Automated Termination Proofs for Haskell by Term Rewriting

JÜRGEN GIESL, RWTH Aachen University
MATTHIAS RAFFELSIEPER, TU Eindhoven
PETER SCHNEIDER-KAMP, University of Southern Denmark
STEPHAN SWIDERSKI, RWTH Aachen University
RENÉ THIEMANN, University of Innsbruck

There are many powerful techniques for automated termination analysis of term rewriting. However, up to now they have hardly been used for real programming languages. We present a new approach which permits the application of existing techniques from term rewriting to prove termination of most functions defined in Haskell programs. In particular, we show how termination techniques for ordinary rewriting can be used to handle those features of Haskell which are missing in term rewriting (e.g., lazy evaluation, polymorphic types, and higher-order functions). We implemented our results in the termination prover AProVE and successfully evaluated them on existing Haskell libraries.

Categories and Subject Descriptors: F.3.1 [**Logic and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Automatic analysis of algorithms*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Functional programming, Haskell, termination analysis, term rewriting, dependency pairs

ACM Reference Format:
Giesl, J., Raffelsieper, M., Schneider-Kamp, P., Swiderski, S., and Thiemann, R. 2011. Automated termination proofs for Haskell by term rewriting. ACM Trans. Program. Lang. Syst. 33, 2, Article 7 (January 2011), 39 pages.
DOI = 10.1145/1890028.1890030 http://doi.acm.org/10.1145/1890028.1890030

1. INTRODUCTION

In the area of term rewriting, techniques for automated termination analysis have been studied for decades. While early work focused on the development of suitable well-founded orders (see, e.g., Dershowitz [1987] for an overview), in the last 10 years much more powerful methods were introduced which can handle large and realistic term

This work was supported by the Deutsche Forschungsgsmeinschaft DFG under grant Gi 274/5-2 and the DFG Research Training Group 1298 (*AlgoSyn*).
Authors' addresses: J. Giesl, LuFG Informatik 2, RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany; email: giesl@informatik.rwth-aachen.de; M. Raffelsieper, Department of Mathematics and Computer Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands; email: m.raffelsieper@tue.nl; P. Schneider-Kamp, Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, DK-5230 Odense, M, Denmark; email: petersk@imada.sdu.dk; S. Swiderski, LuFG Informatik 2, RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany; email: swiderski@informatik.rwth-aachen.de; R. Thiemann, Institute of Computer Science, University of Innsbruck, Technikerstr. 21a, 6020 Innsbruck, Austria; email: rene.thiemann@uibk.ac.at.

DOI 10.1145/1890028.1890030 http://doi.acm.org/10.1145/1890028.1890030

rewrite systems (TRSs); see, for example, Endrullis et al. [2008], Geser et al. [2004], Hirokawa and Middeldorp [2005], Giesl et al. [2006c], and Zantema [2003]. Moreover, numerous powerful automatic tools for termination analysis of TRSs have been developed whose power is demonstrated at the annual International Termination Competition.[1]

However, in order to make methods for termination analysis of term rewriting applicable in practice, one has to adapt them to real existing programming languages. In this article, we show for the first time that termination techniques from term rewriting are indeed very useful for termination analysis of functional programming languages. Specifically, we consider the language Haskell [Peyton Jones 2003], which is one of the most popular functional programming languages.

Since term rewriting itself is a Turing-complete programming language, in principle it is of course possible to translate any program from any programming language into an equivalent TRS and then prove termination of the resulting TRS. However, in general, it is not clear how to obtain an automatic translation that creates TRSs which are *suitable for existing automated termination techniques*. In other words, a naive translation of programs into TRSs is likely to produce very complicated TRSs whose termination can hardly be shown by existing automated techniques and tools.

Although functional programming languages are in some sense "close" to term rewriting, the application and adaption of TRS techniques for termination analysis of Haskell is still challenging for several reasons:

—Haskell has a *lazy evaluation* strategy. However, most TRS techniques ignore such evaluation strategies and try to prove that *all* (or all innermost) reductions terminate.[2]
—Defining equations in Haskell are handled from top to bottom. In contrast, for TRSs, *any* rule may be used for rewriting.
—Haskell has polymorphic types, whereas TRSs are untyped.
—In Haskell programs with infinite data objects, only certain functions are terminating. But most TRS methods try to prove termination of *all* terms.
—Haskell is a *higher-order* language, whereas most automatic termination techniques for TRSs only handle first-order rewriting.

There are many papers on verification of functional programs (see, e.g., Kobayashi [2009], Ong [2006], Rondon et al. [2008] for some of the most recent approaches). However, up to now there exist only few techniques for automated termination analysis of functional languages. Methods for first-order languages with strict evaluation strategy were for example developed in Giesl [1995], Lee et al. [2001], Manolios and Vroon [2006], and Walther [1994], where the *size-change* method of Lee et al. [2001] was also extended to the higher-order setting [Sereni and Jones 2005; Sereni 2007]. The static call graph constructed by the methods of Sereni and Jones [2005] and Sereni [2007] is related to the graphs constructed in our approach in order to analyze termination. However, the size-change method fixes one particular order to compare values for each data type. (This also holds for higher-order types whose values are closures. These closures are typically compared by the subtree order.) Here our approach is more flexible, because the orders to compare values are not fixed. Instead, we translate all data objects (including objects of higher-order type) into first-order terms and afterwards,

[1]For more information on the competition, we refer to `http://termination-portal.org/wiki/Termination_Competition`.

[2]Very recently, there has been work on termination analysis of rewriting under an *outermost* evaluation strategy [Endrullis and Hendriks 2009; Gnaedig and Kirchner 2008; Raffelsieper and Zantema 2009; Thiemann 2009], which, however, does not correspond to the lazy evaluation strategy of Haskell (as illustrated later in Section 2.2).

one can use existing techniques from term rewriting to automatically generate suitable well-founded orders comparing these terms. For a thorough comparison of the size-change method with techniques from term rewriting, we refer to Thiemann and Giesl [2005].

For higher-order languages, several papers study how to ensure termination by typing (e.g., Abel [2004], Barthe et al. [2000], Blanqui [2004], and Xi [2002]) and Telford and Turner [2000] define a restricted language where all evaluations terminate. A successful approach for automated termination proofs for a small Haskell-like language was developed in Panitz and Schmidt-Schauß [1997] and extended and implemented in Panitz [1997].[3] This approach is related to the technique of Gnaedig and Kirchner [2008], which handles outermost evaluation of untyped first-order rewriting. However, these are all "stand-alone" methods which do not allow the use of modern termination techniques from term rewriting. Indeed, the general opinion of the Haskell community was that "current advances in automatic termination proofs are still limited, especially for lazy programs" [Xu et al. 2009].

In our approach we build upon the method of Panitz and Schmidt-Schauß [1997], but we adapt it in order to make TRS techniques applicable.[4] As shown by our experimental evaluation, the coupling with modern powerful TRS techniques solves the previous limitations of termination methods for lazy functional programs. Now automated termination proofs for functions from real Haskell libraries indeed become feasible.

We recapitulate Haskell in Section 2 and introduce our notion of "termination." As described in Section 3, to analyze termination, our method first generates a corresponding *termination graph* (similar to the "termination tableaux" in Panitz and Schmidt-Schauß [1997]). But in contrast to Panitz and Schmidt-Schauß [1997], then our method transforms the termination graph into *dependency pair problems* which can be handled by existing techniques from term rewriting (Section 4). Our approach can deal with any termination graph, whereas Panitz and Schmidt-Schauß [1997] can only handle termination graphs of a special form ("without crossings").[5] While the dependency pair problems in Section 4 still contain higher-order functions, in Section 5 we improve the construction in order to obtain *first-order* dependency pair problems. Section 6 extends our approach to handle more types of Haskell, in particular *type classes*. We implemented all our contributions in the termination prover AProVE [Giesl et al. 2006b]. Section 7 presents extensive experiments which show that our approach gives rise to a very powerful fully automated termination tool. More precisely, when testing it on existing standard Haskell libraries, it turned out that AProVE can fully automatically prove termination of the vast majority of the functions in the libraries. This shows for the first time that:

—it is possible to build a powerful automated termination analyzer for a functional language like Haskell and that
—termination techniques from term rewriting can be successfully applied to real programming languages in practice.

[3]In addition to methods which analyze the termination behavior of programs, there are also several results on ensuring the termination of program optimization techniques like partial evaluation, for example, Glenstrup and Jones [2005]. Here, in particular the approach of Sørensen and Glück [1995] uses graphs that are similar to the termination graphs in Panitz and Schmidt-Schauß [1997] and in our approach.

[4]Alternatively as discussed in Giesl and Middeldorp [2004], one could try to simulate Haskell's evaluation strategy by *context-sensitive rewriting* [Lucas 1998]. But in spite of recent progress in that area (e.g., Giesl and Middeldorp [2004], and Alarcón et al. [2006, 2008]), termination of context-sensitive rewriting is still hard to analyze automatically.

[5]We will illustrate the difference in more detail in Example 5.2.

2. HASKELL

A real programming language like Haskell offers many syntactical constructs which ease the formulation of programs, but which are not necessary for the expressiveness of the language. To analyze termination of a Haskell program, it is first checked for syntactical correctness and for being well typed. To simplify the subsequent termination analysis, our termination tool then transforms the given Haskell program into an equivalent Haskell program which only uses a subset of the constructs available in Haskell. We now give the syntax and semantics for this subset of Haskell. In this subset, we only use certain easy patterns and terms (without "λ"), and we only allow function definitions without "case" expressions or conditionals. So we only permit case analysis by pattern-matching left-hand sides of defining equations.[6]

Indeed, any Haskell program can automatically be transformed into a program from this subset; see, Swiderski [2005]. For example, in our implementation, lambda abstractions are removed by a form of lambda lifting. More precisely, we replace every Haskell term "$\backslash\, t_1 \ldots t_n \to t$" with the free variables $x_1, \ldots, x_m$ by "$\mathsf{f}\, x_1 \ldots x_m$". Here, f is a new function symbol with the defining equation $\mathsf{f}\, x_1 \ldots x_m\, t_1 \ldots t_n = t$.

2.1. Syntax of Haskell

In our subset of Haskell, we permit user-defined data types such as

$$\mathsf{data\ Nats} = \mathsf{Z} \mid \mathsf{S\ Nats} \qquad\qquad \mathsf{data\ List}\ a = \mathsf{Nil} \mid \mathsf{Cons}\ a\ (\mathsf{List}\ a)$$

These data declarations introduce two *type constructors* Nats and List of arity 0 and 1, respectively. So Nats is a type and for every type τ, "List τ" is also a type representing lists with elements of type τ. Apart from user-defined data declarations, there are also predefined data declarations like

$$\mathsf{data\ Bool = False \mid True.}$$

Moreover, there is a predefined binary type constructor $\to$ for function types. So if τ_1 and τ_2 are types, then $\tau_1 \to \tau_2$ is also a type (the type of functions from τ_1 to τ_2). Since Haskell's type system is polymorphic, it also has *type variables* like a which stand for any type, and "List a" is the type of lists where the elements can have any type a. So the set of types is the smallest set which contains all type variables and where "$d\, \tau_1 \ldots \tau_n$" is a type whenever d is a type constructor of arity m and $\tau_1, \ldots, \tau_n$ are types with $n \leq m$.[7]

For each type constructor like Nats, a data declaration also introduces its *data constructors* (e.g., Z and S) and the types of their arguments. Thus, Z has arity 0 and is of type Nats and S has arity 1 and is of type Nats $\to$ Nats.

Apart from data declarations, a program has function declarations.

Example 2.1 (take *and* from). In the following example, "from x" generates the infinite list of numbers starting with x and "take $n\, xs$" returns the first n elements of the list xs. The type of from is "Nats $\to$ (List Nats)" and take has the type "Nats $\to$ (List a) $\to$ (List a)" where $\tau_1 \to \tau_2 \to \tau_3$ stands for $\tau_1 \to (\tau_2 \to \tau_3)$. Such type declarations can also be included in the Haskell program.

[6]Of course, it would be possible to restrict ourselves to programs from an even smaller "core" Haskell subset. However, this would not simplify the subsequent termination analysis any further. In contrast, the resulting programs would usually be less readable, which would also make it harder for the user to understand their (automatically generated) termination proofs.

[7]Moreover, Haskell also has several built-in primitive data types (e.g., Int, Char, Float, ...) and its type system also features *type classes* in order to permit overloading of functions. To ease the presentation, we ignore these concepts at the moment and refer to Section 6 for an extension of our approach to built-in data types and type classes.

$\mathsf{from} :: \mathsf{Nats} \to (\mathsf{List}\,\mathsf{Nats})$
$\mathsf{from}\,x = \mathsf{Cons}\,x\,(\mathsf{from}\,(\mathsf{S}\,x))$

$\mathsf{take} :: \mathsf{Nats} \to (\mathsf{List}\,a) \to (\mathsf{List}\,a)$
$\mathsf{take}\,\mathsf{Z}\,xs = \mathsf{Nil}$
$\mathsf{take}\,n\,\mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\,(\mathsf{S}\,n)\,(\mathsf{Cons}\,x\,xs) = \mathsf{Cons}\,x\,(\mathsf{take}\,n\,xs)$

In general, the equations in function declarations have the form "$f\,\ell_1 \ldots \ell_n = r$" for $n \geq 0$. The function symbols f at the "outermost" position of left-hand sides are called *defined*. So the set of function symbols is the disjoint union of the (data) constructors and the defined function symbols. All defining equations for f must have the same number of arguments n (called f's *arity*). The right-hand side r is an arbitrary *term*, whereas $\ell_1, \ldots, \ell_n$ are special terms, so-called *patterns*. Moreover, the left-hand side must be *linear*, that is, no variable may occur more than once in "$f\,\ell_1 \ldots\ \ell_n$".

The set of *terms* is the smallest set containing all variables, function symbols, and *well-typed* applications $(t_1\,t_2)$ for terms t_1 and t_2. As usual, "$t_1\,t_2\,t_3$" stands for "$((t_1\,t_2)\,t_3)$", etc. The set of *patterns* is the smallest set with all variables and all linear terms "$c\,t_1 \ldots t_n$" where c is a constructor of arity n and $t_1, \ldots, t_n$ are patterns.

The *positions* of t are $Pos(t) = \{\varepsilon\}$ if t is a variable or function symbol. Otherwise, $Pos(t_1\,t_2) = \{\varepsilon\} \cup \{1\,\pi \mid \pi \in Pos(t_1)\} \cup \{2\,\pi \mid \pi \in Pos(t_2)\}$. As usual, we define $t|_\varepsilon = t$ and $(t_1\,t_2)|_{i\,\pi} = t_i|_\pi$ for $i \in \{1, 2\}$. The term q is a *subterm* of t, that is, $q \trianglelefteq t$, if a position π of t exists such that $t|_\pi = q$. The *head* of t is $t|_{1^n}$ where n is the maximal number such that $1^n \in Pos(t)$. So the head of $t = \mathsf{take}\,n\,xs$ (i.e., "$(\mathsf{take}\,n)\,xs$") is $t|_{11} = \mathsf{take}$. Let $\mathcal{V}(t)$ denote the set of variables of a term.

2.2. Operational Semantics of Haskell

Given an underlying program, for any term t we define the position $\mathbf{e}(t)$ where the next evaluation step has to take place due to Haskell's lazy evaluation strategy. In general, $\mathbf{e}(t)$ is the top position ε. There are two exceptions. First, consider terms "$f\,t_1 \ldots t_n\,t_{n+1} \ldots t_m$" where $\mathrm{arity}(f) = n$ and $m > n$. Here, f is applied to too many arguments. Thus, one considers the subterm "$f\,t_1 \ldots t_n$" at position 1^{m-n} to find the evaluation position. The other exception is when one has to evaluate a subterm of "$f\,t_1 \ldots t_n$" in order to check whether a defining f-equation "$f\,\ell_1 \ldots\ \ell_n = r$" will afterwards become applicable at top position. We say that an equation $\ell = r$ from the program is *feasible* for a term t and define the corresponding *evaluation position* $\mathbf{e}_\ell(t)$ *with respect to* ℓ if $\mathrm{head}(\ell) = \mathrm{head}(t) = f$ for some f and either[8]:

(a) ℓ matches t (then we define $\mathbf{e}_\ell(t) = \varepsilon$), or
(b) for the leftmost outermost position[9] π where $\mathrm{head}(\ell|_\pi)$ is a constructor and where $\mathrm{head}(\ell|_\pi) \neq \mathrm{head}(t|_\pi)$, the symbol $\mathrm{head}(t|_\pi)$ is defined or a variable. Then $\mathbf{e}_\ell(t) = \pi$.

So in Example 2.1, if t is the term "$\mathsf{take}\,u\,(\mathsf{from}\,m)$" where u and m are variables, then the defining equation "$\mathsf{take}\,\mathsf{Z}\,xs\ =\ \mathsf{Nil}$" is feasible for t. For $\ell = \mathsf{take}\,\mathsf{Z}\,xs$, the corresponding evaluation position is $\mathbf{e}_\ell(t) = 1\,2$. The reason is that $\pi = 1\,2$ is the leftmost outermost position where $\mathrm{head}(\ell|_\pi) = \mathsf{Z}$ is a constructor that is different from

[8]To simplify the presentation, in the article we do not regard data constructors with strictness annotations "!". However, by adapting the definition of $\mathbf{e}_\ell(t)$, our approach can easily handle strictness annotations as well. Indeed, in our implementation we permit constructors with strictness annotations. By using such constructors, one can also express operators like "seq" which enforce a special evaluation strategy. More precisely, one can define a new data type "data Strict a = BeStrict !a", a function "seq2 (BeStrict x) y = y", and then replace every call "seq t_1 t_2" by "seq2 (BeStrict t_1) t_2".

[9]The *leftmost outermost position* can be formally defined as the smallest position with respect to $<_{lex}$. Here, $<_{lex}$ is the lexicographic order on positions where a position $\pi_1 = m_1 \ldots m_k$ is smaller than a position $\pi_2 = n_1 \ldots n_\ell$ if there is an $i \in \{1, \ldots, \min(k+1, \ell)\}$ such that $m_j = n_j$ for all $j < i$, and $m_i < n_i$ if $i \leq k$. So for example, $\varepsilon <_{lex} 1 <_{lex} 1\,1 <_{lex} 1\,2 <_{lex} 2$.

$\mathrm{head}(t|_\pi) = u$, which is a variable. Thus, to decide whether the defining equation is applicable to t, one would first have to evaluate t at the position $\mathbf{e}_\ell(t) = 1\,2$.

On the other hand, the defining equation "take Z xs = Nil" is not feasible for the term s = take (S u) (from m), since $\mathrm{head}(s|_{1\,2})$ = S is neither defined nor a variable. In other words, this defining equation will never become applicable to s. But the second defining equation "take n Nil = Nil" is feasible for s. For ℓ' = take n Nil, we obtain $\mathbf{e}_{\ell'}(s) = 2$, as $\mathrm{head}(\ell'|_2)$ = Nil is a constructor and $\mathrm{head}(s|_2)$ = from is defined. So to find out whether "take n Nil = Nil" is applicable to s, one would have to evaluate its subterm "from m".

Since Haskell considers the order of the program's equations, a term t is evaluated below the top (at position $\mathbf{e}_\ell(t)$), whenever (b) holds for the *first* feasible equation $\ell = r$ (even if an evaluation with a *subsequent* defining equation would be possible at top position). Thus, this is no ordinary leftmost outermost evaluation strategy. By taking the order of the defining equations into account, we can now define the position $\mathbf{e}(t)$ where the next evaluation step has to take place.

Definition 2.2 (*Evaluation Position* $\mathbf{e}(t)$). For any term t, we define

$$\mathbf{e}(t) = \begin{cases} 1^{m-n}\,\pi, & \text{if } t = (f\,t_1 \ldots t_n\,t_{n+1} \ldots t_m),\ f \text{ is defined, } m > n = \mathrm{arity}(f), \\ & \text{and } \pi = \mathbf{e}(\ f\,t_1 \ldots t_n\ ) \\ \mathbf{e}_\ell(t)\,\pi, & \text{if } t = (f\,t_1 \ldots t_n),\ f \text{ is defined, } n = \mathrm{arity}(f), \text{ there are} \\ & \text{feasible equations for } t \text{ (the first one is "}\ell = r\text{"), } \mathbf{e}_\ell(t) \neq \varepsilon, \\ & \text{and } \pi = \mathbf{e}(t|_{\mathbf{e}_\ell(t)}) \\ \varepsilon, & \text{otherwise} \end{cases}$$

So if t = take u (from m) and s = take (S u) (from m), then $t|_{\mathbf{e}(t)} = u$ and $s|_{\mathbf{e}(s)}$ = from m.

We now present Haskell's operational semantics by defining the *evaluation relation* $\to_{\mathsf{H}}$. For any term t, it performs a rewrite step at position $\mathbf{e}(t)$ using the *first* applicable defining equation of the program. So terms like "x Z" or "take Z" are normal forms: If the head of t is a variable or if a symbol is applied to too few arguments, then $\mathbf{e}(t) = \varepsilon$ and no rule rewrites t at top position. Moreover, a term $s = (f\ s_1 \ldots s_m)$ with a defined symbol f and $m \geq \mathrm{arity}(f)$ is a normal form if no equation in the program is feasible for s. If additionally $\mathrm{head}(s|_{\mathbf{e}(s)})$ is a defined symbol g, then we call s an *error term* (i.e., then g is not defined for some argument patterns). We consider such error terms as terminating, since they do not start an infinite evaluation (indeed, Haskell aborts the program when trying to evaluate an error term).

For terms $t = (c\ t_1 \ldots t_n)$ with a constructor c of arity n, we also have $\mathbf{e}(t) = \varepsilon$ and no rule rewrites t at top position. However, here we permit rewrite steps below the top, that is, $t_1, \ldots, t_n$ may be evaluated with $\to_{\mathsf{H}}$. This corresponds to the behavior of Haskell interpreters like Hugs [Jones and Peterson 1999] which evaluate terms until they can be displayed as a string. To transform data objects into strings, Hugs uses a function "show". This function can be generated automatically for user-defined types by adding "deriving Show" behind the data declarations. This default implementation of the show function would transform every data object "$c\ t_1 \ldots t_n$" into the string consisting of "c" and of show t_1, ..., show t_n. Thus, show would require that all arguments of a term with a constructor head have to be evaluated.

Definition 2.3 (*Evaluation Relation* $\to_{\mathsf{H}}$). We have $t \to_{\mathsf{H}} s$ iff either:

(1) t rewrites to s at the position $\mathbf{e}(t)$ using the first equation of the program whose left-hand side matches $t|_{\mathbf{e}(t)}$, or
(2) $t = (c\ t_1 \ldots t_n)$ for a constructor c of arity n, $t_i \to_{\mathsf{H}} s_i$ for some $1 \leq i \leq n$, and $s = (c\ t_1 \ldots t_{i-1}\ s_i\ t_{i+1} \ldots t_n)$.

For example, we have the infinite evaluation $\mathsf{from}\,m \to_{\mathsf{H}} \mathsf{Cons}\,m(\mathsf{from}\,(\mathsf{S}\,m)) \to_{\mathsf{H}} \mathsf{Cons}\,m(\mathsf{Cons}\,(\mathsf{S}\,m)(\mathsf{from}\,(\mathsf{S}\,(\mathsf{S}\,m)))) \to_{\mathsf{H}} \ldots$ On the other hand, the following evaluation is finite due to Haskell's lazy evaluation strategy: $\mathsf{take}\,(\mathsf{S}\,\mathsf{Z})(\mathsf{from}\,m) \to_{\mathsf{H}} \mathsf{take}\,(\mathsf{S}\,\mathsf{Z})(\mathsf{Cons}\,m(\mathsf{from}\,(\mathsf{S}\,m))) \to_{\mathsf{H}} \mathsf{Cons}\,m(\mathsf{take}\,\mathsf{Z}\,(\mathsf{from}\,(\mathsf{S}\,m))) \to_{\mathsf{H}} \mathsf{Cons}\,m\,\mathsf{Nil}$. Note that while evaluation in Haskell uses sharing to increase efficiency, we ignored this in Definition 2.3, since sharing in Haskell does not influence the termination behavior.

The reason for permitting nonground terms in Definition 2.2 and Definition 2.3 is that our termination method in Section 3 evaluates Haskell *symbolically*. Here, variables stand for arbitrary *terminating* terms. Definition 2.4 introduces our notion of termination (which also corresponds to the notion of "termination" examined in Panitz and Schmidt-Schauß [1997]).[10]

Definition 2.4 (H-*Termination*). The set of H-*terminating* ground terms is the smallest set of ground terms t such that:

(a) t does not start an infinite evaluation $t \to_{\mathsf{H}} \ldots$,
(b) if $t \to_{\mathsf{H}}^{*} (f\ t_1 \ldots t_n)$ for a defined function symbol f, $n <$ arity(f), and the term t' is H-terminating, then $(f\ t_1 \ldots t_n\, t')$ is also H-terminating, and
(c) if $t \to_{\mathsf{H}}^{*} (c\ t_1 \ldots t_n)$ for a constructor c, then $t_1, \ldots, t_n$ are also H-terminating.

A term t is H-*terminating* iff $t\sigma$ is H-terminating for all substitutions σ with H-terminating ground terms. Throughout the article, we always restrict ourselves to substitutions of the correct types. These substitutions σ may also introduce new defined function symbols with arbitrary defining equations.

So a term is only H-*terminating* if all its applications to H-terminating terms H-terminate, too. Thus, "from" is not H-terminating, as "from Z" has an infinite evaluation. But "take u (from m)" is H-terminating: when instantiating u and m by H-terminating ground terms, the resulting term has no infinite evaluation.

Example 2.5 (nonterm). To illustrate that one may have to introduce new defining equations to examine H-termination, consider the function nonterm of type Bool $\to$ (Bool $\to$ Bool) $\to$ Bool:

$$\mathsf{nonterm\ True}\ x = \mathsf{True} \qquad \mathsf{nonterm\ False}\ x = \mathsf{nonterm}\ (x\ \mathsf{True})\ x$$

The term "nonterm False x" is not H-terminating: one obtains an infinite evaluation if one instantiates x by the function mapping all arguments to False. But for this instantiation, one has to extend the program by an additional function with the defining equation $\mathsf{g}\ y = \mathsf{False}$. In full Haskell, such functions can of course be represented by lambda terms and indeed, "nonterm False $(\backslash y \to \mathsf{False})$" starts an infinite evaluation.

3. FROM HASKELL TO TERMINATION GRAPHS

Our goal is to prove H-termination of a *start term* t. By Definition 2.4, H-*termination* of t means that $t\sigma$ is H-terminating for all substitutions σ with H-terminating ground terms. Thus, t represents a (usually infinite) set of terms and we want to prove that they are all H-terminating. Without loss of generality, we can restrict ourselves to normal ground substitutions σ, that is, substitutions where $\sigma(x)$ is a ground term in normal form with respect to $\to_{\mathsf{H}}$ for all variables x in t.

We consider the program of Example 2.1 and the start term $t = \mathsf{take}\,u\,(\mathsf{from}\,m)$. As mentioned before, here the variables u and m stand for arbitrary H-*terminating*

[10]As discussed in Panitz and Schmidt-Schauß [1997], there are also other possible notions of termination like "lazy termination," which, however, can be encoded via H-termination; see Panitz and Schmidt-Schauß [1997] and Raffelsieper [2007].

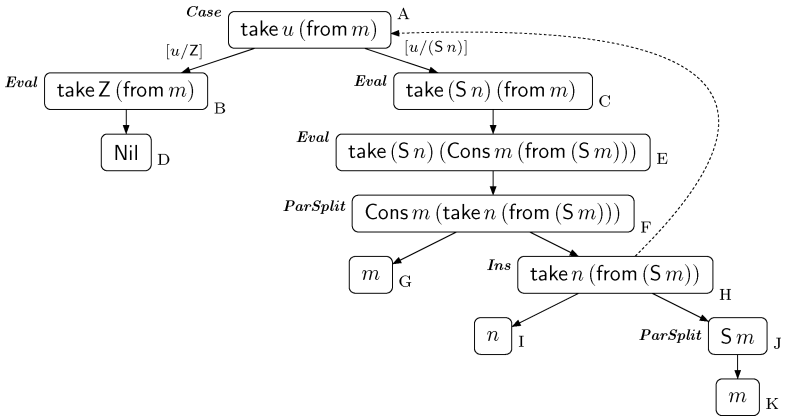

Fig. 1. Termination graph for "take u (from m)".

terms. A naive approach would be to consider the defining equations of all needed functions (i.e., take and from) as rewrite rules and to prove termination of the resulting rewrite system. However, this disregards Haskell's lazy evaluation strategy. So due to the nonterminating rule for "from", we would fail to prove H-termination of t.

Therefore, our approach begins by evaluating the start term a few steps. This gives rise to a so-called *termination graph*. Instead of transforming defining Haskell equations directly into rewrite rules, we transform the termination graph into rewrite rules. (Actually, we transform it into so-called "dependency pair problems," as described in Section 4.) The advantage is that the initial evaluation steps in this graph take the evaluation strategy and the types of Haskell into account and therefore, this is also reflected in the resulting rewrite rules.

To construct a termination graph for the start term t, we begin with the graph containing only one single node, marked with t. Similar to Panitz and Schmidt-Schauß [1997], we then apply *expansion rules* repeatedly to the leaves of the graph in order to extend it by new nodes and edges. As usual, a *leaf* is a node with no outgoing edges. We have obtained a *termination graph* for t if no expansion rule is applicable to its leaves anymore. Afterwards, we try to prove H-termination of all terms occurring in the termination graph, as described in Section 4. A termination graph for the start term "take u (from m)" is depicted in Figure 1. We now describe our five expansion rules intuitively. Their formal definition is given in Definition 3.1.

When constructing termination graphs, the goal is to *evaluate* terms. However, $t = \mathsf{take}\, u\, (\mathsf{from}\, m)$ cannot be evaluated with $\to_{\mathsf{H}}$, since it has a variable u at its evaluation position $\mathbf{e}(t)$. The evaluation can only continue if we know how u is going to be instantiated. Therefore, the first expansion rule is called ***Case Analysis*** (or "***Case***", for short). It adds new child nodes where u is replaced by all terms of the form $(c\ x_1 \ldots x_n)$. Here, c is a constructor of the appropriate type and $x_1, \ldots, x_n$ are fresh variables. The edges to these children are labeled with the respective substitutions $[u/(c\, x_1 \ldots x_n)]$. In our example, u is a variable of type Nats. Therefore, the ***Case*** rule adds two child nodes B and C to our initial node A, where u is instantiated by Z and by $(\mathsf{S}\, n)$, respectively. Since the children of A were generated by the ***Case*** rule, we call A a "***Case*** node". Every node in the graph has the following property: If all its children are marked with H-terminating terms, then the node itself is also marked with an H-terminating term. Indeed, if the terms in nodes B and C are H-terminating, then the term in node A is H-terminating as well.

Now the terms in nodes B and C can indeed be evaluated. Therefore, the ***Evaluation*** rule ("***Eval***") adds the nodes D and E resulting from one evaluation step with $\to_{\mathsf{H}}$. Moreover, E is also an ***Eval*** node, since its term can be evaluated further to the term in node F. So the ***Case*** and ***Eval*** rules perform a form of *narrowing* that respects the evaluation strategy and the types of Haskell. This is similar to evaluation in functional-logic programming languages (e.g., Hanus [2007]).

The term Nil in node D cannot be evaluated and therefore, D is a leaf of the termination graph. But the term "Cons m (take n (from (S m)))" in node F may be evaluated further. Whenever the head of a term is a constructor like Cons or a variable,[11] then one only has to consider the evaluations of its arguments. We use a ***Parameter Split*** rule ("***ParSplit***") which adds new child nodes with the arguments of such terms. Thus, we obtain the nodes G and H. Again, H-termination of the terms in G and H obviously implies H-termination of the term in node F.

The node G remains a leaf since its term m cannot be evaluated further for any normal ground instantiation. For node H, we could continue by applying the rules ***Case***, ***Eval***, and ***ParSplit*** as before. However, in order to obtain finite graphs (instead of infinite trees), we also have an ***Instantiation*** rule ("***Ins***"). Since the term in node H is an *instance* of the term in node A, one can draw an *instantiation edge* from the instantiated term to the more general term (i.e., from H to A). We depict instantiation edges by dashed lines. These are the only edges which may point to already existing nodes (i.e., one obtains a tree if one removes the instantiation edges from a termination graph).

To guarantee that the term in node H is H-terminating whenever the terms in its child nodes are H-terminating, the ***Ins*** rule has to ensure that one only uses instantiations with H-terminating terms. In our example, the variables u and m of node A are instantiated with the terms n and (S m), respectively. Therefore, in addition to the child A, the node H gets two more children I and J marked with n and (S m). Finally, the ***ParSplit*** rule adds J's child K, marked with m.

To illustrate the last of our five expansion rules, we consider a different start term, namely "take". If a defined function has "too few" arguments, then by Definition 2.4 we have to apply it to additional H-terminating arguments in order to examine H-termination. Therefore, we have a ***Variable Expansion*** rule ("***VarExp***") which adds a child marked with "take x" for a fresh variable x. Another application of ***VarExp*** gives "take $x\ xs$". The remaining termination graph is constructed by the rules discussed before. We can now give a formal definition of our expansion rules.

Definition 3.1 (*Termination Graph*). Let G be a graph with a leaf marked with the term t. We say that G can be *expanded* to G' (denoted "$G \Rrightarrow G'$") if G' results from G by adding new **ch**ild nodes marked with the elements of $\mathbf{ch}(t)$ and by adding edges from t to each element of $\mathbf{ch}(t)$. Only in the ***Ins*** rule, we also permit the addition of an edge to an already existing node, which may then lead to cycles. All edges are marked by the identity substitution unless stated otherwise.

- ***Eval***: $\mathbf{ch}(t) = \{\tilde{t}\}$, if $t = (f\, t_1 \ldots t_n)$, f is a defined symbol, $n \geq \text{arity}(f)$, $t \to_{\mathsf{H}} \tilde{t}$.
- ***Case***: $\mathbf{ch}(t) = \{t\sigma_1, \ldots, t\sigma_k\}$, if $t = (f\, t_1 \ldots t_n)$, f is a defined function symbol, $n \geq \text{arity}(f)$, $t|_{\mathbf{e}(t)}$ is a variable x of type "$d\,\tau_1 \ldots \tau_m$" for a type constructor d, the type constructor d has the data constructors c_i of arity n_i (where $1 \leq i \leq k$), and $\sigma_i = [x/(c_i\, x_1 \ldots x_{n_i})]$ for pairwise different fresh variables $x_1, \ldots, x_{n_i}$. The edge from t to $t\sigma_i$ is marked with the substitution σ_i.
- ***VarExp***: $\mathbf{ch}(t) = \{t\, x\}$, if $t = (f\, t_1 \ldots t_n)$, f is a defined function symbol, $n < \text{arity}(f)$, x is a fresh variable.

[11] The reason is that "$x\, t_1 \ldots t_n$" H-terminates iff the terms $t_1, \ldots, t_n$ H-terminate.

ParSplit: $\mathbf{ch}(t) = \{t_1, \ldots, t_n\}$, if $t = (c\, t_1 \ldots t_n)$, c is a constructor or variable, $n > 0$.
Ins: $\mathbf{ch}(t) = \{s_1, \ldots, s_m, \tilde{t}\}$, if $t = (f\ t_1 \ldots t_n)$, t is not an error term, f is a defined symbol, $n \geq \text{arity}(f)$, $t = \tilde{t}\sigma$ for some term $\tilde{t}$, $\sigma = [x_1/s_1, \ldots, x_m/s_m]$, where $\mathcal{V}(\tilde{t}) = \{x_1, \ldots, x_m\}$. Here, either $\tilde{t} = (x\ y)$ for fresh variables[12] x and y or $\tilde{t}$ is an ***Eval*** or ***Case*** node. If $\tilde{t}$ is a ***Case*** node, then it must be guaranteed that all paths starting in $\tilde{t}$ reach an ***Eval*** node or a leaf with an error term after traversing only ***Case*** nodes. This ensures that every cycle of the graph contains at least one ***Eval*** node. The edge from t to $\tilde{t}$ is called an *instantiation edge*.

If the graph already has a node marked with $\tilde{t}$, then instead of adding a new child marked with $\tilde{t}$, one may add an edge from t to the already existing node $\tilde{t}$.

Let G_t be the graph with a single node marked with t and no edges. G is a *termination graph* for t iff $G_t \Rrightarrow^* G$ and G is in normal form with respect to $\Rrightarrow$.

If one disregards ***Ins***, then for each leaf there is at most one rule applicable (and no rule is applicable to leaves consisting of just a variable, a constructor, or an error term). However, the ***Ins*** rule introduces indeterminism. Instead of applying the ***Case*** rule on node A in Figure 1, we could also apply ***Ins*** and generate an instantiation edge to a new node with $\tilde{t} = (\mathsf{take}\, u\, ys)$. Since the instantiation is $[ys/(\mathsf{from}\, m)]$, the node A would get an additional child node marked with the non-H-terminating term $(\mathsf{from}\, m)$. Then our approach in Section 4 which tries to prove H-termination of *all* terms in the termination graph would fail, whereas it succeeds for the graph in Figure 1. Therefore, in our implementation we developed a heuristic for constructing termination graphs. It tries to avoid unnecessary applications of ***Ins*** (since applying ***Ins*** means that one has to prove H-termination of more terms), but at the same time it ensures that the expansion terminates, that is, that one really obtains a termination graph. For details of this heuristic we refer to Swiderski [2005].

Of course, in practice termination graphs can become quite large (e.g., a termination graph for "take u [(m::Int) ..]" using the built-in functions of the Hugs Prelude [Jones and Peterson 1999] already contains 94 nodes).[13] Nevertheless, our experiments in Section 7 will show that constructing termination graphs within automated termination proofs is indeed feasible in practice.

Example 3.2 (tma). An instantiation edge to $\tilde{t} = (x\, y)$ is needed to obtain termination graphs for functions like tma which are applied to "**t**oo **m**any" **a**rguments in recursive calls.[14]

$$\mathsf{tma} :: \mathsf{Nats} \to a$$
$$\mathsf{tma}\,(\mathsf{S}\, m) = \mathsf{tma}\, m\, m$$

We get the termination graph in Figure 2. After applying ***Case*** and ***Eval***, we obtain "$\mathsf{tma}\, m\, m$" in node D which is not an instance of the start term "$\mathsf{tma}\, n$" in node A. Of course, we could continue with ***Case*** and ***Eval*** infinitely often, but to obtain a termination graph, at some point we need to apply the ***Ins*** rule. Here, the only possibility is to regard $t = (\mathsf{tma}\, m\, m)$ as an instance of the term $\tilde{t} = (x\, y)$. Thus, we obtain an instantiation edge to the new node E. As the instantiation is $[x/(\mathsf{tma}\, m), y/m]$, we get additional child nodes F and G marked with "$\mathsf{tma}\, m$" and m, respectively. Now we can "close" the graph, since "$\mathsf{tma}\, m$" is an instance of the start term "$\mathsf{tma}\, n$" in node A. So the instantiation edge to the special term $(x\, y)$ is used to remove "superfluous" arguments

[12] See Example 3.2 for an explanation why instantiation edges to terms $(x\ y)$ can be necessary.
[13] See `http://aprove.informatik.rwth-aachen.de/eval/Haskell/take_from.html`.
[14] Note that tma is not Hindley-Milner typeable (but has a principal type). Hence, Haskell can verify the given type of tma, but it cannot infer the type of tma itself.

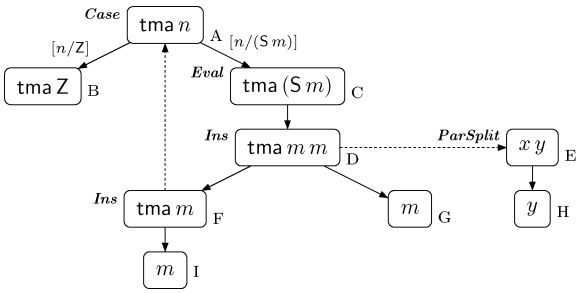

Fig. 2. Termination graph for "tma n".

(i.e., it effectively reduces the analysis of "tma $m\,m$" in node D to the analysis of "tma m" in node F). Of course, in any termination graph it suffices to have at most one node of the form "$(x\,y)$". To expand the node "$(x\,y)$" further, one uses the ***ParSplit*** rule to create its child node with the term y.

Theorem 3.3 shows that by the expansion rules of Definition 3.1 one can always obtain normal forms.[15]

THEOREM 3.3 (EXISTENCE OF TERMINATION GRAPHS). *The relation $\Rightarrow$ is normalizing, that is, for any term t there exists a termination graph.*

4. FROM TERMINATION GRAPHS TO DP PROBLEMS

Now we present a method to prove H-termination of all terms in a termination graph. To this end, we want to use existing techniques for termination analysis of term rewriting. One of the most popular termination techniques for TRSs is the *dependency pair* (DP) method [Arts and Giesl 2000]. In particular, the DP method can be formulated as a general framework which permits the integration and combination of *any* termination technique for TRSs [Giesl et al. 2005a, 2006c; Hirokawa and Middeldorp 2005, 2007]. This *DP framework* operates on so-called *DP problems* $(\mathcal{P}, \mathcal{R})$. Here, $\mathcal{P}$ and $\mathcal{R}$ are TRSs where $\mathcal{P}$ may also have rules $\ell \to r$ where r contains extra variables not occurring in ℓ. $\mathcal{P}$'s rules are called *dependency pairs*. The goal of the DP framework is to show that there is no infinite *chain*, that is, no infinite reduction $s_1\sigma_1 \to_{\mathcal{P}} t_1\sigma_1 \to^*_{\mathcal{R}} s_2\sigma_2 \to_{\mathcal{P}} t_2\sigma_2 \to^*_{\mathcal{R}} \ldots$ where $s_i \to t_i \in \mathcal{P}$ and σ_i are substitutions. In this case, the DP problem $(\mathcal{P}, \mathcal{R})$ is called *finite*. See, for example, Giesl et al. [2005a, 2006c] and Hirokawa and Middeldorp [2005, 2007] for an overview of techniques to prove finiteness of DP problems.[16]

Instead of transforming termination graphs into TRSs, the information available in the termination graph can be better exploited if one transforms these graphs into DP problems.[17] Then finiteness of the resulting DP problems implies H-termination of all terms in the termination graph.

Note that termination graphs still contain higher-order terms (e.g., applications of variables to other terms like "$x\,y$" and partial applications like "take u"). However, most methods and tools for automated termination analysis only operate on first-order TRSs.

[15] All proofs can be found in the online appendix accessible in the ACM Digital Library.

[16] In the DP literature, one usually does not consider rules with extra variables on right-hand sides, but almost all existing termination techniques for DPs can also be used for such rules. (For example, finiteness of such DP problems can often be proved by eliminating the extra variables by suitable *argument filterings* [Arts and Giesl 2000; Giesl et al. 2005a].)

[17] We will discuss the disadvantages of a transformation into TRSs at the end of this section.

Therefore, one option would be to translate higher-order terms into *applicative* first-order terms containing just variables, constants, and a binary symbol ap for function application; see Kennaway et al. [1996], Giesl et al. [2005b, 2006a], and Hirokawa et al. [2008]. Then terms like "$x\,y$", "take u", and "take $u\,xs$" would be transformed into the first-order terms $\mathsf{ap}(x, y)$, $\mathsf{ap}(\mathsf{take}, u)$, and $\mathsf{ap}(\mathsf{ap}(\mathsf{take}, u), xs)$, respectively. In Section 5, we will present a more sophisticated way to translate the higher-order terms from the termination graph into first-order terms. But at the moment, we disregard this problem and transform termination graphs into DP problems that may indeed contain higher-order terms.

Recall that if a node in the termination graph is marked with a non-H-terminating term, then one of its children is also marked with a non-H-terminating term. Hence, every non-H-terminating term corresponds to an infinite path in the termination graph. Since a termination graph only has finitely many nodes, infinite paths have to end in a cycle. Thus, it suffices to prove H-termination for all terms occurring in *strongly connected components (SCCs)* of the termination graph. Moreover, one can analyze H-termination separately for each SCC. Here, an SCC is a maximal subgraph G' of the termination graph such that for all nodes n_1 and n_2 in G' there is a nonempty path from n_1 to n_2 traversing only nodes of G'. (In particular, there must also be a nonempty path from every node to itself in G'.) The termination graph for "$\mathsf{take}\,u\,(\mathsf{from}\,m)$" in Figure 1 has just one SCC with the nodes A, C, E, F, H. The following definition is needed to generate dependency pairs from SCCs of the termination graph.

Definition 4.1 (*DP Path*). Let G' be an SCC of a termination graph containing a path from a node marked with s to a node marked with t. We say that this path is a *DP path* if it does not traverse instantiation edges, if s has an incoming instantiation edge in G', and if t has an outgoing instantiation edge in G'.

So in Figure 1, the only DP path is A, C, E, F, H. Since every infinite path has to traverse instantiation edges infinitely often, it also has to traverse DP paths infinitely often. Therefore, we generate a dependency pair for each DP path. If there is no infinite chain with these dependency pairs, then no term corresponds to an infinite path, so all terms in the graph are H-terminating.

More precisely, whenever there is a DP path from a node marked with s to a node marked with t and the edges of the path are marked with $\sigma_1, \ldots, \sigma_k$, then we generate the dependency pair $s\sigma_1 \ldots \sigma_k \to t$. In Figure 1, the first edge of the DP path is labeled with the substitution $[u/(\mathsf{S}\,n)]$ and all remaining edges are labeled with the identity. Thus, we generate the dependency pair

$$\mathsf{take}\,(\mathsf{S}\,n)\,(\mathsf{from}\,m) \to \mathsf{take}\,n\,(\mathsf{from}\,(\mathsf{S}\,m)). \tag{1}$$

The resulting DP problem is $(\mathcal{P}, \mathcal{R})$ where $\mathcal{P} = \{(1)\}$ and $\mathcal{R} = \varnothing$.[18] When using an appropriate translation into first-order terms as sketched before, automated termination tools (such as AProVE [Giesl et al. 2006b], TTT2 [Korp et al. 2009], and others) can easily show that this DP problem is finite. Hence, the start term "$\mathsf{take}\,u\,(\mathsf{from}\,m)$" is H-terminating in the original Haskell program.

Similarly, finiteness of the DP problem $(\{\mathsf{tma}\,(\mathsf{S}\,m) \to \mathsf{tma}\,m\}, \varnothing)$ for the start term "$\mathsf{tma}\,n$" from Figure 2 is also easy to prove automatically.

The construction of DP problems from the termination graph must be done in such a way that there is an infinite chain whenever the termination graph contains a non-H-terminating term. Indeed, in this case there also exists a DP path in the termination graph whose first node s is not H-terminating. We should construct the DP problems in such a way that s also starts an infinite chain. Clearly if s is not H-terminating,

[18] Definition 4.9 will explain how to generate $\mathcal{R}$ in general.

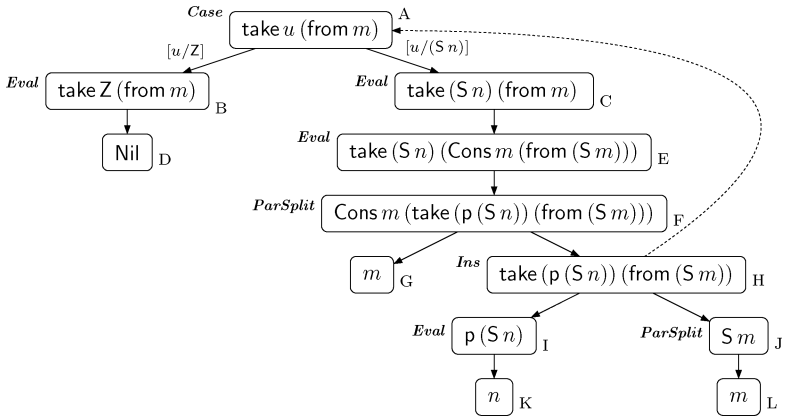

Fig. 3. Termination graph for new "take u (from m)".

then there is a normal ground substitution σ where $s\sigma$ is not H-terminating either. There must be a DP path from s to a term t labeled with the substitutions $\sigma_1, \ldots, \sigma_k$ such that $t\sigma$ is also not H-terminating and such that σ is an instance of $\sigma_1 \ldots \sigma_k$ (as σ is a *normal* ground substitution and the substitutions $\sigma_1, \ldots, \sigma_k$ originate from ***Case*** analyses that consider all possible constructors of a data type). So the first step of the desired corresponding infinite chain is $s\sigma \to_{\mathcal{P}} t\sigma$. The node t has an outgoing instantiation edge to a node $\tilde{t}$ which starts another DP path. So to continue the construction of the infinite chain in the same way, we now need a non-H-terminating instantiation of $\tilde{t}$ with a normal ground substitution. Obviously, $\tilde{t}$ matches t by some matcher μ. But while $\tilde{t}\mu\sigma$ is not H-terminating, the substitution $\mu\sigma$ is not necessarily a normal ground substitution. The reason is that t and hence μ may contain defined symbols. The following example demonstrates this problem.

Example 4.2 (take *with* p). A slightly more challenging example is obtained by replacing the last take rule in Example 2.1 by the following two rules, where p computes the **p**redecessor function.

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

We consider the start term "take u (from m)" again. The resulting termination graph is shown in Figure 3. The only DP path is A, C, E, F, H, which would result in the dependency pair $\mathsf{take}\,(\mathsf{S}\,n)\,(\mathsf{from}\,m) \to t$ with $t = \mathsf{take}\,(\mathsf{p}\,(\mathsf{S}\,n))\,(\mathsf{from}\,(\mathsf{S}\,m))$. Now t has an instantiation edge to node A with $\tilde{t} = \mathsf{take}\,u\,(\mathsf{from}\,m)$. The matcher is $\mu = [u/(\mathsf{p}\,(\mathsf{S}\,n)), m/(\mathsf{S}\,m)]$. So $\mu(u)$ is not normal.

In Example 4.2, the problem of defined symbols in right-hand sides of dependency pairs can be avoided by already evaluating the right-hand sides of dependency pairs as much as possible. To this end, we define an appropriate function **ev**. Before presenting the formal definition of **ev** in Definition 4.4, we will motivate it step by step. More precisely, we will discuss how $\mathbf{ev}(t)$ should be defined for different kinds of nodes t.

So instead of a dependency pair $s\sigma_1 \ldots \sigma_k \to t$ we now generate the dependency pair $s\sigma_1 \ldots \sigma_k \to \mathbf{ev}(t)$. For a node marked with t, essentially $\mathbf{ev}(t)$ is the term reachable from t by traversing only ***Eval*** nodes. So in our example we have $\mathbf{ev}(\mathsf{p}\,(\mathsf{S}\,n)) = n$, since node I is an ***Eval*** node with an edge to node K. Moreover, we will define **ev** in such a way that $\mathbf{ev}(t)$ can also evaluate subterms of t if t is an ***Ins*** node or a ***ParSplit*** node with a constructor as head. We obtain $\mathbf{ev}(\mathsf{S}\,m) = \mathsf{S}\,m$ for node J and $\mathbf{ev}(\mathsf{take}\,(\mathsf{p}\,(\mathsf{S}\,n))$

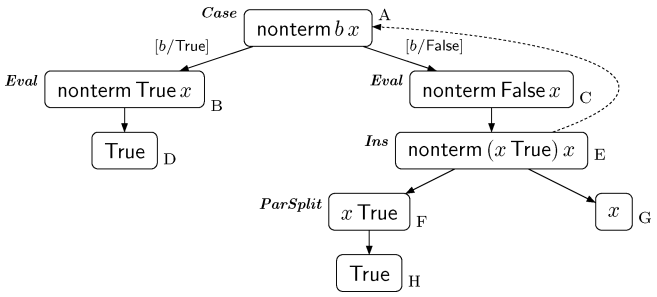

Fig. 4. Termination graph for "nonterm $b\,x$".

(from (S m))) = take n (from (S m)) for node H. Thus, the resulting DP problem is again $(\mathcal{P}, \mathcal{R})$ with $\mathcal{P} = \{(1)\}$ and $\mathcal{R} = \varnothing$.

To show how $\mathbf{ev}(t)$ should be defined for ***ParSplit*** nodes where head(t) is a variable, we consider the function nonterm from Example 2.5 again. The termination graph for the start term "nonterm $b\,x$" is given in Figure 4. We obtain a DP path from node A with the start term to node E with "nonterm (x True) x" labeled with the substitution $[b/\mathsf{False}]$. So the resulting DP problem only contains the dependency pair "nonterm False x $\rightarrow$ $\mathbf{ev}$(nonterm (x True) x)". If we defined $\mathbf{ev}(x\,\mathsf{True}) = x\,\mathsf{True}$, then $\mathbf{ev}$ would not modify the term "nonterm (x True) x". But then the resulting DP problem would be finite and one could falsely prove H-termination. (The reason is that the DP problem contains no rule to transform any instance of "x True" to False.) But as discussed in Section 3, x can be instantiated by arbitrary H-terminating functions and then, "x True" can evaluate to any term of type Bool. Therefore, we should define $\mathbf{ev}$ in such a way that it replaces subterms like "x True" by fresh variables.

Let $\mathbf{U}_G$ be the set of all ***ParSplit*** nodes[19] with variable heads in a termination graph G. In other words, this set contains nodes whose evaluations can lead to any term of a given type.

$$\mathbf{U}_G = \{t \mid t \text{ is a } \textbf{\textit{ParSplit}} \text{ node in } G \text{ with } t = (x\,t_1\ \ldots\ t_n)\}$$

Recall that if t is an ***Ins*** node or a ***ParSplit*** node with a constructor head, then $\mathbf{ev}$ proceeds by evaluating subterms of t. More precisely, let $t = \tilde{t}[x_1/s_1, ..., x_m/s_m]$, where either $\tilde{t} = (c\ x_1 \ldots x_m)$ for a constructor c (then t is a ***ParSplit*** node) or t is an ***Ins*** node and there is an instantiation edge to $\tilde{t}$. In both cases, t also has the children $s_1, \ldots, s_m$. As mentioned before, we essentially define $\mathbf{ev}(t) = \tilde{t}[x_1/\mathbf{ev}(s_1), \ldots, x_m/\mathbf{ev}(s_m)]$. However, whenever there is a path from s_i to a term from $\mathbf{U}_G$ (i.e., to a term $(x \ldots)$ that $\mathbf{ev}$ approximates by a fresh variable), then instead of $\mathbf{ev}(s_i)$ one should use a fresh variable in the definition of $\mathbf{ev}(t)$. A fresh variable is needed because then an instantiation of s_i could in principle evaluate to any value.

Example 4.3 ($\mathbf{ev}$ *for* ***Ins*** *Nodes*). Consider the following program:

$$\mathsf{f\ Z}\ z = \mathsf{f}\ (\mathsf{id}\ z\ \mathsf{Z})\ z \qquad\qquad \mathsf{id}\ x = x$$

The termination graph for the start term "f $x\,z$" is depicted in Figure 5. From the DP path A, C, D, we obtain the dependency pair "f Z $z \rightarrow$ $\mathbf{ev}$(f (id z Z) z)". Note that node D is an ***Ins*** node where (f (id z Z) z) = (f $x\,z$) $[x/(\mathsf{id}\,z\,\mathsf{Z}),\ z/z]$, that is, here we have $\tilde{t} = (\mathsf{f}\ x\ z)$.

If one defined $\mathbf{ev}$ on ***Ins*** nodes by simply applying it to the child subterms, then we would get $\mathbf{ev}$(f (id z Z) z) = f $\mathbf{ev}$(id z Z) $\mathbf{ev}$(z). Clearly, $\mathbf{ev}(z) = z$. Moreover, (id z Z) (i.e.,

[19]To simplify the presentation, we identify nodes with the terms they are labeled with.

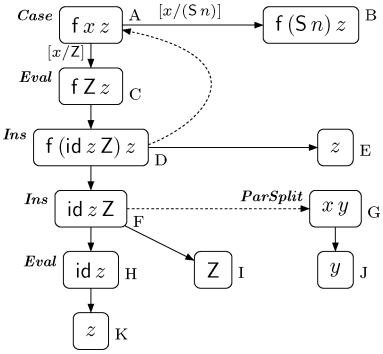

Fig. 5. Termination graph for "f x z".

node F) is again an ***Ins*** node where (id z Z) = (x y)[x/(id z), y/Z]. Thus, **ev**(id z Z) = **ev**(id z) **ev**(Z) = z Z. Hence, the resulting dependency pair "f Z z → **ev**(f (id z Z) z)" would be "f Z z → f (z Z) z". The resulting DP problem would contain no rules $\mathcal{R}$. As this DP problem is finite, then we could falsely prove H-termination of f.

However, there is a path from node F with the child subterm (id z Z) to node G with the term (x y) which is in $\mathbf{U}_G$. As discussed, when computing **ev**(f (id z Z) z), **ev** should not be applied to child subterms like F, but instead, one should replace such child subterms by fresh variables. So we obtain **ev**(f (id z Z) z) = f u z for a fresh variable u. The resulting dependency pair f Z z → f u z indeed gives rise to an infinite chain.

Let the set $\mathbf{PU}_G$ contain all nodes from which there is a path to a node in $\mathbf{U}_G$. So in particular, we also have $\mathbf{U}_G \subseteq \mathbf{PU}_G$. For instance, in Example 4.3 we have $\mathbf{U}_G = \{\text{G}\}$ and $\mathbf{PU}_G = \{\text{A}, \text{C}, \text{D}, \text{F}, \text{G}\}$. Now we can define **ev** formally.

Definition 4.4 (**ev**). Let G be a termination graph with a node t. Then

$$\mathbf{ev}(t) = \begin{cases} x, & \text{for a fresh variable } x, \text{ if } t \in \mathbf{U}_G \\ t, & \text{if } t \text{ is a leaf, a } \textbf{\textit{Case}} \text{ node, or a } \textbf{\textit{VarExp}} \text{ node} \\ \mathbf{ev}(\tilde{t}), & \text{if } t \text{ is an } \textbf{\textit{Eval}} \text{ node with child } \tilde{t} \\ \tilde{t}[x_1/t_1, \ldots, x_m/t_m], & \\ & \text{if } t = \tilde{t}[x_1/s_1, \ldots, x_m/s_m] \text{ and} \\ & \quad — \ t \text{ is an } \textbf{\textit{Ins}} \text{ node with the children } s_1, \ldots, s_m, \tilde{t} \text{ or} \\ & \quad\ \ t \text{ is a } \textbf{\textit{ParSplit}} \text{ node, } \tilde{t} = (c\ x_1 \ldots x_m) \text{ for a constructor } c \\ & \quad — \ t_i = \begin{cases} y_i, & \text{for a fresh variable } y_i, \text{ if } s_i \in \mathbf{PU}_G \\ \mathbf{ev}(s_i), & \text{otherwise} \end{cases} \end{cases}$$

Our goal was to construct the DP problems in such a way that there is an infinite chain whenever s is the first node in a DP path and $s\sigma$ is not H-terminating for a normal ground substitution σ. As discussed before, then there is a DP path from s to t such that the chain starts with $s\sigma \to_{\mathcal{P}} \mathbf{ev}(t)\sigma$ and such that $t\sigma$ and hence $\mathbf{ev}(t)\sigma$ is also not H-terminating. The node t has an instantiation edge to some node $\tilde{t}$. Thus, $t = \tilde{t}[x_1/s_1, \ldots, x_m/s_m]$ and $\mathbf{ev}(t) = \tilde{t}[x_1/\mathbf{ev}(s_1), \ldots, x_m/\mathbf{ev}(s_m)]$, if we assume for simplicity that the s_i are not from $\mathbf{PU}_G$.

In order to continue the construction of the infinite chain, we need a non-H-terminating instantiation of $\tilde{t}$ with a normal ground substitution. Clearly, if $\tilde{t}$ is instantiated by the substitution $[x_1/\mathbf{ev}(s_1)\sigma, \ldots, x_m/\mathbf{ev}(s_m)\sigma]$, then it is again not H-terminating. However, the substitution $[x_1/\mathbf{ev}(s_1)\sigma, \ldots, x_m/\mathbf{ev}(s_m)\sigma]$ is not necessarily normal. The problem is that **ev** stops performing evaluations as soon as one reaches a ***Case*** node

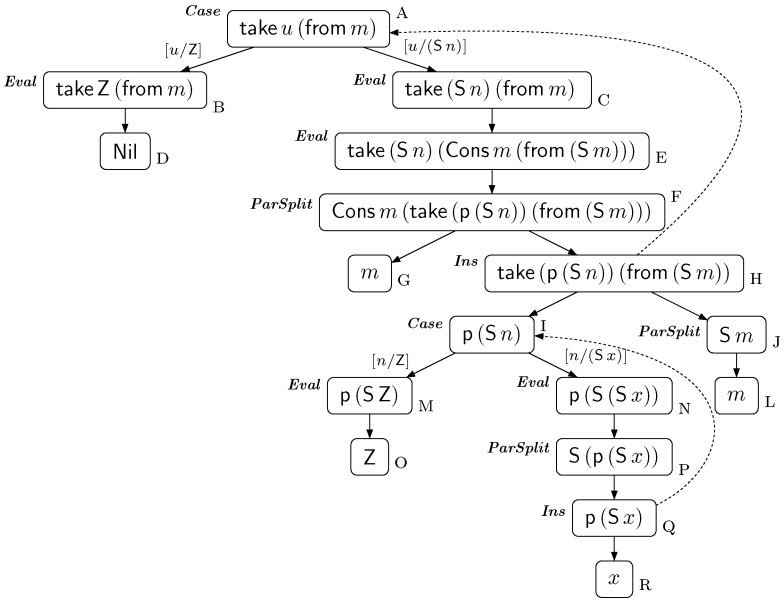

Fig. 6. Termination graph for "take u (from m)" with modified p-equations.

(i.e., **ev** is not propagated over edges originating in ***Case*** nodes). A similar problem is due to the fact that **ev** is also not propagated over instantiation edges.

Therefore, we now generate DP problems which do not just contain dependency pairs $\mathcal{P}$, but they also contain all rules $\mathcal{R}$ which might be needed to evaluate $\mathbf{ev}(s_i)\sigma$ further. Then we obtain $s\sigma \to_{\mathcal{P}} \mathbf{ev}(t)\sigma \to^*_{\mathcal{R}} \tilde{t}\sigma'$ for a normal ground substitution σ' where $\tilde{t}\sigma'$ is not H-terminating. Since $\tilde{t}$ is again the first node in a DP path, now this construction of the chain can be continued in the same way infinitely often. Hence, we obtain an infinite chain.

Example 4.5 (take *with Recursive* p). To illustrate this, we replace the equation for p in Example 4.2 by the following two defining equations:

$$\mathsf{p}(\mathsf{S}\,\mathsf{Z}) = \mathsf{Z} \qquad \mathsf{p}(\mathsf{S}\,x) = \mathsf{S}(\mathsf{p}\,x)$$

For the start term "take u (from m)", we obtain the termination graph depicted in Figure 6. So I is now a ***Case*** node. Thus, instead of (1) we have the dependency pair

$$\mathsf{take}\,(\mathsf{S}\,n)\,(\mathsf{from}\,m) \;\to\; \mathsf{take}\,(\mathsf{p}\,(\mathsf{S}\,n))\,(\mathsf{from}\,(\mathsf{S}\,m)), \tag{2}$$

since now $\mathbf{ev}(\mathsf{p}\,(\mathsf{S}\,n)) = \mathsf{p}\,(\mathsf{S}\,n)$. Hence, the resulting DP problem must contain all rules $\mathcal{R}$ that might be used to evaluate $\mathsf{p}\,(\mathsf{S}\,n)$ when instantiated by a normal ground substitution σ.

So for any term t, we want to detect the rules that might be needed to evaluate $\mathbf{ev}(t)\sigma$ further for normal ground substitutions σ. To this end, we first compute the set $\mathbf{con}(t)$ of those terms that are reachable from t, but where the computation of **ev** stopped. In other words, $\mathbf{con}(t)$ contains all terms which might give rise to further **con**tinuing evaluations that are not captured by **ev**. To compute $\mathbf{con}(t)$, we traverse all paths starting in t. If we reach a ***Case*** node s, we stop traversing this path and insert s into $\mathbf{con}(t)$. Moreover, if we traverse an instantiation edge to some node $\tilde{t}$, we also stop and insert $\tilde{t}$ into $\mathbf{con}(t)$. So in the example of Figure 6, we obtain $\mathbf{con}(\mathsf{p}\,(\mathsf{S}\,n)) = \{\mathsf{p}\,(\mathsf{S}\,n)\}$, since I is now a ***Case*** node. If we had started with the term $t = \mathsf{take}\,(\mathsf{S}\,n)\,(\mathsf{from}\,m)$ in

node C, then we would reach the ***Case*** node I and the node A which is reachable via an instantiation edge. So $\mathbf{con}(t) = \{\mathsf{p}(\mathsf{S}\,n), \mathsf{take}\,u\,(\mathsf{from}\,m)\}$. Moreover, **con** also stops at leaves and at ***VarExp*** nodes t, since they are in normal form with respect to $\to_{\mathsf{H}}$. Thus, here $\mathbf{con}(t) = \varnothing$. Finally, note that **con** is initially applied to ***Ins*** nodes (i.e., to terms on right-hand sides of dependency pairs). Hence, if a (sub)term t is in $\mathbf{PU}_G$, then **ev** already approximates the result of t's evaluation by a fresh variable. Thus, one also defines $\mathbf{con}(t) = \varnothing$ for all $t \in \mathbf{PU}_G$.

Definition 4.6 (**con**). Let G be a termination graph with a node t. Then

$$\mathbf{con}(t) = \begin{cases} \varnothing, & \text{if } t \text{ is a leaf or a } \textbf{\textit{VarExp}} \text{ node or } t \in \mathbf{PU}_G \\ \{t\}, & \text{if } t \text{ is a } \textbf{\textit{Case}} \text{ node} \\ \{\tilde{t}\} \cup \mathbf{con}(s_1) \cup \ldots \cup \mathbf{con}(s_m), & \text{if } t \text{ is an } \textbf{\textit{Ins}} \text{ node with} \\ & \text{the children } s_1, \ldots, s_m, \tilde{t} \text{ and an instantiation edge from } t \text{ to } \tilde{t} \\ \bigcup_{t' \text{child of } t} \mathbf{con}(t'), & \text{otherwise} \end{cases}$$

Now we can define how to extract a DP problem $\mathbf{dp}_{G'}$ from every SCC G' of the termination graph. As mentioned, we generate a dependency pair $s\sigma_1 \ldots \sigma_k \to \mathbf{ev}(t)$ for every DP path from s to t labeled with $\sigma_1, \ldots, \sigma_k$ in G'. If $t = \tilde{t}[x_1/s_1, \ldots, x_m/s_m]$ has an instantiation edge to $\tilde{t}$, then the resulting DP problem must contain all rules that can be used to reduce the terms in $\mathbf{con}(s_1) \cup \ldots \cup \mathbf{con}(s_m)$. For any term q, let $\mathbf{rl}(q)$ be the rules that can be used to reduce $q\sigma$ for normal ground substitutions σ. We will give the definition of **rl** afterwards.

Definition 4.7 (**dp**). For a termination graph containing an SCC G', we define $\mathbf{dp}_{G'} = (\mathcal{P}, \mathcal{R})$. Here, $\mathcal{P}$ and $\mathcal{R}$ are the smallest sets such that:

—"$s\sigma_1 \ldots \sigma_k \to \mathbf{ev}(t)$" $\in \mathcal{P}$ and
—$\mathbf{rl}(q) \subseteq \mathcal{R}$,

whenever G' contains a DP path from s to t labeled with $\sigma_1, \ldots, \sigma_k$, $t = \tilde{t}[x_1/s_1, \ldots, x_m/s_m]$ has an instantiation edge to $\tilde{t}$, and $q \in \mathbf{con}(s_1) \cup \ldots \cup \mathbf{con}(s_m)$.

In Example 4.5, the termination graph in Figure 6 has two SCCs G_1 (consisting of the nodes A, C, E, F, H) and G_2 (consisting of I, N, P, Q). Finiteness of the two DP problems $\mathbf{dp}_{G_1}$ and $\mathbf{dp}_{G_2}$ can be proved independently. The SCC G_1 only has the DP path from A to H leading to the dependency pair (2). So we obtain $\mathbf{dp}_{G_1} = (\{(2)\}, \mathcal{R}_1)$ where $\mathcal{R}_1$ contains $\mathbf{rl}(q)$ for all $q \in \mathbf{con}(\mathsf{p}(\mathsf{S}\,n)) \cup \mathbf{con}(\mathsf{S}\,m) = \{\mathsf{p}(\mathsf{S}\,n)\}$. Thus, $\mathcal{R}_1 = \mathbf{rl}(\mathsf{p}(\mathsf{S}\,n))$, that is, $\mathcal{R}_1$ will contain rules to evaluate p, but no rules to evaluate take.[20] Such rules are not needed in $\mathcal{R}_1$ since the evaluation of take is already captured by the dependency pair (2). The SCC G_2 only has the DP path from I to Q. Hence, $\mathbf{dp}_{G_2} = (\mathcal{P}_2, \mathcal{R}_2)$ where $\mathcal{P}_2$ consists of the dependency pair "$\mathsf{p}(\mathsf{S}(\mathsf{S}\,x)) \to \mathsf{p}(\mathsf{S}\,x)$" (since $\mathbf{ev}(\mathsf{p}(\mathsf{S}\,x)) = \mathsf{p}(\mathsf{S}\,x)$) and $\mathcal{R}_2$ contains $\mathbf{rl}(q)$ for all $q \in \mathbf{con}(x) = \varnothing$, that is, $\mathcal{R}_2 = \varnothing$. Thus, finiteness of $\mathbf{dp}_{G_2}$ can easily be proved automatically.

For every term s, we now show how to extract a set of rules $\mathbf{rl}(s)$ such that every evaluation of $s\sigma$ for a normal ground substitution σ corresponds to a reduction with $\mathbf{rl}(s)$.[21] The only expansion rules which transform terms into "equal" ones are ***Eval*** and ***Case***. This leads to the following definition.

[20] Formally, this is because Definition 4.7 only includes $\mathbf{con}(s_i)$ but not $\mathbf{con}(\tilde{t})$ in $\mathcal{R}$.

[21] More precisely, $s\sigma \to_{\mathsf{H}}^* q$ implies $s\sigma \to_{\mathbf{rl}(s)}^* q'$ for a term q' which is "at least as evaluated" as q (i.e., one can evaluate q further to q' if one also permits evaluation steps below or beside the evaluation position of Definition 2.2. For more details, see the proofs in the online appendix in the ACM Digital Library).

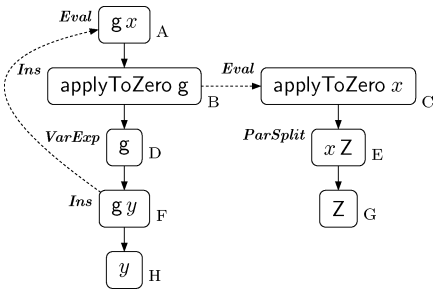

Fig. 7. Termination graph for "g x".

Definition 4.8 (*Rule Path*). A path from a node marked with s to a node marked with t is a *rule path* if s and all other nodes on the path except t are ***Eval*** or ***Case*** nodes and t is not an ***Eval*** or ***Case*** node. So t may also be a leaf.

In Figure 6, there are two rule paths starting in node I. The first one is I, M, O (since O is a leaf) and the other one is I, N, P (since P is a ***ParSplit*** node).

While DP paths give rise to dependency pairs, rule paths give rise to rules. Therefore, if there is a rule path from s to t labeled with $\sigma_1, \ldots, \sigma_k$, then $\mathbf{rl}(s)$ contains the rule $s\sigma_1 \ldots \sigma_k \to \mathbf{ev}(t)$. In addition, $\mathbf{rl}(s)$ must also contain all rules required to evaluate $\mathbf{ev}(t)$ further, that is, all rules in $\mathbf{rl}(q)$ for $q \in \mathbf{con}(t)$.[22]

Definition 4.9 (**rl**). For a node labeled with s, $\mathbf{rl}(s)$ is the smallest set with:

—"$s\sigma_1 \ldots \sigma_k \to \mathbf{ev}(t)$" $\in \mathbf{rl}(s)$ and
—$\mathbf{rl}(q) \subseteq \mathbf{rl}(s)$,

whenever there is a rule path from s to t labeled with $\sigma_1, \ldots, \sigma_k$, and $q \in \mathbf{con}(t)$.

In Example 4.5, we obtained the DP problem $\mathbf{dp}_{G_1} = (\{(2)\}, \mathbf{rl}(\mathsf{p}(\mathsf{S}\,n)))$. Here, $\mathbf{rl}(\mathsf{p}(\mathsf{S}\,n))$ consists of

$$\mathsf{p}(\mathsf{S}\,\mathsf{Z}) \to \mathsf{Z} \qquad \text{(due to the rule path from I to O)} \tag{3}$$

$$\mathsf{p}(\mathsf{S}(\mathsf{S}\,x)) \to \mathsf{S}(\mathsf{p}(\mathsf{S}\,x)) \qquad \text{(due to the rule path from I to P)}, \tag{4}$$

as **ev** does not modify the right-hand sides of (3) and (4). Moreover, the requirement "$\mathbf{rl}(q) \subseteq \mathbf{rl}(\mathsf{p}(\mathsf{S}\,n))$ for all $q \in \mathbf{con}(\mathsf{Z})$ and all $q \in \mathbf{con}(\mathsf{S}(\mathsf{p}(\mathsf{S}\,x)))$" does not add further rules. The reason is that $\mathbf{con}(\mathsf{Z}) = \varnothing$ and $\mathbf{con}(\mathsf{S}(\mathsf{p}(\mathsf{S}\,x))) = \{\mathsf{p}(\mathsf{S}\,n)\}$. Now finiteness of $\mathbf{dp}_{G_1} = (\{2\}, \{(3), (4)\})$ is also easy to show automatically.

Example 4.10 (applyToZero). Next consider the following program and the corresponding termination graph in Figure 7.

$$\mathsf{g}\,x = \mathsf{applyToZero}\,\mathsf{g} \quad \mathsf{applyToZero}\,x = x\,\mathsf{Z}$$

This example shows that one also has to traverse edges resulting from ***VarExp*** when constructing dependency pairs. Otherwise one would falsely prove H-termination. Since the only DP path goes from node A to F, we obtain the DP problem $(\{\mathsf{g}\,x \to \mathsf{g}\,y\}, \mathcal{R})$ with

[22]So if $t = \tilde{t}[x_1/s_1, \ldots, x_m/s_m]$ has an instantiation edge to $\tilde{t}$, then $\mathbf{rl}(t)$ also includes all rules of $\mathbf{rl}(\tilde{t})$, since $\mathbf{con}(t) = \{\tilde{t}\} \cup \mathbf{con}(s_1) \cup \ldots \cup \mathbf{con}(s_m)$. In contrast, in the definition of **dp** (Definition 4.7) we only consider the rules $\mathbf{rl}(q)$ for $q \in \mathbf{con}(s_1) \cup \ldots \cup \mathbf{con}(s_m)$, but not $\mathbf{rl}(\tilde{t})$. The reason is that if t is a right-hand side of a dependency pair, then the evaluations of $\tilde{t}$ are already captured by the dependency pairs, as remarked in Footnote 20.

$\mathcal{R} = \mathbf{rl}(y) = \varnothing$. This problem is not finite and indeed, "$\mathsf{g}\,x$" is not H-terminating, since $\mathsf{g}\,\mathsf{Z} \to_{\mathsf{H}} \mathsf{applyToZero}\,\mathsf{g} \to_{\mathsf{H}} \mathsf{g}\,\mathsf{Z} \to_{\mathsf{H}} \ldots$ In contrast, the definition of **rl** stops at ***VarExp*** nodes.

The following theorem states the soundness of our approach.

THEOREM 4.11 (SOUNDNESS). *Let G be a termination graph. If the DP problems $\mathbf{dp}_{G'}$ are finite for all SCCs G' of G, then all nodes in G are* H*-terminating.*[23]

Example 4.12 (*Incompleteness of Our Approach*). The converse of Theorem 4.11 does not hold. Consider the following program.

$$\begin{aligned}&\mathsf{stuck} \;::\; \mathsf{Bool} \to \mathsf{Bool} \to (\mathsf{Bool} \to \mathsf{Bool}) \to a\\ &\mathsf{stuck}\,\mathsf{True}\,\mathsf{False}\,b \;=\; \mathsf{stuck}\,(b\,\mathsf{True})\,(b\,\mathsf{True})\,b\end{aligned}$$

Clearly, the term $(\mathsf{stuck}\,x\,y\,b)$ is H-terminating, because there is no Boolean function b which returns both True and False when applying it to True. Nevertheless, there exists no termination graph for the start term $(\mathsf{stuck}\,x\,y\,b)$ where the resulting DP problem would be finite. The only dependency pair obtained from the termination graph is $(\mathsf{stuck}\,\mathsf{True}\,\mathsf{False}\;\;b) \;\to\; \mathbf{ev}(\mathsf{stuck}\,(b\,\mathsf{True})\,(b\,\mathsf{True})\,b)$, that is, $(\mathsf{stuck}\,\mathsf{True}\,\mathsf{False}\;\;b) \;\to\; (\mathsf{stuck}\,x\,y\,b)$. Hence, the resulting DP problem is obviously not finite.

Although we have chosen to transform termination graphs into DP problems, it would also be possible to transform termination graphs into TRSs instead and then prove termination of the resulting TRSs. However, this approach has several disadvantages. For example, if the termination graph contains a ***VarExp*** node or a ***ParSplit*** node with a variable as head, then we would obtain rules with extra variables on right-hand sides and thus, the resulting TRSs would never be terminating. In contrast, a DP problem $(\mathcal{P}, \mathcal{R})$ with extra variables in $\mathcal{P}$ can still be finite, since dependency pairs from $\mathcal{P}$ are only applied at top positions in chains. Note that, due to the definition of **ev**, there are never any extra variables in the rules $\mathcal{R}$ of the resulting DP problems $(\mathcal{P}, \mathcal{R})$.

5. FROM HIGHER-ORDER TERMS TO FIRST-ORDER DP PROBLEMS

Up to now, the termination graphs still contain higher-order terms and thus, higher-order terms also occur in the DP problems resulting from these graphs. As discussed in the beginning of Section 4, this is problematic since most approaches for automated termination analysis of term rewriting focus on first-order rewriting only. In Section 4, we already mentioned a possible solution to this problem: higher-order terms could be represented as applicative first-order terms using a special binary function symbol ap (i.e., "$\mathsf{map}\;x\;xs$" would be transformed into $\mathsf{ap}(\mathsf{ap}(\mathsf{map}, x), xs)$). But in spite of some recent approaches for termination analysis of applicative TRSs [Giesl et al. 2005b; Hirokawa et al. 2008], termination techniques are usually considerably more powerful on "ordinary" nonapplicative rewrite systems. Therefore, in this section we present an improvement which first *renames* the terms in the termination graph into first-order terms.[24] Here, we benefit from the structure of the termination graph and thus, we do not construct "applicative terms" as before. Afterwards, the DP problems

[23]Instead of $\mathbf{dp}_{G'} = (\mathcal{P}, \mathcal{R})$, for H-termination it suffices to prove finiteness of $(\mathcal{P}^\sharp, \mathcal{R})$, as shown in the online appendix. Here, $\mathcal{P}^\sharp$ results from $\mathcal{P}$ by replacing each rule $(f\,t_1 \ldots t_n) \to (g\,s_1 \ldots s_m)$ in $\mathcal{P}$ by $(f^\sharp\,t_1 \ldots t_n) \to (g^\sharp\,s_1 \ldots s_m)$, where $f^\sharp$ and $g^\sharp$ are fresh "*tuple*" function symbols; see Arts and Giesl [2000].

[24]The only exception are terms from $\mathbf{U}_G$. These are not renamed into first-order terms, since **ev** replaces them by fresh variables anyway when constructing DP problems from the termination graph.

are constructed from this *renamed* termination graph. This results in DP problems that only contain first-order terms. In this way, we avoid the disadvantages of the "brute-force method" that simply converts all terms into applicative form. Another advantage of these renamed termination graphs is that they allow us to treat types more accurately, as will be explained in Section 6.2.3.

The basic idea for this renaming is to introduce a new function symbol for (almost) every node of the termination graph. Consider again the program from Example 4.5 and the associated termination graph from Figure 6. For node A, we introduce a new function symbol f and replace "take u (from m)" by a term built with f. As arguments of f we take the variables occurring in "take u (from m)". So "take u (from m)" is replaced by the term "f u m". This means that any term of the form "take t_1 (from t_2)" can now be represented as "f t_1 t_2".

Formally, to perform this renaming of the term "take u (from m)" into "f u m", we use a renaming function $\mathbf{r}$ which is applied to each node of the termination graph. The function $\mathbf{r}$ gives new names to the nodes in the graph, but it does not modify variables and it also does not modify constructors if they occur as head symbols.

More precisely, each ***Eval***, ***Case***, or ***VarExp*** node t is renamed to the term $\mathbf{r}(t) = (f_t\, x_1 \ldots x_n)$, where f_t is a new *renaming function symbol* for the term t and $\mathcal{V}(t) = \{x_1, \ldots, x_n\}$. Here we always assume that there is a total order on the variables to determine the order of the sequence $x_1, \ldots, x_n$. Thus, instead of using t in the construction of the left-hand sides of dependency pairs and rules, we now use $\mathbf{r}(t)$. Note that all variables occurring in t are still contained in $\mathbf{r}(t)$. So if t is the start node of a DP path or rule path, then when constructing the left-hand sides of dependency pairs or rules, the substitutions on the DP path or rule path are now applied to $\mathbf{r}(t)$. In Example 4.5, this means that the substitution $[u/(\mathsf{S}\, n)]$ on the DP path from node A to H must be applied to the renamed term "f u m" in node A. So the left-hand side of the dependency pair corresponding to the DP path from A to H is "f (S n) m".

Next we explain how $\mathbf{r}$ operates on ***Ins*** nodes. For an ***Ins*** node t with $\mathbf{ch}(t) = \{s_1, \ldots, s_m, \tilde{t}\}$, where t is connected to node $\tilde{t}$ via an instantiation edge and where $t = \tilde{t}[x_1/s_1, \ldots, x_m/s_m]$, we do not introduce a fresh renaming function symbol f_t. Instead, we reuse the function symbol $f_{\tilde{t}}$ already introduced for $\tilde{t}$. The reason is that t is an instance of $\tilde{t}$. So while[25] $\mathbf{r}(\tilde{t}) = (f_{\tilde{t}}\ x_1 \ldots x_m)$, we now define $\mathbf{r}(t) = (f_{\tilde{t}}\ \mathbf{r}(s_1) \ldots \mathbf{r}(s_m))$. Hence, now $\mathbf{r}(t)$ is also an instance of $\mathbf{r}(\tilde{t})$. So for node H in the termination graph of Example 4.5, we obtain

$$\mathbf{r}(\,\mathsf{take}\,(\mathsf{p}\,(\mathsf{S}\, n))\,(\mathsf{from}\,(\mathsf{S}\, m))\,) \;=\; \mathsf{f}\;\; \mathbf{r}(\mathsf{p}\,(\mathsf{S}\, n))\;\; \mathbf{r}(\mathsf{S}\, m).$$

Finally, we also define $\mathbf{r}$ on ***ParSplit*** nodes. Here, the head symbol is not changed and $\mathbf{r}$ is only applied to the arguments. So for a ***ParSplit*** node $t = (c\, t_1 \ldots t_n)$ where $\mathbf{ch}(t) = \{t_1, \ldots, t_n\}$, we have $\mathbf{r}(t) = (c\, \mathbf{r}(t_1) \ldots \mathbf{r}(t_n))$. This holds both for constructors and variables c. So for node J we have $\mathbf{r}(\mathsf{S}\, m) = \mathsf{S}\, \mathbf{r}(m) = \mathsf{S}\, m$, since $\mathbf{r}$ does not modify leaves of the termination graph like "m".

For the ***Case*** node I, we introduce a new function symbol g and obtain $\mathbf{r}(\mathsf{p}\,(\mathsf{S}\, n)) = \mathsf{g}\, n$. So "g t_3" stands for any term of the form "p (S t_3)". So instead of the DP

$$\mathsf{take}\,(\mathsf{S}\, n)\,(\mathsf{from}\, m) \;\to\; \mathsf{take}\,(\mathsf{p}\,(\mathsf{S}\, n))\,(\mathsf{from}\,(\mathsf{S}\, m)), \tag{2}$$

in Example 4.5 we now obtain the DP

$$\mathsf{f}\,(\mathsf{S}\, n)\, m \;\to\; \mathsf{f}\,(\mathsf{g}\, n)\,(\mathsf{S}\, m). \tag{5}$$

[25]However, $\mathbf{r}(\tilde{t}) = (f_{\tilde{t}}\ x_1 \ldots x_m)$ only holds if $\tilde{t}$ is not the special term $(x\ y)$; see Definition 3.1. In this special case where $\tilde{t} \in \mathbf{U}_G$, we use a fresh renaming function symbol f_t for the ***Ins*** node t.

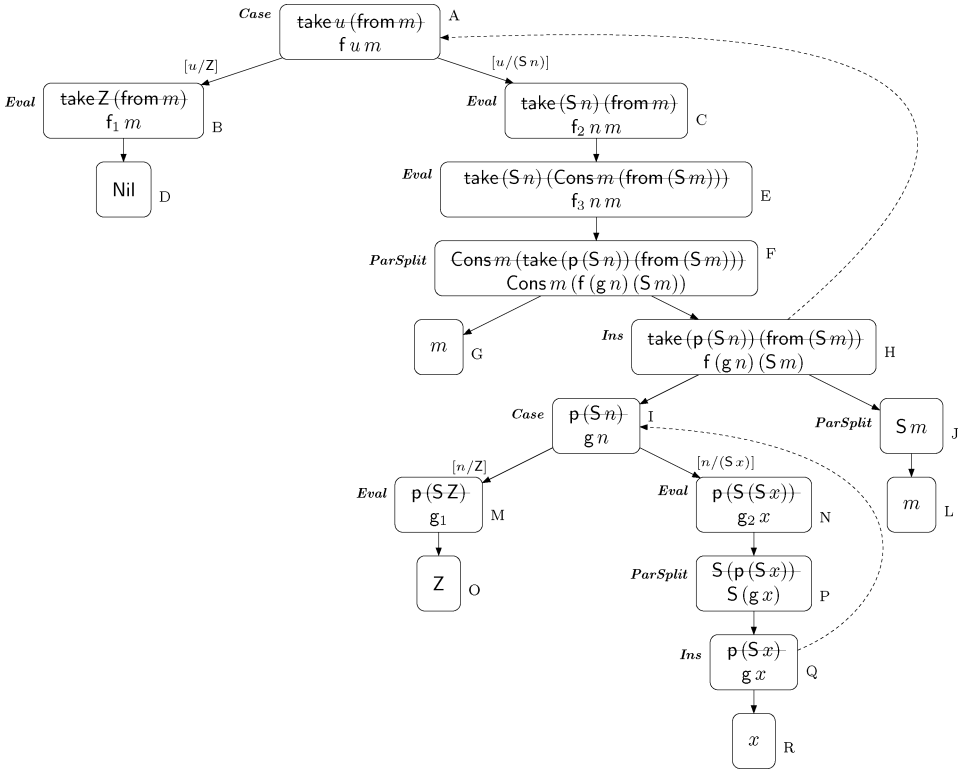

Fig. 8. Renamed termination graph for "take u (from m)".

This DP makes clear that the structure of the term (represented by f) does not change when going from the left- to the right-hand side and that only the two "arguments" of the term are modified. In the corresponding DP problem, up to now we had the following rules.

$$\mathsf{p}(\mathsf{S}\,\mathsf{Z}) \to \mathsf{Z} \qquad \text{(due to the rule path from I to O)} \tag{3}$$
$$\mathsf{p}(\mathsf{S}(\mathsf{S}\,x)) \to \mathsf{S}(\mathsf{p}(\mathsf{S}\,x)) \qquad \text{(due to the rule path from I to P)} \tag{4}$$

Since the start node I of the rule paths has been renamed from "$\mathsf{p}(\mathsf{S}\,n)$" to "$\mathsf{g}\,n$", we now have the following rules.

$$\mathsf{g}\,\mathsf{Z} \to \mathsf{Z} \qquad \text{(due to the rule path from I to O)} \tag{6}$$
$$\mathsf{g}(\mathsf{S}\,x) \to \mathsf{S}(\mathsf{g}\,x) \qquad \text{(due to the rule path from I to P)} \tag{7}$$

instead. Indeed, the right-hand side of (7) is $\mathbf{ev}(\mathbf{r}(\mathsf{S}(\mathsf{p}(\mathsf{S}\,x)))) = \mathbf{ev}(\mathsf{S}\ \mathbf{r}(\mathsf{p}(\mathsf{S}\,x))) = \mathbf{ev}(\mathsf{S}(\mathsf{g}\,\mathbf{r}(x))) = \mathbf{ev}(\mathsf{S}(\mathsf{g}\,x)) = \mathsf{S}(\mathsf{g}\,x)$. So instead of the problem $(\{(2)\}, \{(3), (4)\})$ we now obtain the DP problem $(\{(5)\}, \{(6), (7)\})$ from the renamed termination graph. The whole termination graph which results from Figure 6 by the renaming $\mathbf{r}$ is depicted in Figure 8. To summarize, we obtain the following definition of $\mathbf{r}$.

Definition 5.1 (**r**). Let G be a termination graph and let t be a node in G. Then

$$\mathbf{r}(t) = \begin{cases} t, & \text{if } t \text{ is a leaf} \\ (f_t\, x_1 \ldots x_n), & \text{if } t \text{ is an } \textbf{\textit{Eval}}, \textbf{\textit{Case}}, \textbf{\textit{VarExp}}, \\ & \text{or } \textbf{\textit{Ins}} \text{ node with instantiation edge to } \tilde{t} \in \mathbf{U}_G, \\ & \text{where } \mathcal{V}(t) = \{x_1, \ldots, x_n\} \\ (c\, \mathbf{r}(t_1) \ldots \mathbf{r}(t_n)), & \text{if } t \text{ is a } \textbf{\textit{ParSplit}} \text{ node with} \\ & \text{head}(t) = c \text{ and } \mathbf{ch}(t) = \{t_1, \ldots, t_n\} \\ (f\, \mathbf{r}(s_1) \ldots \mathbf{r}(s_m)), & \text{if } t \text{ is an } \textbf{\textit{Ins}} \text{ node with} \\ & t = \tilde{t}[x_1/s_1, \ldots, x_m/s_m], \\ & \mathbf{ch}(t) = \{s_1, \ldots, s_m, \tilde{t}\},\ \tilde{t} \notin \mathbf{U}_G, \\ & \text{and } \mathbf{r}(\tilde{t}) = (f\, x_1 \ldots x_m) \end{cases}$$

Now the main soundness theorem (Theorem 4.11) still holds: if all DP problems for the SCCs of the renamed termination graph are finite, then all terms in the original termination graph are H-terminating.

Note that when using this renaming, all terms in DP problems resulting from a termination graph correspond to *first-order* terms. The left-hand sides of DPs and rules are constructed by renaming the start node (which results in a term of the form $(f\, x_1 \ldots x_n)$ for a fresh function symbol f) and by applying all substitutions on a path to this term. The ranges of these substitutions contain only terms of the form $(c\, y_1 \ldots y_m)$ where c is a constructor with arity$(c) = m$. The right-hand sides of DPs and rules are of the form $\mathbf{ev}(t)$, where **ev** replaces all subterms of the form $(x\, t_1 \ldots t_k)$ by fresh variables. So the renaming solves the problems with higher-order terms containing "partial applications" of functions.

Example 5.2 (mapTree). To demonstrate this, we consider one of the most classical examples of higher-order functions, that is, the map function on lists which is then used to define a map function mapTree on variadic trees.

$$\begin{array}{l} \mathsf{map} :: (a \to b) \to (\mathsf{List}\ a) \to (\mathsf{List}\ b) \\ \mathsf{map}\ x\ \mathsf{Nil} = \mathsf{Nil} \\ \mathsf{map}\ x\ (\mathsf{Cons}\ y\ ys) = \mathsf{Cons}\ (x\ y)\ (\mathsf{map}\ x\ ys) \\ \\ \mathsf{data}\ \mathsf{Tree}\ a\ = \mathsf{Node}\ a\ (\mathsf{List}\ (\mathsf{Tree}\ a)) \\ \\ \mathsf{mapTree} :: (a \to b) \to (\mathsf{Tree}\ a) \to (\mathsf{Tree}\ b) \\ \mathsf{mapTree}\ g\ (\mathsf{Node}\ e\ ts) = \mathsf{Node}\ (g\ e)\ (\mathsf{map}\ (\mathsf{mapTree}\ g)\ ts) \end{array}$$

For the term "mapTree $g\ t$", we obtain the termination graph in Figure 9, where the terms of the original termination graph are crossed out and are replaced by the terms of the renamed termination graph. The termination graph has one SCC G consisting of the nodes A, B, C, E, H, J, K, and L. If one considers the original termination graph, then $\mathbf{dp}_G$ contains the following dependency pairs.

$$\begin{array}{rcl} \mathsf{mapTree}\ g\ (\mathsf{Node}\ e\ (\mathsf{Cons}\ y\ ys)) & \to & \mathsf{mapTree}\ g\ y \\ \mathsf{mapTree}\ g\ (\mathsf{Node}\ e\ (\mathsf{Cons}\ y\ ys)) & \to & \mathsf{map}\ (\mathsf{mapTree}\ g)\ ys \\ \mathsf{map}\ (\mathsf{mapTree}\ g)\ (\mathsf{Cons}\ y\ ys) & \to & \mathsf{mapTree}\ g\ y \\ \mathsf{map}\ (\mathsf{mapTree}\ g)\ (\mathsf{Cons}\ y\ ys) & \to & \mathsf{map}\ (\mathsf{mapTree}\ g)\ ys \end{array}$$

Here, sometimes mapTree is applied to two arguments, but the left- and right-hand sides of dependency pairs also contain the higher-order subterm "mapTree g" where the second argument of mapTree is missing. Thus, these dependency pairs do not correspond to ordinary first-order term rewriting, but they represent a challenging form of recursion: mapTree calls itself recursively via a partial application in an argument

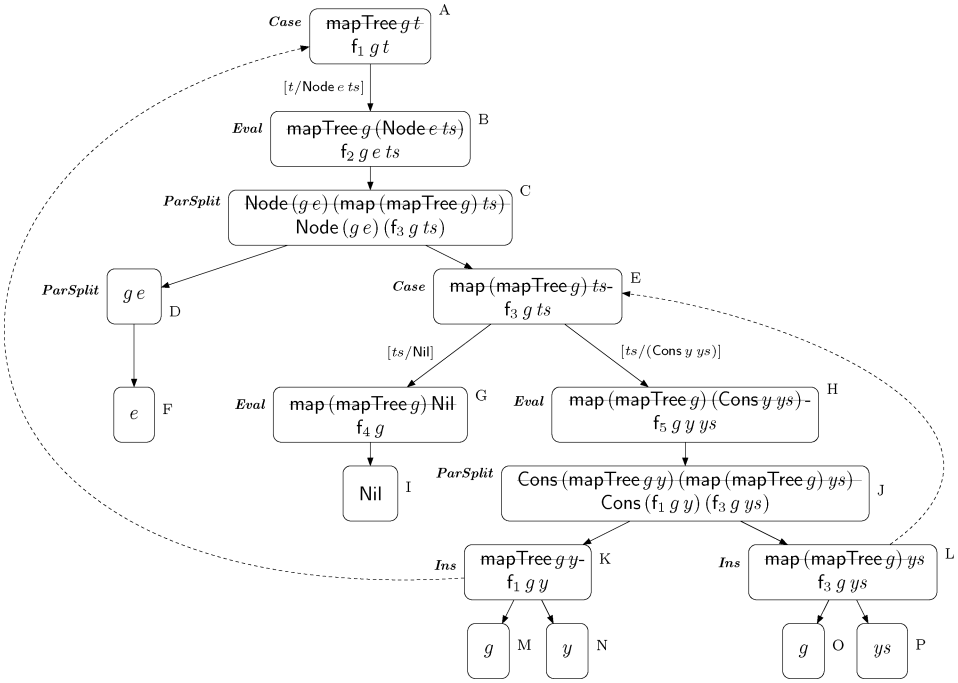

Fig. 9. Renamed termination graph for "mapTree $g\,t$".

to the higher-order function map. However, this recursion structure is substantially simplified when considering the renamed termination graph. Now one obtains the following dependency pairs without any partial applications.

$$\begin{aligned} \mathsf{f}_1\ g\ (\mathsf{Node}\ e\ (\mathsf{Cons}\ y\ ys)) &\to \mathsf{f}_1\ g\ y \\ \mathsf{f}_1\ g\ (\mathsf{Node}\ e\ (\mathsf{Cons}\ y\ ys)) &\to \mathsf{f}_3\ g\ ys \\ \mathsf{f}_3\ g\ (\mathsf{Cons}\ y\ ys) &\to \mathsf{f}_1\ g\ y \\ \mathsf{f}_3\ g\ (\mathsf{Cons}\ y\ ys) &\to \mathsf{f}_3\ g\ ys \end{aligned}$$

Note that this example cannot be handled by the approach of Panitz and Schmidt-Schauß [1997] since one obtains a termination graph with "crossings." The main reason why our approach can deal with arbitrary termination graphs is the difference in the identification of recursive calls. Panitz and Schmidt-Schauß [1997] build so-called *recursive pairs* and directly construct ordering constraints. A recursive pair is only built from the target and the source node of each instantiation edge, but not from the target of one instantiation edge and the source of *another different* instantiation edge. So for this example, Panitz and Schmidt-Schauß [1997] would only build the recursive pairs (A, K) and (E, L). In contrast, we also construct dependency pairs for the the DP paths from A to L and from E to K which are necessary to achieve a sound approach for termination graphs with crossings.

To translate the DPs and rules that result from the renamed termination graph into real first-order terms, we introduce new function symbols f_n for every function symbol f and every $n \in \mathbb{N}$, and apply the following translation $\mathbf{tr}$.

$$\begin{aligned} \mathbf{tr}(x) &= x && \text{for variables } x \\ \mathbf{tr}((f\,t_1 \ldots t_n)) &= f_n(\mathbf{tr}(t_1), \ldots, \mathbf{tr}(t_n)) && \text{for function symbols } f \end{aligned}$$

After this translation, there is no connection between a function symbol f applied to two arguments (which is translated to the symbol f_2) and an application where f is only applied to one argument (which is translated to the symbol f_1). Indeed, such remaining partial applications of functions no longer pose a problem. More precisely, if a function symbol f is only applied to m arguments with $m < \text{arity}(f)$, then by rewriting with DPs or rules, it will never be possible to "supply" the missing arguments which would be needed to evaluate f. The reason is that the DPs and rules no longer contain any subterms of the form $(x\, t_1 \ldots t_n)$. Moreover, it is no longer possible to have rules in the DP problems which add more and more arguments to a function symbol.

Example 5.3. To illustrate this, consider again the tma program from Example 3.2 and the corresponding termination graph in Figure 2. If we abbreviate the fresh function symbol "$f_{\mathsf{tma}\,n}$" by tma, then applying **rl** to the renamed node A yields the rule "$\mathsf{tma}\,(\mathsf{S}\,m) \to \mathsf{tma}\,m\,m$". Hence, after applying **tr**, we obtain the rule

$$\mathsf{tma}_1(\mathsf{S}_1(m)) \to \mathsf{tma}_2(m, m).$$

This rule does not correspond to the real behavior of tma, since the connection between tma_1 and tma_2 is lost. However, this problem only occurs in SCCs of the termination graph where an argument of a function symbol is "removed" via an instantiation edge to the node "$x\ y$". Thus, nodes like A are predecessors of "$x\ y$", that is, they are contained in $\mathbf{PU}_G$. According to the definition of **con**, one never creates any rules for such nodes, even if "$\mathsf{tma}\,n$" occurred in the recursive call of another function.

The renaming **r** has another important benefit. It ensures that one obtains DPs and rules that are nonoverlapping. As shown in Giesl et al. [2005a], for such DP problems it suffices to prove that they are finite with respect to *innermost* rewriting. This has important advantages since, in general, proving *innermost* termination is significantly easier than proving *full* termination automatically.

6. IMPROVED HANDLING OF TYPES

In this section, we improve our method by considering also built-in primitive data types (Section 6.1) and by handling type classes and overloading (Section 6.2).

6.1. Predefined Data Types

To treat the predefined data types Int, Integer, Char, Float, and Double of Haskell, we use a very straightforward approach and simply represent them using the following data declarations.

```
data Nats = Z | S Nats                  data Char = Char Nats
data Int = Pos Nats | Neg Nats          data Float = Float Int Int
data Integer = Integer Int              data Double = Double Int Int
```

So our termination analyzer internally converts every integer number into a term built with the constructors Pos and Neg which take arguments of type Nats. Hence, 1 is converted into "Pos (S Z)" and the number 0 has two possible representations "Pos Z" and "Neg Z".[26] Our representation of integers does not handle overflows, that is, it treats fixed-precision integers (Int) in the same way as arbitrary-precision integers

[26]Of course, this representation has disadvantages when considering "large" numbers. Very recently, there has been work on extending termination methods from term rewriting such that they can also deal with rewrite rules containing integers and other predefined data types [Falke and Kapur 2008; Fuhs et al. 2009]. Then such data types do not have to be represented by *terms* anymore. In future work, we will investigate the use of such extensions of term rewriting for termination analysis of Haskell. Nevertheless, our experiments in Section 7 show that even our straightforward encoding of data types already yields an extremely powerful method for automated termination analysis.

(Integer). In fact, the Haskell specification [Peyton Jones 2003] does not determine how to deal with overflow.[27]

Similarly we represent characters (Char) by the type Nats. Floating-point numbers are internally represented as fractions and we ignore their underflow and overflow, too. (Again, the Haskell specification [Peyton Jones 2003] does not determine the implementation of floating-point numbers.) Other built-in data types like Haskell's lists and tuples simply correspond to user-defined "ordinary" types with a different syntax. Thus, they are internally translated into the corresponding ordinary types.

For each type, we internally implement the required primitive functions by appropriate defining equations. For example, we use the following implementations for addition of integers and multiplication of floating-point numbers.

$$
\begin{array}{l}
\mathsf{primPlusInt} :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\
\mathsf{primPlusInt}\ \ (\mathsf{Pos}\ x)\ (\mathsf{Neg}\ y) = \mathsf{primMinusNats}\ x\ y \\
\mathsf{primPlusInt}\ \ (\mathsf{Neg}\ x)\ (\mathsf{Pos}\ y) = \mathsf{primMinusNats}\ y\ x \\
\mathsf{primPlusInt}\ \ (\mathsf{Pos}\ x)\ (\mathsf{Pos}\ y) = \mathsf{Pos}\ (\mathsf{primPlusNats}\ x\ y) \\
\mathsf{primPlusInt}\ \ (\mathsf{Neg}\ x)\ (\mathsf{Neg}\ y) = \mathsf{Neg}\ (\mathsf{primPlusNats}\ x\ y)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{primPlusNats} :: \mathsf{Nats} \to \mathsf{Nats} \to \mathsf{Nats} \\
\mathsf{primPlusNats}\ \mathsf{Z}\ \ \mathsf{Z}\ \ = \mathsf{Z} \\
\mathsf{primPlusNats}\ \mathsf{Z}\ \ (\mathsf{S}\ y) = \mathsf{S}\ y \\
\mathsf{primPlusNats}\ (\mathsf{S}\ x)\ \ \mathsf{Z}\ \ = \mathsf{S}\ x \\
\mathsf{primPlusNats}\ (\mathsf{S}\ x)\ \ (\mathsf{S}\ y) = \mathsf{S}\ (\mathsf{S}\ (\mathsf{primPlusNats}\ x\ y))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{primMinusNats} :: \mathsf{Nats} \to \mathsf{Nats} \to \mathsf{Int} \\
\mathsf{primMinusNats}\ \mathsf{Z}\ \ \mathsf{Z}\ \ = \mathsf{Pos}\ \mathsf{Z} \\
\mathsf{primMinusNats}\ \mathsf{Z}\ \ (\mathsf{S}\ y) = \mathsf{Neg}\ (\mathsf{S}\ y) \\
\mathsf{primMinusNats}\ (\mathsf{S}\ x)\ \ \mathsf{Z}\ \ = \mathsf{Pos}\ (\mathsf{S}\ x) \\
\mathsf{primMinusNats}\ (\mathsf{S}\ x)\ \ (\mathsf{S}\ y) = \mathsf{primMinusNats}\ x\ y
\end{array}
$$

$$
\begin{array}{l}
\mathsf{primMulFloat} :: \mathsf{Float} \to \mathsf{Float} \to \mathsf{Float} \\
\mathsf{primMulFloat}\ \ (\mathsf{Float}\ x1\ x2)\ (\mathsf{Float}\ y1\ y2) = \\
\qquad \mathsf{Float}\ (\mathsf{primMulInt}\ x1\ y1)\ (\mathsf{primMulInt}\ x2\ y2)
\end{array}
$$

The result of more complex primitive functions is "approximated" by free variables. For example, consider the primitive sine function primSinFloat for the type Float. The result of "primSinFloat t" for a term t is irrelevant for the termination behavior of most programs in practice. Thus, instead of implementing a complex algorithm to compute such results, we decided to return a fresh variable for each evaluation of primSinFloat. More precisely, we introduce a new primitive function terminator of type a, which is replaced by a new fresh variable when building dependency pairs or rules (i.e., **ev**(terminator) is a fresh variable). Then we use equations like the following to define complex functions like primSinFloat.

$$\mathsf{primSinFloat} = \mathsf{terminator}$$

Finally, we describe how we handle Haskell's built-in IO functions. Since devices could be nonresponding, IO functions that read from a device are considered to be potentially nonterminating. Furthermore, primitive functions handling exceptions are also considered to be possibly nonterminating, since exceptions bypass the usual evaluation

[27]Moreover, we treat the functions primMinInt and primMaxInt that return the smallest and largest number of type Int as being non-H-terminating. In this way, our method does not prove termination for programs depending on the handling of overflow errors via primMinInt and primMaxInt.

process of Haskell. All other IO functions (in particular those that only write to some device) are considered to be terminating.

6.2. Type Classes and Overloading

Up to now, we did not permit any use of type classes when analyzing the termination behavior of Haskell programs. In this section, we will extend our approach appropriately. Section 6.2.1 illustrates the concept of type classes in Haskell. Afterwards, Section 6.2.2 describes how to construct termination graphs for Haskell programs with type classes and how to obtain DP problems from these termination graphs. Finally, Section 6.2.3 shows how to adapt the renaming refinement from Section 5 to Haskell programs with type classes.

6.2.1 Type Classes in Haskell

Example 6.1 (Size *and* Max). Now we also permit programs with class and instance declarations like the following.

```
data Maybe b = Nothing | Just b

class Size b where
  size :: b → Nats

instance Size Bool where
  size x = S Z

instance Size Nats where
  size x = S x

instance Size b ⇒ Size (Maybe b) where
  size (Just x) = S (size x)
  size Nothing = S Z

class Max b where
  max :: b

instance Max Bool where
  max = True

instance Max b ⇒ Max (Maybe b) where
  max = Just max

headSize :: Size b ⇒ List b → Nats
headSize (Cons x xs) = size x
```

The first class declaration introduces the new *type class* Size with the *method* size. This means that if a type τ is an *instance* of the class Size, then there must be a function size of type $\tau \to$ Nats. The idea is that for any object t, "size t" should compute the number of data constructors in t. Similarly, the type class Max with the method max is introduced. Here, the idea is that for any type τ of class Max, max :: τ should return a "largest" object of type τ. To declare that a type is an instance of a class, an instance declaration is used. Here, the type Bool is declared to be an instance of the class Size. In the corresponding instance declaration, the function size of type Bool → Nats is defined. So this implementation of size is executed whenever size is applied to an argument of type Bool. For example, we have size True = S Z.

The next instance declaration that follows states that the type Nats is also an instance of the class Size and it implements the function size of type Nats → Nats. Thus, size is *overloaded* and when evaluating a term "size t", it depends on the type of the argument term t which implementation of size is executed. For example, we have size (S Z) = S (S Z) .

The function headSize returns the size of the first element in a list. It can be applied to any argument of type "List τ", provided that size is defined on arguments of type τ. In other words, it can be applied whenever τ is an instance of the type class Size. So the type of the function headSize is not "List $b \to$ Nats" (since the type variable b may not be instantiated by arbitrary types), but its type is "Size $b \Rightarrow$ List $b \to$ Nats". Here, "Size b" is a *class context* which ensures that the type variable b may only be instantiated by types from the class Size. In general, class contexts take the form

"$(C_1\ \tau_1, \ldots, C_n\ \tau_n)$", where the C_i are classes and the τ_i are types. Such a context means that the type variables in $\tau_1, \ldots, \tau_n$ may only be instantiated in such a way that the resulting instantiated types are instances of the classes $C_1, \ldots, C_n$.

Class contexts can also be used in instance declarations and class declarations. For example, "instance Size b $\Rightarrow$ Size (Maybe b)" means that the type "Maybe τ" is an instance of the class Size, provided that τ is also an instance of the class Size. Similar statements hold for the instance declarations of the class Max. For more details about type classes in Haskell we refer to Peyton Jones [2003].

In order to handle the overloading of functions properly, we now consider annotated terms. An *annotated term* is a term where every variable and every function symbol is annotated with a corresponding type. For example, $\frac{\mathsf{Nil}}{\mathsf{List}\,a}$ denotes the term Nil annotated by its type "List a". Similarly, $\frac{\mathsf{Nil}}{\mathsf{List\,Nats}}$ is also an annotated term, but this time, Nil is not annotated by its most general type. A more complex annotated term is $\frac{\mathsf{Cons}}{\mathsf{Nats}\to\mathsf{List\,Nats}\to\mathsf{List\,Nats}}\ \frac{\mathsf{Z}}{\mathsf{Nats}}\ \frac{\mathsf{Nil}}{\mathsf{List\,Nats}}$. If we additionally want to state the class context for an annotated term, then we add it in front. This results in annotated terms like $\mathsf{Size}\ b \Rightarrow \frac{\mathsf{headSize}}{\mathsf{List}\ b\to\mathsf{Nats}}$ since the term headSize has the type "List $b \to$ Nats" and the class context "Size b". Another example for an annotated term is $\mathsf{Size}\ a \Rightarrow \frac{\mathsf{size}}{a\to\mathsf{Nats}}\ \frac{x}{a}$. Of course, annotated terms have to be well typed. So for terms of the form $(\frac{t_1}{\tau_1}\ \frac{t_2}{\tau_2}\ \ldots\ \frac{t_n}{\tau_n})$ we must have $\tau_1 = \tau_2 \to \tau_3 \to \ldots \to \tau_n \to \tau_{n+1}$. Moreover, if a term contains several occurrences of the same variable (i.e., $\frac{x}{\tau_1}, \ldots, \frac{x}{\tau_n}$), then we must have $\tau_1 = \ldots = \tau_n$.

From now on, we only consider annotated terms (which we simply refer to as "terms"). While $\mathcal{V}(t)$ denotes the set of (object) variables of a term, we now also define $\mathcal{V}_\mathsf{T}(t)$ to be the set of type variables occurring in an (annotated) term t. So if t is $\mathsf{Size}\ a \Rightarrow \frac{\mathsf{size}}{a\to\mathsf{Nats}}\ \frac{x}{a}$, then we have $\mathcal{V}(t) = \{x\}$ and $\mathcal{V}_\mathsf{T}(t) = \{a\}$. If a function f is annotated with a type τ and there is no defining equation $\ell = r$ with $\mathrm{head}(\ell) = f$ where the type of f in ℓ is the same or more general[28] than the type τ, then we say that $\frac{f}{\tau}$ is *indetermined*. So $\frac{\mathsf{size}}{a\to\mathsf{Nats}}$ in a term with the class context "Size a" is indetermined, whereas $\frac{\mathsf{size}}{\mathsf{Bool}\to\mathsf{Nats}}$ is not indetermined. Note also that an indetermined function symbol has no arity (e.g., $\mathrm{arity}(\frac{\mathsf{size}}{a\to\mathsf{Nats}})$ is undefined). The reason is that an overloaded function may have different arities in the different instances of a class. In other words, f may have n arguments in the defining equations of one instance declaration, but m arguments in the defining equations of another instance declaration, where m can be different from n.

Similarly, concerning the operational semantics of Haskell, now an equation $\ell = r$ with $\mathrm{head}(\ell) = f$ can only be *feasible* for a term t with $\mathrm{head}(t) = f$ if the type of f in ℓ is the same or more general than the type of f in t.

Example 6.2 (*Feasible Equations for Overloaded Functions*). To illustrate this, consider the program of Example 6.1 and let t be the annotated term $\mathsf{Size}\ a \Rightarrow \frac{\mathsf{size}}{a\to\mathsf{Nats}}\ \frac{x}{a}$. The left-hand side of the first defining equation of size (in the instance declaration of Bool) is $\frac{\mathsf{size}}{\mathsf{Bool}\to\mathsf{Nats}}\ \frac{x}{\mathsf{Bool}}$. This equation is not feasible, since the type "Bool $\to$ Nats" of "size" in ℓ is not the same or more general than the type "Size $a \Rightarrow a \to$ Nats" of "size" in t. So in fact, no equation is feasible for t.[29]

[28] Of course, here one also has to take the respective class contexts into account.

[29] This would even hold if there were a (default) implementation of the function size in the declaration of the class Size. The reason is that such a default implementation is only copied into those instance declarations of Size where an implementation of the function size is missing. So even then, there would not be any defining equation where "size" has the type "Size $a \Rightarrow a \to$ Nats".

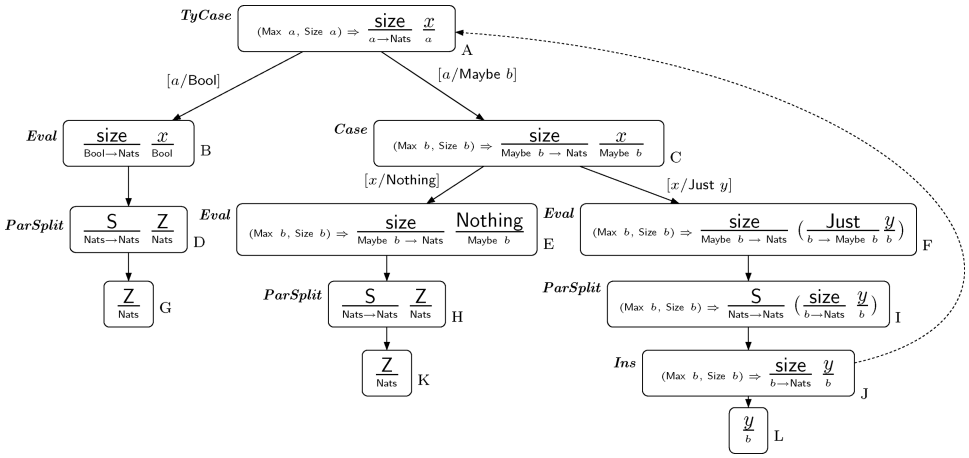

Fig. 10. Termination graph for "size x".

In an analogous way, one also has to take the types of annotated terms into account during pattern matching and during evaluations with "$\rightarrow_{\mathsf{H}}$".

6.2.2 Termination Graphs and DP Problems for Haskell with Type Classes

The next example illustrates the construction of termination graphs for programs using type classes. We consider the program of Example 6.1 and the following start term t.

$$(\mathsf{Max}\ a,\ \mathsf{Size}\ a)\ \Rightarrow \frac{\mathsf{size}}{a \to \mathsf{Nats}}\ \frac{x}{a} \tag{8}$$

In the beginning, the graph only consists of the node A marked with the start term, as in Figure 10. Now we try to evaluate t. But due to the overloading of the function size, this is not possible since it is unclear which instance of size should be used. An evaluation is only possible if we know how the type variable a is instantiated. More precisely, there is no feasible defining equation "size $\ldots = \ldots$" where the type of size is the same or more general than the type "$(\mathsf{Max}\ a,\ \mathsf{Size}\ a) \Rightarrow a \to \mathsf{Nats}$" of size in the term t. In other words, this occurrence of the function size is indetermined. Therefore, we introduce a new ***Type Case*** rule ("***TyCase***") which works on type variables in essentially the same way that the ***Case*** rule works on object variables. To apply the ***TyCase*** rule, we first check how the type variable a in size's type "$(\mathsf{Max}\ a,\ \mathsf{Size}\ a) \Rightarrow a \to \mathsf{Nats}$" could be instantiated in order to make a defining equation of size applicable. Here, we proceed in two steps. In the first step, we ignore all class contexts like $(\mathsf{Max}\ a,\ \mathsf{Size}\ a)$. Then size has the type "$a \to \mathsf{Nats}$" in the term t. Now we compute all possible substitutions of the type variable a that would allow defining equations of size to become applicable. In general, for any function f and any type τ, let **instances**(f, τ) be the set of the most general substitutions $\{\delta_1, \ldots, \delta_m\}$ such that there are defining equations for f in which f has the type $\delta_1(\tau), \ldots, \delta_m(\tau)$ (or a more general one). So for example, we obtain

$$\mathbf{instances}(\mathsf{size}, a \to \mathsf{Nats}) = \{[a/\mathsf{Bool}],\ [a/\mathsf{Nats}],\ [a/\mathsf{Maybe}\ b]\}. \tag{9}$$

The general definition of **instances** is as follows. Without loss of generality, here we assume that the type variables in the type τ of the function f are disjoint from the type variables occurring in class and instance declarations. (Otherwise, the variables in the

class and instance declarations are renamed.)

$$\textbf{instances}(f,\tau) = \{\delta \mid \begin{array}{l} f \text{ is a method declared in a declaration ``class } C\ b \ldots\text{''},\\ \text{in this class declaration, } f \text{ has the type } \tau',\\ \text{there is a declaration ``instance } cx \Rightarrow C\ \tau'' \ldots\text{''}, \text{ and}\\ \delta \text{ is the mgu of } \tau \text{ and } \tau'[b/\tau'']\} \end{array}$$

Here, "cx" denotes an arbitrary class context.

To illustrate the definition, consider again our example, where f is the function size and τ is the type $a \to \mathsf{Nats}$. The function size is a method of the class Size (i.e., C is Size) and in this class declaration, size has the type τ' which is $b \to \mathsf{Nats}$. Now we consider every instance of the class Size. For example, Bool is declared to be an instance of the class Size. So here, cx is an empty class context and τ'' is the type Bool. Hence, in the defining size equation of this instance declaration, the function size has the type $\tau'[b/\tau'']$ which is $\mathsf{Bool} \to \mathsf{Nats}$. Therefore, the first substitution δ in the set **instances**(size, $a \to \mathsf{Nats}$) is the mgu of τ and $\tau'[b/\tau'']$, that is, of $a \to \mathsf{Nats}$ and $\mathsf{Bool} \to \mathsf{Nats}$. Thus, $\delta = [a/\mathsf{Bool}]$. By considering the other two instances of the class Size, one also obtains the other substitutions in **instances**(size, $a \to \mathsf{Nats}$); see (9).

So if f occurs in a term t where f has the type τ, then **instances**(f,τ) computes all substitutions δ which would have to be applied to the type τ in order to resolve the overloading of f and to apply actual defining f-equations. However, the term t usually also has a class context and this class context could rule out some of the possible substitutions δ in **instances**(f,τ). For example, the term t from (8) has the class context "(Max a, Size a)". Hence, not all substitutions from **instances**(size, $a \to \mathsf{Nats}$) can really be used. The first substitution $[a/\mathsf{Bool}]$ is indeed possible, since Bool is an instance of both classes Max and Size. In other words, the instantiated class context (Max Bool, Size Bool) is valid and can be *reduced* to the empty class context. Hence, the first child node of t that is created by the ***TyCase*** rule is marked with the term "size x" where size has the type $\mathsf{Bool} \to \mathsf{Nats}$ and the class context is empty.

In contrast, the second substitution $[a/\mathsf{Nats}]$ from **instances**(size, $a \to \mathsf{Nats}$) is ruled out by the class context of t. When instantiating (Max a, Size a) with this substitution, the resulting class context (Max Nats, Size Nats) could be reduced to (Max Nats) since Nats is an instance of Size, but the remaining context is invalid since Nats is not an instance of Max. So the substitution $[a/\mathsf{Nats}]$ may not be used when applying the ***TyCase*** rule to t.

Finally, when applying the substitution $[a/\mathsf{Maybe}\ b]$, we obtain the instantiated class context (Max (Maybe b), Size (Maybe b)). According to the instance declarations this can be reduced, since "Size (Maybe b)" holds whenever "Size b" holds (and similar for Max). This results in the reduced class context (Max b, Size b). So the second child node of t that is created by the ***TyCase*** rule is marked with the term "size x" where size has the type "Maybe $b \to$ Nats" and it has the class context "(Max b, Size b)". In other words, the node A gets the child nodes B and C which are marked with

$$\frac{\mathsf{size}}{\mathsf{Bool} \to \mathsf{Nats}}\ \frac{x}{\mathsf{Bool}} \quad \text{and} \quad (\mathsf{Max}\ b,\ \mathsf{Size}\ b) \Rightarrow \frac{\mathsf{size}}{\mathsf{Maybe}\ b \to \mathsf{Nats}}\ \frac{x}{\mathsf{Maybe}\ b}.$$

To perform this reduction of class contexts, we define the following relation $\mapsto$ on class contexts. Whenever a class context contains the constraint "$C\ (d\ \tau_1 \ldots \tau_m)$" for a class C, a type constructor d, and types $\tau_1, \ldots, \tau_m$ and whenever there exists a declaration

$$\mathsf{instance}\ (C_1\ a_{i_1}, \ldots, C_n\ a_{i_n}) \Rightarrow C\ (d\ a_1 \ldots a_m) \ldots$$

with $n \geq 0$, then one can replace the constraint "$C\ (d\ \tau_1 \ldots \tau_m)$" by the new constraints $C_1\ \tau_{i_1}, \ldots, C_n\ \tau_{i_n}$. For example, consider the constraint "Size (Maybe b)". So here C is the class Size, the type constructor d is Maybe, and τ_1 is the type variable b. Due to the

instance declaration

$$\mathsf{instance\ Size}\ b\ \Rightarrow\ \mathsf{Size\ (Maybe}\ b)\ldots$$

(where C_1 corresponds to the class Size), we obtain the reduction

$$\mathsf{Size\ (Maybe}\ b) \mapsto \mathsf{Size}\ b.$$

It is clear that the relation $\mapsto$ is confluent and terminating (as cyclic dependencies of classes or "overlapping" instance declarations are forbidden in Haskell). For any class context cx, let **reduce**(cx) be the normal form of cx with respect to $\mapsto$. In other words, **reduce**(cx) is the result of applying $\mapsto$ repeatedly as long as possible to cx. For example, we have

—**reduce**$(\mathsf{(Max\ Bool,\ Size\ Bool)}) = (\,)$, since
(Max Bool, Size Bool) $\mapsto$ (Max Bool) $\mapsto$ ( )
—**reduce**$(\mathsf{(Max\ Nats,\ Size\ Nats)}) = \mathsf{(Max\ Nats)}$, since
(Max Nats, Size Nats) $\mapsto$ (Max Nats)
—**reduce**$(\mathsf{(Max\ (Maybe}\ b),\ \mathsf{Size\ (Maybe}\ b))) = \mathsf{(Max}\ b,\ \mathsf{Size}\ b)$, since
(Max (Maybe b), Size (Maybe b)) $\mapsto$ (Max (Maybe b), Size b) $\mapsto$ (Max b, Size b)

Recall that we considered an annotated term t containing a function f of type τ and our goal was to compute all specializations of τ such that the function f can be evaluated. To this end, we first ignored the class context cx of the term t. Then **instances**(f, τ) contains all most general substitutions δ of the type variables in τ such that defining f-equations would become applicable if f had the type $\delta(\tau)$. However, now we have to filter out those substitutions δ from **instances**(f, τ) which contradict the class context cx. To this end, we apply the type substitution δ also on the class context cx and then reduce the instantiated class context, that is, we build **reduce**$(\delta(cx))$. If **reduce**$(\delta(cx))$ is not invalid, then the ***TyCase*** rule generates a child node marked with t, but where the types of all variables and function symbols in t are refined by applying δ to them. Moreover, the former class context cx is replaced by **reduce**$(\delta(cx))$. A reduced class context is *invalid* whenever it contains a constraint like "$C\ (d\ \tau_1 \ldots \tau_m)$" where d is a type constructor. (Indeed, then $(d\ \tau_1 \ldots \tau_m)$ is not an instance of the class C, because otherwise "$C\ (d\ \tau_1 \ldots \tau_m)$" would have been reduced further with the relation $\mapsto$.) So the only constraints which may occur in a valid reduced class context are of the form "$C\ (a\ \tau_1 \ldots \tau_m)$" where a is a type variable. For instance, a valid reduced class context could contain a constraint like "Max a" (where a is a type variable and there are no argument types $\tau_1 \ldots \tau_m$, i.e., $m = 0$). It could also contain a constraint like "Max (a Bool)".

Given a set S of type substitutions and a class context cx, **filter**(S, cx) removes all substitutions δ from S where the instantiated class context $\delta(cx)$ would be invalid (i.e., where **reduce**$(\delta(cx))$ contains constraints like "$C\ (d\ \tau_1 \ldots \tau_m)$").

$$\textbf{filter}(S, cx)\ =\ \{\delta \in S\ \mid\ \textbf{reduce}(\delta(cx)) \text{ does not contain any constraint } \text{"}C\ (d\ \tau_1 \ldots \tau_m)\text{" where } d \text{ is a type constructor}\}$$

When computing the children of node A in Figure 10, S is **instances**$(\mathsf{size}, a \rightarrow \mathsf{Nats})$, that is, $S = \{[a/\mathsf{Bool}],\ [a/\mathsf{Nats}],\ [a/\mathsf{Maybe}\ b]\}$. Moreover, cx is $\mathsf{(Max}\ a,\ \mathsf{Size}\ a)$. The substitutions $[a/\mathsf{Bool}]$ and $[a/\mathsf{Maybe}\ b]$ lead to valid reduced class contexts, as **reduce**$(\mathsf{(Max\ Bool,\ Size\ Bool)}) = (\,)$ and **reduce**$(\mathsf{(Max\ (Maybe}\ b),\ \mathsf{Size\ (Maybe}\ b))) = \mathsf{(Max}\ b,\ \mathsf{Size}\ b)$. But the substitution $[a/\mathsf{Nats}]$ yields an invalid reduced class context, as **reduce**$(\mathsf{(Max\ Nats,\ Size\ Nats)}) = \mathsf{(Max\ Nats)}$. So here the reduced context has the form $(C\ d)$ for the class $C = \mathsf{Max}$ and the type constructor $d = \mathsf{Nats}$. Hence, we obtain

$$\textbf{filter}(S, cx) = \{[a/\mathsf{Bool}],\ [a/\mathsf{Maybe}\ b]\}.$$

For this reason, in the graph in Figure 10 the node A only has two children B and C corresponding to the substitutions $[a/\mathsf{Bool}]$ and $[a/\mathsf{Maybe}\ b]$, but no child corresponding to the substitution $[a/\mathsf{Nats}]$.

For the nodes B and C we can apply the other expansion rules as before. Once we have reached node J, we can again apply the ***Ins*** rule, because the term in node J is an instance of the term in node A. Here, one not only has to instantiate the *object* variable x in a suitable way, but one also has to instantiate the *type* variable a of the term in node A, that is, x and a are instantiated by y and b, respectively.

Thus, we now adapt the definition of *termination graphs* (Definition 3.1) which describes how to extend a graph with a leaf marked with t by adding new children $\mathbf{ch}(t)$. Now the nodes in the graph are marked with *annotated* terms. The expansion rules of Definition 3.1 remain the same, except for three changes. The first change consists of adding a new ***TyCase*** rule in addition to the previous five rules for extending graphs. In the following three expansion rules, we make the class contexts explicit and let t denote an annotated term without class context.

TyCase: $\mathbf{ch}(cx{\Rightarrow}t) = \{cx_1{\Rightarrow}\delta_1(t), \ldots, cx_m{\Rightarrow}\delta_m(t)\}$ *if* $\mathrm{head}(t)$ *is a defined function symbol and* $\mathrm{head}(t|_{\mathbf{e}(t)}) = \frac{g}{\tau}$ *for a defined function symbol* g *which is indetermined. Here, let* $\{\delta_1, \ldots, \delta_m\} = \mathbf{filter}(\mathbf{instances}(g, \tau), cx)$ *and let* $cx_i = \mathbf{reduce}(\delta_i(cx))$*, for* $1 \leq i \leq m$*. The edge from* $cx{\Rightarrow}t$ *to* $cx_i{\Rightarrow}\delta_i(t)$ *is marked with the substitution* δ_i*.*

The second change concerns the ***VarExp*** rule. This rule is used to add additional arguments whenever a function symbol is applied to less arguments than its arity requires. But for indetermined functions, the arity is not yet clear and thus, here one should apply the ***TyCase*** rule first. Thus, the ***VarExp*** rule is now restricted to function symbols that are not indetermined.

VarExp: $\mathbf{ch}(cx{\Rightarrow}t) = \{cx{\Rightarrow}t\ \frac{x}{\tau_{n+1}}\}$*, if* $t = (\frac{f}{\tau}\ t_1 \ldots t_n)$*,* τ *has the form* $\tau_1 \to \ldots \to \tau_n \to \tau_{n+1} \to \tau'$*,* $\frac{f}{\tau}$ *is a defined function symbol which is not indetermined,* $n < \mathrm{arity}(\frac{f}{\tau})$*,* x *is a fresh variable.*

The last change compared to Definition 3.1 is that in the ***Ins*** rule, we now also take the types and the class context into account.

Ins: $\mathbf{ch}(cx{\Rightarrow}t) = \{cx_1{\Rightarrow}s_1, \ldots, cx_m{\Rightarrow}s_m, \tilde{cx}{\Rightarrow}\tilde{t}\}$*, if* $t = (\frac{f}{\tau}\ t_1 \ldots t_n)$*,* t *is not an error term,*[30] $\frac{f}{\tau}$ *is a defined symbol which is not indetermined,* $n \geq \mathit{arity}(\frac{f}{\tau})$*,* $t = \tilde{t}\sigma$ *for some term* $\tilde{t}$*,* $\sigma = [x_1/s_1, \ldots, x_m/s_m,\ a_1/\tau_1, \ldots, a_\ell/\tau_\ell]$*, where* $\mathcal{V}(\tilde{t}) = \{x_1, \ldots, x_m\}$ *and* $\mathcal{V}_\mathsf{T}(\tilde{t}) = \{a_1, \ldots, a_\ell\}$*. Moreover, the instantiated class context* $\sigma(\tilde{cx})$ *must be the same or more general than the context* cx*, that is,* $\mathbf{reduce}(\sigma(\tilde{cx})) \subseteq \mathbf{reduce}(cx)$*. Here, either* $\tilde{t} = (x\ y)$ *for fresh variables* x *and* y *or* $\tilde{t}$ *is an* ***Eval****,* ***Case****, or* ***TyCase*** *node. If* $\tilde{t}$ *is a* ***Case*** *or* ***TyCase*** *node, then it must be guaranteed that all paths starting in* $\tilde{t}$ *reach an* ***Eval*** *node or a leaf with an error term after traversing only* ***Case*** *or* ***TyCase*** *nodes.*

Now Theorem 3.3 still holds, that is, for every (annotated) start term there exists a termination graph.

[30]Note that the definition of error terms from Section 2 now also has to be adapted, since a term where further evaluations are only blocked because of an indetermined overloaded function symbol are no error terms. So an (annotated) term s is an *error term* if there is no feasible equation for s, if $\mathrm{head}(s)$ and $\mathrm{head}(s|_{\mathbf{e}(s)}) = \frac{g}{\tau}$ are defined function symbols, and if $\frac{g}{\tau}$ is not indetermined. Moreover, a ***TyCase*** node without children in the termination graph (i.e., where $\mathbf{filter}(\mathbf{instances}(g, \tau), cx) = \varnothing$) also corresponds to an error term. In the following, we will not consider such nodes as ***TyCase*** nodes anymore, but simply as error terms. So when speaking of ***TyCase*** nodes, we now assume that they have at least one child.

The extraction of DP problems from the termination graph works as in Section 4. The only change is that we have to extend the functions **ev** and **con** from Definition 4.4 and 4.6 to ***TyCase*** nodes. Here, we simply treat ***TyCase*** nodes in the same way as ***Case*** nodes. So for any annotated term t in a ***TyCase*** node, we define

$$\mathbf{ev}(t) = t \qquad \text{and} \qquad \mathbf{con}(t) = \{t\}.$$

Similarly, ***TyCase*** nodes are also treated like ***Case*** nodes in the definition of *rule paths* (Definition 4.8).

So from the only SCC of the termination graph in Figure 10, we obtain the DP problem $(\{t_1 \to t_2\}, \varnothing)$, where

$$t_1 = (\mathsf{Max}\ (\mathsf{Maybe}\ b),\ \mathsf{Size}\ (\mathsf{Maybe}\ b)) \ \Rightarrow\ \frac{\mathsf{size}}{(\mathsf{Maybe}\ b) \to \mathsf{Nats}}\left(\frac{\mathsf{Just}}{b \to \mathsf{Maybe}\ b}\ \frac{y}{b}\right)$$

$$t_2 = (\mathsf{Max}\qquad b\ ,\ \mathsf{Size}\qquad b) \ \Rightarrow\ \frac{\mathsf{size}}{b \to \mathsf{Nats}}\qquad\qquad \frac{y}{b}$$

6.2.3 First-Order DP Problems for Type Classes

Now we show how to adapt the technique of Section 5 which renames the higher-order terms in the termination graph to first-order terms such that it can also deal with type classes. To rename a term t in a ***TyCase*** node, we again proceed as for ***Case*** nodes and replace the term by a new function symbol f_t. The arguments of f_t are the variables occurring in t. However, since we now deal with *annotated* terms, we can also benefit from the type information in these terms. So in addition to the object variables in $\mathcal{V}(t)$, the type variables in $\mathcal{V}_\mathsf{T}(t)$ are also given to f_t as additional arguments. So each ***Eval***, ***Case***, ***TyCase***, or ***VarExp*** node t is renamed to the term $\mathbf{r}(t) = (f_t\, x_1 \ldots x_n\, a_1 \ldots a_\ell)$, where f_t is a new function symbol for the term t, $\mathcal{V}(t) = \{x_1, \ldots, x_n\}$, and $\mathcal{V}_\mathsf{T}(t) = \{a_1, \ldots, a_\ell\}$. Note that while t is an annotated term, the renamed term $\mathbf{r}(t)$ is no longer annotated. Now the definition of $\mathbf{r}$ in Definition 5.1 can be refined as follows for any termination graph G and any node t in G.

$$\mathbf{r}(t) = \begin{cases} \bar{t}, & \text{if } t \text{ is a leaf, where } \bar{t} \text{ results from the annotated term } t \text{ by removing all annotations} \\ (f_t\, x_1 \ldots x_n\, a_1 \ldots a_\ell), & \text{if } t \text{ is an } \textbf{\textit{Eval}}, \textbf{\textit{Case}}, \textbf{\textit{TyCase}}, \textbf{\textit{VarExp}}, \text{ or } \textbf{\textit{Ins}} \text{ node with instantiation edge to } \tilde{t} \in \mathbf{U}_G, \text{ where } \mathcal{V}(t) = \{x_1, \ldots, x_n\} \text{ and } \mathcal{V}_\mathsf{T}(t) = \{a_1, \ldots, a_\ell\} \\ (c\ \mathbf{r}(t_1) \ldots \mathbf{r}(t_n)), & \text{if } t \text{ is a } \textbf{\textit{ParSplit}} \text{ node with } \mathsf{head}(t) = \frac{c}{\tau} \text{ and } \mathbf{ch}(t) = \{t_1, \ldots, t_n\} \\ (f\ \mathbf{r}(s_1) \ldots \mathbf{r}(s_m)\, \tau_1 \ldots \tau_\ell), & \text{if } t \text{ is an } \textbf{\textit{Ins}} \text{ node with } t = \tilde{t}[x_1/s_1, \ldots, x_m/s_m,\ a_1/\tau_1, \ldots, a_\ell/\tau_\ell], \mathbf{ch}(t) = \{s_1, \ldots, s_m, \tilde{t}\},\ \tilde{t} \notin \mathbf{U}_G, \text{ and } \mathbf{r}(\tilde{t}) = (f\, x_1 \ldots x_m\, a_1 \ldots a_\ell) \end{cases}$$

Again, our main soundness theorem (Theorem 4.11) still holds: if all DP problems for the SCCs of the renamed termination graph are finite, then all terms in the original termination graph are H-terminating.

Finally, we again use the translation **tr** from Section 5 to obtain first-order terms. Here, **tr** also transforms types into terms (i.e., now there are function symbols for type constructors like List, Maybe, etc.).[31] Note that due to the renaming $\mathbf{r}$ we then again

[31]Note that due to constructor classes like Monad, there can be type variables m whose type is a *class*. As an example consider the following program.

$$\mathsf{f} :: \mathsf{Monad}\, m \Rightarrow a \to m\, b \to c$$
$$\mathsf{f}\, x\, y = \mathsf{f}\, y\, y$$

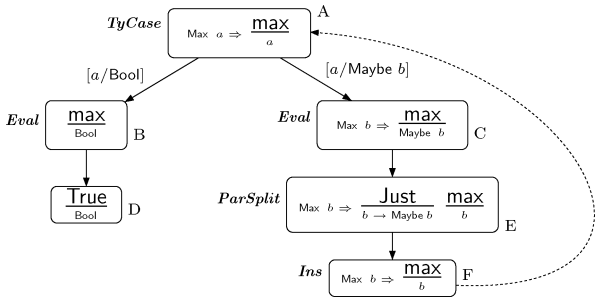

Fig. 11. Original termination graph for "max".

obtain nonoverlapping DPs and rules. For overloaded functions, this nonoverlappingness is only due to the fact that the renaming $\mathbf{r}$ transforms the types into additional arguments. As in Section 5, consequently it suffices to prove that the resulting DP problems are finite with respect to *innermost* rewriting.

Example 6.3. To illustrate the renaming for programs with type classes, we consider the program from Example 6.1 and the start term t in (8) again. When applying the renaming to the termination graph in Figure 10, instead of the dependency pair given at the end of Section 6.2.2, we now obtain the dependency pair

$$\mathsf{f}(\,\mathsf{Just}(y),\ \mathsf{Maybe}(b)\,)\ \to\ \mathsf{f}(y, b).$$

(Here, we wrote "Maybe" instead of "Maybe_1", etc., to ease readability.) Now it is straightforward to prove automatically that the corresponding DP problem is finite.

The following example shows why it is crucial to add the type variables from $\mathcal{V}_{\mathsf{T}}(t)$ when translating terms with the function $\mathbf{r}$. Indeed, in this example this addition is required in order to prove termination, since only the type of the term is decreasing in the recursive call. This is another important advantage of the renaming, because in this way, type information can be taken into account although the rewrite rules in DP problems only contain untyped terms.

Example 6.4. Consider the program of Example 6.1 and the start term t:

$$\mathsf{Max}\,a\ \Rightarrow\frac{\mathsf{max}}{a}$$

Its (original nonrenamed) termination graph is shown in Figure 11. If one ignores the types, then the DP constructed from the only DP path A, C, E, F is "$\mathsf{max} \to \mathsf{max}$". Showing termination for this dependency pair must fail, due to the existence of the infinite chain "$\mathsf{max} \to_{\mathcal{P}} \mathsf{max} \to_{\mathcal{P}} \ldots$". However, in reality, the function max terminates. For example, the term $\frac{\mathsf{max}}{\mathsf{Maybe\,Bool}}$ has the following reduction.

$$\frac{\mathsf{max}}{\mathsf{Maybe\,Bool}} \to_{\mathsf{H}} \frac{\mathsf{Just}}{\mathsf{Bool}\ \to\ \mathsf{Maybe\,Bool}}\ \frac{\mathsf{max}}{\mathsf{Bool}} \to_{\mathsf{H}} \frac{\mathsf{Just}}{\mathsf{Bool}\ \to\ \mathsf{Maybe\,Bool}}\ \frac{\mathsf{True}}{\mathsf{Bool}}$$

The type of the function symbol max decreases whenever the recursive equation "$\mathsf{max} = \mathsf{Just}\,\mathsf{max}$" is applied. In other words, the start term t is terminating since

To avoid "higher-order" terms resulting from types, we represent applications of type variables to other types in applicative notation using the special symbol ap. So from the renamed termination graph of this example, we obtain a dependency pair of the form $\mathsf{h}(x, y, m, a, b, c) \to \mathsf{h}(y, y, m, \mathsf{ap}(m, b), b, c)$. As long as such an applicative notation is only used for the representation of types, this does not yield problems for automated termination proofs, because then ap is not a defined symbol (i.e., there is no rule or dependency pair with ap at the root position of the left-hand side).

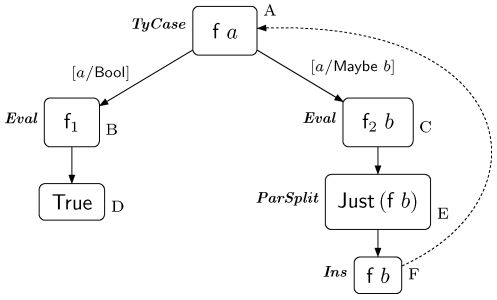

Fig. 12. Renamed termination graph for "max".

the type decreases in the DP path A, C, E, F. This becomes clear when considering the corresponding renamed termination graph in Figure 12. From this renamed graph, we obtain the DP problem with the following dependency pair.

$$\mathsf{f}(\mathsf{Maybe}(b)) \;\rightarrow\; \mathsf{f}(b)$$

Since we have encoded types as extra arguments, finiteness of this DP problem is easy to show automatically.

7. EXPERIMENTS

We implemented our technique in the automated termination prover AProVE [Giesl et al. 2006b]. The implementation accepts the full Haskell 98 language defined in Peyton Jones [2003]. However, we do not handle extensions like quantified types that are available in several implementations of Haskell. Our goal was to make all recent advances in automated termination analysis of term rewriting applicable for termination analysis of Haskell. The power of termination tools is compared at the annual International Termination Competition and AProVE has been the most powerful system for automated termination analysis of term rewriting in all these competitions so far. Therefore, to solve the DP problems resulting from Haskell programs, in our implementation we apply the termination techniques and the strategy used by AProVE in the most recent competition 2009.

A main application of automated termination analysis in practice is to prove termination of as many auxiliary functions as possible. In contrast, the "main" function in a program is often nonterminating. For example, this main function may read input from the user repeatedly and it only terminates if the user explicitly "quits" the program. But it is important that most of the auxiliary functions used by this main function are terminating. Therefore, to assess the power of our method, we evaluated our implementation with the standard libraries FiniteMap, List, Monad, Prelude, and Queue from the distribution of the popular Haskell interpreter Hugs [Jones and Peterson 1999]. These are indeed typical auxiliary functions that are used in many real Haskell applications. As described in Section 6.1, we supplied implementations for primitive functions like primPlusInt, primMulFloat, ... that are declared but not implemented in the Hugs prelude. Moreover, we also wanted to evaluate the effects of the renaming improvement in Section 5. Therefore, in addition to the full version of AProVE that contains all contributions of the current article, we also tested a version (called $\mathsf{AProVE}_{\mathrm{PLAIN}}$) where we ignored the results of Section 5. Since some of AProVE's termination techniques are only applicable for proving *innermost* termination, they could not be used in the variant $\mathsf{AProVE}_{\mathrm{PLAIN}}$.

We ran AProVE and $\mathsf{AProVE}_{\mathrm{PLAIN}}$ on a test corpus consisting of 1272 examples. Here, we tried to prove H-termination of every exported function in the preceding Hugs

libraries. For each such function, we first attempted a termination proof for its "most general" version. Moreover, whenever the most general type of a function had a class context "$C\,a$", then we also tried termination proofs for all versions of the function where the type variable a was instantiated with an instance of the type class C. The reason is that by considering all instances separately, we get a finer analysis for those cases where the most general form of the function does not H-terminate (or cannot be shown H-terminating), whereas the function can still be proved H-terminating for certain instances.

The following table summarizes the results of our experimental evaluation. Each termination proof was performed with a time limit of 5 minutes. Here, **YES** indicates the number of functions where proving H-termination succeeded. **MAYBE** gives the number of examples where the H-termination proof failed within 5 minutes and **TIMEOUT** shows the number of functions where no proof could be found within the time limit. It should be mentioned that it is impossible to prove H-termination for all 1272 examples, since at least 49 of the functions are actually not H-terminating.

Version	YES	MAYBE	TIMEOUT
AProVE$_{\text{PLAIN}}$	717 (56.37 %)	104 (8.12 %)	451 (35.46 %)
AProVE	999 (78.54 %)	68 (5.35 %)	205 (16.12 %)

The table shows that our approach is indeed very powerful for analyzing the termination behavior of Haskell functions, in particular, of auxiliary functions used in Haskell programs. Discounting the functions known to be not H-terminating, AProVE can prove H-termination for 81.68 % of all functions in these libraries. The examples where AProVE fails in proving H-termination are mostly functions on rational numbers, IO functions, and several of those functions that require a special evaluation strategy (due to the use of the "seq" operator). A **MAYBE** typically results from cases where the resulting DP problems are not finite. (As illustrated in Example 4.12, due to the incompleteness of our approach, this does not necessarily imply non-H-termination of the original Haskell function). The results are similar to the results in the Termination Competitions, where a shorter time limit of 1 minute was used. Indeed, 950 of the 999 successful termination proofs already succeed within 1 minute (and many of these proofs need only a few seconds). So in fact, the time limit has no big impact on the success rate, as long as one permits runtimes of at least 1 minute.

Moreover, the table also shows that AProVE is substantially more powerful than AProVE$_{\text{PLAIN}}$. In other words, the improvements of Section 5 lead to a significant increase in power. Here, it is also interesting to investigate in which modules these gains in power were achieved. The next table shows the numbers broken down according to the modules.

Module	Version	YES	MAYBE	TIMEOUT
FiniteMap	AProVE$_{\text{PLAIN}}$	116 (36.13 %)	16 (4.98 %)	189 (58.87 %)
	AProVE	258 (80.37 %)	0 (0.00 %)	63 (19.62 %)
List	AProVE$_{\text{PLAIN}}$	64 (36.78 %)	29 (16.66 %)	81 (46.55 %)
	AProVE	168 (96.56 %)	4 (2.29 %)	2 (1.14 %)
Monad	AProVE$_{\text{PLAIN}}$	68 (85.00 %)	11 (13.75 %)	1 (1.25 %)
	AProVE	69 (86.25 %)	11 (13.75 %)	0 (0.00 %)
Prelude	AProVE$_{\text{PLAIN}}$	464 (67.05 %)	48 (6.93 %)	180 (26.01 %)
	AProVE	499 (72.10 %)	53 (7.65 %)	140 (20.23 %)
Queue	AProVE$_{\text{PLAIN}}$	5 (100.00 %)	0 (0.00 %)	0 (0.00 %)
	AProVE	5 (100.00 %)	0 (0.00 %)	0 (0.00 %)

So in particular for the modules FiniteMap and List, the number of functions where H-termination can be proved is more than doubled by the contributions of Section 5. This is due to the high number of higher-order functions in these libraries. For these functions, the better handling of higher-order terms by the renaming technique of Section 5 is very advantageous. Furthermore, the gain is more than 5 % for the Prelude. This is also due to the better handling of higher-order functions, but also due to the fact that the renaming technique results in nonoverlapping DPs and rules. Therefore, it suffices to prove only *innermost* termination. The techniques to prove finiteness of DP problems for innermost rewriting are considerably more powerful than the corresponding techniques for full rewriting.[32]

8. CONCLUSION

We presented a technique for automated termination analysis of Haskell which works in three steps: First, it generates a termination graph for the given start term. Then it extracts DP problems from the termination graph. Finally, one uses existing methods from term rewriting to prove finiteness of these DP problems.

A preliminary version of parts of this article was already presented in Giesl et al. [2006a]. However, the present article extends Giesl et al. [2006a] substantially.

(a) In Giesl et al. [2006a], Haskell terms with higher-order functions were converted into applicative first-order terms for termination analysis. In contrast, in the current article we presented a technique to rename the terms in termination graphs which avoids the problems of applicative terms; see Section 5. Moreover, in this way we can improve the handling of types and convert Haskell termination problems to termination problems for *innermost* rewriting.
(b) In Giesl et al. [2006a], we only considered a restricted version of Haskell without type classes. In contrast, in Section 6 of the current article, we extended our approach to deal with type classes and overloading. Moreover, in Giesl et al. [2006a] we did not handle any predefined data types of Haskell, whereas we now included such data types.
(c) In contrast to Giesl et al. [2006a], the online appendix of the present article accessible in the ACM Digital Library, contains the full proofs for the theorems. We have also included a detailed description of our experimental evaluation in Section 7.

Moreover, compared to Giesl et al. [2006a], several details were added and improved throughout the article.

In this article, we have shown for the first time that termination techniques from term rewriting are indeed suitable for termination analysis of an existing functional programming language. Term rewriting techniques are also suitable for termination analysis of other kinds of programming languages. In Schneider-Kamp et al. [2009, 2010], we recently adapted the dependency pair method in order to prove termination of Prolog programs and in Otto et al. [2010] and Brockschmidt et al. [2010], it was adapted to prove termination of Java Bytecode. While there have been impressive recent results on automated termination analysis of imperative languages (e.g., Albert et al. [2008], Berdine et al. [2006], Bradley et al. [2005], Chawdhary et al. [2008], Colon and Sipma [2002], Cook et al. [2006], Podelski and Rybalchenko [2004a, 2004b], Spoto et al. [2010], and Tiwari [2004]) the combination of these results with approaches based

[32]To access the implementation via a Web interface, for further information on our experiments, and for further details of our method, we refer to:

`http://aprove.informatik.rwth-aachen.de/eval/Haskell/`

on rewriting yields substantial improvements, in particular for programs operating on user-defined data types.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

We thank the reviewers for many helpful suggestions and remarks.

REFERENCES

ABEL, A. 2004. Termination checking with types. *RAIRO Theor. Inform. Appl. 38*, 4, 277–319.

ALARCÓN, B., GUTIÉRREZ, R., AND LUCAS, S. 2006. Context-Sensitive dependency pairs. In *Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'06).* Lecture Notes in Computer Science, vol. 4337, Springer, 298–309.

ALARCÓN, B., EMMES, F., FUHS, C., GIESL, J., GUTIÉRREZ, R., LUCAS, S., SCHNEIDER-KAMP, P., AND THIEMANN, R. 2008. Improving context-sensitive dependency pairs. In *Proceedings of the International Conference on Logic Programming, Artificial Intelligence and Reasoning (LPAR'08).* Lecture Notes in Artificial Intelligence, Springer, vol. 5330, 636–651.

ALBERT, E., ARENAS, P., CODISH, M., GENAIM, S., PUEBLA, G., AND ZANARDINI, D. 2008. Termination analysis of Java bytecode. In *Proceedings of the International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'08).* Lecture Notes in Computer Science, vol. 5051, Springer, 2–18.

ARTS, T. AND GIESL, J. 2000. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci. 236*, 133–178.

BARTHE, G., FRADE, M. J., GIMÉNEZ, E., PINTO, L., AND UUSTALU, T. 2000. Type-Based termination of recursive definitions. *Math. Struct. Comput. Sci. 14*, 1, 1–45.

BERDINE, J., COOK, B., DISTEFANO, D., AND O'HEARN, P. 2006. Automatic termination proofs for programs with shape-shifting heaps. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'06).* Lecture Notes in Computer Science, vol. 4144, Springer, 386–400.

BLANQUI, F. 2004. A type-based termination criterion for dependently-typed higher-order rewrite systems. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA'04).* Lecture Notes in Computer Science, vol. 3091, Springer, 24–39.

BRADLEY, A. R., MANNA, Z., AND SIPMA, H. B. 2005. Termination of polynomial programs. In *Proceedings of the Conference on Verification, Model-Checking and Abstract Interpretation (VMCAI'05).* Lecture Notes in Computer Science, vol. 3385, Springer, 113–129.

BROCKSCHMIDT, M., OTTO, C., VON ESSEN, C., AND GIESL, J. 2010. Termination graphs for Java bytecode. In *Verification, Induction, Termination Analysis*. Lecture Notes in Computer Science, vol. 6463, Springer, 17–37.

CHAWDHARY, A., COOK, B., GULWANI, S., SAGIV, M., AND YANG, H. 2008. Ranking abstractions. In *Proceedings of the European Symposium on Programming (ESOP'08).* Lecture Notes in Computer Science, vol. 4960, Springer, 148–162.

COLÓN, M. AND SIPMA, H. 2002. Practical methods for proving program termination. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'02).* Lecture Notes in Computer Science, vol. 2034, Springer, 442–454.

COOK, B., PODELSKI, A., AND RYBALCHENKO, A. 2006. Termination proofs for systems code. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'06).* ACM Press, 415–426.

DERSHOWITZ, N. 1987. Termination of rewriting. *J. Symb. Comput. 3*, 1-2, 69–116.

ENDRULLIS, J., WALDMANN, J., AND ZANTEMA, H. 2008. Matrix interpretations for proving termination of term rewriting. *J. Autom. Reason. 40*, 2-3, 195–220.

ENDRULLIS, J. AND HENDRIKS, D. 2009. From outermost to context-sensitive rewriting. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA'09).* Lecture Notes in Computer Science, vol. 5595, Springer, 305–319.

FALKE, S. AND KAPUR, D. 2008. Dependency pairs for rewriting with built-in numbers and semantic data structures. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA'08).* Lecture Notes in Computer Science, vol. 5117, Springer, 94–109.

FUHS, C., GIESL, J., PLÜCKER, M., SCHNEIDER-KAMP, P., AND FALKE, S. 2009. Proving termination of integer term rewriting. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA'09)*. Lecture Notes in Computer Science, vol. 5595, Springer, 32–47.

GESER, A., HOFBAUER, D., AND WALDMANN, J. 2004. Match-Bounded string rewriting systems. *Applic. Algebr. Engin., Comm. Comput. 15*, 3, 149–171.

GIESL, J. 1995. Termination analysis for functional programs using term orderings. In *Proceedings of the International Static Analysis Symposium (SAS'95)*. Lecture Notes in Computer Science, vol. 983, Springer, 154–171.

GIESL, J. AND MIDDELDORP, A. 2004. Transformation techniques for context-sensitive rewrite systems. *J. Funct. Program. 14*, 4, 379–427.

GIESL, J., THIEMANN, R., AND SCHNEIDER-KAMP, P. 2005a. The dependency pair framework: Combining techniques for automated termination proofs. In *Proceedings of the International Conference on Logic Programming, Artificial Intelligence and Reasoning (LPAR'04)*. Lecture Notes in Artificial Intelligence, vol. 3452, 301–331.

GIESL, J., THIEMANN, R., AND SCHNEIDER-KAMP, P. 2005b. Proving and disproving termination of higher-order functions. In *Proceedings of the International Workshop on Frontiers of Combining Systems (FroCoS'05)*. Lecture Notes in Artificial Intelligence, vol. 3717, 216–231.

GIESL, J., SWIDERSKI, S., SCHNEIDER-KAMP, P., AND THIEMANN, R. 2006a. Automated termination analysis for Haskell: From term rewriting to programming languages. In *Proceedings of the International Conference on Rewriting Techniques and Applidations (RTA'06)*. Lecture Notes in Computer Science, Springer, vol. 4098, Springer, 297–312.

GIESL, J., SCHNEIDER-KAMP, P., AND THIEMANN, R. 2006b. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proceedings of the International Conference on Automated Reasoning (IJCAR'06)*. Lecture Notes in Artificial Intelligence, Springer, vol. 4130, 281–286.

GIESL, J., THIEMANN, R., SCHNEIDER-KAMP, P., AND FALKE, S. 2006c. Mechanizing and improving dependency pairs. *J. Autom. Reason. 37*, 3, 155–203.

GLENSTRUP, A. J. AND JONES, N. D. 2005. Termination analysis and specialization-point insertion in offline partial evaluation. *ACM Trans. Program. Lang. Syst. 27*, 6, 1147–1215.

GNAEDIG, I. AND KIRCHNER, H. 2008. Termination of rewriting under strategies. *ACM Trans. Comput. Logic 10*, 3.

HANUS, M. 2007. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP'07)*. Lecture Notes in Computer Science, vol. 4670, Springer, 45–75.

HIROKAWA, N. AND MIDDELDORP, A. 2005. Automating the dependency pair method. *Inform. Comput. 199*, 1-2, 172–199.

HIROKAWA, N. AND MIDDELDORP, A. 2007. Tyrolean termination tool: Techniques and features. *Inform. Comput. 205,* 4, 474–511.

HIROKAWA, N., MIDDELDORP, A., AND ZANKL, H. 2008. Uncurrying for termination. In *Proceedings of the International Conference on Logic Programming, Artificial Intelligence and Reasoning (LPAR'08)*. Lecture Notes in Artificial Intelligence, vol. 5330, 667–681.

JONES, M. AND PETERSON, J. 1999. The Hugs 98 user manual. http://www.haskell.org/hugs/

KENNAWAY, R., KLOP, J. W., SLEEP, M. R., AND DE VRIES, F. J. 1996. Comparing curried and uncurried rewriting. *J. Symb. Comput. 21*, 1, 15–39.

KOBAYASHI, N. 2009. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'09)*. ACM Press, 416–428.

KORP, M., STERNAGEL, C., ZANKL, H., AND MIDDELDORP, A. 2009. Tyrolean termination tool 2. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA'09)*. Lecture Notes in Computer Science, vol. 5595, Springer, 295–304.

LEE, C. S., JONES, N. D., AND BEN-AMRAM, A. M. 2001. The size-change principle for program termination. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'01)*. ACM Press, 81–92.

LUCAS, S. 1998. Context-Sensitive computations in functional and functional logic programs. *J. Funct. Logic Program.1*, 1–61.

MANOLIOS, P. AND VROON, D. 2006. Termination analysis with calling context graphs. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'06)*. Lecture Notes in Computer Science, vol. 4144, Springer, 401–414.

ONG, C.-H. L. 2006. On model-checking trees generated by higher-order recursion schemes. In *Proceedings of the Annual IEEE Symposium on Logic in Computer Science (LICS'06)*. IEEE, 81–90.

OTTO, C., BROCKSCHMIDT, M., VON ESSEN, C., AND GIESL, J. 2010. Automated termination analysis of Java bytecode by term rewriting. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA'10)*. LIPIcs, vol. 6, 259–276.

PANITZ, S. E. AND SCHMIDT-SCHAUß, M. 1997. TEA: Automatically proving termination of programs in a non-strict higher-order functional language. In *Proceedings of the International Static Analysis Symposium (SAS'97)*. Lecture Notes in Computer Science, vol. 1302, Springer, 345–360.

PANITZ, S. E. 1997. Generierung statischer Programminformation zur Kompilierung verzögert ausgewerteter funktionaler Programmiersprachen. Ph.D. thesis, University of Frankfurt.

PEYTON JONES, S. 2003. *Haskell 98 Languages and Libraries: The Revised Report*. Cambridge University Press.

PODELSKI, A. AND RYBALCHENKO, A. 2004a. A complete method for the synthesis of linear ranking functions. In *Proceedings of the Conference on Verification, Model-Checking and Abstract Interpretation (VMCAI'04)*. Lecture Notes in Computer Science, vol. 2937, Springer, 239–251.

PODELSKI, A. AND RYBALCHENKO, A. 2004b. Transition invariants. In *Proceedings of the Annual IEEE Symposium on Logic in Computer Science (LICS'04)*. IEEE, 32–41.

RAFFELSIEPER, M. 2007. Improving efficiency and power of automated termination analysis for Haskell. Diploma thesis, RWTH Aachen. http://aprove.informatik.rwth-aachen.de/eval/Haskell/DA/Raffelsieper_DA.pdf

RAFFELSIEPER, M. AND ZANTEMA, H. 2009. A transformational approach to prove outermost termination automatically. In *Proceedings of the International Workshop on Reduction Strategies in Rewriting and Programming (WRS'08)*. ENTCS, vol. 237, 3–21.

RONDON, P. M., KAWAGUCHI, M., AND JHALA, R. 2008. Liquid types. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'08)*. ACM Press, 159–169.

SCHNEIDER-KAMP, P., GIESL, J., SEREBRENIK, A., AND THIEMANN, R. 2009. Automated termination proofs for logic programs by term rewriting. *ACM Trans. Comput.Logic 11*, 1.

SCHNEIDER-KAMP, P., GIESL, J., STRÖDER, T., SEREBRENIK, A., AND THIEMANN, R. 2010. Automated termination analysis for logic programs with cut. *Theory Pract. Logic Program. 10*, 4-6, 365–381.

SERENI, D. AND JONES, N. D. 2005. Termination analysis of higher-order functional programs. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS'05)*. Lecture Notes in Computer Science, vol. 3780, Springer, 281–297.

SERENI, D. 2007. Termination analysis and call graph construction for higher-order functional programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*. ACM Press, 71–84.

SØRENSEN, M. H. AND GLÜCK, R. 1995. An algorithm of generalization in positive supercompilation. In *Proceedings of the International Logic Programming Symposium (ILPS'95)*. MIT Press, 465–479.

SPOTO, F., MESNARD, F., AND PAYET, É. 2010. A termination analyser for Java Bytecode based on path length. *ACM Trans. Program. Lang. Syst. 32*, 3.

SWIDERSKI, S. 2005. Terminierungsanalyse von Haskellprogrammen. Diploma thesis, RWTH Aachen. http://aprove.informatik.rwth-aachen.de/eval/Haskell/DA/Swiderski_DA.pdf

TELFORD, A. AND TURNER, D. 2000. Ensuring termination in ESFP. *J. Universal Comput. Sci. 6*, 4, 474–488.

THIEMANN, R. AND GIESL, J. 2005. The size-change principle and dependency pairs for termination of term rewriting. *Applic. Algebr. Engin., Comm. Comput. 16*, 4, 229–270.

THIEMANN, R. 2009. From outermost termination to innermost termination. In *Proceedings of the Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'09)*. Lecture Notes in Computer Science, vol. 5404, Springer, 533–545.

TIWARI, A. 2004. Termination of linear programs. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'04)*. Lecture Notes in Artificial Intelligence, Springer, vol. 3097, 70–82.

WALTHER, C. 1994. On proving the termination of algorithms by machine. *Artif. Intell. 71*, 1, 101–157.

XI, H. 2002. Dependent types for program termination verification. *Artif. Intell. 15*, 1, 91–131.

XU, D. N., PEYTON JONES, S., AND CLAESSEN, K. 2009. Static contract checking for Haskell. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '09)*. ACM Press, 41–52.

ZANTEMA, H. 2003. Termination. In *Term Rewriting Systems*, Terese, Ed. Cambridge University Press, Chapter 6, 181–259.

Received July 2009; revised March 2010; accepted May 2010