (10) and only need to prove termination of $\mathcal{R}' = \mathcal{R} \setminus \mathcal{R}_\succ = \{(9)\}$. In the second step, we consider its projection to integer arguments $\mathcal{IA}(\mathcal{R}')$.

$$\mathsf{f}(v) \to \mathsf{f}(v') \qquad [\![v > 0 \wedge v' = v - 1]\!] \tag{13}$$

Now we can easily obtain a reduction pair $(\gtrsim_{\mathcal{IA}}, \succ_{\mathcal{IA}})$ with $(\mathcal{IA}(\mathcal{R}'))_{\succ_{\mathcal{IA}}} = \{(13)\}$ $= \mathcal{IA}(\mathcal{R}')$. Again, we can lift this to $\mathcal{R}'$ and thus prove its termination. Note how this termination proof projects away integer information in the first step, but due to our lifting technique, we can make use of it again in the second proof step.

3.2.2 Termination Proving with Term Height Abstraction

As an alternative to the projection technique in Sect. 3.2.1, we use an integer abstraction for terms that is similar to the path-length abstraction for heap structures [56]. This allows us to handle examples that require reasoning about term structure and integers at the same time. For this, we replace terms by their "term height", i.e., by the number of nested constructors. For example, lists are represented by their length, and trees are represented by their height. Then we adapt the rewrite rules such that they constrain heights instead of matching and replacing terms. However, as the abstraction of terms to integers is fixed and somewhat coarse, it fails for algorithms whose termination relies on an intricate manipulation of data structures. For example, in-place tree-to-list flattening algorithms (which require treating "left" and "right" subtrees differently) cannot be handled. For a term t, we define its *term height* as follows.

$$\mathsf{th}(t) = \begin{cases} 0 & \text{if } t \in \mathcal{V} \cup \mathbb{Z} \\ 1 + \max\{\mathsf{th}(t_i) \mid 1 \leq i \leq n\} & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

Based on this, we want to transform terms $t = f(t_1, \ldots, t_n)$ with $f \in \Sigma_d$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathbb{Z} \cup \Sigma_c, \mathcal{V})$ into terms $\pi_{\mathsf{th}}(t) = f(h_1, \ldots, h_n)$, where the h_i approximate the term height $\mathsf{th}(t_i)$, and adapt the rewrite rules accordingly. Subterms t_i with $t_i \in \mathbb{Z}$ are not changed by the abstraction π_{th}. So from a rule $\mathsf{f}(\mathsf{C}(v, xs)) \to \mathsf{f}(xs)$, we want to derive automatically that the height h of the argument on the left-hand side is at least 1, and that the height h' of the argument on the right-hand side is at least 1 smaller than h. This yields the rule $\mathsf{f}(h) \to \mathsf{f}(h') \, [\![h \geq 1 \wedge h' \leq h - 1]\!]$, for which we can easily prove termination.

Example 9 Consider the rewrite sequence obtained for $\mathsf{C}(1, \mathsf{C}(4, \mathsf{C}(7, xs)))$ with the int-TRS $\mathcal{R} = \{\mathsf{f}(\mathsf{C}(v, xs)) \to \mathsf{f}(xs)\}$. Below each term t we denote the term $\pi_{\mathsf{th}}(t)$, i.e., the result of replacing non-integer subterms by their height.

$$\underbrace{\mathsf{f}(\mathsf{C}(1, \mathsf{C}(4, \mathsf{C}(7, xs))))}_{\mathsf{f}(3)} \hookrightarrow \underbrace{\mathsf{f}(\mathsf{C}(4, \mathsf{C}(7, xs)))}_{\mathsf{f}(2)} \hookrightarrow \underbrace{\mathsf{f}(\mathsf{C}(7, xs))}_{\mathsf{f}(1)} \hookrightarrow \underbrace{\mathsf{f}(xs)}_{\mathsf{f}(0)}$$

We now define a rule translation Π_{th} that is "compatible" with π_{th}, i.e., where a rewrite step $s \hookrightarrow_\rho t$ with a rule ρ implies $\pi_{\mathsf{th}}(s) \hookrightarrow_{\Pi_{\mathsf{th}}(\rho)} \pi_{\mathsf{th}}(t)$. Then, evaluations can be reproduced in the translated rewrite system, and hence, the translation preserves non-termination. In other words, a termination proof of the translated rewrite system also implies termination of the original rewrite system.

To this end, consider a rule $f(\ell_1, \ldots, \ell_n) \to g(r_1, \ldots, r_m) \, [\![\varphi]\!]$. As mentioned above, we do not need to change those arguments of f and g that are not term arguments. When replacing a term argument ℓ_i by a variable h_i representing its height, we use that the height of any instantiation of ℓ_i in a rule application will be at least $\mathsf{th}(\ell_i)$ since $\mathsf{th}(\ell_i\sigma) \geq \mathsf{th}(\ell_i)$ for any data substitution σ. Thus, we add a constraint $h_i \geq \mathsf{th}(\ell_i)$.

For a term t, let $\mathcal{V}_{\mathcal{TA}}(t)$ be the variables occurring at term positions in t, i.e., $\mathcal{V}_{\mathcal{TA}}(t) = \{x \in \mathcal{V} \mid t \text{ has a subterm } f(t_1, \ldots, t_n) \text{ with } (f, i) \in \mathcal{TA} \text{ and } t_i = x\}$. For all variables $x \in \mathcal{V}_{\mathcal{TA}}(\ell_i)$, we know that whenever we instantiate ℓ_i with σ in the application of a rule, the height of $\sigma(x)$ will be at most the height of $\sigma(\ell_i)$, and hence $x \leq h_i$. This can be made more precise by taking into account how deeply

"nested" x appears in ℓ_i. For instance, in $\ell_i = \mathsf{C}(2, \mathsf{C}(1, x))$, the height of ℓ_i is 2 plus the height of x because x is nested two levels deep in ℓ_i. Thus, the *nesting level*[9] of x in ℓ_i is $\mathsf{nl}(\ell_i, x) = 2$. So in general, if h_i is the height of ℓ_i, then the difference between h_i and the height of x will be at least as large as the nesting level $\mathsf{nl}(\ell_i, x)$ of x in ℓ_i. Hence, we add the constraint $x + \mathsf{nl}(\ell_i, x) \leq h_i$.

Finally, note that the height h'_i of any argument r_i on the right-hand side is no larger than the maximum of $\mathsf{th}(r_i)$ and of the heights of the variables occurring at term arguments in r_i plus their respective nesting levels. Thus, we add the constraint $h'_i \leq \max\{\mathsf{th}(r_i), \max\{x + \mathsf{nl}(r_i, x) \mid x \in \mathcal{V}_{\mathcal{TA}}(r_i)\}\}$. In practice, representing the maximum is not feasible due to its inherent blowup, as our int-TRSs support no pre-defined max operator. Thus, each max constraint would have to be represented by a disjunction of all possible cases. Therefore, in AProVE, we replace each max term by the sum of its arguments if we cannot statically determine the maximum element.

Definition 10 (Term Height Projection for Rules) Let $\mathcal{TA}$ be as in Def. 6 and $\rho = f(\ell_1, \ldots, \ell_n) \to g(r_1, \ldots, r_m)\,[\![\varphi]\!]$ be a rewrite rule. Then we define the *term height projection for* ρ as $\Pi_{\mathsf{th}}(\rho) = f(\ell'_1, \ldots, \ell'_n) \to g(r'_1, \ldots, r'_m)\,[\![\varphi \wedge \psi]\!]$ where ℓ'_i is a fresh variable h_i if $(f, i) \in \mathcal{TA}$, and otherwise we have $\ell'_i = \ell_i$. Similarly, r'_i is a fresh variable h'_i if $(g, i) \in \mathcal{TA}$ and otherwise $r'_i = r_i$. The constraint ψ is defined as follows.

$$\begin{aligned}\psi = \bigwedge_{\substack{1 \leq i \leq n \\ (f,i) \in \mathcal{TA}}} \Bigg(h_i \geq \mathsf{th}(\ell_i) \wedge \bigwedge_{x \,\in\, \mathcal{V}_{\mathcal{TA}}(\ell_i)} (x + \mathsf{nl}(\ell_i, x) \leq h_i \wedge x \geq 0) \Bigg) \\ \wedge \bigwedge_{\substack{1 \leq i \leq m \\ (g,i) \in \mathcal{TA}}} \Bigg(h'_i \geq \mathsf{th}(r_i) \wedge \bigwedge_{x \,\in\, \mathcal{V}_{\mathcal{TA}}(r_i)} (x + \mathsf{nl}(r_i, x) \leq h'_i \wedge x \geq 0) \\ \wedge\, h'_i \leq \max\{\mathsf{th}(r_i), \max\{x + \mathsf{nl}(r_i, x) \mid x \in \mathcal{V}_{\mathcal{TA}}(r_i)\}\} \Bigg)\end{aligned}$$

We extend Π_{th} to int-TRSs $\mathcal{R}$ by defining $\Pi_{\mathsf{th}}(\mathcal{R}) = \{\Pi_{\mathsf{th}}(\rho) \mid \rho \in \mathcal{R}\}$.

Theorem 11 (Soundness of Term Height Projection) *Let $\mathcal{R}$ be an int-TRS. If $\Pi_{\mathsf{th}}(\mathcal{R})$ terminates, then $\mathcal{R}$ also terminates.*

The following examples demonstrate how well suited the term height projection is to prove termination of standard iterations on user-defined data structures.

Example 12 For $\mathcal{R} = \{\mathsf{f}(\mathsf{C}(v, xs)) \to \mathsf{f}(xs)\}$ from Ex. 9, $\Pi_{\mathsf{th}}(\mathcal{R})$ consists of the rule

$$\mathsf{f}(h_1) \to \mathsf{f}(h'_1) \;\; [\![h_1 \geq 1 \wedge xs + 1 \leq h_1 \wedge xs \geq 0 \wedge h'_1 \geq 0 \wedge xs \leq h'_1 \wedge h'_1 \leq xs]\!]$$

Note that the constraint of the rule can be simplified to $h_1 \geq 1 \wedge h'_1 \leq h_1 - 1 \wedge h'_1 \geq 0$. Now indeed, we have $\mathsf{f}(3) \hookrightarrow_{\Pi_{\mathsf{th}}(\mathcal{R})} \mathsf{f}(2) \hookrightarrow_{\Pi_{\mathsf{th}}(\mathcal{R})} \mathsf{f}(1) \hookrightarrow_{\Pi_{\mathsf{th}}(\mathcal{R})} \mathsf{f}(0)$ and termination can easily be proved.

[9] The nesting level of a variable x in a term $t = f(t_1, \ldots, t_n)$ is $\mathsf{nl}(t, x) = 1 + \max\{\mathsf{nl}(t_i, x) \mid 1 \leq i \leq n, x \in \mathcal{V}_{\mathcal{TA}}(t_i)\}$ if $x \in \mathcal{V}_{\mathcal{TA}}(t)$, where $\mathsf{nl}(x, x) = 0$, and $\mathsf{nl}(t, x) = \infty$ if $x \notin \mathcal{V}_{\mathcal{TA}}(t)$.

Configuration	Term.	Failure	Timeout	avg. Res. (s)	avg. Run. (s)
AProVE-ITRS	239	12	60	11.50	67.42
AProVE-TermHeight	251	10	50	5.67	52.98
AProVE-Project	266	30	15	6.44	20.94
AProVE-IntTerm	285	7	19	6.36	24.51
AProVE-IntTerm-noSimp	146	4	161	18.84	164.21
AProVE-IntTerm-onlySlice	159	5	147	18.78	151.50
AProVE-IntTerm-onlyChain	277	9	25	6.17	30.85

Fig. 1 Experimental Results for Termination of int-TRSs

Similarly, for the TRS $\mathcal{R}' = \{\mathsf{f}(\mathsf{C}(v_1, \mathsf{C}(v_2, xs))) \to \mathsf{f}(\mathsf{C}(v_2, xs))\}$, $\Pi_{\mathsf{th}}(\mathcal{R}')$ consists of the rule

$$\mathsf{f}(h_1) \to \mathsf{f}(h_1') \, [\![h_1 \geq 2 \wedge xs + 2 \leq h_1 \wedge xs \geq 0 \wedge h_1' \geq 1 \wedge xs + 1 \leq h_1' \wedge h_1' \leq xs + 1]\!]$$

Here, we generated the condition $h_1' \leq \max\{1, xs+1\}$, which we simplified to $h_1' \leq xs + 1$. The constraint can be further simplified to $h_1 \geq 2 \wedge h_1' \leq h_1 - 1 \wedge h_1' \geq 1$. Again, termination is easily proved.

3.3 Evaluation

To show the usefulness of our techniques from Sect. 3, we evaluated different versions of AProVE on the standard benchmarks from version 8 of the *Termination Problem Data Base* (*TPDB*) for Java programs.[10] The *TPDB* is the collection of problems used in the *Termination Competition* [62]. However, here we excluded benchmarks that are known to be non-terminating, resulting in a collection of 311 examples. The results of these experiments are displayed in Fig. 1.

We performed our experiments on a computer with 6 GB of RAM and an Intel i7 CPU clocked at 3.07 GHz using a timeout of 300 seconds for each example (running with a higher timeout of 6000 s did not yield additional results). The column "Term." shows the number of examples where termination could be proved. "Failure" are those examples where AProVE failed within 300 s and "Timeout" are the examples where AProVE was stopped after 300 s. The last two columns document the average runtime (in seconds), where "avg. Res." is the average restricted to successful proof attempts and "avg. Run." is the average for all examples.

All evaluated versions of AProVE used our frontend for Java. In the first four experiments, we enabled all simplification techniques and used different variants of the termination backend described in Sect. 3.2. In AProVE-ITRS, we used *integer term rewrite systems* [33] instead of int-TRSs. While this formalism is more expressive than int-TRSs, it has the drawback of being less efficient, resulting in a lower of number of examples whose termination can be proved. Therefore, the remaining versions of AProVE in Fig. 1 used int-TRSs in the backend.

In AProVE-TermHeight, we enabled only the term height projection from Thm. 11, and then applied standard techniques for ITSs. In AProVE-Project, we used

[10] The results for the C benchmarks are similar.

only the lifting of reduction pairs from Thm. 8, and applied standard TRS and ITS techniques. Finally, AProVE-IntTerm combines both techniques and is the most powerful configuration in Fig. 1.

In a second group of experiments, we used AProVE-IntTerm as basis, and varied the applied simplification techniques (Sect. 3.1). In AProVE-IntTerm-noSimp, we disabled all such techniques, in AProVE-Term-onlySlice, we used only the slicing technique from Thm. 3, and in AProVE-IntTerm-onlyChain, we used only the chaining technique from Thm. 5. The experiments clearly show that both of the simplifications are useful and that their combination (in AProVE-IntTerm) leads to the most powerful configuration. So the contributions of Sect. 3 are indeed crucial for applying AProVE to real programs.

4 Automation via SAT and SMT Solving

To solve the arising search problems in (non-)termination or complexity proofs, AProVE uses encodings to satisfiability problems of logics. Satisfiability of propositional formulas can be checked by SAT solvers, whereas more complex logics require a SAT modulo theory (SMT) solver for an appropriate theory. Depending on the kinds of numbers and the possible nesting depth of function symbols from $\Sigma_c \cup \Sigma_d$ in rewrite rules we use the logics *Quantifier-Free Linear Integer Arithmetic* (QF_LIA), *Quantifier-Free Non-Linear Integer Arithmetic* (QF_NIA), *Quantifier-Free Linear Real Arithmetic* (QF_LRA), and *Quantifier-Free Non-Linear Real Arithmetic* (QF_NRA). Any improvements to the SAT and SMT solvers applicable to these logics would thus also benefit program verification in AProVE.

In the following, we review the search problems tackled by SAT or SMT solving in AProVE and explain which underlying SMT logic we use for their encoding. For SAT solving, AProVE uses the tools MiniSat [24] and SAT4J [47]. Like AProVE, SAT4J is implemented in Java and hence, AProVE calls it for small SAT instances, where it is very efficient. MiniSat is used on larger SAT instances, but as it is invoked as an external process, it leads to a small overhead. For SMT solving, AProVE uses Z3 [21], Yices [22], and SMTInterpol [15]. Similarly to SAT4J, SMTInterpol is written in Java and thus, avoids the overhead for calling a non-Java tool. In Sect. 4.1 we discuss the use of SMT solving in AProVE's frontends, whereas Sect. 4.2 focuses on the application of SAT and SMT solvers in AProVE's backend. AProVE's techniques for SMT solving over non-linear integer arithmetic can also be accessed directly, allowing to use AProVE as an SMT solver for QF_NIA, cf. Sect. 4.3. Finally, in Sect. 4.4 we describe the low-level optimizations that AProVE uses for its SAT encodings.

4.1 Techniques in the Programming Languages Frontends

There are two main applications for SMT solving in our frontends. During the construction of the symbolic execution graph, some executions are infeasible and thus do not need to be considered. For the integer fragment of the analyzed programs, AProVE uses (incomplete) heuristics to detect typical unsatisfiable conditions quickly, and external SMT solvers for more complex cases.

The other main application for SMT solving in AProVE's frontends is prov-

ing *non*-termination of Java programs. The transformation from a programming language to (int-)TRSs corresponds to an over-approximation of the original program, and hence, non-termination proofs operate on the symbolic execution graph instead of the resulting (int-)TRSs. In [11], we presented two such techniques for Java, where both strongly rely on SMT solving. The target logics are QF_LIA and QF_NIA (the latter is used if the program has non-linear operations).

4.2 Techniques in the Backend

In AProVE's backend, SAT and SMT solving is used to automatically find termination proofs. As mentioned in Sect. 3.2, for integer transition systems, AProVE uses the approaches from [2, 8, 27, 53] to search for linear ranking functions. To make use of information about the start of computations, AProVE also provides an implementation of the safety prover-based T2 algorithm [13, 19], using a variant of the Impact safety proving method [48]. For these, AProVE uses SMT solving for the logics QF_LIA and QF_LRA.

To prove termination of term rewrite systems, AProVE can find termination arguments from several classes of well-founded orders by encoding the search into SAT or SMT problems. We now describe the use of SAT and SMT solving for these orders in AProVE. Here, we put most emphasis on orders based on *polynomial interpretations* [45], because these are the orders that are used most often for those TRSs that result from the transformations of programs.

Polynomial Interpretations. Essentially, a polynomial interpretation maps function symbols f to polynomials $f_{\mathcal{P}ol}$ over $\mathbb{N}$. This mapping extends homomorphically to terms, i.e., $[x]_{\mathcal{P}ol} = x$ for variables x and $[f(t_1, \ldots, t_n)]_{\mathcal{P}ol} = f_{\mathcal{P}ol}([t_1]_{\mathcal{P}ol}, \ldots, [t_n]_{\mathcal{P}ol})$ for terms $f(t_1, \ldots, t_n)$. Then to compare two terms $s \succ t$, one has to perform a comparison $[s]_{\mathcal{P}ol} > [t]_{\mathcal{P}ol}$ of polynomials over the natural numbers.[11]

A termination prover has to automatically *find* suitable interpretations $\mathcal{P}ol$. For this, AProVE generates SMT formulas w.r.t. the logic QF_NIA with template unknowns for the coefficients. These formulas are solved via a *bitblasting* approach [30]. Here, bitblasting refers to representing the template unknowns as fixed-width bitvectors and to encoding integer operations as Boolean circuits. Using bitblasting is successful for this application as coefficients from $\{0, \ldots, k\}$ for small values of k are usually sufficient in practice (e.g., $k = 3$; see also the discussion in the experiments section of [30]). Since not finding a solution for the constraints does not imply non-termination, being incomplete is not problematic for our application.

We also implemented variants of polynomial interpretations over $\mathbb{Q}^+$ or $\mathbb{R}^+$ [32]. For efficiency, AProVE again reduces the resulting search problems to QF_NIA instead of using dedicated QF_NRA solvers. While QF_NIA is undecidable and QF_NRA is decidable, decision procedures for QF_NRA are extremely inefficient.

AProVE applies its bitblasting approach to solve QF_NIA also for other extensions of classic polynomial interpretations, such as polynomial interpretations with negative coefficients to prove bounded increase [39], polynomial interpretations with max and min operators [30, 31], matrix interpretations [26], and partly

[11] So this concept of polynomial interpretations encompasses both polynomial ranking functions for defined function symbols and size measures / norms / abstractions for constructors of data structures.

strongly monotonic polynomial interpretations suitable for a combination with inductive theorem proving [34]. Moreover, AProVE uses polynomial interpretations not only for termination proving, but also for inferring bounds for the *runtime complexity* of TRSs [51] and Prolog programs [41].

Arctic Matrix Interpretations. Matrix interpretations [26] are an extension of polynomial interpretations where terms are mapped to tuples instead of single numbers. In *arctic* matrix interpretations [44], one uses the max-plus semi-ring instead of the conventional plus-times semi-ring. The constraints to solve when searching for such interpretations can be represented as QF_LIA constraints. However, for such interpretations AProVE applies a SAT encoding to bitvectors with *unary* arithmetic (a variant of *order encoding* [61]). To represent k different numbers, this encoding uses bitvectors of length $k-1$, whereas the "usual" encoding to binary arithmetic only requires bitvectors of length $\lceil \log_2(k) \rceil$. In our experiments, our SAT-based iterative deepening approach for the search space outperforms dedicated SMT solvers for QF_LIA since on our instances, solutions can often be found within a small search space. Moreover, our encodings to unary arithmetic outperform more compact SAT encodings to binary arithmetic for the constraints of arctic matrix interpretations because of the improved propagation of such encodings [16].

KBO and RPO. AProVE also uses an SMT encoding for *Knuth-Bendix orders (KBO)* [64]. Here the target logics are QF_LIA or QF_LRA. For the *recursive path order (RPO)* and its variants, AProVE applies dedicated SAT encodings [17].

4.3 AProVE as an SMT Solver for QF_NIA

AProVE also provides an SMT-LIB 2 [5] frontend to access its bitblasting technique [30] for the QF_NIA logic. In fact, AProVE participated successfully in the *SMT-COMP* [54] competitions of 2010, 2011, 2014, and 2015 for QF_NIA. Since the examples in the underlying SMT-LIB library stem from various application domains with different requirements, an iterative deepening approach was used to determine the search space to be encoded to SAT. With this approach, AProVE reached the first place in this category of *SMT-COMP* in 2011, 2014, and 2015.

The performance on some QF_NIA instances could be improved further if AProVE were extended to detect when the search space is bounded. Then for a formula φ like $x \geq 10 \wedge y = x \wedge y \leq 20 \wedge \cdots$ we would detect that φ is satisfiable over $\mathbb{Z}$ iff φ is satisfiable over $\{10, 11, \ldots, 20\}$ for the unknowns x, y. This would not only improve efficiency, but it would also allow AProVE to return UNSAT, as UNSAT over $[10, 20]$ for x, y would also imply UNSAT over $\mathbb{Z}$. We have not implemented this optimization, though, as the instances obtained from termination and complexity analysis typically do not exhibit such patterns and UNSAT results are usually not interesting for termination and complexity analysis.

To access the SMT-LIB 2 frontend of AProVE, the following command line can be used, where `foo.smt2` contains the satisfiability problem.

```
java -ea -jar aprove.jar -d diologic -m smtlib foo.smt2
```

Here, `diologic` denotes *Diophantine logic* (Diophantine (i.e., integer polynomial) inequations connected by arbitrary Boolean connectives), which in SMT-LIB ter-

minology is the logic QF_NIA. The flag `-m` specifies the output format. If `smtlib` is used, AProVE prints `sat` or `unknown` as a very first answer. Note that this way of using AProVE as an SMT solver for QF_NIA incurs the overhead of starting the Java virtual machine, loading the Java Bytecode for AProVE and relevant libraries, and just-in-time-compiling this bytecode. While such an overhead is acceptable in many applications, it is prohibitively expensive when AProVE is used to solve a large number of dynamically generated small instances (e.g., this is common in termination analysis). For such applications, AProVE offers a "server" mode, where it is started once and then receives a stream of problems, outputting results in the same order. For example, this approach was chosen to integrate AProVE as an SMT solving backend for the logic programming termination analyzer Polytool [49].

4.4 Low-Level Optimizations for SAT Solving in AProVE

AProVE uses some low-level optimizations on the generated SAT formulas which are crucial for the efficiency of the subsequent SAT solving. Virtually all SAT solvers take as input only propositional formulas in conjunctive normal form (CNF). Thus, the formulas resulting from the encoding of the search problems in AProVE are transformed to equisatisfiable CNFs before calling external SAT solvers. For this transformation, AProVE uses the Tseitin conversion built into SAT4J.

Tseitin's transformation incurs a linear overhead in the size of the formula converted by replacing each non-atomic subformula $x \circ y$ by a fresh Boolean variable z and by adding clauses to enforce that z and $x \circ y$ must be equivalent.

To minimize this overhead, AProVE uses optimizations based on identities over Boolean formulas, e.g., cancellation for exclusive-or ($x \oplus x \equiv 0$) or trivially valid implications ($0 \rightarrow x \equiv 1$). In addition to such local use of identities, AProVE globally identifies cases of equivalent subformulas by treating equality for $\wedge$ and $\vee$ modulo associativity, commutativity, and multiplicity, e.g., identifying both $x \vee (y \vee x)$ and $(x \vee y) \vee x$ with $(x \vee y)$. Thus, only one additional variable is introduced by Tseitin's transformation for all occurrences of such equivalent subformulas.

In principle, we could first construct the formulas from our encodings and then post-process them using these optimizations. This process would be efficient enough for the local identities, but vastly inefficient for global ones. Thus, in AProVE we widely make use of *structural hashing*, i.e., we represent formulas as a directed acyclic graph where all syntactically equal subformulas (modulo associativity, commutativity, and multiplicity for $\wedge$ and $\vee$) share the same node.

5 Certification of Generated Proofs

Like any large software product, AProVE had (and very likely still has) bugs. To allow a verification of its results, it can export generated termination, non-termination, or complexity proofs as machine-readable CPF (Certification Problem Format)[12] files by clicking on the button  of the Proof Tree View. Independent certifiers can then check the validity of all proof steps. Examples for such certifiers are CeTA [63], CiME/Coccinelle [18], and CoLoR/Rainbow [7]. Their correctness has

[12] See `http://cl-informatik.uibk.ac.at/software/cpf/`

been formally proved using Isabelle/HOL [50] or Coq [6]. To certify a proof in AProVE's GUI, one can also call CeTA directly using the button ☑ of the Proof Tree View. At the moment, certification is available for most proof techniques operating on term rewrite systems, but we cannot yet certify proof techniques directly operating on Java, C, Haskell, or Prolog.

The "AProVE" entry in Eclipse's main menu allows to modify the configuration of AProVE. Most notably, AProVE can run in either *full* or *certified* mode. Moreover, *online* certification can be enabled or disabled. Finally, AProVE can use its *default* proof strategy or a user-defined *custom* proof strategy, where the choice of the strategy is independent of certification. In the following we explain the difference between the *full* and the *certified* mode as well as the difference between *offline* and *online* certification.

Full Mode with Partial Certificates. In the *full mode* of AProVE, arbitrary proof techniques may be used. However, not all of these techniques can be exported to CPF to be certified afterwards. In order to still provide certifiable proofs in the full mode of AProVE, we generate *partial certificates*. Those steps which are covered by CPF are exported for certification, and all remaining ones are ignored. More precisely, we extended CPF by an additional element unknownProof for proof steps which are not supported by CPF. During certification, unknownProof is treated as an axiom of the form $P_0 \longleftarrow P_1 \wedge \cdots \wedge P_n$. This allows to prove $P_1, \ldots, P_n$ instead of the desired property P_0. Each P_i can be an arbitrary property such as (non-)termination of some TRS, and P_i's subproof can be checked by the certifier again. In this way, it is possible to certify large parts of every termination proof generated by AProVE. For example, now 90% of AProVE's proof steps for termination analysis of the 4367 TRSs in the *TPDB* can be certified by CeTA.

Moreover, we added a new CPF element unknownInput for properties that cannot be expressed in CPF, like termination of a Java program. The only applicable proof step to such a property is unknownProof (i.e., the proof step from the Java program to an (int-)TRS is not supported by CPF either). Using unknownInput, CPF files for *every* proof can be generated. Now the program transformations in AProVE's frontends correspond to unknown proof steps on unknown inputs, but the reasoning in AProVE's backend can still be checked by a certifier (i.e., proof steps can transform unknownInput into objects that are expressible in CPF). To implement partial certification, AProVE analyzes the generated proof tree and for each proof step, it is checked whether it can be exported to CPF or not.

Certified Mode. In the *certified mode*, AProVE is restricted to proof techniques that can be exported to CPF and subsequently be verified. In principle, this could be restricted further to the techniques supported by a particular chosen certifier. These restrictions mean that certain proof techniques have to be disabled completely and that for other proof techniques, some optimizations must be turned off. Often, these restrictions have to be combined, e.g., some optimizations in the process of generating ordering constraints may not be used while also restricting the search to a certain class of well-founded orders. So the certified mode performs *a priori restrictions* on *proof techniques*, whereas in the full mode, certification is done *a posteriori* on the generated *proof* without looking at the configuration of the proof techniques.

However, on input problems which correspond to unknownInput (i.e., where there is no CPF export at all), even in certified mode *all* techniques may be used in the proof search. For example, this holds for the transformations in AProVE's frontends, since currently, CPF is defined only for problems resulting from term rewriting. Once an intermediate problem is reached which corresponds to a well-defined problem type of CPF, only certifiable techniques are used. For such problems, the proofs in the certified mode are fully certifiable, but at the cost of having to restrict the proof techniques and thereby the power of AProVE. Thus, for some examples, no proof can be found in the certified mode, whereas the proof succeeds in the full mode.

Previous versions of AProVE did not have such a certified mode. Therefore, specialized proof strategies had to be maintained and used to make sure that only certifiable techniques were used. This maintenance of two strategies – the default one and the one for certifiable proofs – resulted in quite some overhead. Moreover, non-expert users could not easily determine which techniques were allowed to ensure certifiable proof output. In contrast, in the current AProVE version, the choice between certified and full mode is independent of the strategy. Thus, the same default strategy can be used for both the certified and the full mode. Here, the essential idea is that the restrictions on the proof techniques are not encoded in the strategy as before, but instead they are enforced at runtime. These runtime checks are hardcoded in AProVE via a mixture of whitelists and blacklists with a minimal amount of effort: by default, no new technique is admitted, and by overwriting the implementation of the CPF-export method, a proof technique becomes amenable for certification. Afterwards one can (partially) deactivate the technique for specific certifiers which do not support the technique, or turn off specific optimizations. As a result, the previous stand-alone certified strategy of AProVE for CeTA (a 27k text-file) was replaced by a few lines of code. We expect that such a certified mode can also be useful for other termination analysis tools to ease their development and make their techniques more accessible for certification.

To summarize, in both the full and the certified mode, AProVE may now generate partial certificates which may contain unknown proofs and unknown inputs. However, in the certified mode only certifiable techniques are applied whenever possible, i.e., unknownProofs are only possible for unknownInputs.

Online Certification. Note that regardless of whether one uses the full or the certified mode, the generated certificate will contain only proof steps that contribute to the final proof. In particular, if AProVE was aborted during proof search, then nothing is certified at all; and if AProVE finds a proof of non-termination, then those proofs steps which claim termination of other parts of the input are not checked. This is perfectly fine if only the validity of generated proofs should be established. However, the idea of *online certification* is to use certification as a debugging utility for AProVE itself, where the aim is to increase the coverage of the performed resp. attempted proof steps.

In online certification, every CPF-exportable proof step is immediately checked by the certifier, no matter whether this step contributes to the final proof, and no matter whether a full proof can be found at all. Since the final result is unknown at this point (e.g., it is unknown whether the input is terminating or non-terminating), AProVE often has to export two proofs for each proof step, namely

a partial termination proof and a partial non-termination proof. Here, partiality refers to the fact that we assume termination (resp. non-termination) of the resulting problem, which is the way how implications can be expressed in CPF. So one has to check whether a proof step was sound when attempting to prove termination and whether it was sound when attempting to prove non-termination.

Clearly, online certification imposes some runtime overhead, but it also has the highest coverage w.r.t. error detection. At the moment, online certification is only available for the certifier CeTA.

Thanks to the new concepts of partial certificates and online certification, three bugs of AProVE have been revealed (and fixed) which could be exploited to prove termination of a non-terminating TRS. These bugs had not been discovered before by certification, as the errors occurred when analyzing TRSs resulting from logic programs.

6 Conclusion

In this system description, we presented a new version of AProVE to analyze termination of TRSs and programs for four languages from prevailing programming paradigms (Java, C, Haskell, and Prolog). Moreover, AProVE analyzes the runtime complexity of Prolog programs and TRSs. We are currently working on extending AProVE's complexity analysis to Java as well [14, 29].

AProVE's power is demonstrated by its regular performance in the annual *Termination Competition* [62] and the *SV-COMP* competition [60], where it won most categories related to termination of Java, C, Haskell, Prolog, and to termination or innermost runtime complexity of TRSs. AProVE's automatically generated termination proofs can be exported to (partially) check them by automatic certifiers. Our tool is available as a plug-in of the Eclipse software development environment. Moreover, the frontends of AProVE for the different programming languages can also be used separately to couple them with alternative backends. AProVE is available for download and can be accessed via a web interface [4].

References

1. Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Removing useless variables in cost analysis of Java Bytecode. In *SAC '08*, pages 368–375, 2008.
2. Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS '10*, pages 117–133, 2010.
3. María Alpuente, Santiago Escobar, and Salvador Lucas. Removing redundant arguments automatically. *TPLP*, 7(1-2):3–35, 2007.
4. AProVE. `http://aprove.informatik.rwth-aachen.de/`.
5. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB standard: Version 2.5. Technical report, The University of Iowa, 2015. Available at `http://smt-lib.org/`.
6. Yves Bertot and Pierre Castéran. *Coq'Art*. Springer, 2004.
7. Frédéric Blanqui and Adam Koprowski. CoLoR: A Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science*, 4:827–859, 2011.
8. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *CAV '05*, pages 491–504, 2005.

9. Tim Bray. The JavaScript object notation (JSON) data interchange format. 2014. RFC 7159.
10. Marc Brockschmidt, Carsten Otto, and Jürgen Giesl. Modular termination proofs of recursive Java Bytecode programs by term rewriting. In *RTA '11*, pages 155–170, 2011.
11. Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated detection of non-termination and `NullPointerExceptions` for Java Bytecode. In *FoVeOOS '11*, pages 123–141, 2012.
12. Marc Brockschmidt, Richard Musiol, Carsten Otto, and Jürgen Giesl. Automated termination proofs for Java programs with cyclic data. In *CAV '12*, pages 105–122, 2012.
13. Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *CAV '13*, pages 413–429, 2013.
14. Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Analyzing runtime and size complexity of integer programs. *ACM TOPLAS*, 38(4):13:1–13:50, 2016.
15. Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In *SPIN '12*, pages 248–254, 2012.
16. Michael Codish, Yoav Fekete, Carsten Fuhs, Jürgen Giesl, and Johannes Waldmann. Exotic semiring constraints (extended abstract). In *SMT '12*, pages 87–96, 2012.
17. Michael Codish, Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. SAT solving for termination proofs with recursive path orders and dependency pairs. *JAR*, 49(1):53–93, 2012.
18. Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Automated certified proofs with CiME3. In *RTA '11*, pages 21–30, 2011.
19. Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. lexicographic termination proving. In *TACAS '13*, pages 47–61, 2013.
20. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252, 1977.
21. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS '08*, pages 337–340, 2008.
22. Bruno Dutertre and Leonardo Mendonça de Moura. The Yices SMT solver, 2006. Tool paper at `http://yices.csl.sri.com/tool-paper.pdf`.
23. Eclipse. `http://www.eclipse.org/`.
24. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT '03*, pages 502–518, 2004.
25. Fabian Emmes, Tim Enger, and Jürgen Giesl. Proving non-looping non-termination automatically. In *IJCAR '12*, pages 225–240, 2012.
26. Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *JAR*, 40(2–3):195–220, 2008.
27. Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination analysis of C programs using compiler intermediate languages. In *RTA '11*, pages 41–50, 2011.
28. Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder. Inferring lower bounds for runtime complexity. In *RTA '15*, pages 334–349, 2015.
29. Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl. Lower runtime bounds for integer programs. In *IJCAR '16*, pages 550–567, 2016.
30. Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. SAT solving for termination analysis with polynomial interpretations. In *SAT '07*, pages 340–354, 2007.
31. Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. Maximal termination. In *RTA '08*, pages 110–125, 2008.
32. Carsten Fuhs, Rafael Navarro-Marset, Carsten Otto, Jürgen Giesl, Salvador Lucas, and Peter Schneider-Kamp. Search techniques for rational polynomial orders. In *AISC '08*, pages 109–124, 2008.
33. Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp, and Stephan Falke. Proving termination of integer term rewriting. In *RTA '09*, pages 32–47, 2009.
34. Carsten Fuhs, Jürgen Giesl, Michael Parting, Peter Schneider-Kamp, and Stephan Swiderski. Proving termination by dependency pairs and inductive theorem proving. *JAR*, 47(2):133–160, 2011.
35. Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated termination proofs with AProVE. In *RTA '04*, pages 210–220, 2004.

36. Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. Proving and disproving termination of higher-order functions. In *FroCoS '05*, pages 216–231, 2005.
37. Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing and improving dependency pairs. *JAR*, 37(3):155–203, 2006.
38. Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *IJCAR '06*, pages 281–286, 2006.
39. Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp. Proving termination by bounded increase. In *CADE '07*, pages 443–459, 2007.
40. Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski, and René Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM TOPLAS*, 33(2):7:1–7:39, 2011.
41. Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs. Symbolic evaluation graphs and term rewriting – A general methodology for analyzing logic programs. In *PPDP '12*, pages 1–12, 2012.
42. Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving termination of programs automatically with AProVE. In *IJCAR '14*, pages 184–191, 2014.
43. Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder. Proving termination of programs with bitvector arithmetic by symbolic execution. In *SEFM '16*, pages 234–252, 2016.
44. Adam Koprowski and Johannes Waldmann. Max/plus tree automata for termination of term rewriting. *Acta Cybernetica*, 19(2):357–392, 2009.
45. Dallas Lankford. On proving term rewriting systems are Noetherian. Technical Report Memo MTP-3, Louisiana Technical University, 1979.
46. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–88, 2004.
47. Daniel Le Berre and Anne Parrain. The SAT4J library, release 2.2. *JSAT*, 7:59–64, 2010.
48. Ken McMillan. Lazy abstraction with interpolants. In *CAV '06*, pages 123–136, 2006.
49. Manh Thang Nguyen, Danny De Schreye, Jürgen Giesl, and Peter Schneider-Kamp. Polytool: Polynomial interpretations as a basis for termination analysis of logic programs. *TPLP*, 11(1):33–63, 2011.
50. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.
51. Lars Noschinski, Fabian Emmes, and Jürgen Giesl. Analyzing innermost runtime complexity of term rewriting by dependency pairs. *JAR*, 51(1):27–56, 2013.
52. Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *RTA '10*, pages 259–276, 2010.
53. Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI '04*, pages 239–251, 2004.
54. *SMT-COMP*. `http://www.smt-comp.org/`.
55. Fausto Spoto, Lunjin Lu, and Fred Mesnard. Using CLP simplifications to improve Java Bytecode termination analysis. *ENTCS*, 253(5):129–144, 2009.
56. Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyser for Java Bytecode based on path-length. *ACM TOPLAS*, 32(3):8:1–8:70, 2010.
57. Thomas Ströder, Peter Schneider-Kamp, and Jürgen Giesl. Dependency triples for improving termination analysis of logic programs with cut. In *LOPSTR '10*, pages 184–199, 2011.
58. Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp. Proving termination and memory safety for programs with pointer arithmetic. In *IJCAR '14*, pages 208–223, 2014.
59. Thomas Ströder, Cornelius Aschermann, Florian Frohn, Jera Hensel, and Jürgen Giesl. AProVE: Termination and memory safety of C programs (competition contribution). In *TACAS '15*, pages 417–419, 2015.
60. *SV-COMP*. `http://sv-comp.sosy-lab.org/`.
61. Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.
62. *Termination Comp*. `http://termination-portal.org/wiki/Termination_Competition`.
63. René Thiemann and Christian Sternagel. Certification of termination proofs using CeTA. In *TPHOLs '09*, pages 452–468, 2009.
64. Harald Zankl, Nao Hirokawa, and Aart Middeldorp. KBO orientability. *JAR*, 43(2):173–201, 2009.

A Proofs

To define the removal of arguments formally, we use a suitable *argument filter*.

Definition 13 (Argument Filter [37]) An *argument filter* π for a signature Σ maps every n-ary function symbol to a (possibly empty) list $[i_1, \ldots, i_k]$ with $1 \leq i_1 < \ldots < i_k \leq n$. The signature Σ_π consists of all function symbols $f \in \Sigma$, but for $\pi(f) = [i_1, \ldots, i_k]$, the arity of f in Σ_π is k. Every argument filter π induces a mapping from $\mathcal{T}(\Sigma, \mathcal{V})$ to $\mathcal{T}(\Sigma_\pi, \mathcal{V})$ as follows.

$$\pi(t) = \begin{cases} t & \text{if } t \text{ is a variable} \\ f(\pi(t_{i_1}), \ldots, \pi(t_{i_k})) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } \pi(f) = [i_1, \ldots, i_k]. \end{cases}$$

For every int-TRS $\mathcal{R}$, we define $\pi(\mathcal{R}) = \{\pi(\ell) \to \pi(r) \,[\![\varphi]\!] \mid \ell \to r \,[\![\varphi]\!] \in \mathcal{R}\}$.

Let N be the set of needed argument positions of Def. 2. Then we define the *needless argument filter* π_N as $\pi_N(f) = [\,]$ for $f \in \mathbb{Z}$ and $\pi_N(f) = [i_1, \ldots, i_k]$ for any $f \in \Sigma_c \cup \Sigma_d$, where $\{i_1, \ldots, i_k\} = \{i \mid (f, i) \in N\}$ and $i_j < i_{j+1}$ for all j.

Theorem 3 (Soundness of Slicing) *Let $\mathcal{R}$ be an int-TRS and let $\mathcal{R}'$ result from filtering away all argument positions that are not needed according to Def. 2, i.e., $\mathcal{R}' = \pi_N(\mathcal{R})$. Then $\mathcal{R}$ terminates iff $\mathcal{R}'$ terminates.*

Proof In the following, we often write $s \hookrightarrow^{\sigma}_{\rho} t$ to denote the rule ρ and the substitution σ used in the rewrite step.

Soundness: For any argument filter π, non-termination of $\mathcal{R}$ implies non-termination of $\pi(\mathcal{R})$, since every (root) rewrite step $s \hookrightarrow_{\ell \to r \,[\![\varphi]\!]} t$ implies that we have $\pi(s) \hookrightarrow_{\pi(\ell) \to \pi(r) \,[\![\varphi]\!]} \pi(t)$.

Completeness: For readability, in the following we write "π" instead of "π_N". Let $\overline{s}_1 \hookrightarrow^{\overline{\sigma}_1}_{\pi(\ell_1) \to \pi(r_1) \,[\![\varphi_1]\!]} \overline{s}_2 \hookrightarrow^{\overline{\sigma}_2}_{\pi(\ell_2) \to \pi(r_2) \,[\![\varphi_2]\!]} \cdots$ be an infinite reduction w.r.t. $\pi(\mathcal{R})$. Our goal is to construct an infinite reduction w.r.t. $\mathcal{R}$.

W.l.o.g., we can assume that the left-hand sides ℓ_i of the rules in $\mathcal{R}$ only contain function symbols that also occur on right-hand sides of $\mathcal{R}$. The reason is that all other rules can be used only a finite number of times at the beginning of a reduction and thus, any infinite reduction has a suffix that satisfies our assumption.

Note that if there are subterms $f(t_1, \ldots, t_n)$ and $f(t'_1, \ldots, t'_n)$ on left-hand sides where $(f, i) \notin N$, then there exists a term u_i such that both t_i and t'_i match u_i. The reason is that some right-hand side must contain a subterm $f(u_1, \ldots, u_i, \ldots, u_n)$ and by the "*matching*" condition in Def. 2, u_i has the desired property. In other words, one can always find a term u which is matched by all ith arguments of f on left-hand sides. We say that such a term u is (f, i)-*compatible*. A term s has the *lhs matching property* iff the following holds: if $(f, i) \notin N$, then any f-subterm of s has some (f, i)-compatible term on its ith argument. Here, an "f-subterm" is a term with f on its root position.

We now construct a term s_1 with $\pi(s_1) = \overline{s}_1$ that starts an infinite reduction w.r.t. $\mathcal{R}$. To this end, we define a function $\mathit{unfilter} : \mathcal{T}(\Sigma_\pi, \mathcal{V}) \to \mathcal{T}(\Sigma, \mathcal{V})$ with $\mathit{unfilter}(x) = x$ for $x \in \mathcal{V}$ and $\mathit{unfilter}(f(t_{i_1}, \ldots, t_{i_k})) = f(t'_1, \ldots, t'_n)$ if the arity of f in Σ is n. Here, we have $t'_i = \mathit{unfilter}(t_i)$ if $i \in \pi(f)$ and t'_i is some (f, i)-compatible term if $i \notin \pi(f)$. Then we choose $s_1 = \mathit{unfilter}(\overline{s}_1)$. By construction, we then have $\pi(s_1) = \overline{s}_1$ and s_1 has the lhs matching property.

Any matcher $\overline{\sigma}_1$ for the filtered reduction can now be extended to a matcher for the non-filtered case. So from $\pi(\ell_1)\overline{\sigma}_1 = \overline{s}_1$, our construction of s_1, and the "*non-linearity*" condition in Def. 2, we can conclude that there is a substitution σ_1 with $\ell_1\sigma_1 = s_1$ such that $\pi(\ell_1\sigma_1) = \overline{s}_1 = \pi(\ell_1)\overline{\sigma}_1$. By the condition "*constraint*" in Def. 2, we can define σ_1 to coincide with $\overline{\sigma}_1$ on all variables from $\widehat{\varphi}_1$. By the condition "*propagation*", σ_1 can be extended to φ_1's remaining variables such that $\varphi_1\sigma_1 = \varphi_1\overline{\sigma}_1$ is valid, and hence, $s_1 \hookrightarrow^{\sigma_1}_{\ell_1\to r_1[\![\varphi_1]\!]} s_2$, where $s_2 = r_1\sigma_1$. Due to the condition "*propagation*" in Def. 2, the root symbols of needed subterms in $s_2 = r_1\sigma_1$ coincide with the corresponding ones in $\overline{s}_2 = \pi(r_1)\overline{\sigma}_1$. Thus, we obtain $\pi(s_2) = \overline{s}_2$. By construction, s_2 also has the lhs matching property. Hence, by repeating the above construction, one finally obtains an infinite reduction $s_1 \hookrightarrow^{\sigma_1}_{\ell_1\to r_1[\![\varphi_1]\!]} s_2 \hookrightarrow^{\sigma_2}_{\ell_2\to r_2[\![\varphi_2]\!]} \cdots$ □

The idea of the *chaining* simplification is to combine rules by narrowing. Narrowing of a term (or rule) is similar to performing a standard rewrite step. However, whereas in rewriting we apply a rule $\ell \to r$ to a term t by finding a *matcher* σ such that $\ell\sigma = t$, in narrowing, we search for a *unifier* μ such that $\ell\mu = t\mu$ holds.

Definition 14 (Narrowing) A term t' is a *narrowing* of the term t with the int-rule $\ell \to r\,[\![\varphi]\!]$ using the unifier μ (written $t \rightsquigarrow^{\mu}_{\ell\to r[\![\varphi]\!]} t'$) if μ is the most general unifier of ℓ and t and $t' = r\mu$. Here we always assume that ℓ and t are variable disjoint (otherwise the variables in the rule are renamed).

Note that our form of narrowing differs from the narrowing of *dependency pairs* in [37], where narrowing steps take place only below the root position, whereas here we narrow only at the root position.

As mentioned, $\mathcal{R}_{\to f} = \{\ell \to r\,[\![\varphi]\!] \in \mathcal{R} \mid \mathsf{root}(r) = f\}$ consists of all rules where f is the root symbol of the right-hand side and $\mathcal{R}_{f\to} = \{\ell \to r\,[\![\varphi]\!] \in \mathcal{R} \mid \mathsf{root}(\ell) = f\}$ are all rules where f is the root of the left-hand side.

Definition 15 (f-chained System $\mathcal{R}^{-f}$) Let $\mathcal{R}$ be an int-TRS and let $f \in \Sigma_d$ such that $\mathcal{R}_{\to f} \cap \mathcal{R}_{f\to} = \varnothing$. Then we define the *$f$-chained system* as $\mathcal{R}^{-f} = (\mathcal{R} \setminus (\mathcal{R}_{\to f} \cup \mathcal{R}_{f\to})) \cup \mathcal{R}'$, where $\mathcal{R}' = \{\ell\mu \to r'\ [\![(\varphi\wedge\overline{\varphi})\mu]\!] \mid \ell \to r\,[\![\varphi]\!] \in \mathcal{R}_{\to f},\ \overline{\ell} \to \overline{r}\,[\![\overline{\varphi}]\!] \in \mathcal{R}_{f\to},\ r \rightsquigarrow^{\mu}_{\overline{\ell}\to\overline{r}[\![\overline{\varphi}]\!]} r'\}$.

Theorem 5 (Soundness of Chaining) *Let $\mathcal{R}$ be an int-TRS and f be as in Def. 15. Then $\mathcal{R}$ terminates iff $\mathcal{R}^{-f}$ terminates.*

Proof

Soundness: Let $t_1 \hookrightarrow^{\sigma_1}_{\ell_1\to r_1[\![\varphi_1]\!]} t_2 \hookrightarrow^{\sigma_2}_{\ell_2\to r_2[\![\varphi_2]\!]} \cdots$ be an infinite $\mathcal{R}$-reduction where we assume that $\ell_i \to r_i\,[\![\varphi_i]\!]$ and $\ell_j \to r_j\,[\![\varphi_j]\!]$ are variable disjoint whenever $i \neq j$.

W.l.o.g. let $\ell_1 \to r_1\,[\![\varphi_1]\!] \notin \mathcal{R}_{f\to}$. For every i where $\ell_i \to r_i\,[\![\varphi_i]\!] \in \mathcal{R}_{\to f}$ we have $\ell_{i+1} \to r_{i+1}\,[\![\varphi_{i+1}]\!] \in \mathcal{R}_{f\to}$, since $\mathsf{root}(r_i) = f$.

Moreover, $t_i = \ell_i\sigma_i$, $t_{i+1} = r_i\sigma_i = \ell_{i+1}\sigma_{i+1}$, and both $\varphi_i\sigma_i$ and $\varphi_{i+1}\sigma_{i+1}$ are valid. So there is a $\mu = \mathsf{mgu}(r_i, \ell_{i+1})$ such that $r_i \rightsquigarrow^{\mu}_{\ell_{i+1}\to r_{i+1}[\![\varphi_{i+1}]\!]} r_{i+1}\mu$ and $\ell_i\mu \to r_{i+1}\mu\,[\![(\varphi_i \wedge \varphi_{i+1})\mu]\!] \in \mathcal{R}' \subseteq \mathcal{R}^{-f}$, with $\mathcal{R}'$ as in Def. 15.

As $\ell_i \to r_i\,[\![\varphi_i]\!]$ and $\ell_{i+1} \to r_{i+1}\,[\![\varphi_{i+1}]\!]$ are variable disjoint, there exists a substitution γ such that $\mu\gamma$ is like σ_i for all variables in $\ell_i \to r_i\,[\![\varphi_i]\!]$ and like σ_{i+1} for all variables in $\ell_{i+1} \to r_{i+1}\,[\![\varphi_{i+1}]\!]$. Hence, $\ell_i\mu\gamma = \ell_i\sigma_i = t_i$, $r_{i+1}\mu\gamma = r_{i+1}\sigma_{i+1} = t_{i+2}$, and both $\varphi_i\mu\gamma = \varphi_i\sigma_i$ and $\varphi_{i+1}\mu\gamma = \varphi_{i+1}\sigma_{i+1}$ are valid. Thus, $t_i \hookrightarrow^{\gamma}_{\ell_i\mu\to r_{i+1}\mu[\![(\varphi_i\wedge\varphi_{i+1})\mu]\!]} t_{i+2}$.

By iterating this replacement of rule applications from $\mathcal{R}_{\to f}$, we also eliminate all applications of rules from $\mathcal{R}_{f\to}$, since they are always preceded by a rule from $\mathcal{R}_{\to f}$. In this way, we obtain an infinite reduction w.r.t. $\mathcal{R}^{-f}$.

Completeness: Every infinite reduction w.r.t. $\mathcal{R}^{-f}$ can be transformed into an infinite reduction w.r.t. $\mathcal{R}$. The reason is that whenever t rewrites to t' with a rule from $\mathcal{R}' \subseteq \mathcal{R}^{-f}$, then t also rewrites to t' in 2 steps with $\mathcal{R}$. To see this, let $t \hookrightarrow^{\sigma}_{\ell\mu\to r' \,[\![(\varphi\wedge\overline{\varphi})\mu]\!]} t'$. Hence, $\varphi\mu\sigma$ and $\overline{\varphi}\mu\sigma$ are valid. Here, both $\ell \to r \,[\![\varphi]\!]$ and $\overline{\ell} \to \overline{r} \,[\![\overline{\varphi}]\!]$ are from $\mathcal{R}$ and $r \rightsquigarrow^{\mu}_{\overline{\ell}\to\overline{r}\,[\![\overline{\varphi}]\!]} r'$ (i.e., $r\mu = \overline{\ell}\mu$ and $r' = \overline{r}\mu$). Thus, $t = \ell\mu\sigma \hookrightarrow^{\mu\sigma}_{\ell\to r\,[\![\varphi]\!]} r\mu\sigma = \overline{\ell}\mu\sigma \hookrightarrow^{\mu\sigma}_{\overline{\ell}\to\overline{r}\,[\![\overline{\varphi}]\!]} \overline{r}\mu\sigma = r'\sigma = t'$. □

For any term t, we can define $\mathcal{IA}(t)$ and $\mathcal{TA}(t)$ formally using argument filters. We define the *integer argument projection* that removes all term arguments as $\pi_{\mathcal{IA}}(f) = [\,]$ for $f \in \mathbb{Z}$ and $\pi_{\mathcal{IA}}(f) = [i_1, \ldots, i_k]$ for any $f \in \Sigma_c \cup \Sigma_d$, where $\{i_1, \ldots, i_k\} = \{i \mid (f, i) \in \mathcal{IA}\}$ and $i_j < i_{j+1}$ for all j. The *term argument projection* $\pi_{\mathcal{TA}}$ is defined analogously, retaining only term arguments. Then we have $\mathcal{IA}(t) = \pi_{\mathcal{IA}}(t)$ and $\mathcal{TA}(t) = \pi_{\mathcal{TA}}(t)$.

Before proving Thm. 8, we recapitulate the definition of reduction pairs.

Definition 16 (Reduction Pairs [37]) We call $(\succsim, \succ)$ a *reduction pair* iff $\succsim$ is reflexive, transitive, and closed under substitutions (i.e., $s \succsim t$ implies $s\sigma \succsim t\sigma$ for all σ), $\succ$ is closed under substitutions and well founded, and $\succ$ and $\succsim$ are compatible (i.e., $\succ \circ \succsim \subseteq \succ$ or $\succsim \circ \succ \subseteq \succ$). For a reduction pair and an int-TRS $\mathcal{R}$, we define the sets $\mathcal{R}_{\succ} = \{\ell \to r \,[\![\varphi]\!] \mid \varphi \implies \ell \succ r\}$ and $\mathcal{R}_{\succsim} = \{\ell \to r \,[\![\varphi]\!] \mid \varphi \implies \ell \succsim r\}$.

Note that in contrast to the standard definition of reduction pairs [37], here $\succsim$ does not have to be closed under contexts since we only regard rewrite steps at the root position.

Theorem 8 (Reduction Pairs From Projections) *Let $\mathcal{R}$ be an int-TRS. If $(\succsim_{\mathcal{IA}}, \succ_{\mathcal{IA}})$ is a reduction pair for $\mathcal{IA}(\mathcal{R})$, then $(\succsim, \succ)$ is a reduction pair for $\mathcal{R}$ where $t_1 \succ t_2$ holds iff $\mathcal{IA}(t_1) \succ_{\mathcal{IA}} \mathcal{IA}(t_2)$ and $t_1 \succsim t_2$ holds iff $\mathcal{IA}(t_1) \succsim_{\mathcal{IA}} \mathcal{IA}(t_2)$. The same holds for the restriction $\mathcal{TA}$.*

Proof We show that transitivity of $\succsim_{\mathcal{IA}}$ implies transitivity of $\succsim$. Note that $t_1 \succsim t_2$ and $t_2 \succsim t_3$ implies $\mathcal{IA}(t_1) \succsim_{\mathcal{IA}} \mathcal{IA}(t_2)$ and $\mathcal{IA}(t_2) \succsim_{\mathcal{IA}} \mathcal{IA}(t_3)$ by definition. By transitivity of $\succsim_{\mathcal{IA}}$, we have $\mathcal{IA}(t_1) \succsim_{\mathcal{IA}} \mathcal{IA}(t_3)$ which implies $t_1 \succsim t_3$. For all other properties, the proof is completely analogous. □

To prove Thm. 11, we need a few intermediate lemmas. First, we show that th is "monotonic" w.r.t. data substitutions, i.e., for any term t, its term height $\mathsf{th}(t)$ is not greater than the term height of any instantiation $t\sigma$.

Lemma 17 (Term Height is Monotonic w.r.t. Substitutions) *Let $t \in \mathcal{T}(\mathbb{Z} \cup \Sigma_c, \mathcal{V})$ and $\sigma : \mathcal{V} \to \mathcal{T}(\mathbb{Z} \cup \Sigma_c, \mathcal{V})$. Then $\mathsf{th}(t) \leq \mathsf{th}(t\sigma)$.*

Proof The proof is by induction on the term structure. In the base case, we either have $t \in \mathbb{Z}$ and then $t = t\sigma$ and thus $\mathsf{th}(t) = \mathsf{th}(t\sigma)$, or $t \in \mathcal{V}$ and then $\mathsf{th}(t) = 0 \leq \mathsf{th}(t\sigma)$. In the induction step, let $t = f(t_1, \ldots, t_n)$. Then we have

$$\begin{aligned}
\mathsf{th}(s) &= 1 + \max\{\mathsf{th}(t_i) \mid 1 \leq i \leq n\} \\
&\leq 1 + \max\{\mathsf{th}(t_i\sigma) \mid 1 \leq i \leq n\} \qquad \text{by the induction hypothesis} \\
&= \mathsf{th}(t\sigma)
\end{aligned}$$

□

We also need the following lemma about the relation between the term height of a variable occurring at a term position of t and the term height of t.

Lemma 18 (Lower Bounds for Term Height) *Let $t \in \mathcal{T}(\mathbb{Z} \cup \Sigma_c, \mathcal{V})$, $x \in \mathcal{V}_{\mathcal{TA}}(t)$, and $\sigma : \mathcal{V} \to \mathcal{T}(\mathbb{Z} \cup \Sigma_c, \mathcal{V})$. Then $\mathsf{th}(x\sigma) + \mathsf{nl}(t, x) \leq \mathsf{th}(t\sigma)$.*

Proof We prove the lemma by induction. In the base case, $x \in \mathcal{V}_{\mathcal{TA}}(t)$ implies $t = x$. Then $\mathsf{th}(x\sigma) + \mathsf{nl}(x, x) = \mathsf{th}(x\sigma)$. In the induction step, let $t = f(t_1, \ldots, t_n)$. Then we have

$$\begin{aligned}
& \mathsf{th}(x\sigma) + \mathsf{nl}(t, x) \\
= \ & \mathsf{th}(x\sigma) + 1 + \max\{\mathsf{nl}(t_i, x) \mid 1 \leq i \leq n, x \in \mathcal{V}_{\mathcal{TA}}(t_i)\} && \text{as } x \in \mathcal{V}_{\mathcal{TA}}(t) \\
= \ & 1 + \max\{\mathsf{th}(x\sigma) + \mathsf{nl}(t_i, x) \mid 1 \leq i \leq n, x \in \mathcal{V}_{\mathcal{TA}}(t_i)\} \\
\leq \ & 1 + \max\{\mathsf{th}(t_i\sigma) \mid 1 \leq i \leq n, x \in \mathcal{V}_{\mathcal{TA}}(t_i)\} && \text{by the ind. hypothesis} \\
\leq \ & 1 + \max\{\mathsf{th}(t_i\sigma) \mid 1 \leq i \leq n\} \\
= \ & \mathsf{th}(t\sigma)
\end{aligned}$$

□

Finally, we also prove the following lemma about the relation between the term height of an instantiated term $t\sigma$ and the term heights $\mathsf{th}(x\sigma)$ of the variables x occurring in t. In this way, we obtain an upper bound for the term height $\mathsf{th}(t\sigma)$.

Lemma 19 (Upper Bounds for Term Height) *Let $t \in \mathcal{T}(\mathbb{Z} \cup \Sigma_c, \mathcal{V})$ and $\sigma : \mathcal{V} \to \mathcal{T}(\mathbb{Z} \cup \Sigma_c, \mathcal{V})$. Then $\mathsf{th}(t\sigma) \leq \max\{\mathsf{th}(t), \max\{\mathsf{th}(x\sigma) + \mathsf{nl}(t, x) \mid x \in \mathcal{V}_{\mathcal{TA}}(t)\}\}$.*

Proof Again, we prove the lemma by induction. In the base case, we consider three cases. For $t = x \in \mathcal{V}$, we have $\mathsf{th}(x\sigma) = \max\{\mathsf{th}(x), \mathsf{th}(x\sigma) + \mathsf{nl}(x, x)\}$, since $\mathsf{th}(x) = \mathsf{nl}(x, x) = 0$. For $t \in \Sigma_c$, we have $\mathsf{th}(t\sigma) = \mathsf{th}(t) = 1 = \max\{\mathsf{th}(t)\}$. Finally, for $t \in \mathbb{Z}$, we have $\mathsf{th}(t\sigma) = 0$, which is a lower bound for any term height. In the induction step, let $t = f(t_1, \ldots, t_n)$.

$$\begin{aligned}
\mathsf{th}(t\sigma) &= 1 + \max\{\mathsf{th}(t_i\sigma) \mid 1 \leq i \leq n\} \\
&\leq 1 + \max\{\max\{\mathsf{th}(t_i), \\
&\qquad \max\{\mathsf{th}(x\sigma) + \mathsf{nl}(t_i, x) \mid x \in \mathcal{V}_{\mathcal{TA}}(t_i)\}\} \mid 1 \leq i \leq n\} \text{ by the ind. hyp.} \\
&= \max\{1 + \max\{\mathsf{th}(t_i) \mid 1 \leq i \leq n\}, \\
&\qquad 1 + \max\{\mathsf{th}(x\sigma) + \mathsf{nl}(t_i, x) \mid 1 \leq i \leq n, x \in \mathcal{V}_{\mathcal{TA}}(t_i)\}\} \\
&= \max\{\mathsf{th}(t), \max\{\mathsf{th}(x\sigma) + 1 + \mathsf{nl}(t_i, x) \mid 1 \leq i \leq n, x \in \mathcal{V}_{\mathcal{TA}}(t_i)\}\} \\
&\leq \max\{\mathsf{th}(t), \max\{\mathsf{th}(x\sigma) + \mathsf{nl}(t, x) \mid x \in \mathcal{V}_{\mathcal{TA}}(t)\}\} \text{ as } \bigcup_{1 \leq i \leq n} \mathcal{V}_{\mathcal{TA}}(t_i) = \mathcal{V}_{\mathcal{TA}}(t)
\end{aligned}$$

□

For the proof of Thm. 11, we define the replacement of non-integer subterms by their height formally.

Definition 20 (Term Height Projection π_{th}) Let $t = f(t_1, \ldots, t_n)$ for some $f \in \Sigma_d$, $t_1, \ldots, t_n \in \mathcal{T}(\mathbb{Z} \cup \Sigma_c, \mathcal{V})$. Then $\pi_{\mathsf{th}}(t) = f(\hat{t}_1, \ldots, \hat{t}_n)$ with

$$\hat{t}_i = \begin{cases} t_i & \text{if } (f, i) \notin \mathcal{TA} \\ \mathsf{th}(t_i) & \text{otherwise} \end{cases}$$

We can now prove the soundness of termination proving by the term height projection.

Theorem 11 (Soundness of Term Height Projection) *Let $\mathcal{R}$ be an int-TRS. If $\Pi_{\mathsf{th}}(\mathcal{R})$ terminates, then $\mathcal{R}$ also terminates.*

Proof We show that if $\mathcal{R}$ has an infinite reduction $t_1 \hookrightarrow_{\mathcal{R}} t_2 \hookrightarrow_{\mathcal{R}} \cdots$, then there is also the infinite reduction $\pi_{\mathsf{th}}(t_1) \hookrightarrow_{\Pi_{\mathsf{th}}(\mathcal{R})} \pi_{\mathsf{th}}(t_2) \hookrightarrow_{\Pi_{\mathsf{th}}(\mathcal{R})} \cdots$ To this end, we prove that

$$\begin{array}{lll} s = f(s_1,\ldots,s_n) \hookrightarrow^{\sigma}_{\ell\to r\,[\![\varphi]\!]} & g(t_1,\ldots,t_m) = t & \text{implies} \\ \pi_{\mathsf{th}}(s) = f(\hat{s}_1,\ldots,\hat{s}_n) \hookrightarrow^{\sigma'}_{\Pi_{\mathsf{th}}(\ell\to r\,[\![\varphi]\!])} & g(\hat{t}_1,\ldots,\hat{t}_m) = \pi_{\mathsf{th}}(t) & \end{array}$$

for a suitable substitution σ'. Let $\ell = f(\ell_1,\ldots,\ell_n)$ and $r = g(r_1,\ldots,r_m)$. Then according to Def. 10, we have $\Pi_{\mathsf{th}}(\ell \to r\,[\![\varphi]\!]) = f(\ell'_1,\ldots,\ell'_n) \to g(r'_1,\ldots,r'_m)\,[\![\varphi \wedge \psi]\!]$, where ℓ'_i is a fresh variable h_i if $(f,i) \in \mathcal{TA}$, and otherwise $\ell'_i = \ell_i$. Similarly, r'_i is a fresh variable h'_i if $(g,i) \in \mathcal{TA}$ and otherwise $r'_i = r_i$. We now define σ' as follows.

$$\sigma'(x) = \begin{cases} \mathsf{th}(\sigma(x)) & \text{if } x \in \mathcal{V}_{\mathcal{TA}}(\ell) \cup \mathcal{V}_{\mathcal{TA}}(r) \\ \mathsf{th}(s_i) & \text{if } x = h_i \\ \mathsf{th}(t_i) & \text{if } x = h'_i \\ \sigma(x) & \text{otherwise} \end{cases}$$

Now we show that $f(\hat{s}_1,\ldots,\hat{s}_n) \hookrightarrow^{\sigma'}_{f(\ell'_1,\ldots,\ell'_n)\to g(r'_1,\ldots,r'_m)\,[\![\varphi\wedge\psi]\!]} g(\hat{t}_1,\ldots,\hat{t}_m)$.

First note that the left-hand side of the rule matches $f(\hat{s}_1,\ldots,\hat{s}_n)$ using the matcher σ'. In other words, we have $\ell'_i\sigma' = \hat{s}_i$ for all $1 \le i \le n$. The reason is that if $(f,i) \in \mathcal{TA}$, then $\ell'_i\sigma' = h_i\sigma' = \mathsf{th}(s_i) = \hat{s}_i$. If $(f,i) \notin \mathcal{TA}$, then $\ell'_i\sigma' = \ell_i\sigma' = \ell_i\sigma = s_i = \hat{s}_i$. For a similar reason, we have $g(r'_1,\ldots,r'_m)\sigma' = g(\hat{t}_1,\ldots,\hat{t}_m)$.

It remains to prove that the constraint $(\varphi \wedge \psi)\sigma'$ is valid. As σ' behaves like σ on integer variables, we have $\varphi\sigma' = \varphi\sigma$, and thus validity of $\varphi\sigma'$ follows from validity of $\varphi\sigma$. So we need to consider only the additional conjuncts in ψ.

Conjuncts of the form $h_i \ge \mathsf{th}(\ell_i)$ for $(f,i) \in \mathcal{TA}$ are valid when instantiated with σ' because $(h_i \ge \mathsf{th}(\ell_i))\sigma'$ iff $h_i\sigma' \ge \mathsf{th}(\ell_i)$ iff $\mathsf{th}(s_i) \ge \mathsf{th}(\ell_i)$ iff $\mathsf{th}(\ell_i\sigma) \ge \mathsf{th}(\ell_i)$. The validity of $\mathsf{th}(\ell_i\sigma) \ge \mathsf{th}(\ell_i)$ follows from Lemma 17.

For conjuncts of the form $x + \mathsf{nl}(\ell_i, x) \le h_i$ with $(f,i) \in \mathcal{TA}$ and $x \in \mathcal{V}_{\mathcal{TA}}(\ell_i)$, we have $(x + \mathsf{nl}(\ell_i,x) \le h_i)\sigma'$ iff $x\sigma' + \mathsf{nl}(\ell_i,x) \le h_i\sigma'$ iff $\mathsf{th}(x\sigma) + \mathsf{nl}(\ell_i,x) \le \mathsf{th}(s_i)$ iff $\mathsf{th}(x\sigma) + \mathsf{nl}(\ell_i,x) \le \mathsf{th}(\ell_i\sigma)$. This is a consequence of Lemma 18.

Now we consider conjuncts of the form $x \ge 0$ for $(f,i) \in \mathcal{TA}$ and $x \in \mathcal{V}_{\mathcal{TA}}(\ell_i)$. Here we have $(x \ge 0)\sigma'$ iff $x\sigma' \ge 0$ iff $\mathsf{th}(x\sigma) \ge 0$. This is clearly valid as th always yields a non-negative number. The validity of the corresponding conjuncts for the height variables h'_i on the right-hand side can be shown in an analogous way.

Finally, we consider the conjunct $h'_i \le \max\{\mathsf{th}(r_i), \max\{x + \mathsf{nl}(r_i,x) \mid x \in \mathcal{V}_{\mathcal{TA}}(r_i)\}\}$ where $(g,i) \in \mathcal{TA}$ and $x \in \mathcal{V}_{\mathcal{TA}}(r_i)$. We have $(h'_i \le \max\{\mathsf{th}(r_i), \max\{x + \mathsf{nl}(r_i,x) \mid x \in \mathcal{V}_{\mathcal{TA}}(r_i)\}\})\sigma'$ iff $h'_i\sigma' \le \max\{\mathsf{th}(r_i), \max\{x\sigma' + \mathsf{nl}(r_i,x) \mid x \in \mathcal{V}_{\mathcal{TA}}(r_i)\}\}$ iff $\mathsf{th}(t_i) \le \max\{\mathsf{th}(r_i), \max\{\mathsf{th}(x\sigma) + \mathsf{nl}(r_i,x) \mid x \in \mathcal{V}_{\mathcal{TA}}(r_i)\}\}$ iff $\mathsf{th}(r_i\sigma) \le \max\{\mathsf{th}(r_i), \max\{\mathsf{th}(x\sigma) + \mathsf{nl}(r_i,x) \mid x \in \mathcal{V}_{\mathcal{TA}}(r_i)\}\}$. This is a consequence of Lemma 19. □