Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages*

Jürgen Giesl, Stephan Swiderski, Peter Schneider–Kamp, and René Thiemann

LuFG Informatik 2, RWTH Aachen, Ahornstr. 55, 52074 Aachen, Germany,
{giesl,swiderski,psk,thiemann}@informatik.rwth-aachen.de

Abstract. 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 in order to prove termination of programs in the functional language Haskell. 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.

1 Introduction

We show that termination techniques for term rewrite systems (TRSs) are also useful for termination analysis of programming languages like Haskell. Of course, any program can be translated into a TRS, but in general, it is not obvious how to obtain TRSs *suitable for existing automated termination techniques*. Adapting TRS-techniques for termination of Haskell is challenging for the following reasons:

- Haskell has a *lazy evaluation* strategy. However, most TRS-techniques ignore such evaluation strategies and try to prove that *all* reductions terminate.
- 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 only few techniques for automated termination analysis of functional programs. Methods for first-order languages with strict evaluation strategy were developed in [5, 11, 17]. For higher-order languages, [1, 3, 18] study how to ensure termination by typing and [16] defines a restricted language where all

* Supported by the Deutsche Forschungsgemeinschaft DFG under grant GI 274/5-1. Extended version of a paper [9] which appeared in *Proc. RTA '06*, Seattle, USA, LNCS 4098, pp. 297-312, 2006.

evaluations terminate. A successful approach for automated termination proofs for a small Haskell-like language was developed in [12]. (A related technique is [4], 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. In our approach we build upon the method of [12], but we adapt it in order to make TRS-techniques applicable.[1]

We recapitulate Haskell in Sect. 2 and introduce our notion of "termination". To analyze termination, our method first generates a corresponding *termination graph* (similar to the "termination tableaux" in [12]), cf. Sect. 3. But in contrast to [12], then our method transforms the termination graph into *dependency pair problems* which can be handled by existing techniques from term rewriting (Sect. 4). Our approach in Sect. 4 can deal with any termination graph, whereas [12] can only handle termination graphs of a special form ("without crossings"). We implemented our technique in the termination prover AProVE [10], cf. Sect. 5.

2 Haskell

We now give the syntax and semantics for a subset of Haskell which only uses certain easy patterns and terms (without "λ"), and function definitions without conditions. Any Haskell-program (without type classes and built-in data structures)[2] can automatically be transformed into a program from this subset [15].[3] For example, in our implementation lambda abstractions are removed by replacing every Haskell-term "$\backslash\, t_1 ... t_n \to t$" with the free variables $x_1, ..., x_m$ by "$\mathsf{f}\, x_1 \ldots x_m$". Here, f is a new function symbol with the defining equation $\mathsf{f}\, x_1 ... x_m\, t_1 ... t_n = t$.

2.1 Syntax of Haskell

In our subset of Haskell, we only permit user-defined data structures 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. Moreover, there is a pre-defined binary type constructor $\to$ for function types. Since Haskell's type system is polymorphic, it also has *type variables* like a which stand for any type.

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.

[1] Alternatively, one could simulate Haskell's evaluation strategy by *context-sensitive rewriting* (CSR), cf. [6]. But termination of CSR is hard to analyze automatically.

[2] See Sect. 5 for an extension to type classes and pre-defined data structures.

[3] 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 make interactive termination proofs harder.

Apart from data-declarations, a program has function declarations. Here, "$\mathsf{from}\ x$" generates the infinite list of numbers starting with x and "$\mathsf{take}\ n\ xs$" returns the first n elements of xs. The type of from is "$\mathsf{Nats} \to (\mathsf{List}\ \mathsf{Nats})$" and take has type "$\mathsf{Nats} \to (\mathsf{List}\ a) \to (\mathsf{List}\ a)$" where $\tau_1 \to \tau_2 \to \tau_3$ stands for $\tau_1 \to (\tau_2 \to \tau_3)$.

$$\mathsf{from}\ x = \mathsf{Cons}\ x\ (\mathsf{from}\ (\mathsf{S}\ x)) \qquad \begin{aligned} &\mathsf{take}\ \mathsf{Z}\ xs = \mathsf{Nil}\\ &\mathsf{take}\ n\ \mathsf{Nil} = \mathsf{Nil}\\ &\mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs)\end{aligned}$$

In general, function declarations have the form "$f\ \ell_1 \ldots \ell_n = r$". The function symbols f on 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*, i.e., 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 "$((t_1\ t_2)\ t_3)$". The set of *patterns* is the smallest set with all variables and 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 $(t_1\ t_2)|_{i\,\pi} = t_i|_\pi$. The *head* of t is $t|_{1^n}$ where n is the maximal number with $1^n \in Pos(t)$. So the head of $t = \mathsf{take}\ n\ xs$ (i.e., "$(\mathsf{take}\ n)\ xs$") is $t|_{11} = \mathsf{take}$.

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 outermost strategy. So in most cases, $\mathbf{e}(t)$ is the top position ε. An exception are terms "$f\ t_1 \ldots t_n\ t_{n+1} \ldots t_m$" where $\text{arity}(f) = n$ 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 $\ell = r$ will then become applicable on top position. We say that an equation $\ell = r$ from the program is *feasible* for a term t and define the corresponding *evaluation position* $\mathbf{e}_\ell(t)$ *w.r.t.* ℓ if either

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

Since Haskell considers the order of the program's equations, t is evaluated below the top (on 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.

Definition 1 (Evaluation Position $\mathbf{e}(t)$). *For any term t, we define*

$$\mathbf{e}(t) = \begin{cases} 1^{m-n}\,\pi, & \text{if } t = f\,t_1\,\ldots\,t_n\,t_{n+1}\ldots t_m,\ f \text{ is defined, } m > n = \mathrm{arity}(f), \\ & \text{and } \pi = \mathbf{e}(\,f\,t_1\,\ldots\,t_n\,) \\ \mathbf{e}_\ell(t)\,\pi, & \text{if } t = f\,t_1\,\ldots\,t_n,\ f \text{ is defined, } n = \mathrm{arity}(f), \text{ there are feasible} \\ & \text{equations for } t \text{ (the first is ``}\ell = r\text{''), } \mathbf{e}_\ell(t) \neq \varepsilon, \text{ and } \pi = \mathbf{e}(t|_{\mathbf{e}_\ell(t)}) \\ \varepsilon, & \text{otherwise} \end{cases}$$

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

We now present Haskell's operational semantics by defining the *evaluation relation* $\to_{\mathsf{H}}$. For any term t, it performs a rewrite step on position $\mathbf{e}(t)$ using the *first* applicable defining equation of the program. So terms like "$x\,\mathsf{Z}$" or "$\mathsf{take}\,\mathsf{Z}$" 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 $\mathrm{head}(s|_{\mathbf{e}(s)})$ is a defined symbol g, then we call s an *error term* (i.e., then g is not "completely" defined).

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, i.e., $t_1, \ldots, t_n$ may be evaluated with $\to_{\mathsf{H}}$. This corresponds to the behavior of Haskell-interpreters like Hugs 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 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 (Evaluation Relation $\to_{\mathsf{H}}$). *We have $t \to_{\mathsf{H}} s$ iff either*

(1) t rewrites to s on 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 $\mathsf{from}\,m \to_{\mathsf{H}} \mathsf{Cons}\,m\,(\mathsf{from}\,(\mathsf{S}\,m))$ $\to_{\mathsf{H}} \mathsf{Cons}\,m\,(\mathsf{Cons}\,(\mathsf{S}\,m)\,(\mathsf{from}\,(\mathsf{S}\,m))) \to_{\mathsf{H}} \ldots$ On the other hand, the following evaluation is finite: $\mathsf{take}\,(\mathsf{S}\,\mathsf{Z})\,(\mathsf{from}\,m) \to_{\mathsf{H}} \mathsf{take}\,(\mathsf{S}\,\mathsf{Z})\,(\mathsf{Cons}\,m\,(\mathsf{from}\,(\mathsf{S}\,m))) \to_{\mathsf{H}}$ $\mathsf{Cons}\,m(\mathsf{take}\,\mathsf{Z}\,(\mathsf{from}\,(\mathsf{S}\,m))) \to_{\mathsf{H}} \mathsf{Cons}\,m\,\mathsf{Nil}$.

The reason for permitting non-ground terms in Def. 1 and 2 is that our termination method in Sect. 3 evaluates Haskell *symbolically.* Here, variables stand for arbitrary *terminating* terms. Def. 3 introduces our notion of termination.

Definition 3 (H-Termination). *The set of* H-terminating *ground terms is the smallest set of ground terms t with*

(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, $n < \mathrm{arity}(f)$, 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 (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.

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

$$\mathsf{nonterm}\ \mathsf{True}\ x = \mathsf{True} \qquad\qquad \mathsf{nonterm}\ \mathsf{False}\ x = \mathsf{nonterm}\ (x\ \mathsf{True})\ x \tag{1}$$

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. In full Haskell, such functions can of course be represented by lambda terms and indeed, "nonterm False ($\backslash y \to$ False)" starts an infinite evaluation.

3 From Haskell to Termination Graphs

Our goal is to prove H-termination of a *start term* t. By Def. 3, H-*termination* of t implies 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, i.e., substitutions where $\sigma(x)$ is a ground term in normal form w.r.t. $\to_{\mathsf{H}}$ for all variables x in t.

Regard the start term $t = \mathsf{take}\, u\, (\mathsf{from}\, m)$. A naive approach would be to consider the defining equations of all needed functions (i.e., take and from) as rewrite rules. However, this disregards Haskell's lazy evaluation strategy. So due to the non-terminating rule for "from", we would fail to prove H-termination of t.

Therefore, our approach starts evaluating the start term a few steps. This gives rise to a so-called *termination graph*. Instead of transforming defining Haskell-equations into rewrite rules, we then transform the termination graph into rewrite rules. 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 [12], 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, cf. Sect. 4. We now describe our five expansion rules intuitively using Fig. 1. Their formal definition is given in Def. 4.

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 on 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 $(c\,x_1 \dots x_n)$. Here, c is a constructor of the appropriate type and $x_1, \dots, x_n$ are fresh variables. The edges to these children are labelled with the respective substitutions $[u/(c\,x_1 \dots x_n)]$. 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 by a 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.

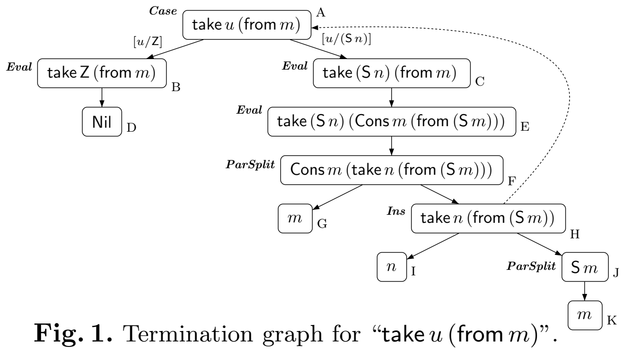


Fig. 1. Termination graph for "$\mathsf{take}\,u\,(\mathsf{from}\,m)$".

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***-rule perform a form of *narrowing* that respects the evaluation strategy and the types of Haskell.

The term Nil in node D cannot be evaluated and therefore, D is a leaf of the termination graph. But the term "$\mathsf{Cons}\,m\,(\mathsf{take}\,n\,(\mathsf{from}\,(\mathsf{S}\,m)))$" in node F may be evaluated further. Whenever the head of a term is a constructor like Cons or a variable,[4] then evaluations can only take place on 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).

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

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 $(\mathsf{S}\,m)$, respectively. Therefore, in addition to the child A, the node H gets two more children I and J marked with n and $(\mathsf{S}\,m)$. Finally, the ***ParSplit***-rule adds J's child K, marked with m.

Now we consider a different start term, viz. "take". If a defined function has "too few" arguments, then by Def. 3 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 would add 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.

Definition 4 (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 "$G \Rightarrow G'$") 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 to add 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 $t = (f\,t_1 \dots t_n)$, f is a defined symbol, $n \geq \mathrm{arity}(f)$, $t \to_{\mathsf{H}} \tilde{t}$*

Case: $\mathbf{ch}(t) = \{t\sigma_1, \dots, t\sigma_k\}$, *if $t = (f\,t_1 \dots t_n)$, f is a defined function symbol, $n \geq \mathrm{arity}(f)$, $t|_{\mathsf{e}(t)}$ is a variable x of type "$d\,\tau_1 ... \tau_m$" 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 $\sigma_i = [x/(c_i\,x_1 \dots x_{n_i})]$ for fresh pairwise different variables $x_1, \dots, x_{n_i}$. The edge from t to $t\sigma_i$ is marked with the substitution σ_i.*

VarExp: $\mathbf{ch}(t) = \{t\,x\}$, *if $t = (f\,t_1 \dots t_n)$, f is a defined function symbol, $n < \mathrm{arity}(f)$, x is a fresh variable*

ParSplit: $\mathbf{ch}(t)=\{t_1, ..., t_n\}$ *if $t=(c\,t_1 ... t_n)$, c is a constructor or variable, $n>0$*

Ins: $\mathbf{ch}(t) = \{s_1, \dots, s_m, \tilde{t}\}$, *if $t = (f\,t_1 \dots t_n)$, t is not an error term, f is a defined symbol, $n \geq \mathrm{arity}(f)$, $t = \tilde{t}\sigma$ for some term $\tilde{t}$, $\sigma = [x_1/s_1, \dots, x_m/s_m]$. Moreover, either $\tilde{t} = (x\,y)$ for fresh variables x and y, or $\tilde{t}$ is an **Eval**-node, or $\tilde{t}$ is a **Case**-node and all paths starting in $\tilde{t}$ reach an **Eval**-node or a leaf with an error term after traversing only **Case**-nodes.[5] The edge from t to $\tilde{t}$ is called an* instantiation edge.
*If the graph already contained a node marked with $\tilde{t}$, then we permit to re-use this node in the **Ins**-rule. So in this case, 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 $G_t \Rightarrow^* G$ and G is in normal form w.r.t. $\Rightarrow$.*

If one disregards ***Ins***, then for each leaf there is at most one rule applicable.[6] However, the ***Ins***-rule introduces indeterminism. Instead of applying the ***Case***-rule on node A in Fig. 1, we could also apply ***Ins*** and generate an instantiation

[5] This ensures that every cycle of the graph contains at least one ***Eval***-node.
[6] No rule is applicable to leaves with variables, constructors, or error terms.

edge to a new node with $\tilde{t} = (\mathsf{take}\, u\, ys)$. Since the instantiation is $[ys/(\mathsf{from}\, m)]$, node A would get an additional child node marked with the non-H-terminating term $(\mathsf{from}\, m)$. Then our approach in Sect. 4 which tries to prove H-termination of *all* terms in the termination graph would fail, whereas it succeeds for the graph in Fig. 1. Therefore, in our implementation we developed a heuristic for constructing termination graphs which tries to avoid unnecessary applications of ***Ins*** (since applying ***Ins*** means that one has to prove H-termination of more terms).

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

$$\mathsf{tma}\,(\mathsf{S}\, m) = \mathsf{tma}\, m\, m \tag{2}$$

Here, tma has the type $\mathsf{Nats} \to a$. We obtain the termination graph in Fig. 2. After applying ***Case*** and ***Eval***, we result in "$\mathsf{tma}\, m\, m$" 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 (i.e., it permits to go from "$\mathsf{tma}\, m\, m$" in node D to "$\mathsf{tma}\, m$" in node F). Thm. 5 shows that by the expansion rules of Def. 4 one can always obtain normal forms.[7]

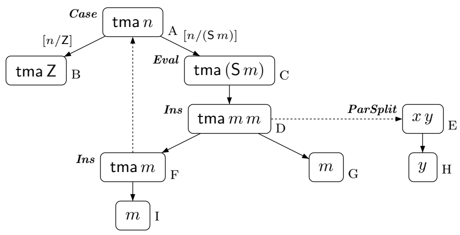


Fig. 2. Termination graph for "$\mathsf{tma}\, n$"

Theorem 5 (Existence of Termination Graphs). *The relation $\Rightarrow$ is normalizing, i.e., 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 techniques for TRSs is the *dependency pair* (DP) method [2]. In particular, the DP method can be formulated as a general framework which permits the integration and combination of *any* termination technique for TRSs [7]. This *DP framework* operates on so-called *DP problems* $(\mathcal{P}, \mathcal{R})$. Here, $\mathcal{P}$ and $\mathcal{R}$ are TRSs that 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*, i.e., no

[7] All proofs can be found in the appendix.

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 [7] for an overview on techniques to prove finiteness of DP problems.[8]

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, cf. the end of this section. In this way, we also do not have to impose any restrictions on the form of the termination graph (as in [12] where one can only analyze certain start terms which lead to termination graphs "without crossings"). 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 "$\mathsf{take}\,u$"). Since most methods and tools for automated termination analysis only operate on first-order TRSs, we translate higher-order terms into *applicative* first-order terms containing just variables, constants, and a binary symbol ap for function application. So terms like "$x\,y$", "$\mathsf{take}\ u$", and "$\mathsf{take}\ u\ xs$" are 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. As shown in [8], the DP framework is well suited to prove termination of applicative TRSs automatically. To ease readability, in the remainder we will not distinguish anymore between higher-order and corresponding applicative first-order terms, since the conversion between these two representations is obvious.

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 cycles resp. 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 non-empty path from n_1 to n_2 traversing only nodes of G'. (In particular, there must also be a non-empty path from every node to itself in G'.) The termination graph for "$\mathsf{take}\,u\,(\mathsf{from}\,m)$" in Fig. 1 has just one SCC with the nodes A, C, E, F, H. The following definition is needed to extract dependency pairs from SCCs of the termination graph.

Definition 6 (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 Fig. 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

[8] In the DP literature, one usually does not regard 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* [2].)

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, i.e., then 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 $\sigma_1, \ldots, \sigma_m$, then we generate the dependency pair $s\sigma_1 \ldots \sigma_m \to t$. In Fig. 1, the first edge of the DP path is labelled with the substitution $[u/(\mathsf{S}\, n)]$ and all remaining edges are labelled with the identity. Thus, we generate the dependency pair

$$\mathsf{take}\,(\mathsf{S}\, n)\,(\mathsf{from}\, m) \to \mathsf{take}\, n\,(\mathsf{from}\,(\mathsf{S}\, m)). \tag{3}$$

The resulting DP problem is $(\mathcal{P}, \mathcal{R})$ where $\mathcal{P} = \{(3)\}$ and $\mathcal{R} = \varnothing$.[9] Automated termination tools 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 Fig. 2 is also easy to prove automatically.

A slightly more challenging example is obtained by replacing the last take-rule by the following two rules, where p computes the **p**redecesor function.

$$\mathsf{take}\,(\mathsf{S}\, n)\,(\mathsf{Cons}\, x\, xs) = \mathsf{Cons}\, x\,(\mathsf{take}\,(\mathsf{p}\,(\mathsf{S}\, n))\, xs) \qquad\qquad \mathsf{p}\,(\mathsf{S}\, x) = x \tag{4}$$

Now the resulting termination graph can be obtained from the graph in Fig. 1 by replacing the subgraph starting with node F by the subgraph in Fig. 3.

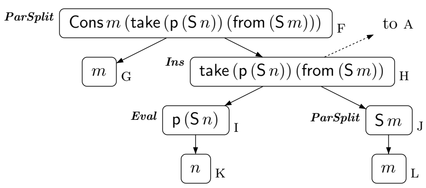


Fig. 3. Subtree at node F of Fig. 1

We want to construct an infinite chain whenever the termination graph contains a non-H-terminating term. In this case, there also exists a DP path with first node s such that s is not H-terminating. So 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 labelled with the substitutions $\sigma_1, \ldots, \sigma_m$ such that σ is an instance of $\sigma_1 \ldots \sigma_m$ and such that $t\sigma$ is also not H-terminating. 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 τ. But while $\tilde{t}\tau\sigma$ is not H-terminating, the substitution $\tau\sigma$ is not necessarily a normal ground substitution. The reason is that t and hence τ may contain defined symbols.

This is also the case in our example. 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 $\tau = [u/(\mathsf{p}\,(\mathsf{S}\, n)), m/(\mathsf{S}\, m)]$. So $\tau(u)$ is not normal.

In this example, the problem can be avoided by already evaluating the right-hand sides of dependency pairs as much as possible. So instead of a dependency

[9] Def. 11 will explain how to generate $\mathcal{R}$ in general.

pair $s\sigma_1 \ldots \sigma_m \to t$ we now generate the dependency pair $s\sigma_1 \ldots \sigma_m \to \mathbf{ev}(t)$. For a node marked with t, $\mathbf{ev}(t)$ is the term reachable from t by traversing only ***Eval***-nodes. So in our example $\mathbf{ev}(\mathsf{p}\,(\mathsf{S}\,n)) = n$, since node I is an ***Eval***-node with an edge to node K. Moreover, $\mathbf{ev}(t)$ can also evaluate subterms of t if t is a ***ParSplit***-node with a constructor as head or an ***Ins***-node without an instantiation edge to the special node "$x\,y$". We obtain $\mathbf{ev}(\mathsf{S}\,m) = \mathsf{S}\,m$ for node J and $\mathbf{ev}(\mathsf{take}\,(\mathsf{p}\,(\mathsf{S}\,n))\,(\mathsf{from}\,(\mathsf{S}\,m))) = \mathsf{take}\,n\,(\mathsf{from}\,(\mathsf{S}\,m))$ for node H. Thus, the resulting DP problem is again $(\mathcal{P}, \mathcal{R})$ with $\mathcal{P} = \{(3)\}$ and $\mathcal{R} = \varnothing$.

To see how $\mathbf{ev}(t)$ must be defined for ***ParSplit***-nodes where head(t) is a variable, we regard the function $\mathsf{nonterm}$ again, cf. (1). In the termination graph for the start term "$\mathsf{nonterm}\,b\,x$", we obtain a DP path from the node with the start term to a node with "$\mathsf{nonterm}\,(x\,\mathsf{True})\,x$" labelled with the substitution $[b/\mathsf{False}]$. So the resulting DP problem only contains the dependency pair "$\mathsf{nonterm}\,\mathsf{False}\,x \to \mathbf{ev}(\mathsf{nonterm}\,(x\,\mathsf{True})\,x)$". If we would define $\mathbf{ev}(x\,\mathsf{True}) = x\,\mathsf{True}$, then $\mathbf{ev}$ would not modify the term "$\mathsf{nonterm}\,(x\,\mathsf{True})\,x$". 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 "$x\,\mathsf{True}$" to False.) But as discussed in Sect. 3, x can be instantiated by arbitrary H-terminating functions and then, "$x\,\mathsf{True}$" can evaluate to any term. Therefore, $\mathbf{ev}$ must replace terms like "$x\,\mathsf{True}$" by fresh variables. Similarly, if t is an ***Ins***-node with an instantiation edge to the node "$x\,y$", then we also define $\mathbf{ev}(t)$ to be a fresh variable. The reason for this will become clear in the sequel, cf. Footnote 11.

Definition 7 (ev). *Let G be a termination graph with a node t.*[10] *Then*

$$\mathbf{ev}(t) = \begin{cases} t, & \text{if } t \text{ is a leaf, a } \textbf{\textit{Case}}\text{-node, or a } \textbf{\textit{VarExp}}\text{-node} \\ x, & \text{for a fresh variable } x \text{ if either} \\ & t \text{ is a } \textbf{\textit{ParSplit}}\text{-node where head}(t) \text{ is a variable or} \\ & t \text{ is an } \textbf{\textit{Ins}}\text{-node with an instantiation edge to "}x\,y\text{"} \\ \mathbf{ev}(\tilde{t}), & \text{if } t \text{ is an } \textbf{\textit{Eval}}\text{-node with child } \tilde{t} \\ \tilde{t}[x_1/\mathbf{ev}(t_1), \ldots, x_n/\mathbf{ev}(t_n)], & \text{if } t = \tilde{t}[x_1/t_1, \ldots, x_n/t_n] \text{ and either} \\ & t \text{ is an } \textbf{\textit{Ins}}\text{-node with the children } t_1, \ldots, t_n, \tilde{t} \text{ or} \\ & t \text{ is a } \textbf{\textit{ParSplit}}\text{-node, and } \tilde{t} = (c\,x_1 \ldots x_n) \text{ for a constructor } c \end{cases}$$

Our goal was to construct 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, 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 $t = \tilde{t}[x_1/t_1, \ldots, x_n/t_n]$ and $\mathbf{ev}(t) = \tilde{t}[x_1/\mathbf{ev}(t_1), \ldots, x_n/\mathbf{ev}(t_n)]$. 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 $[x_1/\mathbf{ev}(t_1)\sigma, \ldots, x_n/\mathbf{ev}(t_n)\sigma]$, then it is again not H-terminating. However, the substitution $[x_1/\mathbf{ev}(t_1)\sigma, \ldots, x_n/\mathbf{ev}(t_n)\sigma]$ is not necessarily normal. The problem is that $\mathbf{ev}$ does not perform those evaluations that correspond to instan-

[10] To simplify the presentation, we identify nodes with the terms they are labelled with.

tiation edges and to edges from ***Case***-nodes. 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 $\mathbf{ev}(t_i)\sigma$ 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'. 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 many times. Hence, we obtain an infinite chain.

As an example, we replace the equation for p in (4) by the following two defining equations:

$$\mathsf{p}\,(\mathsf{S}\,\mathsf{Z}) = \mathsf{Z} \qquad\qquad \mathsf{p}\,(\mathsf{S}\,x) = \mathsf{S}\,(\mathsf{p}\,x) \qquad (5)$$

In the termination graph for "$\mathsf{take}\,u\,(\mathsf{from}\,m)$" from Fig. 1 and 3, the node I would now be replaced by the subtree in Fig. 4. So I is now a ***Case***-node. Thus, instead of (3) we obtain the dependency pair

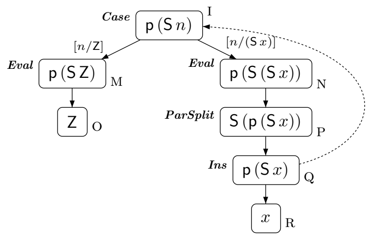


Fig. 4. Subtree at node I of Fig. 3

$$\mathsf{take}\,(\mathsf{S}\,n)\,(\mathsf{from}\,m) \to \mathsf{take}\,(\mathsf{p}\,(\mathsf{S}\,n))\,(\mathsf{from}\,(\mathsf{S}\,m)), \qquad (6)$$

since now $\mathbf{ev}$ does not modify its right-hand side anymore (i.e., $\mathbf{ev}(\mathsf{p}\,(\mathsf{S}\,n)) = \mathsf{p}\,(\mathsf{S}\,n)$). Hence, now the resulting DP problem must contain all rules $\mathcal{R}$ that might be used to evaluate $\mathsf{p}\,(\mathsf{S}\,n)$ when instantiated by σ.

So for any term t, we want to detect 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 $\mathbf{ev}$ stopped. So $\mathbf{con}(t)$ contains all terms which might give rise to further **con**tinuing evaluations that are not captured by $\mathbf{ev}$. 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 Fig. 4, we obtain $\mathbf{con}(\mathsf{p}\,(\mathsf{S}\,n)) = \{\mathsf{p}\,(\mathsf{S}\,n)\}$, since I is now a ***Case***-node. If we 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 $\mathbf{con}(t) = \{\mathsf{p}\,(\mathsf{S}\,n), \mathsf{take}\,u\,(\mathsf{from}\,m)\}$. Finally, **con** also stops at ***VarExp***-nodes (they are in normal form w.r.t. $\to_{\mathsf{H}}$), at ***ParSplit***-nodes whose head is a variable and at ***Ins***-nodes with an instantiation edge to "$x\,y$" (since $\mathbf{ev}$ already "approximates" their result by fresh variables).[11]

[11] The special treatment of ***Ins***-nodes $t = t_1\,t_2$ with an instantiation edge to "$x\,y$" was not considered in the definitions of **ev** and **con** in [9]. If they were handled like ordinary ***Ins***-nodes, then we would obtain $\mathbf{ev}(t) = \mathbf{ev}(t_1)\,\mathbf{ev}(t_2)$. But then $\mathbf{con}(t_1)$ and $\mathbf{con}(t_2)$ could be empty although "$\mathbf{ev}(t_1)\sigma\ \mathbf{ev}(t_2)\sigma$" can be evaluated further for a normal ground substitution σ.

As an example, consider the termination graph resulting from the non-terminating program with the defining equations "$\mathsf{f}\ \mathsf{Z}\ z = \mathsf{f}\ (\mathsf{id}\ z\ \mathsf{Z})\ z$" and "$\mathsf{id}\ x = x$".

Definition 8 (con). *Let G be a termination graph with a node t. Then*

$$\mathbf{con}(t) = \begin{cases} \varnothing, & \textit{if } t \textit{ is a leaf, a } \textbf{\textit{VarExp}}\textit{-node, a } \textbf{\textit{ParSplit}}\textit{-node with variable head,} \\ & \textit{or an } \textbf{\textit{Ins}}\textit{-node with instantiation edge to ``} x\,y \textit{''} \\ \{t\}, & \textit{if } t \textit{ is a } \textbf{\textit{Case}}\textit{-node} \\ \{\tilde{t}\} \cup \mathbf{con}(t_1) \cup \ldots \cup \mathbf{con}(t_n), & \textit{if } t \textit{ is an } \textbf{\textit{Ins}}\textit{-node with the} \\ & \textit{children } t_1, \ldots, t_n, \tilde{t} \textit{ and an instantiation edge from } t \textit{ to } \tilde{t} \\ \bigcup_{t' \textit{ child of } t} \mathbf{con}(t'), & \textit{otherwise} \end{cases}$$

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 $s\sigma_1 \ldots \sigma_m \to \mathbf{ev}(t)$ for every DP path from s to t labelled with $\sigma_1, \ldots, \sigma_m$ in G'. If $t = \tilde{t}[x_1/t_1, \ldots, x_n/t_n]$ has an instantiation edge to $\tilde{t}$, then the resulting DP problem must contain all rules that can be used reduce the terms in $\mathbf{con}(t_1) \cup \ldots \cup \mathbf{con}(t_n)$. For any term s, let $\mathbf{rl}(s)$ be the rules that can be used to reduce $s\sigma$ for normal ground substitutions σ. We will give the definition of $\mathbf{rl}$ afterwards.

Definition 9 (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*

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

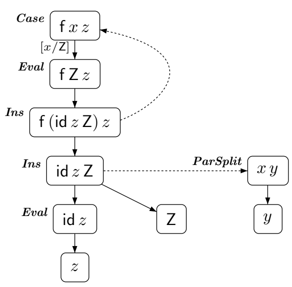
whenever G' contains a DP path from s to t labelled with $\sigma_1, \ldots, \sigma_m$, $t = \tilde{t}[x_1/t_1, \ldots, x_n/t_n]$ has an instantiation edge to $\tilde{t}$, and $q \in \mathbf{con}(t_1) \cup \ldots \cup \mathbf{con}(t_n)$.

In our example with the start term "$\mathsf{take}\, u\, (\mathsf{from}\, m)$" and the p-equations from (5), the termination graph in Fig. 1, 3, and 4 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 (6). So we obtain $\mathbf{dp}_{G_1} = (\{(6)\}, \mathcal{R}_1)$ where $\mathcal{R}_1$ contains $\mathbf{rl}(q)$ for all $q \in \mathbf{con}(\mathsf{p}\,(\mathsf{S}\,n)) = \{\mathsf{p}\,(\mathsf{S}\,n)\}$. Thus, $\mathcal{R}_1 = \mathbf{rl}(\mathsf{p}\,(\mathsf{S}\,n))$. 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 $q \in \mathbf{con}(x) = \varnothing$, i.e., $\mathcal{R}_2 = \varnothing$. 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

From the DP path, we obtain the dependency pair "$\mathsf{f}\ \mathsf{Z}\ z \to \mathbf{ev}(\mathsf{f}\ (\mathsf{id}\ z\ \mathsf{Z})\ z)$", where $\mathbf{ev}(\mathsf{f}\ (\mathsf{id}\ z\ \mathsf{Z})\ z) = \mathsf{f}\ \mathbf{ev}(\mathsf{id}\ z\ \mathsf{Z})\ z$. With our current definition, $\mathbf{ev}(\mathsf{id}\ z\ \mathsf{Z})$ is a fresh variable and therefore, the dependency pair is "$\mathsf{f}\ \mathsf{Z}\ z \to \mathsf{f}\ x\ z$" which gives rise to an infinite chain.

But if we treated "$\mathsf{id}\ z\ \mathsf{Z}$" as an ordinary ***Ins***-node, we would get $\mathbf{ev}(\mathsf{id}\ z\ \mathsf{Z}) = \mathbf{ev}(\mathsf{id}\ z)\ \mathbf{ev}(\mathsf{Z}) = z\ \mathsf{Z}$ and the dependency pair would be "$\mathsf{f}\ \mathsf{Z}\ z \to \mathsf{f}\ (z\ \mathsf{Z})\ z$". The resulting DP problem would contain no rules, since $\mathbf{con}(\mathsf{id}\ z) \cup \mathbf{con}(\mathsf{Z}) = \varnothing$. So then we could falsely prove H-termination of f.



a reduction with $\mathbf{rl}(s)$.[12] The only expansion rules which transform terms into "equal" ones are ***Eval*** and ***Case***. This leads to the following definition.

Definition 10 (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 no* ***Eval****- or* ***Case****-node. So t may also be a leaf.*

In Fig. 4, there are two rule paths starting in node I. The first one is I, M, O (since O is a leaf) and the other 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 labelled with $\sigma_1, \ldots, \sigma_m$, then $\mathbf{rl}(s)$ contains the rule $s\sigma_1 \ldots \sigma_m \to \mathbf{ev}(t)$. In addition, $\mathbf{rl}(s)$ must also contain all rules required to evaluate $\mathbf{ev}(t)$ further, i.e., all rules in $\mathbf{rl}(q)$ for $q \in \mathbf{con}(t)$.[13]

Definition 11 (rl). *For a node labelled with s, $\mathbf{rl}(s)$ is the smallest set with*

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

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

For the start term "$\mathsf{take}\, u\, (\mathsf{from}\, m)$" and the p-equations from (5), we obtained the DP problem $\mathbf{dp}_{G_1} = (\{6\}, \mathbf{rl}(\mathsf{p}\,(\mathsf{S}\,n)))$. Here, $\mathbf{rl}(\mathsf{p}\,(\mathsf{S}\,n))$ consists of

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

$$\mathsf{p}\,(\mathsf{S}\,(\mathsf{S}\,x)) \to \mathsf{S}\,(\mathsf{p}\,(\mathsf{S}\,x)) \qquad \text{(due to the rule path from I to P),} \tag{8}$$

as $\mathbf{ev}$ does not modify the right-hand sides of (7) and (8). 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 $\mathbf{dp}_{G_1} = (\{6\}, \{(7), (8)\})$ is also easy to show automatically.

Finally, consider the following program which leads to the graph in Fig. 5.

$$\mathsf{f}\,x = \mathsf{applyToZero}\,\mathsf{f} \qquad\qquad \mathsf{applyToZero}\,x = x\,\mathsf{Z}$$

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

[12] 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).

[13] So if $t = \tilde{t}[x_1/t_1, \ldots, x_n/t_n]$ has an instantiation edge to $\tilde{t}$, then here we also include all rules of $\mathbf{rl}(\tilde{t})$, since $\mathbf{con}(t) = \{\tilde{t}\} \cup \mathbf{con}(t_1) \cup \ldots \cup \mathbf{con}(t_n)$. In contrast, for the definition of $\mathbf{dp}$ in Def. 9 we only regard the rules $\mathbf{rl}(q)$ for $q \in \mathbf{con}(t_1) \cup \ldots \cup \mathbf{con}(t_n)$, whereas the evaluations of $\tilde{t}$ are captured by the dependency pairs.

The example also illustrates that **rl** and **dp** handle instantiation edges differently, cf. Footnote 13. Since there is a rule path from A to B, we would obtain $\mathbf{rl}(\mathsf{f}\,x) = \{\mathsf{f}\,x \to \mathsf{applyToZero}\,\mathsf{f}\} \cup \mathbf{rl}(\mathsf{applyToZero}\,x)$, since $\mathbf{con}(\mathsf{applyToZero}\,\mathsf{f}) = \mathsf{applyToZero}\,x$. So for the construction of **rl** we also have to include the rules resulting from nodes like C which are only reachable by instantiation edges.[14] We obtain $\mathbf{rl}(\mathsf{applyToZero}\,x) = \{\mathsf{applyToZero}\,x \to z\}$, since $\mathbf{ev}(x\,\mathsf{Z}) = z$ for a fresh variable z. The following theorem states the soundness of our approach.

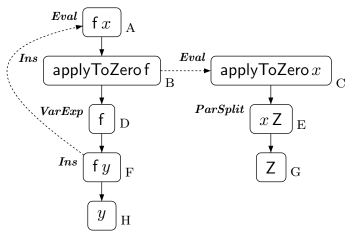


Fig. 5. Termination graph for "f x"

Theorem 12 (Soundness). *Let G be termination graph. If the DP problem $\mathbf{dp}_{G'}$ is finite for all SCCs G' of G, then all nodes t in G are* H*-terminating.*[15]

While we 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 result in 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 $\mathcal{P}$ and $\mathcal{R}$ can still be finite, since dependency pairs from $\mathcal{P}$ are only applied on top positions in chains and since $\mathcal{R}$ need not be terminating for finite DP problems $(\mathcal{P}, \mathcal{R})$.

5 Extensions, Implementation, and Experiments

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.

To ease readability, we did not regard Haskell's *type classes* and built-in data structures in the preceding sections. However, our approach easily extends to these concepts [15]. To deal with type classes, we use an additional ***Case***-rule in the construction of termination graphs, which instantiates type variables by all instances of the corresponding type class. Built-in data structures like Haskell's lists and tuples simply correspond to user-defined types with a different syntax. To deal with integers, we transform them into a notation with the constructors Pos and Neg (which take arguments of type Nats) and provide pre-defined rewrite rules for integer operations like addition, subtraction, etc. Floating-point numbers can be handled in a similar way (e.g., by representing them as fractions).

[14] This is different in the definition of **dp**. Otherwise, we would have $\mathcal{R} = \mathbf{rl}(y) \cup \mathbf{rl}(\mathsf{f}\,x)$.

[15] Instead of $\mathbf{dp}_{G'} = (\mathcal{P}, \mathcal{R})$, for H-termination it suffices to prove finiteness of $(\mathcal{P}^\sharp, \mathcal{R})$. Here, $\mathcal{P}^\sharp$ results from $\mathcal{P}$ by replacing each rule $f(t_1, ..., t_n) \to g(s_1, ..., s_m)$ in $\mathcal{P}$ by $f^\sharp(t_1, ..., t_n) \to g^\sharp(s_1, ..., s_m)$, where $f^\sharp$ and $g^\sharp$ are fresh "*tuple*" function symbols [2].

We implemented our approach in the termination prover AProVE [10]. It accepts the full Haskell 98 language defined in [13] and we successfully evaluated our implementation with standard Haskell-libraries from the Hugs-distribution such as Prelude, Monad, List, FiniteMap, etc. To access the implementation via a web interface and for details on our experiments see `http://aprove.informatik.rwth-aachen.de/eval/Haskell/`.

We conjecture that term rewriting techniques are also suitable for termination analysis of other kinds of programming languages. In [14], we recently adapted the dependency pair method in order to prove termination of *logic* programming languages like Prolog. In future work, we intend to examine the use of TRS-techniques for *imperative* programming languages as well.

References

1. A. Abel. Termination checking with types. *RAIRO - Theoretical Informatics and Applications*, 38(4):277–319, 2004.
2. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
3. G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Math. Structures in Comp. Sc.*, 14(1):1–45, 2004.
4. O. Fissore, I. Gnaedig, and H. Kirchner. Outermost ground termination. In *Proc. WRLA '02*, ENTCS 71, 2002.
5. J. Giesl. Termination analysis for functional programs using term orderings. In *Proc. SAS' 95*, LNCS 983, pages 154–171, 1995.
6. J. Giesl and A. Middeldorp. Transformation techniques for context-sensitive rewrite systems. *Journal of Functional Programming*, 14(4):379–427, 2004.
7. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR '04*, LNAI 3452, pages 301–331, 2005.
8. J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. FroCoS '05*, LNAI 3717, pp. 216-231, 2005.
9. J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages. In *Proc. RTA '06*, LNCS, 2006. To appear.
10. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the DP framework. In *Proc. IJCAR '06*, LNAI, 2006. To appear.
11. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proc. POPL '01*, pages 81–92. ACM Press, 2001.
12. S. E. Panitz and M. Schmidt-Schauss. TEA: Automatically proving termination of programs in a non-strict higher-order functional language. In *Proc. SAS '97*, LNCS 1302, pages 345–360, 1997.
13. S. Peyton Jones (ed.). Haskell 98 *Languages and Libraries: The revised report.* Cambridge University Press, 2003.
14. P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated termination analysis for logic programs by term rewriting. In *Proc. LOPSTR '06*, LNCS, 2006. To appear.
15. S. Swiderski. Terminierungsanalyse von Haskellprogrammen. Diploma Thesis, RWTH Aachen, 2005. See `http://aprove.informatik.rwth-aachen.de/eval/Haskell/`.

16. A. Telford and D. Turner. Ensuring termination in ESFP. *Journal of Universal Computer Science*, 6(4):474–488, 2000.
17. C. Walther. On proving the termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157, 1994.
18. H. Xi. Dependent types for program termination verification. *Higher-Order and Symbolic Computation*, 15(1):91–131, 2002.

A Proofs

Theorem 5 (Existence of Termination Graphs). *The relation $\Rightarrow$ is normalizing, i.e., for any term t, there exists a termination graph.*

Proof. We first construct a graph G in normal form w.r.t. $\Rightarrow$ which contains nodes marked with "$f\,x_1 \ldots x_n$" for all defined symbols f where the x_i are pairwise fresh variables and $\text{arity}(f) = n$. Then, by performing suitable expansion steps and by adding instantiation edges to these nodes, the graph G_t (which only consists of one node which is marked with t) can be transformed into normal form. Here, in the end all nodes which are not reachable from t are removed.

To construct G, we start with the nodes "$f\,x_1 \ldots x_n$" for all defined symbols f. Currently, these nodes are not reached by any edges. By performing ***Case***- and ***Eval***-steps, we obtain paths from "$f\,x_1 \ldots x_n$" to all right-hand sides of f-rules. Then we apply the following procedure: We apply ***ParSplit*** until we end up with leaves of the form "$g\,s_1 \ldots s_k$" for defined symbols g. If $k = \text{arity}(g)$, then we can add an instantiation edge to the already existing node "$g\,x_1 \ldots x_k$" and continue this process with the nodes $s_1, \ldots, s_k$ (i.e., we apply ***ParSplit*** again until we reach leaves with a defined symbol as head). If $k < \text{arity}(g)$, then we perform a number of ***VarExp***-steps to obtain a term where the number of arguments of g is $\text{arity}(g)$. If $k > \text{arity}(g)$, then we perform ***Ins***-steps with instantiation edges to the node "$x\,y$" until we obtain a term where the number of arguments of g is $\text{arity}(g)$. $\square$

To prove the soundness theorem of our approach (Thm. 12), we need a lemma about the semantics of **ev**, **con**, and **rl** (Lemma 18) and a lemma about the semantics of **dp** (Lemma 20). To describe the semantics of **ev**, **con**, and **rl**, we use the following relation $\Rightarrow_{\mathsf{H}}$, which also permits reductions on positions beside or below the evaluation position.

Definition 13 ($\Rightarrow_{\mathsf{H}}$). *We have $t \Rightarrow_{\mathsf{H}} s$ iff t rewrites to s on a position π which is not strictly above $\mathbf{e}(t)$ using the first equation of the program whose left-hand side matches $t|_\pi$.*

The following lemma shows that $\Rightarrow_{\mathsf{H}}$ allows us to simulate all reductions that are performed by the function **ev**.

Lemma 14 (Simulation of ev by $\Rightarrow_{\mathsf{H}}$). *Let G be a termination graph and let σ be a substitution. If $t\sigma \Rightarrow_{\mathsf{H}}^* \mathbf{ev}(t)\sigma$ holds for all* **ParSplit**- *and* **Ins**-*nodes t where $\mathbf{ev}(t)$ is a fresh variable, then $t\sigma \Rightarrow_{\mathsf{H}}^* \mathbf{ev}(t)\sigma$ holds for all nodes t of the termination graph G.*

Proof. We use induction on the edge-relation of G where we remove all instantiation edges. This relation is well founded, since after the removal of the instantiation edges, G is an acyclic graph. We only have to consider the cases where $\mathbf{ev}(t) \neq t$ and where $\mathbf{ev}(t)$ is not a fresh variable.

If t is an ***Eval***-node with child $\tilde{t}$, then we have $t\sigma \to_{\mathsf{H}} \tilde{t}\sigma$ and thus, $t\sigma \Rightarrow_{\mathsf{H}} \tilde{t}\sigma$. By the induction hypothesis we obtain $\tilde{t}\sigma \Rightarrow_{\mathsf{H}}^* \mathbf{ev}(\tilde{t})\sigma$ and hence $t\sigma \Rightarrow_{\mathsf{H}} \tilde{t}\sigma \Rightarrow_{\mathsf{H}}^* \mathbf{ev}(\tilde{t})\sigma = \mathbf{ev}(t)\sigma$.

If $t = (c\ t_1 \dots t_n)$ is a ***ParSplit***-node with a constructor as head, then we directly obtain the result from the induction hypotheses for the children t_i:

$$t\sigma = (c\, t_1\sigma \dots t_n\sigma) \Rightarrow_{\mathsf{H}}^* (c\, \mathbf{ev}(t_1)\sigma \dots \mathbf{ev}(t_n)\sigma) = \mathbf{ev}(t)\sigma$$

Finally, if $t = \tilde{t}[x_1/t_1, \dots, x_n/t_n]$ is an ***Ins***-node with an instantiation edge to an ***Eval***- or ***Case***-node $\tilde{t}$, then we proceed in a similar way by using the induction hypotheses for the t_i:

$$\begin{aligned} t\sigma &= \tilde{t}[x_1/t_1, \dots, x_n/t_n]\sigma \\ &= \tilde{t}\sigma[x_1/t_1\sigma, \dots, x_n/t_n\sigma] \\ &\Rightarrow_{\mathsf{H}}^* \tilde{t}\sigma[x_1/\mathbf{ev}(t_1)\sigma, \dots, x_n/\mathbf{ev}(t_n)\sigma] \\ &= \tilde{t}[x_1/\mathbf{ev}(t_1), \dots, x_n/\mathbf{ev}(t_n)]\sigma \\ &= \mathbf{ev}(t)\sigma \end{aligned}$$

□

Lemma 18 about the semantics of **ev**, **con**, and **rl** shows a kind of converse to Lemma 14, i.e., how to simulate reductions with $\to_{\mathsf{H}}$ by **ev**, **con**, and **rl**. Essentially, our goal is to show that if t is a node in the termination graph and there is a reduction $t\sigma \to_{\mathsf{H}}^* q$, then we also have $\mathbf{ev}(t)\sigma \to^*_{\bigcup_{s \in \mathbf{con}(t)} \mathbf{rl}(s)} q'$ for some term q' with $q \Rightarrow_{\mathsf{H}}^* q'$.

Since the rules $\bigcup_{s \in \mathbf{con}(t)} \mathbf{rl}(s)$ do not take σ into account, this does not hold in general if the terms introduced by σ are evaluated in the reduction $t\sigma \to_{\mathsf{H}}^* q$. A possibility would be to restrict ourselves to *normal* substitutions σ, i.e., to substitutions where $\sigma(x)$ is in normal form w.r.t. $\to_{\mathsf{H}}$ for all variables x. However, to ease the proof of Lemma 18, we consider a slightly larger class of substitutions. We only require that σ is *evaluated enough* in the reduction $t\sigma \to_{\mathsf{H}}^* q$. This means that in this reduction we do not evaluate those terms introduced by σ which may have a non-functional type. (Terms with functional type may lead to future evaluations when supplied with an argument. However, this does not lead to any problems for our desired lemma, because **ev** replaces such subterms by fresh variables.)

We formalize this concept in the two following definitions. Here, drop replaces all non-functional terms by fresh variables, except those terms whose head is already a constructor.

Definition 15 (drop, undrop)**.** *We define a function* drop *from terms to terms as follows:*

- $\mathrm{drop}(t) = t$*, if* t *has a functional type (i.e., an instance of the type* $a \to b$*)*
- $\mathrm{drop}(c\, t_1 \dots t_n) = c\, \mathrm{drop}(t_1) \dots \mathrm{drop}(t_n)$*, if* c *is a constructor of arity* n
- $\mathrm{drop}(t) = x_t$*, otherwise; here* x_t *is some fresh variable*

We define undrop *as the inverse function of* drop*, i.e.,* undrop *is the substitution which replaces every variable x_t by t. For any substitution σ, let σ_{drop} be the substitution with $\sigma_{\mathrm{drop}}(x) = \mathrm{drop}(\sigma(x))$ for all variables x. (Here, we allow substitutions with infinite domains.)*

For example, if we have a data type

$$\mathsf{data}\ \mathsf{D}\ a = \mathsf{C}\ (a \to a)\ \mid \mathsf{E}\ a\ \mid \mathsf{G}\ a\ a$$

and a defined function f of type a, then we obtain $\mathrm{drop}(\mathsf{f}) = x_{\mathsf{f}}$, $\mathrm{drop}(\mathsf{E}\,\mathsf{f}) = \mathsf{E}\,x_{\mathsf{f}}$, $\mathrm{drop}(\mathsf{C}\,\mathsf{f}) = \mathsf{C}\,\mathsf{f}$ (since here, the subterm f has type $a \to a$), and $\mathrm{drop}(\mathsf{G}\,\mathsf{f}) = \mathsf{G}\,\mathsf{f}$ (since here, "$\mathsf{G}\,\mathsf{f}$" has the functional type $a \to \mathsf{D}\,a$).

Now we define that a substitution is *evaluated enough* for a reduction if this reduction could also be performed when replacing all terms introduced by the substitution (except constructors and terms of functional type) by fresh variables. The constructors cannot be replaced since they are needed for pattern matching. The terms of functional types cannot be replaced either since they can be substituted for variables in the heads of subterms (i.e., for variables x in subterms of the form "$x\,t_1 \dots t_n$" and they can be further evaluated when supplied with suitable arguments $t_1 \dots t_n$).

Definition 16 (Evaluated Enough). *We say that a substitution σ is* evaluated enough *in a (possibly infinite) reduction $t\sigma \to_{\mathsf{H}} t_1 \to_{\mathsf{H}} t_2 \to_{\mathsf{H}} \dots$ iff $t\sigma_{\mathrm{drop}} \to_{\mathsf{H}} s_1 \to_{\mathsf{H}} s_2 \to_{\mathsf{H}} \dots$ and $t_i = \mathrm{undrop}(s_i)$ for all i.*

An evaluated enough substitution may introduce terms of functional type which may be evaluated further during a reduction. We have to ensure that such an evaluation only happens if this is required in order to obtain again a subterm with a constructor as head. Otherwise, $t\sigma \to_{\mathsf{H}}^* q$ does not imply $\mathbf{ev}(t)\sigma \to^*_{\bigcup_{s \in \mathbf{con}(t)} \mathbf{rl}(s)} q'$ for some term q' with $q \Rightarrow_{\mathsf{H}}^* q'$. The reason is that t could be a variable x at a leaf of the termination graph and σ could instantiate x by a term of functional type which is then reduced to q.

Therefore, we restrict ourselves to so-called *necessary* reductions $t\sigma \to_{\mathsf{H}}^* q$. These reductions are needed in order to facilitate pattern matching. The idea of necessary reductions is the following restriction: "if one performs reductions at all, then one has to evaluate until the head is a constructor".

Definition 17 (Necessary Reduction). *We say that a reduction $t \to_{\mathsf{H}}^* s$ is* necessary *iff $t = s$ or both $s = (c\ s_1 \dots s_n)$ for some constructor c of arity n and*

- *$t = c\,t_1 \dots t_n$ and all reductions $t_i \to_{\mathsf{H}}^* s_i$ are necessary or*
- *$t = (f\ t_1 \dots t_k) \to_{\mathsf{H}}^* (g\ l_1 \dots l_m) \to_{\mathsf{H}} (c\ r_1 \dots r_n)$ for some defined symbol g and all reductions $r_i \to_{\mathsf{H}}^* s_i$ are necessary.*

Note that whenever t starts a necessary reduction of length greater than zero, then t has a base type, since t can be reduced to $(c\ s_1 \dots s_n)$ for some constructor c of arity n.

Now we can prove the desired lemma which states the needed properties of **ev**, **con**, and **rl**.

Lemma 18 (Properties of ev, con, and rl). *Let G be a termination graph and let t be a node in G. Let $t\sigma \to_{\mathsf{H}}^* q$ be a necessary reduction where σ is evaluated enough. Then we have*[16]

(a) $\mathbf{ev}(t)\sigma \to^*_{\bigcup_{s\in\mathbf{con}(t)}\mathbf{rl}(s)} q'$ *for some term q' with $q \Rrightarrow_{\mathsf{H}}^* q'$*

*(b) If t is a **Case**- or **Eval**-node and if $t\sigma \neq q$, then there is a rule path from t to some term $\hat{t}$ which is labelled by the substitutions $\sigma_1, \dots, \sigma_m$ such that $\sigma = \sigma_1 \dots \sigma_m \tau$ and $\mathbf{ev}(\hat{t})\tau \to^*_{\bigcup_{s\in\mathbf{con}(\hat{t})}\mathbf{rl}(s)} q'$ for some substitution τ and some term q' with $q \Rrightarrow_{\mathsf{H}}^* q'$.*

Proof. The lemma is proved by induction. The induction relation is obtained using the lexicographic combination of the length of the reduction $t\sigma \to_{\mathsf{H}}^* q$ and the edge relation of the graph which results from G by removing all outgoing edges of ***Eval***-nodes. (Due to the condition on the ***Ins***-rule in Def. 4, after removing these edges, the remaining graph is acyclic. Hence, the edge-relation is well founded.) In other words, as induction hypothesis we may assume that the lemma holds for all terms $\tilde{t}$ where the corresponding reduction $\tilde{t}\sigma \to_{\mathsf{H}}^* q$ is shorter or where the reduction has the same length but $\tilde{t}$ is a child of the original term t where t is not an ***Eval***-node.

We first consider the case where $t\sigma = q$. Here, we choose $q' = \mathbf{ev}(t)\sigma$ and by Lemma 14 we obtain $q = t\sigma \Rrightarrow_{\mathsf{H}}^* \mathbf{ev}(t)\sigma = q'$. Thus (a) is fulfilled and (b) is trivially true. So, in the remainder of the proof we assume that $t\sigma \neq q$ and as the reduction is necessary, we know that the head of q is a constructor. We perform case analysis according to the expansion rule applied to generate t's children.

Leaf

If t is a leaf then t is either an error term, a constructor, or a variable x. In the first two cases, $t\sigma$ cannot be reduced with $\to_{\mathsf{H}}$ and hence, we obtain $t\sigma = q$ which is a contradiction. In the last case we also have $t\sigma = q$, as the reduction is necessary and as σ is evaluated enough. Hence, $\mathrm{drop}(t\sigma) = \mathrm{drop}(x\sigma) = x\sigma_{\mathrm{drop}} = t\sigma_{\mathrm{drop}} \to_{\mathsf{H}}^* q'$ for some term q' with $\mathrm{undrop}(q') = q$.

To prove that $t\sigma = q$ also holds in the last case, we show the following claim for arbitrary terms s, p, and p': if $s \to_{\mathsf{H}}^* p$ is a necessary reduction and $\mathrm{drop}(s) \to_{\mathsf{H}}^* p'$ with $\mathrm{undrop}(p') = p$, then $s = p$. Then we use this result for $s = t\sigma$, $p = q$, and $p' = q'$. We perform induction on s:

- If s has a functional type then we obtain $s = p$ by the definition of necessary reductions.
- If $s = (c\ s_1 \dots s_n)$ for some constructor c of arity n, then $p = (c\ p_1 \dots p_n)$ where $s_i \to_{\mathsf{H}}^* p_i$ are necessary reductions. We have $\mathrm{drop}(s) = (c\ \mathrm{drop}(s_1) \dots \mathrm{drop}(s_n)) \to_{\mathsf{H}}^* (c\ \mathrm{drop}(p_1') \dots \mathrm{drop}(p_n')) = p'$ where $p = \mathrm{undrop}(p') = (c\ \mathrm{undrop}(p_1') \dots \mathrm{undrop}(p_n'))$. Since $\mathrm{drop}(s_i) \to_{\mathsf{H}}^* p_i'$ and $\mathrm{undrop}(p_i') = p_i$, the induction hypothesis implies $s_i = p_i$ and hence, $s = p$.

[16] Here, σ may have to be extended to the fresh variables in $\mathbf{ev}(t)$ or $\mathbf{ev}(\hat{t})$ that do not occur in t.

- Otherwise, we have $\mathrm{drop}(s) = x_s$. From $\mathrm{drop}(s) \to_{\mathsf{H}}^* p'$ we conclude $p' = x_s$. Hence, $p = \mathrm{undrop}(p') = \mathrm{undrop}(x_s) = s$.

Eval

If t is an ***Eval***-node with child $\tilde{t}$, then $\mathbf{ev}(t) = \mathbf{ev}(\tilde{t})$, $\mathbf{con}(t) = \mathbf{con}(\tilde{t})$, and $t \to_{\mathsf{H}} \tilde{t}$.

Since every evaluation of $t\sigma$ has to start with this evaluation step, we have the reduction $t\sigma \to_{\mathsf{H}} \tilde{t}\sigma \to_{\mathsf{H}}^* q$ where the reduction of $\tilde{t}\sigma \to_{\mathsf{H}}^* q$ is shorter than the reduction of $t\sigma$ to q. Moreover, the reduction fulfills the requirements of Lemma 18: By Def. 17, it is obvious that the reduction is necessary. As σ is evaluated enough in the original reduction, we obtain $t\sigma_{\mathrm{drop}} \to_{\mathsf{H}}^* p$ where $\mathrm{undrop}(p) = q$. The first step in this reduction is $t\sigma_{\mathrm{drop}} \to_{\mathsf{H}} \tilde{t}\sigma_{\mathrm{drop}}$ and then $\tilde{t}\sigma_{\mathrm{drop}}$ is further reduced to p. Thus, we may conclude $\tilde{t}\sigma_{\mathrm{drop}} \to_{\mathsf{H}}^* p$ which proves that σ is evaluated enough in the reduction of $\tilde{t}\sigma \to_{\mathsf{H}}^* q$. Hence, we can use the induction hypotheses for (a) and (b).

By the induction hypothesis for (a) we have

(a) $\mathbf{ev}(\tilde{t})\sigma \to^*_{\bigcup_{s\in\mathbf{con}(\tilde{t})} \mathbf{rl}(s)} q'$ and thus, $\mathbf{ev}(t)\sigma \to^*_{\bigcup_{s\in\mathbf{con}(t)} \mathbf{rl}(s)} q'$

for some term q' with $q \Rightarrow_{\mathsf{H}}^* q'$.

Next we prove (b). If $\tilde{t}$ is no ***Case***- or ***Eval***-node, then the edge from t to $\tilde{t}$ is a rule path. So by defining $\hat{t} = \tilde{t}$ and $\tau = \sigma$, (b) follows from the induction hypothesis for (a). Thus, here we need (a) in order to prove (b).

Otherwise, if $\tilde{t}$ is a ***Case***- or ***Eval***-node, then we know that the head of $\tilde{t}$ is defined. As the reduction $t\sigma \to_{\mathsf{H}}^* q$ is necessary, the head of q is a constructor and thus, we have $\tilde{t}\sigma \neq q$. By the induction hypothesis for (b) there is a rule path from $\tilde{t}$ to $\hat{t}$ satisfying the conditions in (b). Then there is also a corresponding rule path from t to $\hat{t}$, which proves (b).

Case

If t is a ***Case***-node, then $\mathbf{ev}(t) = t$ and $\mathbf{con}(t) = \{t\}$. Moreover, $t|_{\mathbf{e}(t)}$ is a variable x. As σ is evaluated enough, the term $\sigma(x)$ must be of the form "$c\, t_1 \dots t_n$" for some constructor c of arity n.

One of the children of t is $t\delta$ with $\delta = \{x/(c\ x_1 \dots x_n)\}$ for fresh variables $x_1, \dots, x_n$. Let σ' be like σ, but on these fresh variables we define $\sigma'(x_i) = t_i$ for all i. Then we have $\sigma = \delta\sigma'$. Thus, $t\sigma = t\delta\sigma' \to_{\mathsf{H}}^* q$ is a necessary reduction. We obtain $t\delta\ \sigma'_{\mathrm{drop}} = t\delta\ \sigma_{\mathrm{drop}}\ [x_1/\mathrm{drop}(t_1), \dots, x_n/\mathrm{drop}(t_n)] = t\sigma_{\mathrm{drop}}$. Therefore, since σ is evaluated enough in the original reduction $t\sigma \to_{\mathsf{H}}^* q$, σ' is evaluated enough in the reduction $t\delta\ \sigma' \to_{\mathsf{H}}^* q$.

Since $t\delta$ is smaller than t (it is the child of t) and since the reduction has the same length, the induction hypothesis for (b) now implies that there is a rule path from $t\delta$ to some term $\hat{t}$ which is labelled by $\sigma_1, \dots, \sigma_m$ such that $\sigma' = \sigma_1 \dots \sigma_m \tau$ and $\mathbf{ev}(\hat{t})\tau \to^*_{\bigcup_{s\in\mathbf{con}(\hat{t})} \mathbf{rl}(s)} q'$ for some term q' with $q \Rightarrow_{\mathsf{H}}^* q'$. Hence, there is also a rule path from t to $\hat{t}$ which is labelled by $\delta, \sigma_1, \dots, \sigma_m$ such that $\sigma = \delta\sigma' = \delta\sigma_1 \dots \sigma_m \tau$ which proves (b).

Note that the rule "$t\delta\sigma_1 \ldots \sigma_m \to \mathbf{ev}(\hat{t})$" is contained in $\mathbf{rl}(t)$ and that $\mathbf{rl}(s) \subseteq \mathbf{rl}(t)$ for all $s \in \mathbf{con}(\hat{t})$. Hence, we can now prove (a):

$$\mathbf{ev}(t)\sigma = t\sigma = t\delta\sigma_1 \ldots \sigma_m\tau \to_{\mathbf{rl}(t)} \mathbf{ev}(\hat{t})\tau \to^*_{\mathbf{rl}(t)} q'.$$

Thus, here we need (b) in order to prove (a).

VarExp

If t is a ***VarExp***-node then $t\sigma$ has a functional type and therefore $t\sigma = q$ which is a contradiction.

ParSplit, where head(t) is a constructor

Now we have $t = (c\ t_1 \ldots t_n)$ and $q = (c\ q_1 \ldots q_n)$ with $t_i\sigma \to^*_{\mathsf{H}} q_i$ for all i. As $t\sigma \neq q$ and as the original reduction is necessary, the arity of c must be n. By Def. 17, all the reductions $t_i\sigma \to^*_{\mathsf{H}} q_i$ are necessary. Moreover, $t\sigma_{\text{drop}} = (c\ t_1\sigma_{\text{drop}} \ldots t_n\sigma_{\text{drop}}) \to^*_{\mathsf{H}} (c\ p_1 \ldots p_n)$ where undrop$(p_i) = q_i$ for all i. This implies that σ is evaluated enough in every reduction $t_i\sigma \to^*_{\mathsf{H}} q_i$ as $t_i\sigma_{\text{drop}} \to^*_{\mathsf{H}} p_i$. We have $\mathbf{ev}(t) = (c\ \mathbf{ev}(t_1) \ldots \mathbf{ev}(t_n))$ and $\mathbf{con}(t) = \mathbf{con}(t_1) \cup \ldots \cup \mathbf{con}(t_n)$. Since the reductions $t_i\sigma \to^*_{\mathsf{H}} q_i$ have at most the same length as the reduction from $t\sigma$ to q and since t_i is smaller than t (it is a child of t), the induction hypothesis implies

(a) $\mathbf{ev}(t)\sigma = (c\ \mathbf{ev}(t_1)\sigma\ \ldots \mathbf{ev}(t_n)\sigma) \to^*_{\bigcup_{s\in\mathbf{con}(t_1)\cup\ldots\cup\mathbf{con}(t_n)=\mathbf{con}(t)} \mathbf{rl}(s)} c\ q'_1\ \ldots q'_n$

for terms q'_i with $q_i \Rightarrow^*_{\mathsf{H}} q'_i$. By defining $q' = (c\ q'_1\ \ldots q'_n)$ we therefore obtain $q \Rightarrow^*_{\mathsf{H}} q'$, as desired.

ParSplit, where head(t) is a variable

Now we have $\mathbf{ev}(t) = x$ for a fresh variable x. By extending σ such that $\sigma(x) = q$, we obtain

(a) $\mathbf{ev}(t)\sigma = x\sigma = q \to^*_{\bigcup_{s\in\mathbf{con}(t)} \mathbf{rl}(s)} q.$

Ins

If t is an ***Ins***-node, then we have $t = \tilde{t}[x_1/t_1, \ldots, x_n/t_n]$. If $\tilde{t} = (x\ y)$ then $\mathbf{ev}(t)$ is a fresh variable and we proceed as in the previous case. Otherwise, due to the condition on the ***Ins***-rule in Def. 4, $\tilde{t}$ is an ***Eval***- or a ***Case***-node. Without loss of generality, we assume that $x_1, \ldots, x_n$ are fresh variables not occurring in t or in the domain of σ. Then we obtain $t\sigma = \tilde{t}\sigma[x_1/t_1\sigma, \ldots, x_n/t_n\sigma] \to^*_{\mathsf{H}} q$.

Now instead of the above reduction, we start with first evaluating the subterms $t_i\sigma$ "as much as ever needed in the reduction $t\sigma \to^*_{\mathsf{H}} q$". In this way, each $t_i\sigma$ is evaluated to a term s_i. For a precise definition of s_i, consider the reduction $\tilde{t}\sigma[x_1/t_1\sigma, \ldots, x_n/t_n\sigma] \to^*_{\mathsf{H}} q$. Initially we choose $s_i = t_i\sigma$. If the above reduction is also possible with $\tilde{t}\sigma[x_1/\text{drop}(s_1), \ldots, x_n/\text{drop}(s_n)]$ (yielding a term p with undrop$(p) = q$), then we have found our final terms s_i. Note that then the substitution $\sigma\ [x_1/s_1, \dots, x_n/s_n]$ is evaluated enough in this reduction. Otherwise, at some point the evaluation of $\tilde{t}\sigma[x_1/\mathrm{drop}(s_1), \dots, x_n/\mathrm{drop}(s_n)]$ gets stuck as we have to evaluate a variable x_r that was introduced by applying drop on some s_i. Then we replace s_i by evaluating it further. More precisely, instead of x_r, the reduction requires a term of base type with a constructor as head. In the original reduction, we were able to continue the evaluation. Therefore, the term r must be reducible to a term of the form $(c\ l_1 \dots l_k)$. Then, we replace the subterm r of s_i by $(c\ l_1 \dots l_k)$ and can continue behind the point where we got stuck before. The reason is that for the old definition of s_i we got $\mathrm{drop}(s_i) = C[\mathrm{drop}(r)] = C[x_r]$, but for the new definition of s_i we obtain $\mathrm{drop}(s_i) = C[\mathrm{drop}(c\ l_1 \dots l_k)] = C[c\ \mathrm{drop}(l_1) \dots \mathrm{drop}(l_k)]$ and hence the required constructor c is present. In this way we re-define every s_i until the constructors that are necessary for the reduction are present in each s_i. Note that by construction the reductions $t_i\sigma \to_{\mathsf{H}}^* s_i$ are all necessary according to Def. 17. Furthermore, as σ was evaluated enough to reduce $t\sigma \to_{\mathsf{H}}^* q$ and as this reduction includes all reductions $t_i\sigma \to_{\mathsf{H}}^* s_i$, the substitution σ is evaluated enough in the reduction $t_i\sigma \to_{\mathsf{H}}^* s_i$, too. The length of the reduction $t_i\sigma \to_{\mathsf{H}}^* s_i$ obviously has at most the same length as the original reduction $t\sigma \to_{\mathsf{H}}^* q$. Thus, we can apply the induction hypothesis for these reductions as every t_i is a child of t.

Note that in this way we also obtain the necessary reduction $\tilde{t}\sigma[x_1/s_1, \dots, x_n/s_n] \to_{\mathsf{H}}^* q'$ for a term q' with $q \Rightarrow_{\mathsf{H}}^* q'$, where the reduction $\tilde{t}\sigma[x_1/s_1, ..., x_n/s_n] \to_{\mathsf{H}}^* q'$ has at most the same length as the original reduction $t\sigma \to_{\mathsf{H}}^* q$. The substitution $\sigma\ [x_1/s_1, \dots, x_n/s_n]$ is evaluated enough by construction.[17]

For every reduction $t_i\sigma \to_{\mathsf{H}}^* s_i$ the induction hypothesis (a) implies

$$\mathbf{ev}(t_i)\sigma \to^*_{\bigcup_{s \in \mathbf{con}(t_i)} \mathbf{rl}(s)} s_i' \text{ for some terms } s_i' \text{ with } s_i \Rightarrow_{\mathsf{H}}^* s_i'. \tag{9}$$

Moreover, a further reduction of s_i to s_i' does not destroy the above properties, i.e., for the substitution $\sigma' = \sigma\ [x_1/s_1', \dots, x_n/s_n']$ we still have $\tilde{t}\sigma' \to_{\mathsf{H}}^* q''$ for a term q'' with

$$q \Rightarrow_{\mathsf{H}}^* q''. \tag{10}$$

Again, the reduction $\tilde{t}\sigma' \to_{\mathsf{H}}^* q''$ is necessary, it has at most the same length as the original reduction $t\sigma \to_{\mathsf{H}}^* q$, and the substitution σ' is evaluated enough.

Note that the reduction $\tilde{t}\sigma' \to_{\mathsf{H}}^* q''$ has length greater zero, as $\mathrm{head}(q) = \mathrm{head}(q'')$ is a constructor and $\mathrm{head}(\tilde{t})$ is defined. As $\tilde{t}$ is an ***Eval***- or a ***Case***-node we use induction hypothesis (b) for the child $\tilde{t}$ of t. We obtain a node $\hat{t}$ and a substitution τ such that there is a rule path from $\tilde{t}$ to $\hat{t}$ labelled by substitutions $\sigma_1', \dots, \sigma_m'$ where $\sigma' = \sigma_1' \dots \sigma_m'\tau$ and

$$\mathbf{ev}(\hat{t})\tau \to^*_{\bigcup_{s \in \mathbf{con}(\hat{t})} \mathbf{rl}(s)} q' \text{ for some term } q' \text{ with } q'' \Rightarrow_{\mathsf{H}}^* q'. \tag{11}$$

[17] Note that we cannot guarantee $q = q'$. As an example consider the rules $\mathsf{f} = \mathsf{d}$ and $\mathsf{g}\ x = \mathsf{C}\ x\ x$ and the terms $t = (\mathsf{g}\ \mathsf{f})$, $\tilde{t} = (\mathsf{g}\ x_1)$, and $q = (\mathsf{C}\ \mathsf{d}\ \mathsf{f})$. Then the construction forces us to first evaluate $t_1 = \mathsf{f}$ to $s_1 = \mathsf{d}$. Thus, $\tilde{t}[x_1/s_1] = (\mathsf{g}\,\mathsf{d})$. But then the final result q' is $(\mathsf{C}\ \mathsf{d}\ \mathsf{d})$ which is further evaluated than q.

So the rule $\tilde{t}\sigma'_1 \dots \sigma'_m \to \mathbf{ev}(\hat{t})$ is included in $\mathbf{rl}(\tilde{t})$. As $\mathbf{rl}(s) \subseteq \mathbf{rl}(\tilde{t})$ for all $s \in \mathbf{con}(\hat{t})$ we can now prove the desired statement (a).

$$\begin{array}{lll}
\mathbf{ev}(t)\sigma = & \tilde{t}\sigma[x_1/\mathbf{ev}(t_1)\sigma, \dots, x_n/\mathbf{ev}(t_n)\sigma] & \\
\to^*_{\bigcup_{s\in\mathbf{con}(t_1)\cup\dots\cup\mathbf{con}(t_n)} \mathbf{rl}(s)} & \tilde{t}\sigma[x_1/s'_1, \dots, x_n/s'_n] & \text{by (9)} \\
= & \tilde{t}\sigma' & \\
= & \tilde{t}\sigma'_1 \dots \sigma'_m\tau & \\
\to_{\mathbf{rl}(\tilde{t})} & \mathbf{ev}(\hat{t})\tau & \\
\to^*_{\mathbf{rl}(\tilde{t})} & q' & \text{by (11)}
\end{array}$$

Thus, $\mathbf{ev}(t)\sigma \to^*_{\bigcup_{s\in\mathbf{con}(t)} \mathbf{rl}(s)} q'$. Note that we again needed (b) to prove (a). As desired, we have

$$\begin{array}{rl}
q \Rightarrow^*_{\mathsf{H}} q'' & \text{by (10)} \\
\Rightarrow^*_{\mathsf{H}} q' & \text{by (11)}
\end{array}$$

□

Now we can prove the lemma about the semantics of **dp**. It states the desired connection between the DP problems obtained with **dp** and the termination of the nodes in the termination graph. More precisely, it shows that every non-H-terminating node corresponds to an infinite path in the termination graph and that the rules in a DP problem can be used for the step from the last node of a DP path to the first node in the next DP path.

In order to simplify the proof we introduce a new relation $\hookrightarrow_{\mathsf{H}}$ which we use instead of H-termination. The advantage is that then we can use the notion of "*evaluated enough* substitutions" for $\hookrightarrow_{\mathsf{H}}$-reductions which is not directly possible for H-termination.

Definition 19 **($\hookrightarrow_{\mathsf{H}}$).** *We define $s \hookrightarrow_{\mathsf{H}} t$ iff*

(a) $s \to_{\mathsf{H}} t$,
(b) $s = (f\ s_1 \dots s_n)$ *for a defined function symbol* f *with* $n < \text{arity}(f)$ *and* $t = (f\ s_1 \dots s_n\ t')$ *for an* H*-terminating term* t', *or*
(c) $s = (c\ t_1 \dots t_n)$ *for a constructor* c *and* $t = t_i$ *for some* i.

By Def. 3 it is easy to see that a ground term is not H-terminating iff it starts an infinite $\hookrightarrow_{\mathsf{H}}$-reduction. For a non-ground term t one can only conclude that t terminates w.r.t. $\hookrightarrow_{\mathsf{H}}$ if t is H-terminating, but not vice versa.

Lemma 20 (Properties of dp). *Let G be a termination graph and let s be a node in G. Let σ be a substitution such that $s\sigma$ starts an infinite $\hookrightarrow_{\mathsf{H}}$-reduction, where σ is evaluated enough in this $\hookrightarrow_{\mathsf{H}}$-reduction and where $\sigma(x)$ is terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$ for all variables x. Then there is a path (possibly of length zero) from s to an **Ins**-node $t = \tilde{t}[x_1/t_1, \dots, x_n/t_n]$ labelled with $\sigma_1, \dots, \sigma_m$ and an instantiation edge from t to a node $\tilde{t}$ such that*

– *$\sigma = \sigma_1 \dots \sigma_m\tau$ for some substitution τ such that $t\tau$ is not $\hookrightarrow_{\mathsf{H}}$-terminating*[18]

[18] Again, σ may have to be extended to fresh variables x in t that do not occur in s.

– $\mathbf{ev}(t)\tau \xrightarrow{>\varepsilon}{}^{*}_{\mathcal{R}} \tilde{t}\mu$ *for* $\mathcal{R} = \bigcup_{s \in \mathbf{con}(t_1) \cup \ldots \cup \mathbf{con}(t_n)} \mathbf{rl}(s)$ *such that* $\tilde{t}\mu$ *starts an infinite* $\hookrightarrow_{\mathsf{H}}$*-reduction and* μ *is a substitution which is evaluated enough in this* $\hookrightarrow_{\mathsf{H}}$*-reduction and all* $\mu(x)$ *are terminating w.r.t.* $\hookrightarrow_{\mathsf{H}}$*.*

Proof. The lemma is proved by induction on the edge-relation in the graph obtained from G by removing all instantiation edges. In other words, as induction hypothesis we may assume that the lemma holds for all children of the original term s (except those which are only reachable via instantiation edges). We perform case analysis according to the expansion rule applied on s.

Leaf

If s is a leaf, then s is an error term, a constructor, or a variable x. In the first two cases, $s\sigma$ is a normal form w.r.t. $\hookrightarrow_{\mathsf{H}}$. In the last case, the term $s\sigma = \sigma(x)$ is terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$ by the requirements on σ. Hence, $s\sigma$ cannot start an infinite $\hookrightarrow_{\mathsf{H}}$-reduction which gives a contradiction.

Eval

If s is an ***Eval***-node with child $\tilde{s}$, then we have $s \rightarrow_{\mathsf{H}} \tilde{s}$ and thus, the infinite $\hookrightarrow_{\mathsf{H}}$-reduction of $s\sigma$ has to start with $s\sigma \hookrightarrow_{\mathsf{H}} \tilde{s}\sigma$. Hence, $\tilde{s}\sigma$ is also not terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$ and σ is evaluated enough in the infinite $\hookrightarrow_{\mathsf{H}}$-reduction. Since $\tilde{s}$ is smaller than s (it is the child of s), the induction hypothesis implies the lemma.

Case

If s is a ***Case***-node, then $s|_{\mathsf{e}(s)}$ is a variable x. As σ is evaluated enough and as we have an infinite $\hookrightarrow_{\mathsf{H}}$-reduction, $\sigma(x)$ must be of the form $(c\ s_1 \ldots s_n)$ for a constructor c.

One of the children of s is $s\delta$ with $\delta = \{x/(c\ x_1 \ldots x_n)\}$ for fresh variables $x_1, \ldots, x_n$. So we have $\sigma = \delta\sigma'$ for the substitution σ' with $\sigma'(x_i) = s_i$. Then we obtain $s\sigma = s\delta\sigma'$, i.e., $s\delta\sigma'$ also starts an infinite $\hookrightarrow_{\mathsf{H}}$-reduction. We argue in the same way as for ***Case***-nodes in the proof of Lemma 18 to show that σ' is evaluated enough in the infinite $\hookrightarrow_{\mathsf{H}}$-reduction of $s\delta\sigma'$.

Since $s\delta$ is smaller than s (it is the child of s), the lemma is implied by the induction hypothesis.

VarExp

If s is a ***VarExp***-node, then the infinite $\hookrightarrow_{\mathsf{H}}$-reduction starts with $s\sigma \hookrightarrow_{\mathsf{H}} s\sigma\ s'$ for some H-terminating term s'. Recall that H-termination implies termination w.r.t. $\hookrightarrow_{\mathsf{H}}$. Hence, s' cannot start an infinite $\hookrightarrow_{\mathsf{H}}$-reduction and w.l.o.g. we assume that s' is evaluated "as much as ever needed in the infinite reduction of $s\sigma\ s'$", cf. the proof of Lemma 18 for ***Ins***-nodes. (Alternatively, one can also replace s' by its normal form w.r.t. $\rightarrow_{\mathsf{H}}$.) Let "$s\ x$" be the child of s. Then we extend σ to the fresh variable x by defining $\sigma(x) = s'$. Thus, $(s\ x)\sigma$ is not terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$ and "$s\ x$" is smaller than s (it is the child of s). Moreover, by construction σ remains evaluated enough in the infinite reduction of "$s\ x$" and instantiates all variables with terms that terminate w.r.t. $\hookrightarrow_{\mathsf{H}}$. Thus, the lemma follows from the induction hypothesis.

ParSplit

Now we have $s = (c\ s_1 \ldots s_n)$ for a constructor c or $s = (x\ s_1 \ldots s_n)$ for a variable x. Since $s\sigma$ is not terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$, there must be a s_i such that $s_i\sigma$ is not terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$ either. Since s_i is smaller than s (it is the child of s), the lemma again follows from the induction hypothesis.

Ins

Now we have $s = \tilde{s}[x_1/s_1, \ldots, x_n/s_n]$ and the children of s are $s_1, \ldots, s_n$ and $\tilde{s}$.

We first regard the case that there is an $1 \leq i \leq n$ where $s_i\sigma$ starts an infinite $\hookrightarrow_{\mathsf{H}}$-reduction. Since s_i is smaller than s (it is the child of s), the lemma again follows from the induction hypothesis. Note that if $\tilde{s} = (x_1\ x_2)$ then we always are in this case. The reason is the same as in the ***ParSplit***-case where the head is a variable: If both $s_1\sigma$ and $s_2\sigma$ are terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$, then this is also the case for $(s_1\sigma\ s_2\sigma) = s\sigma$.

Now we regard the case where all $s_i\sigma$ are terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$. We know $\tilde{s} \neq (x_1\ x_2)$ and hence $\mathbf{ev}(s) = \tilde{s}[x_1/\mathbf{ev}(s_1), \ldots, x_n/\mathbf{ev}(s_n)]$. Let $t = s$, $t_i = s_i$, $\tilde{t} = \tilde{s}$, and $\tau = \sigma$. Without loss of generality, we assume that $x_1, \ldots, x_n$ are fresh variables not occurring in t or in the domain of $\tau = \sigma$. Then $s\sigma = t\tau = \tilde{t}\tau[x_1/t_1\tau, \ldots, x_n/t_n\tau]$ starts an infinite $\hookrightarrow_{\mathsf{H}}$-reduction. Clearly, we also have an infinite $\hookrightarrow_{\mathsf{H}}$-reduction for any term which results from $\tilde{t}\tau[x_1/t_1\tau, \ldots, x_n/t_n\tau]$ by first reducing $t_i\tau$ "as much as ever needed in the infinite reduction", cf. the proof of Lemma 18 in case of ***Ins***-nodes. In this way, each $t_i\tau$ evaluates to a term q_i.

By construction, the reduction $t_i\tau \to_{\mathsf{H}}^* q_i$ is necessary and τ is evaluated enough. Hence, Lemma 18 (a) implies $\mathbf{ev}(t_i)\tau \to^*_{\bigcup_{s \in \mathbf{con}(t_i)} \mathbf{rl}(s)} q_i'$ for some term q_i' with $q_i \Rightarrow_{\mathsf{H}}^* q_i'$.

Let μ be like τ, but on the variables $x_1, \ldots, x_n$ we define $\mu(x_i) = q_i'$. Then again, $\tilde{t}\mu$ starts an infinite $\hookrightarrow_{\mathsf{H}}$-reduction. Moreover, by construction μ is evaluated enough and terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$. $\square$

Finally, we can prove the main soundness theorem for our approach. Here, $\xrightarrow{\varepsilon}$ denotes a rewrite step at top position and $\xrightarrow{>\varepsilon}$ denotes a rewrite step which is strictly below the top position.

Theorem 12 (Soundness). *Let G be a termination graph. If the DP problem $\mathbf{dp}_{G'}$ is finite for all SCCs G' of G, then all nodes t in G are H-terminating. More precisely, if there is a non-H-terminating node in G, then there exists an SCC G' with a DP problem $\mathbf{dp}_{G'} = (\mathcal{P}, \mathcal{R})$ such that there is an infinite reduction of the form*

$$s_1 \xrightarrow{\varepsilon}_{\mathcal{P}} t_1 \xrightarrow{>\varepsilon}{}^*_{\mathcal{R}} s_2 \xrightarrow{\varepsilon}_{\mathcal{P}} t_2 \xrightarrow{>\varepsilon}{}^*_{\mathcal{R}} \ldots$$

This implies that there is also an infinite reduction of the form

$$s_1 \xrightarrow{\varepsilon}_{\mathcal{P}^\sharp} t_1 \xrightarrow{>\varepsilon}{}^*_{\mathcal{R}} s_2 \xrightarrow{\varepsilon}_{\mathcal{P}^\sharp} t_2 \xrightarrow{>\varepsilon}{}^*_{\mathcal{R}} \ldots$$

In other words, the DP problem $(\mathcal{P}^\sharp, \mathcal{R})$ is not finite either.

Proof. If t is not H-terminating, then there is a substitution σ with H-terminating terms such that $t\sigma$ is a non-H-terminating ground term and thus, a non-$\hookrightarrow_{\mathsf{H}}$-terminating ground term. As $\sigma(x)$ is H-terminating for all x, we may assume that σ is a normal substitution (i.e., $\sigma(x)$ is a normal form w.r.t. $\to_{\mathsf{H}}$ for all variables x). Then, σ is evaluated enough in every $\to_{\mathsf{H}}$-reduction and also in every $\hookrightarrow_{\mathsf{H}}$-reduction. As H-termination implies termination w.r.t. $\hookrightarrow_{\mathsf{H}}$, we also know that $\sigma(x)$ is terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$ for all variables x.

By Lemma 20 there is an infinite path in G. Since G is finite, this path must end in some SCC G'. We regard the infinite tail of this path which only traverses nodes and edges of G'. By Lemma 20, there must be an infinite sequence of nodes $s^1, t^1, s^2, t^2, \ldots$ and substitutions $\sigma^1, \sigma^2, \sigma^3, \ldots$ such that for all i

- the path from s^i to t^i is a DP path in G' labelled with $\sigma^i_1, \ldots, \sigma^i_{m_i}$ (thus $\mathcal{P}$ contains the rule $s^i\sigma^i_1 \ldots \sigma^i_{m_i} \to \mathbf{ev}(t^i)$)
- $s^i\sigma^i$ is not terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$
- $\sigma^i = \sigma^i_1 \ldots \sigma^i_{m_i}\tau^i$ for substitutions τ^i
- $\mathbf{ev}(t^i)\tau^i \xrightarrow{>\varepsilon}{}^*_{\mathcal{R}} s^{i+1}\sigma^{i+1}$

Thus, we have

$$\begin{array}{ll} s^1\sigma^1 = & s^1\sigma^1_1 \ldots \sigma^1_{m_1}\tau^1 \xrightarrow{\varepsilon}_{\mathcal{P}} \mathbf{ev}(t^1)\tau^1 \xrightarrow{>\varepsilon}{}^*_{\mathcal{R}} \\ s^2\sigma^2 = & s^2\sigma^2_1 \ldots \sigma^2_{m_2}\tau^2 \xrightarrow{\varepsilon}_{\mathcal{P}} \mathbf{ev}(t^2)\tau^2 \xrightarrow{>\varepsilon}{}^*_{\mathcal{R}} \\ s^3\sigma^3 \ \ldots & \end{array}$$

□