Rheinisch-Westfälische Technische Hochschule Aachen
Lehr- und Forschungsgebiet Informatik II
Programmiersprachen und Verifikation

Improving efficiency and power of automated termination analysis for Haskell

Matthias Raffelsieper

Diplomarbeit

im Studiengang Informatik

vorgelegt der
Fakultät für Mathematik, Informatik und Naturwisssenschaften der
Rheinisch-Westfälischen Technischen Hochschule Aachen

im November 2007

Gutachter: Prof. Dr. Jürgen Giesl
Prof. Dr. Ulrik Schroeder

Acknowledgments

I want to thank the following people for their support throughout the creation of this thesis:

- My supervisor Prof. Dr. Jürgen Giesl, for giving me the chance to work on the AProVE team and for providing me with such an interesting topic for this diploma thesis.
- Prof. Dr. Ulrik Schroeder, for agreeing to review this diploma thesis.
- The research assistant Stephan Swiderski for his invaluable support in every aspect of this diploma thesis.
- The research assistants Carsten Fuhs, Peter Schneider-Kamp, and René Thiemann, for a lot of interesting discussions and for a productive and friendly environment.
- The whole AProVE team for a great working atmosphere.
- My fellow students Nicolas Becker, Tobias Buhr, Thorben Keller, Fekry Meawad, and Andreas Tielmann for a lot of feedback and a great time throughout my whole time at the RWTH.
- My parents Norbert and Birgit Raffelsieper, and my siblings Christoph, Thomas, and Lisa, for supporting me all the time.

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst und keine anderen als die angebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 12. November 2007,

Matthias Raffelsieper

Contents

Chapter 1

Introduction

Computers become more and more a part of everyday's life. In the last few decades, the computer has evolved from a technical tool with unclear practical use to an essential piece in our society. This evolution took place, because the *software*, i.e., the programs that are run on computers, could be adapted to first solve all tedious tasks, then to solve tasks that were too complex before. Therefore, it is the versatility through software of modern computers that lead to their success.

Software that is executed by a computer is still being written in most parts by humans. Only a very small amount of software can be generated automatically these days. Since humans will always make small mistakes, one has to ensure that the programs will contain as few mistakes (often called *bugs*) as possible. One approach to avoid bugs is testing, where a number of input values is provided and the output of the program is compared against the expected output. However, testing can never be complete, i.e., there will always be cases which are not covered by a test.

Since computers and software have become so ubiquitous, and since software has sometimes advantages over humans, such as higher speed at certain operations, no concentration issues, etc., computers are also used in highly critical environments, such as airplanes, financial transactions, medical purposes, etc. For these environments, testing the software is not enough. One wants to make sure that no bugs exist in the used components. While for hardware, redundancy is an often employed approach, for software so-called *verification* is used. This is, because for hardware it is usually assumed that the redundant parts fail independently of each other. However, if one simply replicates the software, then bugs in the software are not independent, i.e., upon a wrong input, all instances will then exhibit the same bug. Software verification tries to formally prove that a program does what it was specified to do. This correctness proof is divided into two proofs: First, one shows *partial correctness*, i.e., if an algorithm outputs a value, then this is the correct value. The other part of the proof is, that the algorithm actually always outputs a result for all inputs. To show this latter property, which is called *termination*, a numerous amount of techniques have been developed. Such a technique can never prove termination for all algorithms, this is because it has been proven that termination of a problem is generally undecidable [Tur36]. Most of these techniques focus on *Term Rewrite Systems*, a formalism that is intentionally kept small, but it still allows to for-

mulate every algorithm in it. A description of Term Rewrite Systems is given in [BN98], for example. For Term Rewrite Systems, quite a few automatic tools exist that show termination. One of the most successful tools in this area is the automated termination prover AProVE [GTSKF04, GSKT06], which has won all of the four international termination competitions [MZ07] for Term Rewrite Systems.

However, no algorithm is developed in this formalism. Instead, programmers use high-level programming languages. Therefore, it is an interesting question how to apply the existing techniques for Term Rewrite Systems to these programming languages.

This has recently been the field of quite some investigation. A step towards these programming languages is given in [Has04], where a functional and an imperative programming language are introduced which resembled some of the features modern programming languages have. These programming languages are directly translated into Term Rewrite Systems, for which termination is then tried to be shown. In [Swi05, GSSKT06], a technique was presented which works on full Haskell 98 [Jon03]. Here, from Haskell programs so called *Dependency Pairs* [AG00] are generated directly, which is a technique originally developed for Term Rewrite Systems. This is an advantage over a previous approach presented in [PSS97], where it was tried to directly find an ordering, which worked only for special cases. For imperative programs, a similar approach for a restricted set of Java programs was presented in [Son07].

Another approach to prove termination of the practically used logic programming language Prolog by employing techniques for Term Rewrite Systems, was presented in [Käu05, SKGST07]. For Prolog, it is interesting that tools implementing direct approaches are available which can be compared with the afore mentioned technique. The results of such a comparison are presented in [SKGST07] as well, which indicate that termination proofs based on the presented transformation to Term Rewrite Systems are comparable in strength to the dedicated tools cTI [MB05] and TerminWeb [CT99]. The comparison shows that the transformation to Term Rewrite Systems done by AProVE is able to prove more examples terminating than the dedicated tools. However, it should be mentioned that neither the transformational approach to Term Rewrite Systems nor the direct approaches supersede the other. There are examples where the transformational approach is successful while the direct ones fail, and vice versa. Furthermore, these results show that the transformation to Term Rewrite Systems is a very powerful approach, which has the advantage of a very large research community behind it and therefore more progress is made for it.

Unfortunately, for lazy evaluating programming languages such as Haskell and for real imperative programming languages, there are no such comparisons. This is, because there are no implementations of other approaches available. This would be interesting, as at least for imperative programs there are direct approaches as well, such as for example [CPR06].

For Haskell, another possible approach would be to transform the Haskell program into a Term Rewrite System, since Term Rewrite Systems have a functional semantics, as well. However, for such an approach no good results can be expected. This is, because there are functions that intuitively would be called non-terminating. However, when these are inserted into the right context, then termination of these can be shown, due to the lazy evaluation strategy of Haskell. Thus, we analyze termination not for every function contained in a program,

but for a *start term*, i.e., a program entry point. In a Haskell program that is compiled, this is normally the defined function `main`, but in a Haskell interpreter, one is allowed to enter an arbitrary *ground* term, which is a term without variables that is to be evaluated. Therefore, we will consider termination of a start term for a given Haskell program.

The results of this thesis extend the approach presented in [GSSKT06, Swi05]. It will present a number of improvements to the automatic termination analysis for Haskell, which improve both the runtime for finding a proof and the strength of the approach.

The presentation of the improvements contained in this thesis shall be outlined briefly. In Chapter 2 the programming language Haskell for the standard Haskell 98 [Jon03] is shortly introduced. There, we will also define our notion of termination with respect to a start term. Furthermore, the notation and basic definitions of Term Rewrite Systems and Dependency Pairs are given.

Chapter 3 outlines the approach of [GSSKT06, Swi05] which is the basis we will extend in the following chapters.

First, the approach of Chapter 3 is extended in Chapter 4 to include type classes. This was already included in [Swi05], but here it will be incorporated into the notation and proofs given in [GSSKT06]. Furthermore, we will modify the approach in order to have fresh variables only inside Dependency Pairs, not inside rules.

In Chapter 5, we present a reduction of large Haskell programs to only those parts which might be needed in order to evaluate the start term. This enables us to usually leave out large parts of used libraries which provide a large set of functions, where typically only a few of these are being used.

The next extension of the approach is presented in Chapter 6. There, we care for the creation of first-order terms that are separated according to the recursion structure of the program. This is an improvement compared to the previous approach which used an applicative encoding due to higher-order terms. Such an encoding has quite a few drawbacks that make it harder to prove finiteness of such DP problems. Furthermore, we will include the type class information of the terms into the generated Dependency Pairs. This was not done in [Swi05] and ensures the non-overlappingness of the resulting rules.

Because of this non-overlappingness, we can show in Chapter 7, that it suffices to only prove absence of minimal innermost chains. This sounds counter-intuitive at first, considering that Haskell has a lazy evaluation strategy, which can be described as a leftmost-outermost strategy. But as we will show in this chapter, all of this information was already used in the previous steps and hence the easier minimal innermost chains suffice.

In Chapter 8, we then show the practical relevance of the presented improvements on a large set of start terms. For these, we try to show termination and it is shown that the number of start terms that can be proven terminating is increased from 56.67 % to 76.68 %.

As already discussed above, there are functions in Haskell that are non-terminating, but still a start term using these functions is terminating and can be shown by our approach as such. This occurs, because Haskell has a lazy evaluation strategy, where a function is only evaluated as far as needed. Such functions are called lazy-terminating. One form of it is regarded in Chapter 9. There, it will be shown how to prove a term to be lazy-terminating using the existing techniques for termination. When this could be proven, one is sure that

for all functions requiring only a lazy-terminating argument, the current start term may be used.

In the penultimate Chapter 10, we are not considering termination anymore. Instead, we show how we can prove nontermination of a start term for a given Haskell program. If a case is detected where an infinite evaluation exists, then usually a programming error has been detected. Since a counterexample is produced, a programmer can look into this counterexample and is able to fix programming errors more easily.

Finally, Chapter 11 gives a conclusion of the presented work and presents a few pointers as to what might be interesting topics in the context of the presented approach for Haskell termination analysis, where further research seems promising.

It should be noted that all of the above has also been implemented in the automatic termination prover AProVE [GTSKF04, GSKT06]. This tool accepts Haskell programs conforming to the full Haskell 98 language specification and can analyze start terms for H-Termination, Lazy-Termination, and Non-Termination using the techniques described in this thesis.

Chapter 2

Preliminaries

The reader of this thesis should be familiar with the Haskell programming language in its incorporation "Haskell 98" [Jon03]. Furthermore, the basics of term rewriting and the Dependency Pair Framework should be known to the reader, for reference see for example [BN98, GTSK05b]. The main purpose of this chapter is the introduction of the notations that will be used, therefore only a general description of these concepts will be given.

2.1 Syntax and Semantics of Haskell

The AProVE Haskell implementation, which was mainly implemented by Swiderski [Swi05], accepts the full Haskell 98 standard as defined in [Jon03]. Since Haskell is a complete programming language, its formal definition is rather long. Therefore, this section will only focus on the main components of Haskell and introduces these informally. Furthermore, we define a few properties of Haskell terms that will be used in the remainder of this thesis.

A Haskell program consists of *data definitions*, *function declarations*, and *class/instance definitions*. A data definition introduces a new algebraic, user-defined data type. An example for such a data definition would be

```
data Nats = Z | S Nats
```

Here, the data type `Nats` represents the natural numbers starting at zero by the constant `Z` for zero, and by building the successor of a natural number by prepending the constructor `S` to the given natural number.

Data types can also be polymorphic. This enables definitions of structures such as lists, where the type of the elements is irrelevant to the list. This could for example be done as follows:

```
data List a = Nil | Cons a (List a)
```

The above defines a list data type which represents the empty list with the constructor `Nil`, and the list starting with an element `x` and a tail list `xs` by the term `Cons x xs`.

For the data type `List a`, the type variable `a` determines the type of the elements. This is, because for the data constructor `Cons`, the first argument is defined to be of type `a`. Thus, a Haskell programmer is allowed to write

`List Int` if that instance of a list shall only contain values of the predefined type `Int`, and at the same time `List Nats` would be a list, where all contained elements have the type `Nats`.

Function declarations are given in rules, which are applied in order from top to bottom. A rule consists of a left-hand side and a right-hand side. The right-hand side of a rule may be an arbitrary Haskell expression; whereas for the left-hand side some restrictions apply. The first symbol on the left-hand side is a defined function symbol. This defined function symbol is followed by a number of so-called *patterns*. These patterns are linear, meaning that no variable occurs twice in the set of patterns of a rule. Furthermore, these patterns can be thought of to contain only constructors and variables. There are cases where this is not true, e.g., the `(x+`n`)`-pattern. But for user-defined types this is always the case, and the predefined special patterns can be transformed, which will be described further in chapter 3.

The following example shows two such function definitions.

Example 2.1 (`take from` example)**.**

```
data Nats   = Z   | S Nats
data List a = Nil | Cons a (List a)

from x = Cons x (from (S x))

take  Z     _           = Nil
take  _     Nil         = Nil
take (S n) (Cons x xs) = Cons x (take n xs)
```

The given example is a complete Haskell program. It first contains the declarations for the data types `Nats` and `List a`, which are then used in the definitions of the functions `take` and `from`. If a term shall be evaluated, this is performed in an ordered leftmost-outermost fashion. The ordering, as stated before, is top to bottom; i.e., the rules are tried in the order as they appear in the program. Haskell always reduces as far on the outside as possible, such that functions, which are non-terminating when called directly, can still contribute to the calculation. Such a function is `from` which builds up an infinite list of natural numbers, starting at the number given as argument. However, if this function is used as a second argument of the function `take`, where the first argument is a term having only finite evaluation sequences, then the complete evaluation is a finite run which returns a finite result.

A rule is applied to a term by checking every rule whether its left-hand side *matches* the term with the leftmost-outermost defined symbol. The matching process is performed by checking whether every variable in the left-hand side of a rule can be replaced by a subterm of the term that is to be evaluated, such that both terms are equal after this replacement. If this is the case, then the term can be replaced by the right-hand side of the Haskell rule, where all variables are replaced by the same term they were replaced with previously. Then the search for a rule stops and the process of searching a term to be evaluated starts again with the new term. In case there was no replacement such that the left-hand side of the current rule and the term that has to be evaluated exists, then the next rule is tried.

Consider for example the start term `take (S (S Z)) (from Z)`. Here, a Haskell evaluator will first try to reduce the defined function `take` with the first rule for this function. But since we have the constructor `S` on the first argument, and not the constructor `Z`, this rule is not applicable. In this case, the Haskell evaluator advances to the next rule. For the second rule, it has to be checked whether the second argument of the term matches the constructor `Nil`. But a term starting with the defined function `from` is the second argument of `take` in the term that shall be reduced. Thus, the evaluator must first reduce this term to a term starting with a constructor (a so called *weak head normal form*), in order to decide whether the current rule is applicable or not. The Haskell evaluator now searches for a rule that is applicable to this term. It finds the first and only rule for `from` and checks the arguments. Since the argument in the rule of `from` is a variable, it matches everything. So, `from Z` is replaced by the right-hand side of this rule, where the variable `x` is replaced by its instantiation `Z`. Thus, the term to evaluate further is the term `take (S (S Z)) (Cons Z (from (S Z)))`. Here, the Haskell evaluator again looks for a rule that could reduce the complete term, instead of further reducing the term starting with thefunction `from`. It will now be found that the second rule of `take` is not applicable, since the constructor `Nil` is required, but the constructor `Cons` occurs in this argument. Therefore, the evaluator advances to the third rule of `take`, whose left-hand side matches. Thus, the new term is `Cons Z (take (S Z) (from (S Z)))`. In this new term, the leftmost-outermost position with a defined function is the argument `take (S Z) (from (S Z))` of the constructor `Cons`, which will then be tried to reduce next. In a similar manner as before, this term will be reduced, such that the resulting term will be `Cons Z (Cons (S Z) (take Z (from (S (S Z)))))`. Here, again the term starting with `take` is on the leftmost-outermost position that starts with a defined symbol and must be reduced. The first rule for `take` reduces this term to `Cons Z (Cons (S Z) Nil)`, which does not contain any further defined functions and therefore is the result of this evaluation.

Every term in Haskell is statically typed. These types can be provided, but one is not required to specify them. Then a type inference will determine the most general possible type for every term. For the above example functions, the type of the function `from` would be inferred to be `from :: Nats -> List Nats`. For the function `take`, the type `take :: Nats -> List a -> List a` would be inferred. When a type is explicitly specified, this type must be at least as concrete as the inferred type. One could for example provide the type `take :: Nats -> List Nats -> List Nats`, thereby restricting `take` to lists containing only elements of type `Nats`. However, no type may be specified that is more general than the inferred type, e.g., specifying `take :: Nats -> a -> a` would not be a valid type, since a list is required on the second argument of `take`.

The *arity* of a Haskell function is defined via its type. A function f has $arity(f) = n$, if f has the type f `::` ρ_1 `->` $\ldots$ `->` ρ_n `->` ρ_{n+1}, where ρ_i does not contain the type constructor `->` for every $1 \leq i \leq n+1$.

Classes and instances thereof are used to group together data types and ensure availability of certain defined operations on these data types. An example for such classes and instances of them is given in the following.

Example 2.2 (Classes and Instances)**.**

```
data Nats   = Z     | S Nats
data Bool   = False | True
data List a = Nil   | Cons a (List a)

class HasZero a where
  getZero :: a

instance HasZero Nats where
  getZero = Z

instance HasZero Bool where
  getZero = False

class Addition a where
  plus :: a -> a -> a

instance Addition Nats where
  plus  Z    n = n
  plus (S m) n = S (plus n m)

instance Addition a => Addition (List a) where
  plus  Nil         Nil        = Nil
  plus (Cons x xs) (Cons y ys) = Cons (plus x y) (plus xs ys)

class Addition a => Multiplication a where
  mult :: a -> a -> a
```

In the example, the class `HasZero` ensures that any instance of it will implement the function `getZero`. Such a function which is defined inside a class is called a *class member*. For the different instances, it is the case that the type variable, which was used in the class definition, is instantiated with the specific type. Therefore, in our example, the function definition for the data type `Nats` must return a term of this type, while for the data type `Bool` a term of this type must be returned, as the type variable `a` is instantiated by this type.

For the class `Addition` in the above program, we see that it ensures the existence of a member `plus` for all of its instances. This function adds natural numbers by decrementing the first number and constructing a term that represents the sum of both arguments. For lists, it performs a componentwise addition of the elements. In order to be able to do this, it must ensure that also for the component type, the function `plus` is available. This is done by requiring an instance for `Addition a` in order to build the instance `Addition (List a)`.

The last class contained in the above example is the class `Multiplication` which contains a member `mult`. For this class, it is required that an instance of the class `Multiplication` must be an instance of the class `Addition` at the same time, i.e., we have defined a subclass of `Addition`.

For every type, it is valid to restrict type variables to certain classes by so called *class constraints*. An example for such a type is given in the example below.

Example 2.3 (Class Constraints)**.**

```
isZero :: (HasZero a, Eq a) => a -> Bool
isZero x = x == getZero
```

Here, the type for the argument `x` *is restricted to such types that are an instance of the two classes* `HasZero` *and* `Eq` *(which is a predefined class in Haskell providing the function* `==` *used for the comparison).*

In Haskell 98, a class may only have one type variable as argument. When an instance shall be defined, then only the head symbol of the instance type may be a type constructor, the rest must be type variables. Furthermore, every class constraint that restricts an instance must only contain a single type variable. This makes the calculation of instances for a ground term decidable, since a class can be understood as a set of types, where for all of these types certain operations exist. As seen in the example above, classes may be structured hierarchically, i.e., one can define classes to be a subset of another class. Here, it is allowed to even define classes as a subset of the intersection of a set of other classes, i.e., a class may have multiple superclasses.

As stated before, Haskell considers the rules of a program in top-to-bottom order, and applies rules as far outside as possible. To identify subterms of a term, we define the set of *positions* $Occ(t)$ for a term t.

Definition 2.4 (Positions $Occ(t)$)**.** *For any term t, it holds that $\epsilon \in Occ(t)$ and if $t = (t_1\ t_2)$, then $\{1\pi_1, 2\pi_2 \mid \pi_1 \in Occ(t_1), \pi_2 \in Occ(t_2)\} \subseteq Occ(t)$.*

In Haskell, terms are constructed using the function "juxtaposition" only. This is the reason why positions only consist of the numbers "1" and "2". The function "juxtaposition" associates to the left, for example it holds that `take n xs = (take n) xs`. Among the positions of a term, one position identifies the subterm that has to be evaluated next. This position is called the *evaluation position* of a term.

Definition 2.5 (Evaluation position $\mathbf{e}(t)$, [GSSKT06])**.** *For a Haskell Program HP, a left-hand side l of a rule, and any term t, we define*

$$\mathbf{e}_l(t) = \begin{cases} \epsilon, & \text{if } l \text{ matches } t \\ \pi, & \text{for the leftmost-outermost position } \pi \text{ where } head(l|_\pi) \text{ is a constructor, } head(l|_\pi) \neq \mathrm{head}(t|_\pi), \text{ and the symbol } head(t|_\pi) \text{ is defined or a variable} \end{cases}$$

Using this evaluation position with respect to a left-hand side of a rule, we can define the evaluation position of a term with respect to the Haskell program:

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

It should be noted that in the definition of the evaluation position we also allowed non-ground terms. This is, because for our approach we want to evaluate Haskell *symbolically*. Here, variables shall represent arbitrary terminating terms. For a precise definition of our notion of termination see definition 2.7.

As an example, for the term $t =$ `take u (from m)` and the previously given rules for `take` and `from`, we have that $t|_{\mathbf{e}(t)} =$ `u`. This means that the term introduced for `u` by an instantiation of the term t has to be evaluated first, in order to determine the applicable rule for this term. If we instead knew that this position starts with the constructor `S`, e.g., for the term $s =$ `take (S n) (from m)`, the first rule of `take` is known to not being applicable. Thus, the redex of s is $s|_{\mathbf{e}(s)} =$ `from m`, since we must evaluate this subterm first in order to determine whether the second rule of `take` is applicable.

If we have a term like for example `x Z` or `take (S n)`, then the evaluation position stays at the root position, i.e., it will be ϵ for terms having a variable head and for terms which start with a defined function symbol that is applied to too few arguments. If instead a defined function symbol is applied to enough arguments, but no rule in the program is applicable to it, then we call such a term an *error term*. These error terms are replaced by a term `error []`, where the predefined special function `error` has the semantics that the whole evaluation of the current term immediately stops. This is what makes it different from normal predefined functions. Thus, for the term `p Z`, we can replace this term with `error []`. This behavior is the same as that of a Haskell interpreter; it will stop the evaluation and output an error, usually with an error message like "pattern match failure". This error message is normally the argument of the `error` function, but we will always replace it with the empty string (i.e., the empty list of characters), since we are not interested in these messages.

As could be seen, the evaluation position identifies the term that has to be evaluated next. The evaluation step that will take place at the evaluation position is formally defined as a relation below.

Definition 2.6 (Haskell evaluation relation $\rightarrow_{\mathsf{H}}$, [GSSKT06])**.** *For a Haskell program HP, we define $s \rightarrow_{\mathsf{H}} t$, iff*

(1) s rewrites to t on position $\mathbf{e}(s)$ using the first equation of the program whose left-hand side matches $s|_{\mathbf{e}(s)}$, or

(2) $s = (c\ s_1 \ldots s_n)$ for a constructor c with $arity(c) = n$, $s_i \rightarrow_{\mathsf{H}} t_i$ for some $1 \leq i \leq n$, and $t = (c\ s_1 \ldots s_{i-1}\ t_i\ s_{i+1} \ldots s_n)$.

In this evaluation relation, we allow to descent into constructors, although for a term $t = c\ t_1 \ldots t_n$, where c is a constructor, $\mathbf{e}(t) = \epsilon$ holds. However, Haskell-interpreters, such as for example Hugs [JP99], evaluate a term until they can be displayed as a string. To transform a term into a string representation, these interpreters use a function `show`. For most user-defined data types, such a function can be generated automatically by adding `deriving Show` to the data type declaration. This generated function will transform a term $c\ t_1 \ldots t_n$, where c is a constructor of this type, into the string starting with "c" and being followed by `show` $t_1, \ldots,$ `show` t_n. Therefore, such a `show` function requires all arguments of constructors to be evaluated further, instead of stopping at a weak head normal form. Since we do not want to analyze different implementations of `show` functions, we assume that the arguments of constructors must be evaluated, too,

but the order of evaluation is arbitrary. This is reflected in the above definition of the Haskell evaluation relation $\rightarrow_{\mathsf{H}}$.

This evaluation relation is used to define the notion of H-terminating terms.

Definition 2.7 (H-terminating terms, [GSSKT06])**.** *The set of* H-terminating *ground terms is defined as the smallest set of ground terms t, such that*

(a) t does not start an infinite evaluation $t \rightarrow_{\mathsf{H}} \ldots$,

(b) if $t \rightarrow_{\mathsf{H}}^{*} (f\ t_1 \ldots t_n)$ for a defined function symbol f, $n < arity(f)$, and the term t' is H*-terminating, then $(f\ t_1 \ldots t_n\ t')$ is also* H*-terminating, and*

(c) if $t \rightarrow_{\mathsf{H}}^{*} (c\ t_1 \ldots t_n)$ for a constructor c, then $t_1, \ldots, t_n$ are 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 may also introduce new defined functions with arbitrary defining equations.*

Please note that case (c) of definition 2.7 is not included in case (2) of definition 2.6. This is, because a subterm of a constructor could as well be a function, which would be normal w.r.t. $\rightarrow_{\mathsf{H}}$. The reason to require H-termination of a term which can be extended by an arbitrary argument is, that a context might apply such a function to some arguments. So what we show is, that an H-terminating term can be used as an argument of an H-terminating function, such that the resulting term is still H-terminating. For example, the term `take n (from m)` can be used as an argument of a derived `show` function, giving the term `show (take n (from m))` which still is an H-terminating term. From the term `take n (from m)` we can see that this does not hold, if a non-H-terminating term is used as an argument of an H-terminating function: The argument `from m` is not H-terminating, but the complete term is. However, this is dependent on the context, so for example the term `show (from m)` is not H-terminating, if we used the derived `show` functions for our data types.

Often, we also want to specify the types and the class constraints that a Haskell term has. For this purpose, we will use a notation that can be specified at an arbitrary level of precision, where class constraints and types may be specified additionally to the term structure itself.

Definition 2.8 (Notation of terms)**.** *A Haskell term consists of a set of class constraints $\underline{cs}$, a type ρ, and its structure t. This will be denoted in the following equivalent ways, with different levels of detail:*

- $\underline{cs} \Rightarrow \frac{t}{\rho}$
- $\underline{cs} \Rightarrow t$
- $\frac{t}{\rho}$
- t

When class constraints are specified, they are always specified on the top-level, which corresponds to Haskell's type schema.

The example given below shall demonstrate the different levels of detail in this notation.

Example 2.9 (Different levels of detail in term notation)**.** *Consider the Haskell program for the functions* `take` *and* `from`*, given in example 2.1.*

For the term `take n (from m)`*, the following notations are equivalent:*

- $\emptyset \Rightarrow \dfrac{\dfrac{\dfrac{\texttt{take}}{\texttt{Nats -> List Nats -> List Nats}}\ \dfrac{\texttt{n}}{\texttt{Nats}}}{\texttt{List Nats -> List Nats}}\ \left(\dfrac{\dfrac{\texttt{from}}{\texttt{Nats -> List Nats}}\ \dfrac{\texttt{m}}{\texttt{Nats}}}{\texttt{List Nats}}\right)}{\texttt{List Nats}}$
- $\emptyset \Rightarrow \dfrac{\texttt{take n (from m)}}{\texttt{List Nats}}$
- $\emptyset \Rightarrow$ `take n (from m)`
- $\dfrac{\texttt{take n (from m)}}{\texttt{List Nats}}$
- `take n (from m)`

For an example containing class constraints, reconsider the Haskell program from example 2.2, containing the function `plus` *for different data types. For the term* `plus x y`*, the following notations are equivalent:*

- `Addition a` $\Rightarrow \dfrac{\dfrac{\dfrac{\texttt{plus}}{\texttt{a -> a -> a}}\ \dfrac{\texttt{x}}{\texttt{a}}}{\texttt{a -> a}}\ \dfrac{\texttt{y}}{\texttt{a}}}{\texttt{a}}$
- `Addition a` $\Rightarrow \dfrac{\texttt{plus x y}}{\texttt{a}}$

It should be noted that really every term is associated with a type and a set of class constraints. So we only leave out information that is evident from the context.

In order to be able to quickly extract all free variables in a term, we define a special notation for these. For the term structure itself, these can be simply collected. For type variables, we will descent into the types of subterms, as well, instead of only looking at the type of the complete term.

Definition 2.10 (Variables)**.** *For a Haskell term t, we define the set $\mathcal{V}_{\mathsf{H}}(t) = \{t\}$ if t is a Haskell variable, and $\mathcal{V}_{\mathsf{H}}(t) = \mathcal{V}_{\mathsf{H}}(t_1) \cup \mathcal{V}_{\mathsf{H}}(t_2)$ if $t = (t_1\ t_2)$.*

For the Haskell term t, we define $\mathcal{V}_{\mathsf{T}}(t) = \bigcup_{\underline{s} \trianglelefteq t} \mathcal{V}_{\mathsf{T}}(\rho)$. Here, $\mathcal{V}_{\mathsf{T}}(\rho) = \{\rho\}$ if ρ is a type variable, and $\mathcal{V}_{\mathsf{T}}(\rho) = \mathcal{V}_{\mathsf{T}}(\rho_1) \cup \mathcal{V}_{\mathsf{T}}(\rho_2)$ if $\rho = (\rho_1\ \rho_2)$.

For a set of class constraints $\underline{cs}$, we define the set of type variables of these class constraints as $\mathcal{V}_{\mathsf{T}}(\underline{cs}) = \{a \in \mathcal{V}_{\mathsf{T}}(\rho) \mid C\ \rho \in \underline{cs}\}$.

It should be observed that for a term $t = f\ t_1\ \ldots\ t_n$, it holds that $\mathcal{V}_{\mathsf{H}}(t) = \bigcup_{i=1}^{n} \mathcal{V}_{\mathsf{H}}(t_i)$. The example below shall demonstrate, how these variables are collected.

Example 2.11 (Variables and Type Variables)**.** *As an example, we want to have a function* `length` *that counts the number of elements in a list.*

```
data List a = Nil | Cons a (List a)

length  Nil        = Z
length (Cons _ xs) = S (length xs)
```

We want to consider the term $t =$ `length xs`*. For the variables, it holds that* $\mathcal{V}_{\mathsf{H}}(t) = \mathcal{V}_{\mathsf{H}}(\texttt{length xs}) = \{\texttt{xs}\}$.

In order to determine the type variables of t*, we have to take a look at all types of subterms. Thus, we write the term* t *with full type annotations:*

$$\dfrac{\dfrac{\texttt{length}}{\texttt{List a -> Nats}} \quad \dfrac{\texttt{xs}}{\texttt{List a}}}{\texttt{Nats}}$$

In this notation, the types of all subterms of $t =$ `length xs` *can be seen. Thus* $\mathcal{V}_{\mathsf{T}}(t) = \mathcal{V}_{\mathsf{T}}(\texttt{Nats}) \cup \mathcal{V}_{\mathsf{T}}(\texttt{List a -> Nats}) \cup \mathcal{V}_{\mathsf{T}}(\texttt{List a}) = \{\texttt{a}\}$.

In the example, we see that the type variable `a` *does not occur in the type* `Nats` *of the complete term. This is the reason why the function* $\mathcal{V}_{\mathsf{T}}$ *considers the types of all subterms, too.*

2.2 Term Rewrite Systems

This section briefly introduces the basic notions and concepts that are used in Term Rewriting, and shall chiefly introduce the used notation. For an in-depth discussion on Term Rewriting, please see the book by Baader and Nipkow [BN98], for example.

Let $\mathcal{V}$ be a countably infinite set of *variables* and let Σ be a *signature*, i.e., a finite set of *functions* where every function is associated a natural number which is called its *arity*. Then the set of *terms over* Σ *and* $\mathcal{V}$, denoted $T(\Sigma, \mathcal{V})$ is defined as the smallest set such that $\mathcal{V} \subseteq T(\Sigma, \mathcal{V})$ and if $f \in \Sigma$ and $arity(f) = n$, then $f(t_1, \ldots, t_n) \in T(\Sigma, \mathcal{V})$ for all $t_1, \ldots, t_n \in T(\Sigma, \mathcal{V})$.

For a term $t \in T(\Sigma, \mathcal{V})$, we define the *set of variables occuring in* t, which will be denoted as $\mathcal{V}(t)$, as follows: $\mathcal{V}(t) = t$ if $t \in \mathcal{V}$, and $\mathcal{V}(t) = \bigcup_{i=1}^{n} \mathcal{V}(t_i)$ if $t = f(t_1, \ldots, t_n)$. A term t is called a *ground* term, iff $\mathcal{V}(t) = \emptyset$.

Also, for a term t we define the set of *positions* of t, denoted $Occ(t)$. This set is inductively defined by $\epsilon \in Occ(t)$, and $i\pi \in Occ(t)$ if $t = f(t_1, \ldots, t_n)$, $1 \leq i \leq n$, and $\pi \in Occ(t_i)$. If $\pi \in Occ(t)$, then the *term identified by* π is defined as $t|_\epsilon = t$ and $t|_\pi = t_i|_{\pi'}$ if $t = f(t_1, \ldots, t_n)$ and $\pi = i\pi'$. Two positions $\pi_1, \pi_2 \in Occ(t)$ are called *disjoint*, denoted $\pi_1 \perp \pi_2$, iff there exist $\pi', \pi_1'', \pi_2'' \in \mathbb{N}^*$ and $i, j \in \mathbb{N}$, such that $\pi_1 = \pi' i \pi_1''$, $\pi_2 = \pi' j \pi_2''$ and $i \neq j$.

For two terms s and t and a position $\pi \in Occ(t)$, the term $t[s]_\pi$ results from replacing $t|_\pi$ by s. This can be defined as $t[s]_\epsilon = s$ and if $t = f(t_1, \ldots, t_n)$ and $\pi = i\pi'$, then $t[s]_\pi = f(t_1, \ldots, t_n)[s]_{i\pi'} = f(t_1, \ldots, t_i[s]_{\pi'}, \ldots, t_n)$.

A *substitution* σ is a mapping from $\mathcal{V}$ to $T(\Sigma, \mathcal{V})$, where it must hold that $|\{x \mid \sigma(x) \neq x\}| < \infty$. The application of a substution σ to a term t is denoted $t\sigma$ and is defined as $t\sigma = \sigma(t)$ if $t \in \mathcal{V}$, and $t\sigma = f(t_1\sigma, \ldots, t_n\sigma)$ if $t = f(t_1, \ldots, t_n)$. If for two terms s and t it holds that $s\sigma = t\sigma$ for a substitution σ, then σ is called a *unifier* of s and t. For a term s and a term t, a substitution σ is called a *matcher* of s to t, if $s\sigma = t$.

A term t' is a *subterm* of another term t, symbolically $t \trianglerighteq t'$, iff there exists a position $\pi \in Occ(t)$ such that $t|_\pi = t'$. A term t' is a *proper subterm* of another term t, written $t \triangleright t'$, iff there exists a position $\pi \in Occ(t)$ such that $t|_\pi = t'$ and $\pi \neq \epsilon$.

A *Term Rewrite System (TRS)* $\mathcal{R}$ over $T(\Sigma, \mathcal{V})$ is defined as $\mathcal{R} \subseteq T(\Sigma, \mathcal{V})^2$, where we write $l \to r \in \mathcal{R}$ iff $(l, r) \in \mathcal{R}$. Furthermore, it must hold that $t \notin \mathcal{V}$

and $\mathcal{V}(r) \subseteq \mathcal{V}(l)$. The elements of a TRS are called *rules*. A term s can be *rewritten/reduced* to a term t with $\mathcal{R}$, denoted $s \to_{\mathcal{R},\pi} t$, if $\pi \in Occ(s)$ and there exist a rule $l \to r \in \mathcal{R}$ and a substitution σ, such that $s|_\pi = l\sigma$ and $t = s[r\sigma]_\pi$. The term $s|_\pi$ is called a *redex*. We write $s \to_{\mathcal{R}} t$ iff a $\pi \in Occ(s)$ exists, such that $s \to_{\mathcal{R},\pi} t$. Furthermore, we write $s \xrightarrow{\varepsilon}_{\mathcal{R}} t$ iff $s \to_{\mathcal{R},\epsilon} t$, and we write $s \xrightarrow{>\varepsilon}_{\mathcal{R}} t$ iff $s \to_{\mathcal{R},\pi} t$ and $\pi \neq \epsilon$.

A function symbol f is called a *defined* function symbol, which is denoted $f \in \mathcal{D}_\mathcal{R}$, if there exists a rule of the form $f(t_1, \ldots, t_n) \to r \in \mathcal{R}$. The set $\mathcal{C}_\mathcal{R}$ of *constructor* function symbols is defined as $\mathcal{C}_\mathcal{R} = \Sigma \setminus \mathcal{D}_\mathcal{R}$. If the rewrite system $\mathcal{R}$ is clear from the context, then we also write $\mathcal{D}$ and $\mathcal{C}$.

We call a Term Rewrite System $\mathcal{R}$ an *applicative* Term Rewrite System, iff $\mathcal{D}_\mathcal{R} \ni \mathsf{app}$ for a binary symbol app and where every $f \in \Sigma$ with $f \neq \mathsf{app}$ has $arity(f) = 0$. Otherwise, $\mathcal{R}$ is called a *first order* Term Rewrite System. Thus, a Haskell program can be seen as an applicative Term Rewrite System, since the only defined function having a non-zero arity is "juxtaposition", as was already mentioned in the previous section.

For two relations $\to_1, \to_2 \subseteq M^2$, the *concatenation*, denoted $\to_1 \circ \to_2$, is defined as $\to_1 \circ \to_2 = \{(x, z) \mid \exists y : x \to_1 y \to_2 z\}$. For any relation $\to \subseteq M^2$, we define $\to^0 = \{(x, x) \mid x \in M\}$, $\to^{n+1} = \to \circ \to^n$ for every $n \in \mathbb{N}$, $\to^+ = \bigcup_{i=1}^n \to^i$, and $\to^* = \to^0 \cup \to^+$. We say that a relation $\to \subseteq M^2$ is *reflexive*, iff $x \to x$ for all $x \in M$. It is called *irreflexive*, iff $x \not\to x$ for all $x \in M$. The relation $\to$ is called *well-founded*, iff no infinite sequence $x_0 \to x_1 \to x_2 \to \ldots$ exists, where $x_i \in M$. It can be observed that $\to$ must be irreflexive in order to be well-founded.

A term t is called *terminating* w.r.t. a relation $\to$, iff there is no infinite sequence $t \to t_1 \to t_2 \to \ldots$. A Term Rewrite System $\mathcal{R}$ is said to be *terminating*, iff there is no infinite rewrite sequence for any term t.

In order to restrict the rewrite relation to the Haskell evaluation strategy, we define $\xrightarrow{\mathsf{H}}_\mathcal{R}$ for a rewrite relation $\to_\mathcal{R}$ as follows:

For two terms s and t, it holds that $s \xrightarrow{\mathsf{H}}_\mathcal{R} t$, iff $s \to_{\mathcal{R},\pi} t$ and for all $\pi' \in Occ(s)$, where $s \to_{\mathcal{R},\pi'} t'$ for some term t', it holds that $\pi' = \pi\pi''$ for some $\pi'' \in Occ(s|_\pi)$, or $\pi = \pi_1 i \pi_2$, $\pi' = \pi_1 j \pi_2'$ and $i < j$. The term $s|_\pi$ is called a *leftmost-outermost* redex.

2.3 Dependency Pair Framework

This section briefly recapitulates the basic definitions of the *Dependency Pair Framework (DP Framework)*, as described in [GTSK05b]. It is an extension of the original *Dependency Pairs Approach* which was presented in [AG00]. The main improvements of the DP framework over the Dependency Pair Approach consist of even more modular proofs and by introducing a new component, the TRS $\mathcal{Q}$, which is the reason for choosing this rather complicated form of Dependency Pairs. This TRS $\mathcal{Q}$ serves the purpose of restricting the possible evaluations: A proper subterm of another term where the subterm is reducible w.r.t. $\mathcal{Q}$ blocks all reductions above it, until it has been reduced to a term that is normal w.r.t. $\mathcal{Q}$. Thus, the set $\mathcal{Q}$ can define a certain degree of *innermost* evaluations. This may seem unsuited for the analysis of Haskell at first, but we will see later that this TRS will be of great value when we need only partial innermost evaluations in chapter 10. Because only the information whether a

term is normal w.r.t. $\mathcal{Q}$ is of interest, we will often leave out the right-hand sides of this TRS.

A quadruple $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, f)$ is called a *DP problem*, iff $\mathcal{P}$, $\mathcal{R}$, and $\mathcal{Q}$ are Term Rewrite Systems and $f \in \{\mathsf{a}, \mathsf{m}\}$.

The $\mathcal{Q}$-*restricted rewriting* $\xrightarrow{\mathcal{Q}}_{\mathcal{R}}$ is defined as $s \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t$ iff $s \to_{\mathcal{R}} t$ and no proper subterm of s is reducible with $\to_{\mathcal{Q}}$.

For TRSs $\mathcal{P}$, $\mathcal{R}$, and $\mathcal{Q}$, a possibly infinite sequence $s_1 \to t_1, s_2 \to t_2, \ldots \in \mathcal{P}$ is called a $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-*chain*, iff there are substitutions σ_i such that $t_i\sigma_i \xrightarrow{\mathcal{Q}}{}^*_{\mathcal{R}} s_{i+1}\sigma_{i+1}$ for all i and no proper subterm of a $s_i\sigma_{i+1}$ is reducible with $\to_{\mathcal{Q}}$. A chain is *minimal*, iff there is a substitution σ as above and all $t_i\sigma_i$ are terminating w.r.t. $\xrightarrow{\mathcal{Q}}_{\mathcal{R}}$.

The flag f in a DP problem indicates whether **a**ny, or only **m**inimal chains shall be considered.

A DP problem $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathsf{a})/(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathsf{m})$ is called *finite*, iff it contains no (minimal) infinite $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-chains. A DP problem $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathsf{a})/(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathsf{m})$ is called *infinite*, iff either it contains a (minimal) infinite $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-chain, or $\mathcal{R}$ is not terminating. Please note that a DP problem can be both finite and infinite at the same time, since a DP problem may contain no infinite chain, but a non-terminating set of rules.

For every DP problem $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, f)$, we can assume that the defined symbols in $\mathcal{P}$ and $\mathcal{R}$ are disjoint. If this was not the case, then we could replace $\mathcal{P}$ by $\mathcal{P}^\sharp = \{f^\sharp(s_1, \ldots, s_m) \to g^\sharp(t_1, \ldots, t_n) \mid f(s_1, \ldots, s_m) \to g(t_1, \ldots, t_n) \in \mathcal{P}\}$. Here, the symbols $f^\sharp$ and $g^\sharp$ are fresh symbols that do not occur in $\mathcal{R}$.

Chapter 3

Previous Haskell Termination Approach

This chapter briefly outlines the previous approach to Haskell Termination analysis [Swi05, GSSKT06] which is to be extended in this thesis. This previous approach is an improvement of [PSS97] that performs termination analysis of a small Haskell-like language, but is restricted to a special form of what we call Termination Graphs ("termination tableaux without crossings"). The used notation will follow [GSSKT06] whenever possible.

As a first step, the Haskell program is transformed into a simplified form of Haskell. The simplified Haskell which is used is not Core Haskell, since this is more suited for evaluation purposes, but makes termination analysis harder. These transformations remove lambda- and `if`-expressions, and convert special patterns, for example. For a detailed description please refer to [Swi05], since these transformation were not changed during the development of this thesis.

After the transformations were applied, the result is a Haskell program that only consists of top-level function definitions, where all left-hand sides contain only basic patterns. Basic patterns are such patterns that do not have any special semantics, i.e., they consist only of constructors and variables. An example for a pattern that is not a basic pattern is `y@(x+2)`, which will be transformed into a test for `y >= 2` and every occurrence of the variable `x` will be replaced by the term `(y-2)`. This transformation follows the definition of the `(x+`n`)`-pattern, which is given in [Jon03, Figure 3.2, part 2: Semantics of Case Expressions, case (s)].

The next step consists of constructing the Termination Graph. The formal definition of the Termination Graph will be given in definition 4.12, where it will be extended in order to include type classes. Therefore, we only want to show an example construction of a Termination Graph for a slightly modified variant of the Haskell program in example 2.1 which defined the functions `take` and `from`. The difference is that we do not encode the predecessor computation into the function `take`, but use a function `p` to do this. This function works by moving inwards as far as possible, then reducing the representation of the number 1, which is `S Z`, to the term `Z` representing the number 0.

Example 3.1 (`take from` with explicit predecessor computation)**.**

```
data Nats   = Z   | S Nats
data List a = Nil | Cons a (List a)

from m = Cons m (from (S m))

take Z  _           = Nil
take _  Nil         = Nil
take n (Cons x xs) = Cons x (take (p n) xs)

p (S Z) = Z
p (S n) = S (p n)
```

The Termination Graph for the start term $t =$ `take u (from m)` is shown in figure 3.1. Its construction shall be explained in the following.

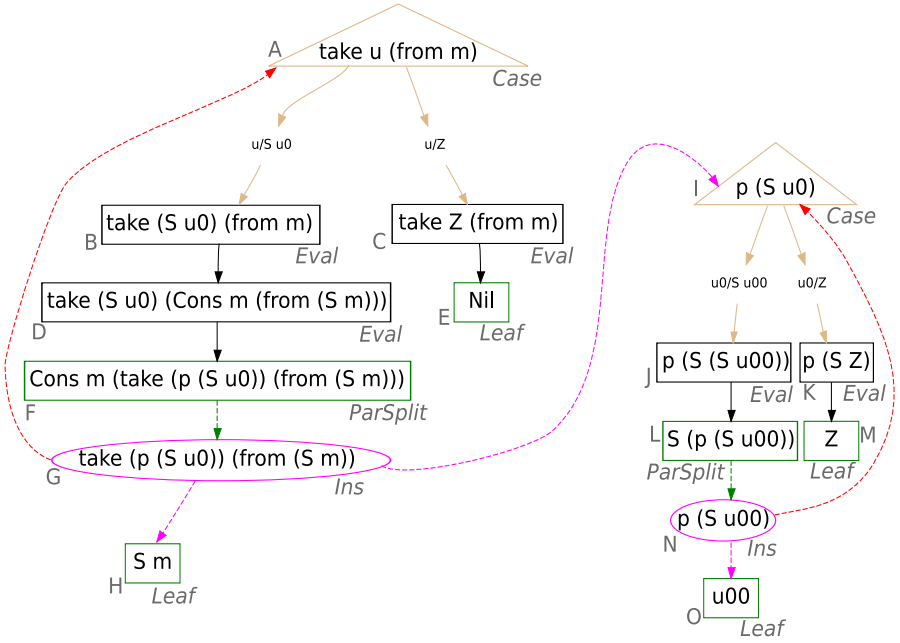

Figure 3.1: Termination Graph for `take u (from m)`

For the start term t in node A, we have already seen that the evaluation position of this term is $t|_{\mathbf{e}(t)} = \mathtt{u}$. Since we have to consider all possible instantiations of this term, we do a ***Case*** split, according to the data constructors that exist for the type of this term. The variable `u` has type `Nats`, so the constructors `Z` and `S` are inserted, yielding nodes B and C. For the node C, the first rule of `take` is applicable, thus the result of this ***Eval*** step is node E. This does not need any further analysis, since it is a terminating term. Thus, it is a leaf in the Termination Graph.

For the node B, the second argument of `take` must be evaluated, such that it can be decided whether the second rule of `take` is applicable or not. For

this purpose, the term `from m` is evaluated one step, therefore the node B is an ***Eval*** step having the child node D. For node D, we see that the second rule of `take` is not applicable, but the third rule is. Therefore, we do another ***Eval*** step, resulting in node F. As said before, we want to allow evaluation inside of constructors, thus we split into the different arguments of the constructor `Cons`. The first argument is just the variable `m` which represents only H-terminating terms by assumption. Therefore, it does not need to be inserted as a new node. For the other argument, we do not see that it is directly H-terminating, thus we make it the child G of node F. As we can see, the term in node G is an instance of the term we already have in node A. We could continue and apply the same steps over and over again. However, we want to construct a finite graph. Thus, we allow to draw an instantiation edge, i.e., we identify the node G as an instance of node A. But we must still regard those subterms which are in the range of the matcher. This is, because these are bound to variables in the instantiated term, and for these we assume that they are instantiated only with H-terminating terms. Thus, we must show this property and create the children H and I for this purpose. Node H is again H-terminating, since it contains only constructors and variables.

For node I, the analysis is performed analogously: a case distinction is made in order to determine the applicable rule for `p`, then evaluation steps apply these rules. Here, we also have an ***Ins***-node, i.e., a term that was identified as an instance of a previous node in the graph, giving another cycle.

From this Termination Graph, DP problems are created. These correspond to the strongly connected components (SCCs) of the Termination Graph. It has to be shown that these SCCs cannot be traversed infinitely often. This is equivalent to showing that the generated DP problems are finite. Since the creation of DP problems is modified in this thesis, this property has to be proven again. Proving the finiteness of DP problems is left to the DP Framework, which incorporates many powerful techniques for this purpose. Thereby, whenever improvements are made in the DP Framework, then these improve the power of the Haskell termination analysis, as well.

In the above example for the start term `take u (from m)`, we have two SCCs, one for the function `take`, and another one for the function `p`. The latter SCC cannot be traversed infinitely often, since there the function `p` is pushed inwards until it reaches the term `S Z`, which is then replaced by `Z`. Thus, the number of recursive calls is limited by the natural number that is represented by its argument. This contains a variable which represents an H-terminating term; therefore this number must be a natural number and cannot be infinite. Furthermore, this function returns the argument decremented by one. This is also the reason, why the SCC for `take` cannot be traversed infinitely often; its first argument will be decreased in every recursive call. How the DP problems are created is not shown here, it will be illustrated in the following Chapter and a modified version of it will be presented in Chapter 6.

Chapter 4

Extension to Type Classes

In order to extend the approach of [GSSKT06] to type classes, an additional expansion rule has to be specified for the construction of Termination Graphs. Also, the type classes of a term must be considered in the other expansion rules. This handling of type classes was already contained in [Swi05], but this chapter shall introduce it into the notation of [GSSKT06]. Without loss of generality it is assumed that type variables, class names, type constructors, function symbols, and variables are pairwise disjoint.

4.1 Generating Class Instances

During the construction of the Termination Graph, we will face the problem, that for a type variable with constraints we need to determine all possible class instances this type variable might be instantiated with. In Haskell 98, only such constraints are valid where the head symbol of the type after the class name is a variable. Therefore, it is required to reduce constraints based on the instance declarations in the Haskell program to this form, otherwise a type is not a valid type, as required in [Jon03].

Example 4.1 (Reduction of class constraints)**.** *Consider the function* `plus`*, which was presented in example 2.2 for different instances.*

If we want to enter the ground term `plus undefined undefined :: List a` *in a Haskell interpreter, then we must add a class constraint to the type, otherwise this is invalid. Since we want to use the function* `plus` *as defined in the instance of the data type* `List a`*, an idea would be to use the term* `plus undefined undefined :: Addition (List a) => List a`*.*

However, this is not a valid term, because of the required reduction mentioned above. Therefore, we have to reduce the constraint according to the instances in the program. The instance for the data type`List a` *has the declaration* `instance Addition a => Addition (List a)`*. Thus, we must add the constraint for the contained elements. So our example term would have to be* `plus undefined undefined :: Addition a => List a`*.*

This reduction according to the instances of a Haskell program is formally defined in the following.

Definition 4.2 (reduce)**.**

$$\text{reduce}(\underline{cs}) = \text{reduceStep}^{\infty}(\underline{cs})$$

The above function reduceStep *does a reduction of the class constraints according to the instances defined in the Haskell program:*

$$\text{reduceStep}((C\ \rho) \uplus \underline{cs}) = \{(C_1\ a_{i_1})\sigma, \ldots, (C_n\ a_{i_n})\sigma\} \uplus \underline{cs}$$

if either

- *an instance* $(C\ (T\ a_1 \ldots a_m))$ *exists in the Haskell program with class constraints* $(C_1\ a_{i_1}), \ldots, (C_n\ a_{i_n})$ *for a type constructor* T $(n \geq 0)$*,*
- σ *matches* $(C\ (T\ a_1 \ldots a_m))$ *to* $(C\ \rho)$*.*

or

- $n = 0$*,*
- *a class constraint* $(C'\ \rho)$ *exists in* $\underline{cs}$*,*
- C' *is a subclass of* C*.*

As Haskell 98 neither allows overlapping instances nor cyclic classes, the result of $\text{reduce}(\underline{cs}) = \text{reduceStep}^{\infty}(\underline{cs})$ *always exists, it is reached after a finite number of steps, and it is unique.*

How this function works shall be shown in the next example.

Example 4.3 (Reduction of class constraints using reduce)**.** *We again make use of the instances defined in example 2.2.*

For the set of class constraints {`Addition (List a)`}*, the above function* reduceStep *is applicable once. Therefore,* reduce({`Addition (List a)`}) = {`Addition a`})*, because for the type* `List a` *the program contains the instance* `Addition a => Addition (List a)`*. As can be observed, this was also done in example 4.1.*

In case of the set {`Addition a`, `Multiplication a`}*, we have that for this set* reduce({`Addition a`, `Multiplication a`}) = {`Multiplication a`}*, because the class* `Multiplication` *is a subclass of the class* `Addition`*. This means that the only class constraint that has to be considered is the class constraint* `Multiplication a`*, since the other class constraint* `Addition a` *is fulfilled for every type contained in the class* `Multiplication`*. Hence, we can drop the class constraint* `Addition a`*.*

Next, we want to select the possible instances for a class member function. This is needed in order to decide on the set of rules that shall be used for the current term.

Example 4.4 (Selection of instances)**.** *We want to analyze termination for the start term* `(plus x y) :: (Addition a, HasZero a) => a` *with respect to the Haskell program given in example 2.2.*

In this term, the function `plus` *refers to a class member of the type class* `Addition`*, whose implementation depends on the type. Therefore, the idea is to consider all possible types. So we enumerate all instances that exist for this class, in our example the instances for the type* `Nats` *and for the type* `List a`*.*

For a formal representation, the function "instances" enumerates all instances of a class, with respect to a type. This type might have been instantiated further, therefore we must instantiate types in a member type schema with the most general unifier.

Definition 4.5 (instances)**.**

$$\begin{aligned}\text{instances}(f,\rho) \quad = \quad \{\delta \quad | \quad & f \textit{ is a member of a class } (C\ a),\\ & f \textit{ has the type } \rho_{f,C} \textit{ in } (C\ a)\\ & \textit{ with disjunct variables from } \rho,\\ & (C\ \rho') \textit{ is an instance of } (C\ a),\\ & \delta \textit{ is the mgu of } \rho \textit{ and } \rho_{f,C}[a/\rho']\}\end{aligned}$$

This definition is now illustrated by applying it to the function `plus` in the start term of example 4.4.

Example 4.6 (instances applied to example 4.4)**.** *In the start term of example 4.4, the function* `plus` *has the type* `a -> a -> a`*. This type unifies with the types of* `plus` *in the two instances of the class* `Addition`*, where for the instance* `Nats` *we have* `Nats -> Nats -> Nats`*, and for the instance* `List b` *we have the type* `List b -> List b -> List b`*. Hence,*

$$\text{instances}(\texttt{plus}, \texttt{a -> a -> a}) = \{[\texttt{a/Nats}],\ [\texttt{a/List b}]\}$$

The most general unifier is used in the above definition, because it might also be the case that a type in an instance is more general than the corresponding type in the start term. So if we had a class `A a` *containing a function* `project :: a -> b -> a`*, then in the start term* `project x Z` *the function* `project` *has the type* `a -> Nats -> a`*. But still, all instances would be applicable. If we assume that only the data type* `Nats` *had an instance of the class* `A`*, then*

$$\text{instances}(\texttt{project}, \texttt{a -> Nats -> a}) = \{[\texttt{a/Nats}, \texttt{b/Nats}]\}$$

In the above example which generated the instances of `plus` for start term `(plus x y) :: (Addition a, HasZero a) => a`, we see that also the instance for the data type `List b` is considered, although there is no instance of the class `HasZero` for this data type in the Haskell program. In order to consider as few of these ill-typed terms as possible, we use the function "filter" to remove those type substitutions where we are sure that they cannot be used in any ground instance of the start term. For this selection, the function "filter" makes use of the function "reduce" and implements the requirement discussed in example 4.1 that all class constraints must start with a type variable as head symbol.

Definition 4.7 (filter)**.**

$$\begin{aligned}\text{filter}(S, \underline{cs}) \quad = \quad \{\delta \in S \quad | \quad & \forall (C\ \rho) \in \text{reduce}(\underline{cs}\delta) :\\ & (C\ \rho) \textit{ is of the form } (C\ (b\ \rho_1 \ldots \rho_{n_\rho}))\\ & \textit{where } b \textit{ is a type variable and } C \textit{ is a class}\}\end{aligned}$$

The condition that a remaining class constraint $(C\ \rho)$ must be of the form $(C\ (b\ \rho_1 \ldots \rho_n))$ rules out instances for which other constraints could not be satisfied, as mentioned above. An example shall demonstrate, how the function "filter" reduces the type substitutions that determine the different instances to consider further.

Example 4.8 (Filtering of instances)**.** *In this example, we consider the generated type substitutions that resulted from the function* instances *for the start term* `(plus x y) :: (Addition a, HasZero a) => a`*, which were discussed in example 4.6.*

$$\text{instances}(\texttt{plus}, \texttt{a -> a -> a}) = \{[\texttt{a/Nats}],\ [\texttt{a/List b}]\}$$

For these substitutions, the function filter *checks whether the new constraints exist. For this purpose, it applies every substitution to the initial set of class constraints, reduces these, and keeps only those where no type constructor occurs as head symbol of a class constraint.*

For the substitution $\delta_1 = [\texttt{a/Nats}]$*, we first reduce the class constraints:*

$$\begin{aligned}
&\text{reduce}(\{\texttt{Addition a, HasZero a}\}\delta_1)\\
&= \text{reduce}(\{\texttt{Addition Nats, HasZero Nats}\})\\
&= \text{reduce}(\{\texttt{HasZero Nats}\}) && \textit{(because of } \texttt{instance Addition Nats}\textit{)}\\
&= \text{reduce}(\emptyset) && \textit{(because of } \texttt{instance HasZero Nats}\textit{)}\\
&= \emptyset
\end{aligned}$$

Trivially, all conditions in $\text{filter}(\{\delta_1\}, \{\texttt{Addition a, HasZero a}\})$ *are fulfilled, hence this type substitution introduces an instance whose implementation of* `plus` *must be considered.*

For $\delta_2 = [\texttt{a/List b}]$*, we again reduce the instantiated class constraints:*

$$\begin{aligned}
&\text{reduce}(\{\texttt{Addition a, HasZero a}\}\delta_2)\\
&= \text{reduce}(\{\texttt{Addition (List b), HasZero (List b)}\})\\
&= \text{reduce}(\{\texttt{Addition b, HasZero (List b)}\})\\
&\qquad\qquad \textit{(because of } \texttt{instance Addition b => Addition (List b)}\textit{)}\\
&= \{\texttt{Addition b, HasZero (List b)}\}
\end{aligned}$$

Here, no further reductions are possible. For the first remaining constraint, it holds that it has the form $C\ (b\ \rho_1 \ldots \rho_n)$*, where* $C = \texttt{Addition}$*,* $b = \texttt{b}$*, and* $n = 0$*. The other remaining constraint* `HasZero (List b)` *does not have the form* $C\ (b\ \rho_1 \ldots \rho_n)$ *for a type variable* b*. Thus, this is an instance that does not need not be considered. And in fact, there is no instance of the class* `HasZero` *for the type* `List b`*, which means that its implementation of the function* `plus` *cannot be used for the given term.*

When a term is evaluated or split into its subterms, only those class constraints need to be considered further, for which the type variables still exist in the term.

Example 4.9 (Reducing class constraints to covered class constraints)**.** *We extend the Haskell program from example 2.2 defining the function* `plus` *by another function.*

```
f :: (Addition a, Multiplication b) => a -> b -> a
f x y = plus x x
```

Here, in the term `plus x x` *on the right-hand side of the rule, we have that the constraint* `Multiplication b` *is no longer of interest, as the type* `b` *is not present. Therefore, this class constraint need not be considered further.*

This reduction to the types that still exist in the term is performed by the function coveredConstraints.

Definition 4.10 (coveredConstraints)**.** *For a set $\underline{cs}$ of class constraints and a set of variables X, the function $step_{\underline{cs}}$ determines all variables of constraints that share a variable with X:*

$$step_{\underline{cs}}(X) := X \cup \bigcup_{\substack{c \in \underline{cs} \\ \mathcal{V}(c) \cap X \neq \emptyset}} \mathcal{V}_\mathsf{T}(c)$$

This function reaches a fixed point after a finite number of steps, since it is bounded by $\underline{cs}$.

Using the variables that the class constraints share with the type variables, we can define the set of class constraints that are still considered as covered *for a term:*

$$\text{coveredConstraints}(\underline{cs} \Rightarrow t) = \left\{ c \in \underline{cs} \;\middle|\; step^{\infty}_{\underline{cs}}(\mathcal{V}_\mathsf{T}(t)) \supseteq \mathcal{V}_\mathsf{T}(c) \right\}$$

The example below shows how this function works on the function given in example 4.9.

Example 4.11 (Using coveredConstraints to reduce class constraints)**.** *We now show how to apply* coveredConstraints *to the right-hand side of the rule given in example 4.9.*

For the term $t =$ `(Addition a, Multiplication b)`$\Rightarrow \underset{\mathtt{a}}{\mathtt{plus\ x\ x}}$*, the set of type variables occuring in t is $step^{\infty}_{\underline{cs}}(\mathcal{V}_\mathsf{T}(t)) = \{\mathtt{a}\}$. Using these type variables, we have* collectCCs$(t) = \{$`Addition a`$\}$*. Thus, it has been found out that the class constraint* `Multiplication b` *does not need to be considered further.*

4.2 Extending the Termination Graph

In the Termination Graph, type classes are handled in a similar way as the distinction of data type constructors. When the term on the evaluation position is a variable, we enumerate all possible constructor terms. Then, the instantiation of the variable will determine which rule is applicable. For type classes, we now might have a term on the evaluation position, where a type variable with some class constraint has to be evaluated. Then, we will enumerate all possible instances of this class. Therefore, the instantiation of this type variable will determine which set of rules is to be applied to the term.

Using the functions for type classes that were defined previously, we extend definition 4 of [GSSKT06] in order to include the handling of type classes. Again, this was already described in [Swi05], but here it is included in the notation of [GSSKT06], where type classes were not considered.

Definition 4.12 (Termination Graph with type classes)**.** *Let TG be a graph with a leaf marked with a term t. We say that TG can be expanded to TG' (denoted "$TG \Rightarrow TG'$") if TG' results from TG 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. If not explicitly stated otherwise, all child nodes in $\mathbf{ch}(t)$ *are assigned the set of class constraints that are still covered by type variables contained in the types of subterms of the child, i.e., for a child* $\underline{cs} \Rightarrow t' \in \mathbf{ch}(t)$*, this child node is not labelled with* $\underline{cs} \Rightarrow t'$*, but instead it is inserted into the Termination Graph labelled with* $\text{coveredConstraints}(\underline{cs}, t') \Rightarrow t'$*.*

Eval: $\mathbf{ch}(t) = \{\tilde{t}\}$*, if* $t = (f\ t_1 \dots t_n)$*,* f *is a defined symbol,* $n \geq arity(f)$*,* $t \to_{\mathsf{H}} \tilde{t}$

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,* $n \geq arity(f)$*,* $t|_{\mathbf{e}(t)}$ *is a variable* x *of type "*$T\ \rho_1 \dots \rho_m$*" for a type constructor* T*, the type constructor* T *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*.*

TyCase: $\mathbf{ch}(\underline{cs} \Rightarrow t) = \{\underline{cs}_1 \Rightarrow t, \dots, \underline{cs}_m \Rightarrow t\}$ *if* $t = (f\ t_1 \dots t_n)$*,* f *is a defined function symbol,* $n \geq arity(f)$*,* $t|_{\mathbf{e}(t)} = (g\ s_1 \dots s_v)$ *has type* $\rho[a]$ *for a type variable* a*, such that* $\{\delta_1, \dots, \delta_m\} = \text{filter}(\text{instances}(g, \rho[a]), \underline{cs})$ *and* $\underline{cs}_i = \text{reduce}(\underline{cs}\delta_i)$ *for* $1 \leq i \leq m$*. The edge from* $\underline{cs} \Rightarrow t$ *to* $\underline{cs}_i \Rightarrow t$ *is marked with* δ_i*.*

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

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

Ins: $\mathbf{ch}(\underline{cs} \Rightarrow t) = \{\underline{cs}_1 \Rightarrow s_1, \dots, \underline{cs}_m \Rightarrow s_m, \widetilde{\underline{cs}} \Rightarrow \tilde{t}\}$*, if* $t = (f\ t_1 \dots t_n)$*,* t *is not an error term,* f *is a defined symbol,* $n \geq arity(f)$*,* $t = \tilde{t}\sigma$ *for some term* $\tilde{t}$*,* $\sigma = [x_1/s_1, \dots, x_m/s_m, a_1/\rho_1, \dots, a_l/\rho_l]$*, where* $x_i \in \mathcal{V}_{\mathsf{H}}(\tilde{t})$ *and* $a_j \in \mathcal{V}_{\mathsf{T}}(\tilde{t})$ *for all* i, j*. 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, but not a* ***TyCase****-node*[1]*, and all paths starting in* $\tilde{t}$ *reach an* ***Eval****-node or a leaf with an error term after traversing only* ***Case****- and* ***TyCase****-nodes.*[2] *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 reuse 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* $\tilde{t}$*.*
The set of class constraints $\widetilde{\underline{cs}}$ *must be the same or more general than the set of class constraints* $\underline{cs}$*.*

Let TG_t *be the graph with a single node marked with* t *and no edges.* TG *is a* Termination Graph *for* t *iff* $TG_t \Rightarrow^* TG$ *and* TG *is in normal form w.r.t.* $\Rightarrow$*.*

An example for a Termination Graph containing a ***TyCase***-node is given in the following.

Example 4.13 (Termination Graph containing ***TyCase***-nodes)**.** *We want to show termination for the start term* `plus x y` *w.r.t. the program given in example 2.2.*

From this start term, the Termination Graph shown in figure 4.1 is built, where the class constraints are shown for every node. In this graph, one can observe two ***TyCase****-nodes. The first one occurs directly at node A containing the start term, because for this term the instance to use is unknown, which was already discussed in example 4.4 for a similar term.*

[1] This shall ensure that once a type is fixed, the corresponding definition of a class member function is used

[2] Thereby ensuring that all cycles contain at least one ***Eval***-node

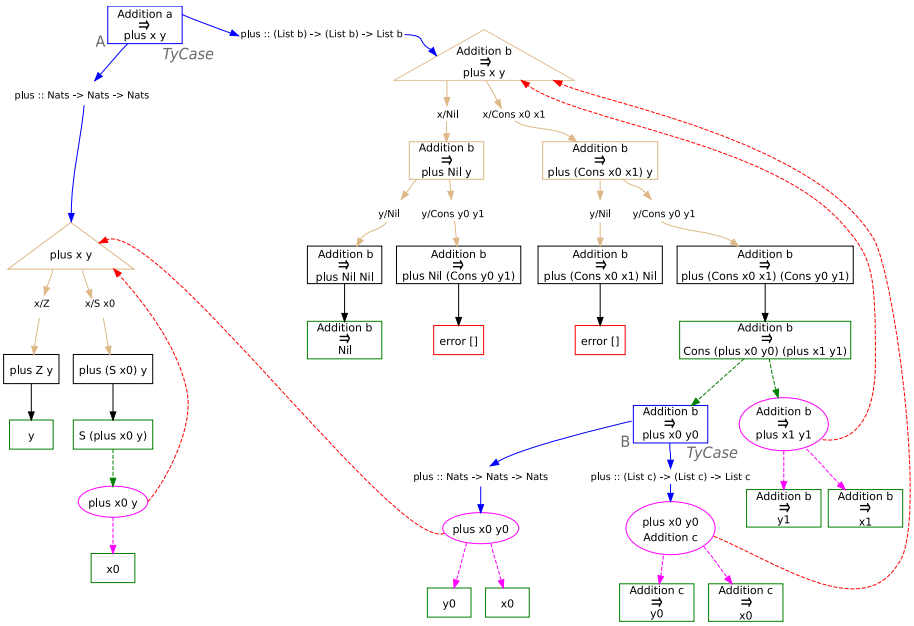

Figure 4.1: Termination Graph for `plus x y` containing ***TyCase***-nodes

For the other ***TyCase****-node B in the graph, the same distinction must be made. This node results from the rule*

`plus (Cons x xs) (Cons y ys) = Cons (plus x y) (plus xs ys)`

for the data type `List a`*. Here, the first argument of the constructor* `Cons` *on the right-hand side of this rule has the type* `b`*. Therefore, the same analysis as for node A has to be made, and therefore the resulting nodes are directly instances of the nodes that resulted from the* ***TyCase*** *expansion at node A.*

The correctness of the extension to type classes will not be proven here, it was already proven in [Swi05] that introducing this case distinction for type variables is correct. In this thesis, the proof of correctness will be included in chapter 6, where renaming is introduced. There, it will be shown that when renaming is applied and the ***TyCase***-expansion is added, then termination of a start term for a Haskell program still follows from the absence of infinite chains in the created DP problems.

Furthermore, it was shown in [Swi05, GSSKT06] that for every start term a Termination Graph can be built. In order to show this, one can introduce a node $f\ x_1 \ldots x_n$ for every defined function symbol f of arity n in the Haskell program, and a node $x\ y$. After having constructed all patterns of such a function, one then can use ***ParSplit***-nodes to dispose constructors and variables. Then, instantiation edges are added, if the arity is fulfilled. In case the arity of the function is less than the number of terms that are supplied as arguments, then a number of instantiation edges to the node $x\ y$ are inserted. Otherwise, it might be the case that too few arguments are applied. Then, a number of ***VarExp***-expansions will be used to bring the term to its defined function's arity.

For ***ParSplit***-nodes with variable head, the result of this application is unknown. Therefore, we want to introduce fresh variables that represent the

unknown result. Since these variables are free on the right-hand side, we want to include them always in the Dependency Pairs that will be created from the Termination Graph. For this purpose, we will use the set of predecessors of ***ParSplit***-nodes with variable head, which is defined in the following. This was already considered in [Swi05].

Definition 4.14 (U_{TG} and PU_{TG})**.** *Let TG be a Termination Graph. The set U_{TG} of all **ParSplit**-nodes with variable head is defined as*

$$\mathsf{U}_{TG} = \{t \in TG \mid \begin{array}{l} t \text{ is a } \textbf{\textit{ParSplit}}\text{-node with } t = (x\ t_1\ \ldots\ t_n) \\ \quad \text{where } x \text{ is a variable and } n > 0\} \end{array}$$

The set PU_{TG} contains all nodes s for which a node $t \in \mathsf{U}_{TG}$ exists, such that a path from s to t exists in TG. Please note that the length of this path may also be zero, i.e., $\mathsf{U}_{TG} \subseteq \mathsf{PU}_{TG}$.

The following example shows, how this function works and where free variables appear. Please note that all examples in the following are presented in a non-applicative form to increase readability.

Example 4.15 (***ParSplit***-node with variable head)**.** *As an example for a function, where we have a **ParSplit**-node with variable head, we want to consider the start term* `foldN n e fs` *for the following Haskell program.*

```
data Nats   = Z   | S Nats
data List a = Nil | Cons a (List a)

appN :: (Nats -> Nats) -> Nats -> Nats -> Nats
appN _  Z    m = m
appN f (S n) m = appN n f (f m)

foldN :: Nats -> Nats -> List (Nats -> Nats) -> Nats
foldN n e  Nil        = e
foldN n e (Cons f fs) = foldN n (appN f n e) fs
```

*As can be seen in the Termination Graph shown in figure 4.2, node M is a **ParSplit**-node with variable head. Thus, we have that $M \in \mathsf{U}_{TG}$. Furthermore, it especially holds that $G \in \mathsf{PU}_{TG}$, since a path exists from node G to node M. When we now create a Dependency Pair for the path from node A to node E, then the subterm* `appN fs0 n e` *in node G is replaced by a fresh variable on the right-hand side of the Dependency Pair. This enables us to get to a DP problem where every variable on the right-hand side also occurs on the left-hand side by filtering this argument position. So the Dependency Pair before filtering is*

$$foldN(n, e, Cons(fs0, fs1)) \quad \rightarrow \quad foldN(n, y, fs1)$$

where now the second argument position of $foldN$ is filtered, such that we get the Dependency Pair

$$foldN(n, Cons(fs0, fs1)) \quad \rightarrow \quad foldN(n, fs1)$$

A detailed description of the filtering is given in Chapter 6, where renaming is introduced.

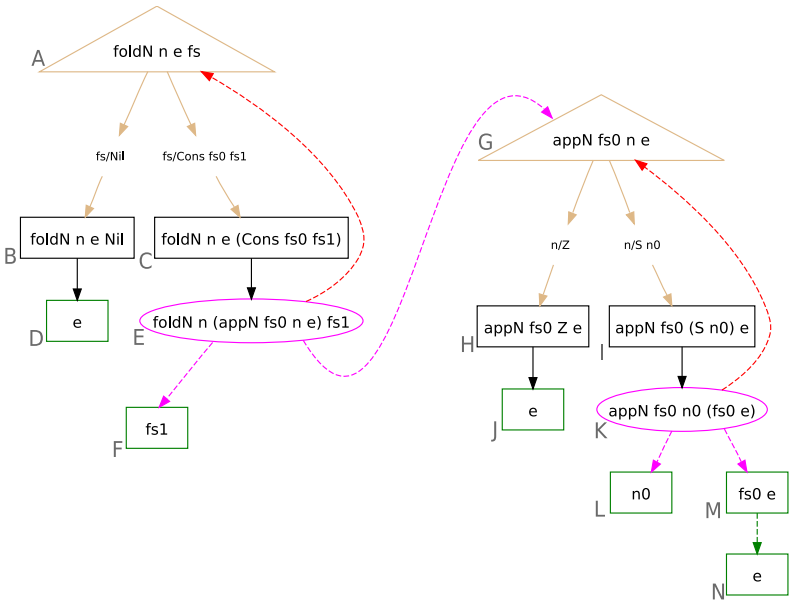

Figure 4.2: Termination Graph for `foldN n e fs` containing a ***ParSplit***-node with variable head

Please note that although all functions presented in the following are extended to also cope with ***TyCase***-nodes in the Termination Graph, the types are not considered in the created DP problems. Therefore, the running example of this section does not contain any type classes to keep the examples smaller.

The function **ev** accumulates ***Eval***-expansions in the Termination Graph. These steps are then merged into one rule, since successive ***Eval***-steps are always applicable. Also, if we can evaluate subterms on a right-hand side of a Dependency Pair further, then we can do so and use the evaluated subterm. This is illustrated in the following example.

Example 4.16 (Evaluation of subterms)**.** *We reconsider example 3.1 again, but replace the rules for the function* `p` *by the new rule* `p (S x) = x`*.*

In the Termination Graph for the start term `take u (from m)` *we see at node I the computation of the predecessor using the new function* `p`*. However, all information for this evaluation is available, which is why this node is directly evaluated by an* ***Eval****-expansion. This evaluation does not need to be reflected in the generated DP problem, since we know how this term is evaluated, namely the term node J is labelled with will always be the result. So instead of the DP problem*

$$\begin{array}{lcl} \mathcal{P} & = & \{ \quad take(S(n), from(m)) \quad \rightarrow \quad take(p(S(n)), from(S(m))) \quad \} \\ \mathcal{R} & = & \{ \quad p(S(n)) \quad \rightarrow \quad n \quad \} \end{array}$$

we can directly generate the following DP problem that has no rules and directly decrements the first argument of take:

$$\begin{array}{lcl} \mathcal{P} & = & \{ \quad take(S(n), from(m)) \quad \rightarrow \quad take(n, from(S(m))) \quad \} \\ \mathcal{R} & = & \emptyset \end{array}$$

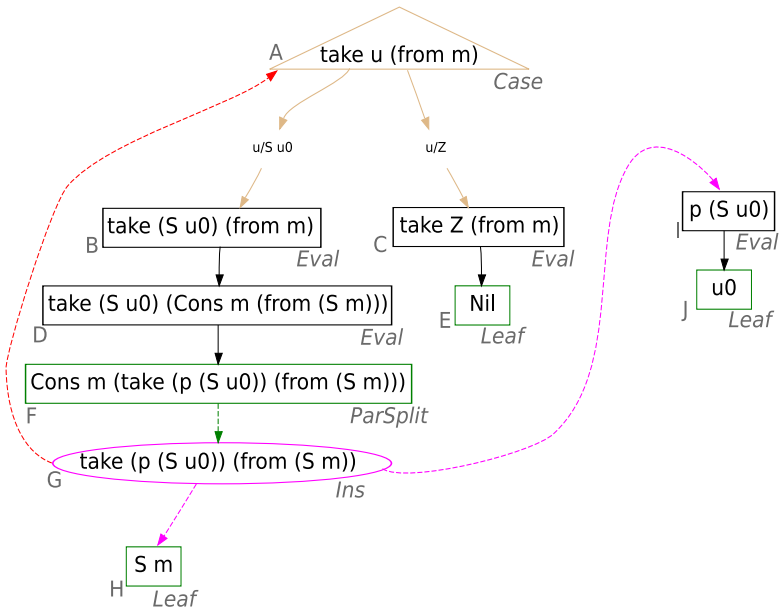

Figure 4.3: Termination Graph for `take u (from m)` with a simple predecessor computation

As stated above, applications which result in unknown values shall already be included in the created Dependency Pairs. For this purpose, the rule of **ev** for ***Ins***-nodes is changed compared to its definition in [GSSKT06], since a right-hand side of a Dependency Pair is created from such a node. Then, we only have to deal with free variables on right-hand sides of Dependency Pairs, instead of also having them on right-hand sides of rules.

This leads to the following formal definition of the function **ev**, where we have the desired effects that evaluations are performed as far as they are known and free variables are directly inserted at ***Ins***-nodes.

Definition 4.17 (**ev** with type classes)**.** *Let TG be a Termination Graph, let t be a node in it. Then:*

$$\mathbf{ev}(t) = \begin{cases} x, & \text{for a fresh variable } x, \text{ if } t \in \mathsf{U}_{TG} \\ t, & \text{if } t \text{ is a leaf, a } \textbf{\textit{Case}}\text{-node, a } \textbf{\textit{VarExp}}\text{-node, or a } \textbf{\textit{TyCase}}\text{-node} \\ \mathbf{ev}(\tilde{t}), & \text{if } t \text{ is an } \textbf{\textit{Eval}}\text{-node with child } \tilde{t} \\ \tilde{t}[x_1/s_1, \ldots, x_n/s_n], & \text{if } t = \tilde{t}[x_1/t_1, \ldots, x_n/t_n] \text{ and either} \\ & \quad t \text{ is an } \textbf{\textit{Ins}}\text{-node with the children } t_1, \ldots, t_n, \tilde{t}, \text{ or} \\ & \quad t \text{ is a } \textbf{\textit{ParSplit}}\text{-node, } \tilde{t} = (c\ x_1 \ldots x_n) \text{ for a constructor } c \\ & \qquad s_i = y_i \text{ for a fresh variable } y_i \text{ if } t_i \in \mathsf{PU}_{TG} \\ & \qquad s_i = \mathbf{ev}(t_i), \text{ otherwise} \end{cases}$$

With this definition of **ev**, we have that in example 4.16 the term created for node G is **ev**(G) = `take n (from (S m))`. This removes the necessity to add rules for this Dependency Pair, giving the DP problem that was already shown in the example. However, this is not always possible. Therefore, we must have the function **con** describing from which nodes rules have to be created. The following example shows, how this works.

Example 4.18 (Reading rules from Termination Graphs)**.** *We consider the start term* `take u (from m)` *again for the Haskell program given in example 3.1, where we use the more complicated predecessor computation given there.*

In the Termination Graph for this start term which was shown in figure 3.1 node I now is a ***Case****-node. Therefore, we do not know which rule is applicable to this term that is a subterm of the* ***Ins****-node G. Therefore, we stop at node I with the function* **ev***, as can be seen in the definition of this function above. Then the function* **con** *collects this node, in order to create rules starting in this node.*

The following definition for the function **con** differs from the definition given in [GSSKT06] by its adaption to type classes, as well. Furthermore, no rules must be created from terms that are a predecessor of an application with unknown result, since these are already approximated by a fresh variable in the created Dependency Pair, as was shown in example 4.15. Thus, no continuation is needed for such nodes.

Definition 4.19 (**con** with type classes)**.** *Let TG be a Termination Graph with a node t. Then:*

$$\mathbf{con}(t) = \begin{cases} \emptyset, & \text{if } t \text{ is a } \textbf{\textit{VarExp}}\text{-node, or } t \in \mathsf{PU}_{TG} \\ \{t\}, & \text{if } t \text{ is a } \textbf{\textit{Case}}\text{-node or a } \textbf{\textit{TyCase}}\text{-node} \\ \{\tilde{t}\} \cup \mathbf{con}(t_1) \cup \ldots \cup \mathbf{con}(t_n), & \text{if } t \text{ is an } \textbf{\textit{Ins}}\text{-node with the} \\ & \text{children } t_1, \ldots, t_n, \tilde{t} \text{ and an instantiation edge from } t \text{ to } \tilde{t} \\ \bigcup_{t' \text{ child of } t} \mathbf{con}(t'), & \text{otherwise} \end{cases}$$

When we now apply this function to the node G of the Termination Graph given in figure 3.1, then we have that it collects the node I, which is the desired behavior as stated in example 4.18.

To read rules starting in a node that was collected using the function **con**, the notion of a *rule path* is used. In [GSSKT06], a rule path is a path in the Termination Graph that starts at an ***Eval***- or a ***Case***-node and reaches the first node that is not an ***Eval***- or a ***Case***-node. So for example in the Termination Graph in figure 3.1, there are two rule paths starting in node I: the first one goes from node I to node L and the second rule path also starts in node I and ends in node M.

The definition of a rule path given in [GSSKT06] has to be extended as well in order to be able to cope with type classes. Here, ***TyCase***-nodes are handled in the same way as ***Case***-nodes.

Definition 4.20 (Rule Path considering type classes)**.** *Let TG be a Termination Graph.*

A path from a node marked with s to a node marked with t in TG is a rule path *if s and all other nodes on the path except t are* ***Eval****-,* ***Case****-, or* ***TyCase****-nodes and t is no* ***Eval****-,* ***Case****-, or* ***TyCase****-node. So t may also be a leaf.*

Thereby, ***TyCase***-nodes are like ***Case***-nodes; the only difference is that the case analysis is performed on type variables rather than on variables contained in the term. Since it is assumed that type variables and term variables have different names, the type substitutions the edges are marked with only apply to type variables.

Finally, we have to formally define from which nodes we have to read Dependency Pairs. As already indicated above, we have that ***Ins***-nodes are a call to some defined function. We want to prove that these calls cannot occur infinitely often. Since the Termination Graph is finite, an infinite evaluation can only occur in a Strongly Connected Component (SCC) of the Termination Graph. Thus, we create Dependency Pairs for the paths to ***Ins***-nodes in these SCCs. For example in the Termination Graph shown in figure 3.1 we have two SCCs: one consisting of the nodes A,B,D,F,G and another one for the nodes I,J,L,N. In node G, for example, a call to the function `take` is made again. This call comes from the node A. Generally, a call to another function results from a path starting in a node with an incoming instantiation edge. Therefore, we say that a *DP Path* starts in a node with an incoming instantiation edge and goes to a node with an outgoing instantiation edge, without traversing any of those edges. So we have that for the first SCC in the example a DP path exists from node A to node G, and in the other SCC we have a DP path from node I to node N.

Compared to the definition of a DP Path given in [GSSKT06], the following definition remains unchanged but still handles the extension to type classes. This is, because a DP Path only forbids instantiation edges, but does not need other distinctions about the type of a node.

Definition 4.21 (DP Path, [GSSKT06])**.** *Let TG be a Termination Graph, let G' be an SCC in it.*

A path p in G' from a node s to a node t is a DP Path*, iff p does not traverse instantiation edges, s has an incoming instantiation edge in G', and t has an outgoing instantiation edge in G'.*

Using these definitions of Rule Paths and DP Paths, we can read DP problems from a Termination Graph. A DP problem is created for every SCC of the Termination Graph. In the SCC, we build a Dependency Pair for every DP Path. On such a DP Path we have to consider the cases that led to another call to a defined function. Therefore, the substitutions the edges are labelled with are collected along the way and applied to the term the start node of the DP path was labelled with. This is shown in the following example.

Example 4.22 (Reading DP problems)**.** *We now want to read DP problems from the Termination Graph shown in figure 3.1, which was constructed for the start term* `take u (from m)` *and the Haskell program shown in example 3.1.*

In the first SCC, we have a DP Path from node A to node G. This path is labelled with the substitution $\delta =$ `[u/S u0]`*. Therefore, the left-hand side of our*

created Dependency Pair is (`take u (from m)`)δ = `take (S u0) (from m)`. *The right hand side is constructed from the term at node G. For this term, we use the function* **ev** *to further evaluate the subterms. Here, the function* **ev** *does not change the term as compared to the term of node G, so the Dependency Pair for this DP Path is*

$$take(S(u0), from(m)) \quad \rightarrow \quad take(p(S(u0)), from(S(m)))$$

In order to check whether this Dependency Pair can call itself again, we must have the add rules for `p` *to the DP problem. For this purpose we consider those nodes, from which Rule Paths start for all of the children of the* ***Ins****-node G. This collection of nodes is performed by the function* **con**. *In the case of node H, there are no nodes in* $\mathbf{con}(H)$, *as there are no defined functions contained. For node I we have* $\mathbf{con}(I) = \{I\}$, *so we have to add rules for all Rule Paths starting in this node.*

For the Rule Path from node I to node M, we also collect the substitutions on the path and apply it to the term of node I. The right hand side is again evaluated as far as we can, however in this case the application of **ev** *to node M does not change the term. Hence, we add the following rule to the created DP problem.*

$$p(S(Z) \quad \rightarrow \quad Z$$

For the other Rule Path from node I to node L, the substitutions are collected as well and applied to the term of node I. Hence, the following rule is also added to the created DP problem.

$$p(S(S(u00))) \quad \rightarrow \quad S(p(S(u00)))$$

However, we have to further follow the children of node L, since we must be able to evaluate the right hand side of the rule further. So we again search for nodes from which rules have to be created starting in node N. At this node, we have $\mathbf{con}(N) = \{I\}$. *For this node, we already added all rules.*

Hence, the DP problem for this SCC is the following:

$$\begin{array}{lcll} \mathcal{P} & = & \{ & take(S(u0), from(m)) \rightarrow take(p(S(u0)), from(S(m))) \quad \} \\ \mathcal{R} & = & \{ & p(S(Z) \rightarrow Z \\ & & & p(S(S(u00))) \rightarrow S(p(S(u00))) \quad \} \end{array}$$

For the SCC containing the function `p`, *a similar construction yields the following DP problem:*

$$\begin{array}{lcll} \mathcal{P} & = & \{ & p(S(S(u00))) \rightarrow p(S(u00)) \quad \} \\ \mathcal{R} & = & \emptyset & \end{array}$$

Formally, this creation of DP problems is performed by the function **dp** that makes use of the function **rl** to create the required rules. The addition of type classes to the Termination Graph does not affect the functions **dp** and **rl**. The difference in their definitions as compared to those given in [GSSKT06] is, that DP problems as defined in [GTSK05b] are used.

Definition 4.23 (**dp**, [GSSKT06])**.** *Let TG be a Termination Graph with an SCC G′.*

We define $\mathbf{dp}_{G'} = (\mathcal{P}, \emptyset, \mathcal{R}, \mathsf{a})$, *where* $\mathcal{P}$ *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 the substitutions* $\sigma_1, \ldots, \sigma_m$, $t = \tilde{t}[x_1/t_1, \ldots, x_n/t_n]$ *has an instantiation edge to* $\tilde{t}$, *and* $q \in \mathbf{con}(t_1) \cup \ldots \cup \mathbf{con}(t_n)$.

Definition 4.24 (**rl**, [GSSKT06])**.** *Let TG be a Termination Graph.*
For a node s, *we define* $\mathbf{rl}(s)$ *as the smallest set satisfying*

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

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

In order to ease the presentation in the following chapters, we define a few shorthand notations.

Definition 4.25. *Let TG be a Termination Graph.*
We say that a set of rules $\mathcal{U}$ *has been* generated from *TG, iff an SCC* G' *exists in TG such that* $\mathbf{dp}_{G'} = (\mathcal{P}, \emptyset, \mathcal{R}, \mathsf{a})$ *and either* $\mathcal{U} = \mathcal{P}$ *or* $\mathcal{U} = \mathcal{R}$.
A DP problem $(\mathcal{P}, \emptyset, \mathcal{R}, \mathsf{a})$ *is said to have* resulted from *TG, iff an SCC* G' *exists in TG such that* $\mathbf{dp}_{G'} = (\mathcal{P}, \emptyset, \mathcal{R}, \mathsf{a})$.

The extensions of the Termination Graph and the creation of DP problems, which were presented in this chapter, will be considered in the proofs of chapter 6 that introduces renaming. Thus, those proofs also prove the correctness of these extensions, i.e., from the finiteness of the created DP problems, it can still be concluded that all nodes in the originating Termination Graph are H-terminating.

But first, the next chapter shows that a Haskell program can be reduced to those elements that might be needed for the evaluation of the given start term. This means that all other elements can be disregarded, making the size of the problem smaller and therefore the analysis faster.

Chapter 5

Reduction to Necessary Components

Often, a Haskell program imports a large library, but only makes use of a few functions that are defined within that library. Thus, a lot of superfluous functions are in the scope of the current Haskell program, which are never used by the start term that is to be analyzed. A library that is automatically imported in every Haskell program is the `Prelude`, which provides a large set of functions for very different purposes. However, virtually no Haskell program uses all of the offered functions. Therefore, the idea is to restrict the analysis to only those functions which might be used in the evaluation of the start term.

This chapter shows that a Haskell program can be reduced to those *necessary components* which are needed in order to evaluate a given start term. This eases the analysis, because fewer rules and cases have to be considered. Since it cannot be decided which rules and classes must be included and which are not used, an overapproximation is done, such that all possibly used functions, data types, and classes are included.

This is formalized in the following definition.

Definition 5.1 (Necessary Components reduction)**.** *Let HP be a Haskell-Program.*

We define the reduction $\mathrm{necRed}(t)$ *of the Haskell program according to a term t. Here, if for a defined function f it holds that $f \in \mathrm{necRed}(t)$, then all rules for f are in the reduced Haskell program. Furthermore, if a type $T \in \mathrm{necRed}(t)$, then the data definition for this type is also contained in the reduced Haskell program. Last, if a class or an instance $C\ \rho \in \mathrm{necRed}(t)$, then this class or this instance is included in the reduced Haskell program, respectively.*

The reduction of HP to the necessary components for a term t is defined as follows:

- $\mathrm{necRed}(t) \supseteq \mathrm{necRed}(t')$*, for all subterms t' of t.*
- $\mathrm{necRed}(t) \ni \mathsf{error}$*, where* error *is the error function defined in the Haskell Prelude.*
- $\mathrm{necRed}(t) \ni f$*, if t contains the defined function symbol f.*

- $\text{necRed}(t) \supseteq \text{necRed}(s)$, *if* $f \in \text{necRed}(t)$ *for a defined function symbol* f, *where a rule of* f *with the right-hand side* s *exists in* HP.
- $\text{necRed}(t) \ni T$, *if* $t = \frac{s}{\tau}$ *and* τ *contains the type constructor* T.
- $\text{necRed}(t) \ni T'$, *if* $T \in \text{necRed}(t)$ *for a type constructor* T *and the data constructor definition of the type starting with* T *contains the type constructor* T'.
- $\text{necRed}(t) \ni c$, *if* $T \in \text{necRed}(t)$ *for a type constructor* T *and the type starting with* T *contains the data constructor* c.
- $\text{necRed}(t) \ni (C\ a)$, *if* t *has a class constraint of the class* C.
- $\text{necRed}(t) \ni (C\ \rho)$, *if* $(C\ a) \in \text{necRed}(t)$ *for a class* C *and* $(C\ \rho)$ *is an instance of* C.
- $\text{necRed}(t) \ni T$, *if an instance* $(C\ (T\ a_1 \ldots a_m))$ *exists in* $\text{necRed}(t)$.
- $\text{necRed}(t) \ni (C'\ a)$, *if* $(C\ a) \in \text{necRed}(t)$ *and the class* C *has a class constraint containing the class* C', *or if an instance* $(C\ \rho) \in \text{necRed}(t)$ *exists which has a class constraint containing the class* C'.

The following example shall illustrate how the reduction to the necessary components works.

Example 5.2 (Reduction to Necessary Components)**.** *Consider the Haskell program from example 2.2, where we want to collect the necessary components for the start term* $t =$`plus :: List a -> List a -> List a`.

- `error` $\in \text{necRed}(t)$, *as this holds regardless of the term.*
- `List` $\in \text{necRed}(t)$, *because this type constructor occurs in the type of the term* t.
- `Nil`, `Cons` $\in \text{necRed}(t)$, *since these are the data constructor of the type* `List` *that is in* $\text{necRed}(t)$.
- $\frac{\texttt{plus}}{\texttt{List a -> List a -> List a}} \in \text{necRed}(t)$, *as this defined function occurs in the start term.*
- $\text{necRed}(\texttt{Cons (plus x y) (plus xs ys)}) \subseteq \text{necRed}(t)$, *since this is a right-hand side of a rule for the above function* `plus`.
- $\text{necRed}(\texttt{Addition a} \Rightarrow \frac{\texttt{plus x y}}{\texttt{a}}) \subseteq \text{necRed}(t)$, *as this is a subterm of the above term.*
- `Addition a` $\in \text{necRed}(t)$, *because of the above function* `plus` *that has the class constraint* `Addition a`.
- `Addition Nats`, `Addition (List a)` $\in \text{necRed}(t)$, *as these are instances of the type class* `Addition a`.
- `Nats` $\in \text{necRed}(t)$, *as the instance* `Addition Nats` *is in* $\text{necRed}(t)$.
- `Z`, `S` $\in \text{necRed}(t)$, *since these are the data constructors for* `Nats`.

These are all components of the Haskell program that have to be regarded. Thus, the reduced Haskell program is the following:

```
data Nats   = Z   | S Nats
data List a = Nil | Cons a (List a)

class Addition a where
  plus :: a -> a -> a

instance Addition Nats where
  plus  Z    y = y
  plus (S x) y = S (plus x y)

instance Addition a => Addition (List a) where
  plus  Nil         Nil        = Nil
  plus (Cons x xs) (Cons y ys) = Cons (plus x y) (plus xs ys)
```

The reduction to the necessary components does not affect the termination behavior. This will be shown for the Termination Graph with the start term t, for which the necessary components are computed. For this purpose, a property of necRed on Termination Graphs is shown first, which is used in the proof. In the proof of this lemma, the extension to type classes is also considered.

Lemma 5.3 (Transitivity of necRed in Termination Graphs)**.** *Let TG be a Termination Graph for a start term t.*

Then for every node $\underline{cs} \Rightarrow s$ *in TG and every child* $\underline{cs'} \Rightarrow s'$ *of* $\underline{cs} \Rightarrow s$ *it holds that* $\text{necRed}(\underline{cs'} \Rightarrow s') \subseteq \text{necRed}(\underline{cs} \Rightarrow s)$.

Proof. Case analysis according to the expansion rule applied to $\underline{cs} \Rightarrow s$ is used.

Eval
In the case of an ***Eval***-node, a rule $(f\ t_1 \dots t_n) = r$ for a defined function symbol f must exist which is applied to $s|_{\mathbf{e}(s)} = (f\ t_1 \dots t_n)\sigma$. Therefore, the child s' has the form $s[r\sigma]_{\mathbf{e}(s)}$. Now $\text{necRed}(s') \subseteq \text{necRed}(s) \cup \text{necRed}(r\sigma)$, and since r is a right-hand side of a defined function symbol in s, $\text{necRed}(s) \supseteq \text{necRed}(r)$. Since for all variables x, $\sigma(x)$ is a subterm of s, it holds that $\text{necRed}(s) \supseteq \text{necRed}(s')$.

Case
Here, $s|_{\mathbf{e}(s)} = x$ for a variable x of type $(T\ a_1 \dots a_m)\rho$. Therefore, $T \in \text{necRed}(s)$ and thus $c \in \text{necRed}(s)$ for all data constructors c of the type $(T\ a_1 \dots a_m)$. For every child $(s', \underline{cs})$ of $(s, \underline{cs})$ it holds that $s' = s\delta$ where δ is of the form $[x/(c'\ x_1 \dots x_n)]$ for a data constructor c' of the type $(T\ a_1 \dots a_m)$ and fresh variables $x_1, \dots, x_n$. All type constructors and therefore also the accompanying data constructors of the type of a variable x_i are also contained in $\text{necRed}(s)$, since these type constructors must also appear in the data constructor definition of the type $(T\ a_1 \dots a_m)$. Thus $\text{necRed}(s) \supseteq \text{necRed}(s')$.

TyCase
In this case, $s|_{\mathbf{e}(s)} = (g\ s_1 \dots s_n)$ for a defined function symbol g that is a member of a class $(C\ a) \in \underline{cs}$. Therefore $C \in \text{necRed}(s)$, which implies that every instance $(C\ \rho)$ of the class C is contained in $\text{necRed}(s)$. As the inclusion

of classes is transitively closed, necRed(s) $\supseteq$ necRed(s') for every child $\underline{cs'} \Rightarrow s'$ of $\underline{cs} \Rightarrow s$ where $s' = s$ and $\underline{cs'} = \underline{cs}[a/\rho]$.

VarExp
Now $s = (f\ s_1 \ldots s_{n'})$ where the defined function symbol f has arity $n > n'$. For the only child $\underline{cs} \Rightarrow s'$ of $\underline{cs} \Rightarrow s$ it holds that $s' = (s\ x)$ for a fresh variable x of type $(T\ a_1 \ldots a_m)$. This type is specified in the type of f, therefore it is also contained in necRed(s), yielding necRed(s) $\supseteq$ necRed(s').

ParSplit
For a ***ParSplit***-node the term s has the form $(c\ s_1 \ldots s_n)$ where c is a data constructor or a variable. The children of $\underline{cs} \Rightarrow s$ are $\underline{cs} \Rightarrow s_i$ for $1 \leq i \leq n$. Since every s_i is a subterm of s, the definition of necRed directly gives necRed(s) $\supseteq$ necRed(s_i).

Ins
The term s now has the form $\tilde{s}[x_1/s_1, \ldots, x_n/s_n]$ and the children $s_1, \ldots, s_n, \tilde{s}$. For every s_i the same argument as in the case for ***ParSplit***-nodes it follows that necRed(s) $\supseteq$ necRed(s_i). For the term $\tilde{s}$ it also holds that necRed(s) $\supseteq$ necRed($\tilde{s}$), since the types of the variables $x_1, \ldots, x_n$ are the same as those of $s_1, \ldots, s_n$, the class constraints implied by $\tilde{s}$ are a subset of the class constraints implied by s, and all defined function symbols in $\tilde{s}$ are also contained in s.

□

From this lemma it follows that the Termination Graph can already be built with the reduced program.

Theorem 5.4 (necRed preserves Termination Graphs)**.** *Let HP be a Haskell-Program, TG be a Termination Graph for the Haskell-Program HP with the start term t.*

Then TG is also a Termination Graph for the Haskell-Program necRed(t) *with the start term t.*

Proof. The proposition follows directly from lemma 5.3, since necRed(t) $\supseteq$ necRed(s) for every node s in the Termination Graph TG.

Therefore, every edge between nodes in TG depends only on elements from necRed(t).

□

The presented reduction to the necessary components improves the speed of the analysis when using large libraries, from which only a few functions and types are used. Especially for the Prelude, this is the case most of the time, since it contains functions which are seldomly used together in a program and do not depend on each other.

Thus, it suffices to construct the Termination Graph for a start term using only the reduced Haskell program. This results in a faster construction, because less memory is occupied and fewer rules have to be considered when a term is to be evaluated.

Chapter 6

Renaming

For Termination Graphs, it holds that an instance of a term can only be evaluated to an instance of another term, if there is a path from the first term to the other term. Furthermore, we might have that an SCC is built from a number of cycles, where the evaluation must leave one cycle in order to get to another cycle. However, it might be the case that the terms occuring on these cycles might match each other. Then, the evaluation in the DP problem for that SCC might switch between the rules for the different cycles, while in the original program this was not possible.

In order to distinguish such separated cycles in an SCC, the idea is to rename defined functions in the resulting DP problem, such that a left-hand side of a Dependency Pair can only be reached if the terms resulted from nodes that were connected via an instantiation edge. In order to separate cycles, different nodes are assigned different names, except for the case where there is an instantiation edge, i.e., where we have explicitly identified a term as an instance of another term.

Example 6.1 (Different cycles in Termination Graph)**.** *For the following Haskell program, the start term* `f x' y'` *shall be proven terminating.*

```
data Nats = Z | S Nats

class Terminate a where
    terminate :: Nats -> Nats -> a

instance Terminate Bool where
    terminate (S x) y = terminate x (S y)
    terminate  Z    y = f Zero Zero


instance Terminate Nats where
    terminate x (S y) = terminate (S x) y
    terminate x  Z    = f Zero Zero

f :: Terminate a => Nats -> Nats -> a
f (S x) (S y) = terminate x y
```

As we can see in the program, the call from the function `f` *to the function* `terminate` *is dependent on the return type of* `f`*. However, once an implementation of the function* `terminate` *has been chosen, this implementation does not call the other implementation, because the call to* `terminate` *on the right-hand sides then always have the same return type as the function* `f`*.*

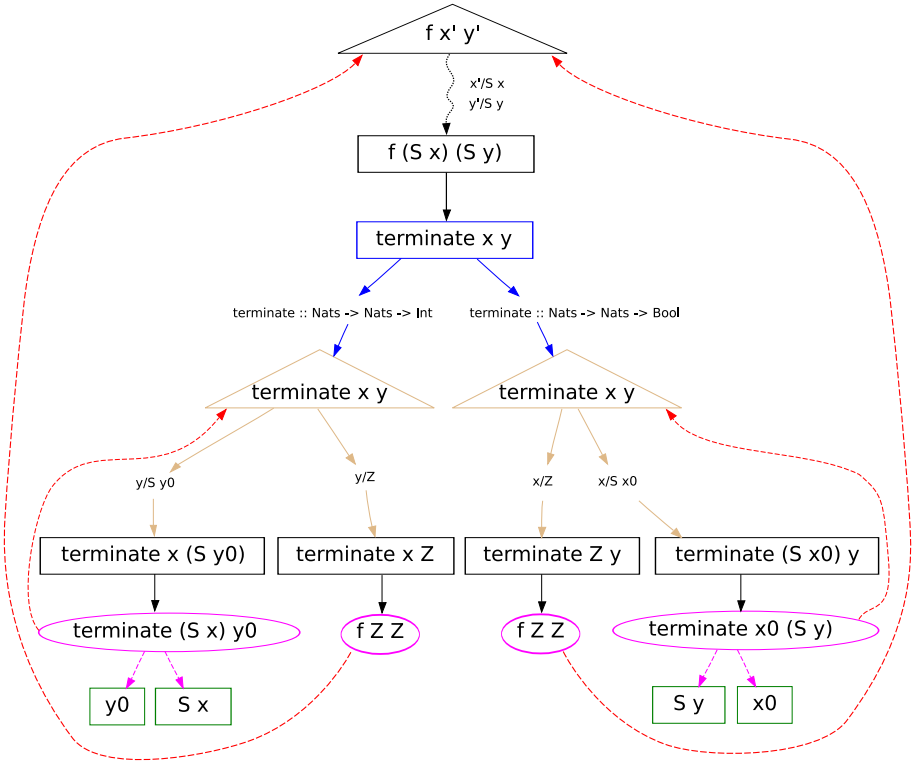

Figure 6.1: Termination Graph for `f x' y'`, illustrating renaming

The Termination Graph is sketched in figure 6.1. Here, the full case analysis for `f x' y'` *has been left out to increase readability; all other cases would lead to the term* `error []`*.*

This Termination Graph contains exactly one SCC. For this SCC, the following DP problem is created without renaming:

$$
\begin{array}{lcll}
\mathcal{P} & = & \{ & f(S(x), S(Z)) \to f(Z, Z) \\
 & & & f(S(Z), S(y)) \to f(Z, Z) \\
 & & & f(S(x), S(S(y0))) \to terminate(S(x), y0) \\
 & & & f(S(S(x0)), S(y)) \to terminate(x0, S(y)) \\
 & & & terminate(x, S(y0)) \to terminate(S(x), y0) \\
 & & & terminate(x, Z) \to f(Z, Z) \\
 & & & terminate(S(x0), y) \to terminate(x0, S(y)) \\
 & & & terminate(Z, y) \to f(Z, Z) \quad \} \\
\mathcal{R} & = & \emptyset &
\end{array}
$$

It can be observed that in this DP problem, the Dependency Pairs starting with `terminate` *reflect both instances. Therefore, an infinite chain exists:*

$$\begin{array}{lll} terminate(S(Z), Z) & \to_{\mathcal{P}} & terminate(Z, S(Z)) \\ & \to_{\mathcal{P}} & terminate(S(Z), Z) \\ & \to_{\mathcal{P}} & \ldots \end{array}$$

The infinite chain exists only, because the instances of the function `terminate` *can call each other, which is not possible in the original Haskell program. This can also be observed in the Termination Graph, where the cycles corresponding to the recursive calls of* `terminate` *are not connected via an instantiation edge. In order to reflect this in the created DP problem, different function names shall be assigned to the different cycles, such that the overlap of the names does not make the cycles overlap.*

So if we assign the new names "new_terminate0*" and "new_terminate*1*" to the two cycles, then the created DP problem would look as follows:*

$$\begin{array}{lllll} \mathcal{P} = \{ & f(S(x), S(Z)) & \to & f(Z, Z) & \\ & f(S(Z), S(y)) & \to & f(Z, Z) & \\ & f(S(x), S(S(y0))) & \to & new\_terminate0(S(x), y0) & \\ & f(S(S(x0)), S(y)) & \to & new\_terminate1(x0, S(y)) & \\ & new\_terminate0(x, S(y0)) & \to & new\_terminate0(S(x), y0) & \\ & new\_terminate0(x, Z) & \to & f(Z, Z) & \\ & new\_terminate1(S(x0), y) & \to & new\_terminate1(x0, S(y)) & \\ & new\_terminate1(Z, y) & \to & f(Z, Z) & \} \\ \mathcal{R} = \emptyset & & & & \end{array}$$

Here, now the two cycles for the two instances of `terminate` *are separated by these names and cannot call each other. This renamed DP problem is finite, as can be shown by the Size-Change processor [TG05], for example.*

Furthermore, not all arguments of a term are required for the evaluation: Only the substitutions a DP or rule path was labelled with determine the applicable rule. Therefore, only these subterms shall be considered, and not the complete term. Because substitutions only change variables of a term, only the variables have to be present in the resulting term. As we will see in the following, this will lead to first-order DP problems which are better suited for automatic termination analysis than the previously used applicative DP problems. This is the case, because the Dependency Graph can be constructed more efficiently. The Dependency Graph for applicative DP problems [GTSK05a] entails a large number of unifications, whereas these unifications can already be avoided if the head symbols are different. Furthermore, we no longer have the problem of functions being used in different arities, which makes the A-transformation [GTSK05a] from applicative DP problems to first-order DP problems fail. Then the Reduction Pair processor [GTSK05b] is less powerful, because standard reduction orders focus on the root symbol, as is also mentioned in [GTSK05a]. An example where the A-Transformation is not applicable is given below.

Example 6.2 (Renaming collecting only variables)**.** *Consider the following Haskell program, which defines the addition of two natural numbers by using a higher-order function. For this program, termination of the start term* `add n m` *shall be analyzed.*

```
data Nats   = Z   | S Nats

appN :: (Nats -> Nats) -> Nats -> Nats -> Nats
appN _  Z    m = m
appN f (S n) m = appN f n (f m)

add :: Nats -> Nats -> Nats
add = appN S
```

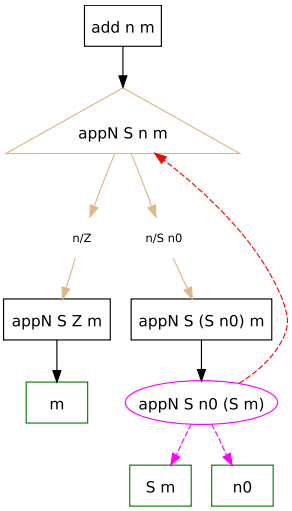

Figure 6.2: Termination Graph for `add n m`, using a higher-order function

For the given start term, the Termination Graph shown in figure 6.2 is constructed. From this Termination Graph, the following applicative DP problem is created:

$$\begin{array}{lcll} \mathcal{P} & = & \{ & \mathsf{app}^{\#}(\mathsf{app}(\mathsf{app}(appN, S), \mathsf{app}(S, n)), m) \quad \rightarrow \\ & & & \mathsf{app}^{\#}(\mathsf{app}(\mathsf{app}(appN, S), n), \mathsf{app}(S, m)) \qquad\qquad \} \\ \mathcal{R} & = & \emptyset & \end{array}$$

This DP problem cannot be handled by the A-Transformation, because of the symbol S occuring both with arity 0 and with arity 1. Therefore, it is rather hard to show finiteness of this DP problem.

However, if only the variables of the DP path are considered, then we have a first-order DP problem:

$$\begin{array}{lcll} \mathcal{P} & = & \{ & appN(S(n0), m) \quad \rightarrow \quad appN(n0, S(m)) \quad \} \\ \mathcal{R} & = & \emptyset & \end{array}$$

This is a finite DP problem, because in every chain the first argument decreases. This can also be shown with the Size-Change processor [TG05], for example.

6.1 Renaming nodes of a Termination Graph

The function bR_{TG} (mnemonic: base-Rename) assigns every node of the Termination Graph TG a new term, using a new function name for every node, except for ***Ins***-nodes: Here, the newly assigned function name of the node the instantiation edge points to is used. These new terms only consider those terms, which are affected by substitutions on edges of the Termination Graph. The idea behind this is, that the rules only demand these, i.e., they suffice to do evaluation steps. This is illustrated in the following example.

Example 6.3 (Renaming nodes of the Termination Graph)**.** *For the start term* `take u (from m)`*, which was considered in example 3.1, the Termination Graph shall be renamed.*

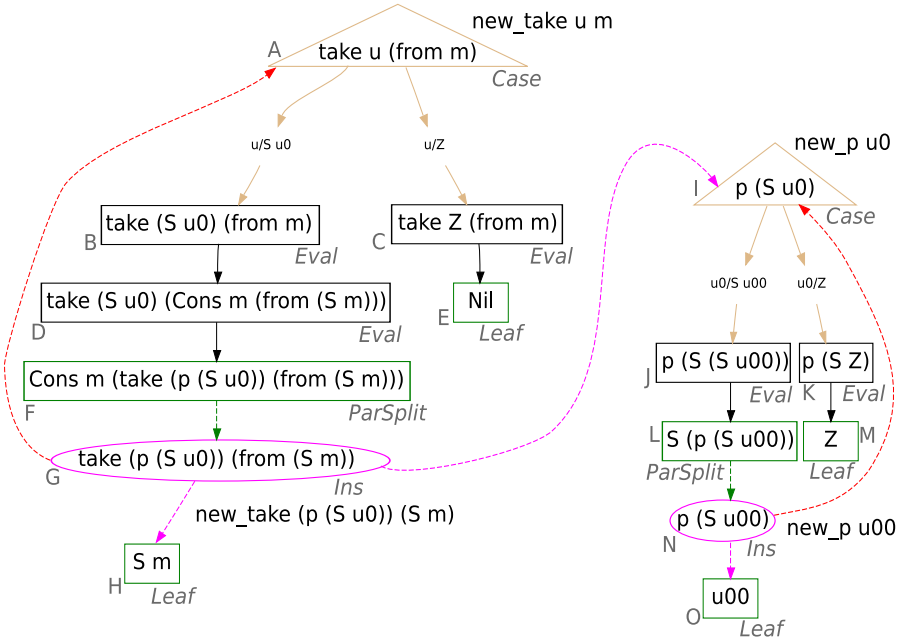

Figure 6.3: Termination Graph for `take u (from m)` with renaming

The Termination Graph where the renamed terms are added is shown in figure 6.3. Here, the fresh name "`new_take`*" is assigned to node A. To build the term, all variables of node A are collected and appended as arguments of this fresh function. Therefore, the renamed term for this node is* `new_take u m`*.*

At the ***Ins****-node G, we have a call to node A again. Therefore, it is assigned the same name "*`new_take`*". As arguments of this term, we use the renamed subterms that are present in node H and in node I. Constructors must not be renamed, because these determine the path to take at* ***Case****-nodes. Therefore, for node H the renamed term is the same as the node is labelled with. For node I, we again assign a new name, in this case "*`new_p`*", and collect all variables. Then this term is inserted into the renamed term of node G. The new term at node N is built in a similar fashion to that of node G. Again, the name of the node the instantiation edge points to is used and the renamed subterms are*

inserted. As the only subterm is the variable `u00` *and variables are not renamed, we have that the renamed term for node N is* `new_p u00`.

How the terms are constructed formally is defined in the following. This function will yield the results presented in the above example.

Definition 6.4 (bR_{TG})**.** *Let TG be a Termination Graph, let t be a node in TG labelled with a term* $(f\ t_1 \dots t_n)$.

If $\{v_1, \dots, v_m\} = \mathcal{V}_{\mathsf{H}}(t)$ *and* $\{a_1, \dots, a_k\} = \mathcal{V}_{\mathsf{T}}(t)$*, then*

$$\mathrm{bR}_{TG}(t) = \begin{cases} t, & \text{if } t \text{ is a leaf} \\ f_t\ v_1 \dots v_m\ a_1 \dots a_k, & \text{if } t \text{ is an } \textbf{\textit{Eval}}\text{-node, a } \textbf{\textit{VarExp}}\text{-node, a } \textbf{\textit{Case}}\text{-node, or a } \textbf{\textit{TyCase}}\text{-node} \\ f\ \mathrm{bR}_{TG}(t_1) \dots \mathrm{bR}_{TG}(t_n), & \text{if } t \text{ is a } \textbf{\textit{ParSplit}}\text{-node} \\ f_{\tilde{t}}\ \mathrm{bR}_{TG}(s_1) \dots \mathrm{bR}_{TG}(s_l)\ \rho_1 \dots \rho_v, & \text{if } t \text{ is an } \textbf{\textit{Ins}}\text{-node labelled with } \tilde{t}[x_1/s_1 \dots x_l/s_l, b_1/\rho_1, \dots, b_v/\rho_v], \\ & \text{an instantiation edge exists from } t \text{ to } \tilde{t}, \text{ and} \\ & \mathrm{bR}_{TG}(\tilde{t}) = f_{\tilde{t}}\ x_1 \dots x_l\ b_1 \dots b_v \end{cases}$$

Here, f_t *is a new function symbol that is assigned to the node t.*

As seen in the definition of H-terminating terms, we need substitutions to argue about whether a term is H-terminating. Therefore, the presented renaming must also be applied to substitutions.

Definition 6.5 (Renaming in substitutions)**.** *Let TG be a Termination Graph, let* σ *be a substitution.*

Then $\sigma_{\mathrm{ren}_{TG}}$ *is defined as* $\sigma_{\mathrm{ren}_{TG}}(x) = \mathrm{bR}_{TG}(\sigma(x))$.

Substitutions may introduce new terms that do not correspond to any node in the Termination Graph. Therefore, we have to leave these parts untouched, while the part that corresponds to a node in the Termination Graph must be renamed.

Definition 6.6 (Application of bR_{TG} to general terms)**.** *If* bR_{TG} *is applied to a term* $s = (g\ s_1 \dots s_u)$ *that is an instance* $s'\sigma$ *of a node labelled with* $s' = (g\ s'_1 \dots s'_u)$*, then* $\mathrm{bR}_{TG}(s) = \mathrm{bR}_{TG}(s')\sigma_{\mathrm{ren}_{TG}}$.

If bR_{TG} *is applied to a term t that is not an instance of a node in TG, then* $\mathrm{bR}_{TG}(t) = t$.

The function bR_{TG} shall be applied when creating DP problems. The idea is to use it prior to applying the substitutions collected on a DP or a rule-path.

Example 6.7 (Application of bR_{TG} to create Dependency Pairs and rules)**.** *We want to create DP problems from the renamed Termination Graph in example 6.3.*

For this purpose, the renamed terms from example 6.3 are being used. Therefore, for the DP Path from node A to node G we have collected the substitution $[u/S(u0)]$ *and hence get the Dependency Pair*

$$\begin{array}{lcl} new\_take(u, m)\ [u/S(u0)] & \to & new\_take(p(u0), S(m)) \\ & = & \\ new\_take(u0, m) & \to & new\_take(p(u0, S(m)) \end{array}$$

For the Rule Paths, the construction is similar. In the example, we have two rule paths starting in node I and ending in node L and node M, respectively. For the Rule Path to node M, we find the substitution $[u0/Z]$ *on the path, and therefore get the rule*

$$\begin{array}{lcl} new\_p(u0)\ [u0/Z] & \to & Z \\ & = & \\ new\_p(Z) & \to & Z \end{array}$$

For the other Rule Path ending in node L, we insert the renamed subterm $new\_p(u00)$ *of node N as argument of the constructor S that occurs at node L. Therefore, the renamed rule for this Rule Path is*

$$\begin{array}{lcl} new\_p(u0)\ [u0/S(u00)] & \to & S(new\_p(u00)) \\ & = & \\ new\_p(S(u00)) & \to & S(new\_p(u00)) \end{array}$$

because we find the substitution $[u0/S(u00)]$ *on the path from node I to node L.*

This construction of renamed Dependency Pairs and rules is performed by the function ren_{TG} that is defined below.

Definition 6.8 (ren_{TG})**.** *Let* $\mathcal{R}$ *be a set of rules generated from a Termination Graph TG.*

Then we define

$$\mathrm{ren}_{TG}(\mathcal{R}) = \{\mathrm{bR}_{TG}(s)\sigma_1 \ldots \sigma_m \to \mathrm{bR}_{TG}(\mathbf{ev}(t)) \mid s\sigma_1 \ldots \sigma_m \to \mathbf{ev}(t) \in \mathcal{R}\}$$

Instead of using the function $\mathbf{dp}_{G'}$ for an SCC G', now the function $\mathbf{dpRen}_{G'}$ is used, which renames the terms. It simply applies the function ren_{TG} to both the Dependency Pairs and the rules.

Definition 6.9 (**dpRen**)**.** *Let TG be a Termination Graph containing an SCC* G'*, and let* $\mathbf{dp}_{G'} = (\mathcal{P}, \emptyset, \mathcal{R}, \mathsf{a})$.

Then $\mathbf{dpRen}_{G'} = (\mathrm{ren}_{TG}(\mathcal{P}), \emptyset, \mathrm{ren}_{TG}(\mathcal{R}), \mathsf{a})$.

Using this function to generate DP problems, we have the following renamed DP problems for example 6.3.

Example 6.10 (Renamed DP problems)**.** *From the Termination Graph shown in figure 6.3, the following renamed DP problems are created, where the Dependency Pairs and rules are as presented in example 6.7:*

$$\begin{array}{lcll} \mathcal{P}_1 & = & \{ & new\_take(S(u0), m) \to new\_take(u0, S(m)) \ \} \\ \mathcal{R}_1 & = & \{ & new\_p(Z) \to Z \\ & & & new\_p(S(u00)) \to S(new\_p(u00)) \ \} \end{array}$$

and

$$\begin{array}{lcll} \mathcal{P}_2 & = & \{ & new\_p(S(u00)) \to new\_p(u00) \ \} \\ \mathcal{R}_2 & = & \emptyset & \end{array}$$

It should be noted that those parts of the renamed terms which are not created from type information are always first order terms. For the types that are appended to the terms we have that higher-order terms might exist, which is the case for constructor classes. Thus, we employ a higher-order encoding for the type information only, where we use an applicative notation using a binary symbol app. Since type information is always an argument of a defined function, the symbol app will be a constructor symbol, so it does not imply the problems of the traditional approach [GTSK05a], where app is a defined symbol.

An example that shall illustrate why the type terms have to be represented in a higher-order encoding is given below.

Example 6.11 (Encoding of type terms)**.** *Consider the following Haskell function*

```
h :: Monad m => a -> (m b) -> Bool
h x mi = h mi mi
```

and the start term `h x mi`.

As can be seen, the type of the first argument of `h` *is changing: On the left-hand side the type of* `x` *is* `a`*, while on the right-hand side the term* `mi` *has the type* `(m b)`*. Here, the latter type is an application of an argument to a variable, which is not allowed in first-order terms.*

To represent this term, a higher-order encoding using a binary constructor app *is employed, as stated previously. Thus, the DP problem that is created looks as follows:*

$$\begin{array}{lcl} \mathcal{P} & = & \{ \quad new\_h(x, mi, a, m, b) \quad \to \quad new\_h(mi, mi, \mathsf{app}(m, b), m, b) \quad \} \\ \mathcal{R} & = & \emptyset \end{array}$$

The approach which was described above might result in problems that violate the variable condition, i.e., where there is a variable on a right-hand side which does not occur on the left-hand side. This is corrected by filtering and by replacing type variables.

Definition 6.12 (Correction of DP problems)**.** *Let* $(\mathcal{P}, \emptyset, \mathcal{R}, \mathsf{a})$ *be a DP problem that resulted from a Termination Graph.*

Whenever there is a variable $x \in \mathcal{V}_{\mathsf{H}}(s_i) \setminus \mathcal{V}_{\mathsf{H}}(l)$ *for a Dependency Pair* $l = F(t_1, \ldots, t_n) \to G(s_1, \ldots, s_m) \in \mathcal{P}$*, the tuple symbol* G *will be filtered, i.e., all terms* $G(u_1, \ldots, u_m)$ *in* $\mathcal{P}$ *will be replaced by* $G(u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_m)$*. This replacement is repeated until there are no more free variables in all right-hand sides of the Dependency Pairs.*

Whenever there is a type variable $a \in \mathcal{V}_{\mathsf{T}}(r) \setminus \mathcal{V}_{\mathsf{T}}(l)$ *that occurs on a right-hand side of a Dependency Pair* $l \to r \in \mathcal{P}$ *or on the right-hand side of a rule* $l \to r \in \mathcal{R}$*, the type variable* a *is replaced by the new type constructor* TyUnknown *which has no data constructors.*

For corrected DP problems, it holds that $\mathcal{P}$ and $\mathcal{R}$ are now first-order Term Rewrite Systems, as introduced in section 2.2. This is, because by construction no left-hand side of a rule is a variable, and because of the correction, all variables of the right-hand side are now contained in the left-hand side of the rule.

As desired, the application of the correction does not change the termination behavior, which will be shown next.

Lemma 6.13 (Termination properties of corrected DP problems)**.** *A DP problem* $(\mathcal{P}, \emptyset, \mathcal{R}, \mathsf{a})$ *that resulted from a Termination Graph is finite if the corrected DP problem is finite.*

Proof. This lemma is shown in two stages. First, it is shown that the repeated filtering does not destroy chains, and therefore any infinite chain is preserved. Second, it is shown that fixing a free type variable to a fresh type constructor (which is unknown in the program), the evaluation of such a term cannot be changed.

Let $(\mathcal{P}, \emptyset, \mathcal{R}, \mathsf{a})$ be a DP problem that resulted from a Termination Graph and let $s_1 \to t_1, s_2 \to t_2, \ldots$ be a (possibly infinite) chain of Dependency Pairs in $\mathcal{P}$, which implies that $t_i \to^*_{\mathcal{R}} s_{i+1}$ for every $i \in \mathbb{N}$. Assume that t_k has the form $G(t_{k,1}, \ldots, t_{k,m})$, where the j-th argument is filtered away in the corrected term. Let $s_{k+1} = G(s_{k+1,1}, \ldots, s_{k+1,m})$. Then $t_k \to^*_{\mathcal{R}} s_{k+1}$, and since there are no rules for the tuple symbol G in $\mathcal{R}$, it must hold that $t_{k,l} \to^*_{\mathcal{R}} s_{k+1,l}$ for every $1 \leq l \leq m$. Therefore, also for the filtered terms that do not contain the j-th argument anymore, it holds that $G(t_{k,1}, \ldots, t_{k,j-1}, t_{k,j+1}, \ldots, t_{k,m}) \to^*_{\mathcal{R}} G(s_{k+1,1}, \ldots, s_{k+1,j-1}, s_{k+1,j+1}, \ldots, s_{k+1,m})$. Hence, every chain in the original DP problem remains a chain in the filtered DP problem, proving the first part.

Assume there is a free type variable a on a right-hand side. Since this type variable must then also been introduced on a right-hand side of a Haskell rule, there cannot be any class constraints for this type variable. This is due to the requirement that any class constraint only refers to variables that are in the type signature [Jon03]. Therefore, no constructors can influence the evaluation, since there are no constructors for a variable type, and no class constraint can influence the choice of rules for the function that is to be evaluated. Therefore, one can safely replace the type variable by an unknown type TyUnknown which has no constructors.

□

The correction of rules shall be illustrated using two examples. First, we present an example for the filtering of free variables on right-hand sides of Dependency Pairs.

Example 6.14 (Filtering)**.** *The start term* `foldl g z xs` *shall be analyzed. Here is the definition of the function* `foldl`*, as given in the Prelude:*

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

From the Termination Graph for the given start term, the following renamed Dependency Pair is created:

$$new\_foldl(g, z, x : xs) \quad \to \quad new\_foldl(g, v, xs)$$

In this Dependency Pair, the fresh variable v is introduced by **ev** *on the right-hand side. This variable must be filtered. For this purpose, the second argument of the function new_foldl is dropped. Thus, the filtered Dependency Pair looks as follows:*

$$new\_foldl(g, x : xs) \quad \to \quad new\_foldl(g, xs)$$

This is now a standard rule that can be processed further.

The second example for the correction of rules illustrates the second step in the correction, namely the replacement of free type variables. These might occur, when a term is used on a right-hand side of a Haskell rule that has a free type variable.

Example 6.15 (Replacement of unbound type variables)**.** *The following Haskell function always returns the* `Int` *number* 0*, but does so in a rather complicated way:*

```
zeroComplicated = length []
```

In order to return 0*, the function* `zeroComplicated` *calls the function* `length` *with the empty list as argument. The empty list* `[]` *has the type* `[a]`*, where the type variable* a *is fresh on the right-hand side, i.e., there is no other term that has the type* a*.*

So if rules have to be created for the function `zeroComplicated`*, the following rule is created*[1]*:*

$$new\_zeroComplicated \quad \rightarrow \quad new\_length([], a)$$

Here, it can be observed that the type variable a *occurs free on the right-hand side of a rule. But this variable does not influence the evaluation of the term* `length []`*. Thus, it could as well be replaced by an unknown type constructor which shall be called* TyUnknown*. This is, because the rules corresponding to the function* `length` *expect a second argument. So the corrected rule will look as follows:*

$$new\_zeroComplicated \quad \rightarrow \quad new\_length([], \mathsf{TyUnknown})$$

This is now a valid first-order rule.

The following section will show that whenever a term is not H-terminating, then there exists an infinite chain for some renamed DP problem, i.e., then a DP problem exists that is not finite.

6.2 Correctness of Renaming

This section shows that when applying the previously presented renaming with a following correction of terms, then we can still deduce the start term to be H-terminating, if we could prove finiteness of all resulting DP problems.

Theorem 6.16 (ren_{TG} preserves soundness)**.** *Let* TG *be a Termination Graph.*

If the DP problem $\mathbf{dpRen}_{G'}$ *is finite for all SCCs* G' *of* TG*, then all nodes* t *in* TG *are* H*-terminating.*

To show theorem 6.16, we will show that a non-H-terminating term implies an infinite chain in a created DP problem. In order to prove this property, the proofs of [GSSKT06] are modified to include type classes and renaming. Furthermore, we modified the definition of the function **ev**, where we introduced fresh variables already in the terms read from ***Ins***-nodes. This is also considered in the following proofs.

[1] In order to achieve this, one must circumvent the function **ev**, so that the result of `length` is not directly inserted. This can be done by inserting an instantiation edge from `length []` to the general function call `length xs`.

We will show theorem 6.16 in four stages. First, a few helper lemmas are introduced. In the second stage, it is shown that every evaluation of subterms can also be done with the renamed rules. Then, the third step shows that an infinite reduction always traverses an ***Ins***-node, for which an infinite reduction exists, as well. Here, the reduction must also consider the extension by arbitrary H-terminating terms, as it is required for H-termination. In the fourth and last step, the theorem will be proven, by giving an infinite chain for such an infinite evaluation.

First, we define a relation that allows rewriting also below or besides the Haskell redex. This will be used in the lemmas in order to reason about reductions, where a corresponding Haskell reduction could be extended in order to reach such a term. This definition is taken directly from [GSSKT06].

Definition 6.17 ($\Rightarrow_{\mathsf{H}}$, [GSSKT06])**.** *For two terms s and t we define $s \Rightarrow_{\mathsf{H}} t$, iff s rewrites to t on a position π that is not strictly above $\mathbf{e}(s)$, using the first equation in the Haskell program that matches $s|_\pi$.*

The relation $\Rightarrow_{\mathsf{H}}$ models the application of the function **ev**, where evaluations may occur on positions that are not on the evaluation position. This was already shown in Lemma 14 of [GSSKT06], however, it has to be modified and shown again due to the changes made to **ev**.

Lemma 6.18 (Modelling **ev** with $\Rightarrow_{\mathsf{H}}$)**.** *Let TG be a Termination Graph and let σ be a substitution.*

If $s\sigma \Rightarrow^*_{\mathsf{H}} x\sigma$ holds for all fresh variables x that are introduced by **ev** *at **ParSplit**- and **Ins**-nodes for a term s, then $t\sigma \Rightarrow^*_{\mathsf{H}} \mathbf{ev}(t)\sigma$ holds for all nodes t of TG.*

Proof. This lemma is shown inductively, using the edge relation of TG after removing all instantiation edges. This is a well-founded relation, since after removing the instantiation edges the graph becomes acyclic.

We only have to consider the cases where $\mathbf{ev}(t) \neq t$ and $t \notin \mathsf{U}_{TG}$.

If t is an ***Eval***-node with child node $\tilde{t}$, then $t\sigma \rightarrow_{\mathsf{H}} \tilde{t}\sigma$. Thus, $t\sigma \Rightarrow^*_{\mathsf{H}} \tilde{t}\sigma$. The induction hypothesis is applicable to $\tilde{t}$, since it is the child of t. It gives us $\tilde{t}\sigma \Rightarrow^*_{\mathsf{H}} \mathbf{ev}(\tilde{t})\sigma$. Since $\mathbf{ev}(t) = \mathbf{ev}(\tilde{t})$, we have $t\sigma \Rightarrow_{\mathsf{H}} \tilde{t}\sigma \Rightarrow^*_{\mathsf{H}} \mathbf{ev}(\tilde{t})\sigma = \mathbf{ev}(t)\sigma$.

In the last case to consider we have $t = \tilde{t}[x_1/t_1, \ldots, x_n/t_n, y_1/s_1, \ldots, y_m/s_m]$ and either t is a ***ParSplit***-node starting with a constructor, or t is an ***Ins***-node with an instantiation edge to a node $\tilde{t}$. We assume that $t_1, \ldots, t_n \notin \mathsf{PU}_{TG}$ and $s_1, \ldots, s_m \in \mathsf{PU}_{TG}$. For every $1 \leq i \leq m$, the term s_i is replaced by a fresh variable z_i in the term $\mathbf{ev}(t)$. For these, we have by assumption that $s_i\sigma \Rightarrow^*_{\mathsf{H}} z_i\sigma$.

The induction hypothesis can be applied to the children t_i, since these are not connected via an instantiation edge. Thereby, we get:

$$\begin{array}{lcl} t\sigma & = & \tilde{t}[x_1/t_1, \ldots, x_n/t_n, y_1/s_1, \ldots, y_m/s_m]\sigma \\ & = & \tilde{t}\sigma[x_1/t_1\sigma, \ldots, x_n/t_n\sigma, y_1/s_1\sigma, \ldots, y_m/s_m\sigma] \\ & \Rightarrow^*_{\mathsf{H}} & \tilde{t}\sigma[x_1/\mathbf{ev}(t_1)\sigma, \ldots, x_n/\mathbf{ev}(t_n)\sigma, y_1/z_1\sigma, \ldots, y_m/z_m\sigma] \\ & = & \tilde{t}[x_1/\mathbf{ev}(t_1), \ldots, x_n/\mathbf{ev}(t_n), y_1/z_1, \ldots, y_m/z_m]\sigma \\ & = & \mathbf{ev}(t)\sigma \end{array}$$

□

Next, we want to show a version of lemma 18 of [GSSKT06], where renaming and the ***TyCase***-expansion have been introduced. Since there, the notions of *necessary reductions* and substitutions that are *evaluated enough* are used, their definitions will be introduced first.

The idea behind necessary reductions is, that some context forces a term to be evaluated to a certain constructor depth. This can already be observed in the definition of the evaluation position, which moves from the top further inwards if a pattern requires a constructor. Such reductions shall be called necessary.

Definition 6.19 (Necessary Reduction, [GSSKT06])**.** *We say that $t \to_{\mathsf{H}}^* q$ is a* necessary reduction*, iff either $t = q$, or both $q = (c\ q_1 \dots q_n)$ for a constructor c of arity n and*

- *$t = (c\ t_1 \dots t_n)$ and all reductions $t_i \to_{\mathsf{H}}^* q_i$ are necessary, or*
- *$t = (f\ t_1 \dots t_m) \to_{\mathsf{H}}^+ (c\ s_1 \dots s_n)$ and all reductions $s_i \to_{\mathsf{H}}^* q_i$ are necessary.*

It should be noted that if $t \to_{\mathsf{H}}^* q$ is a necessary reduction and $t \neq q$, then t has a non-functional type, because it can be reduced to a term $(c\ s_1 \dots s_n)$ for a constructor c of arity n.

Since we do not want substitutions to have an effect on the evaluation position, the notion of evaluated enough substitutions is used. A substitution that is evaluated enough for a reduction must provide a constructor when it is needed. Then, the evaluation position will not be changed.

In order to identify the constructors in a term only, the function drop is used that replaces all non-functional terms by a variable, except for those that already start with a constructor.

Definition 6.20 (drop, undrop, [GSSKT06])**.** *The function* drop *is a mapping from Haskell terms to new Haskell terms. It is defined as:*

- $\mathrm{drop}(t) = t$*, if t has a functional type,*
- $\mathrm{drop}(c\ t_1 \dots t_n) = c\ \mathrm{drop}(t_1) \dots \mathrm{drop}(t_n)$*, if c is a constructor of arity n, and*
- *$drop(t) = x_t$ for a fresh variable x_t, otherwise.*

The function undrop *is defined as the inverse function of* drop*, i.e.,* undrop *is defined as the substitution replacing every x_t by t.*

For a substitution σ, we define σ_{drop} as the substitution, for which $\sigma_{\mathrm{drop}}(x) = \mathrm{drop}(\sigma(x))$ for all variables x. Please note, that here we allow an infinite domain of σ_{drop}.

Using these terms, where constructors and functional terms have been kept, we can now define the requirements imposed onto a substitution that allows for evaluation without having to evaluate a subterm in the domain of the substitution.

Definition 6.21 (Evaluated Enough, [GSSKT06])**.** *For a (possibly infinite) reduction $t\sigma \to_{\mathsf{H}} t_1 \to_{\mathsf{H}} t_2 \to_{\mathsf{H}} \dots$, we say that a substitution σ is* evaluated enough*, 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.*

Finally, an altered version of Lemma 18 from [GSSKT06] can be proven, where renaming and the ***TyCase***-expansion have been included.

Lemma 6.22 (Properties of **ev**, **con**, and **rl** when renaming is used)**.** *Let TG be a Termination Graph and let t be a node in TG, where $t \notin \mathsf{PU}_{TG}$. Let $t\sigma \to_{\mathsf{H}}^* q$ be a necessary reduction where σ is evaluated enough. Then it holds that*

(a) $\mathrm{bR}_{TG}(\mathbf{ev}(t))\sigma_{\mathrm{ren}_{TG}} \to^*_{\bigcup_{s\in\mathbf{con}(t)} \mathrm{ren}_{TG}(\mathbf{rl}(s))} \mathrm{bR}_{TG}(q')$ *for some term q' with $q \Rightarrow_{\mathsf{H}}^* q'$*

*(b) If t is a **Case**-, an **Eval**-, or a **TyCase**-node and if $t\sigma \neq q$, then there is a rule path from t to some term $\hat{t}$ which is labelled with $\sigma_1, \ldots, \sigma_m$ such that*

- $\sigma = \sigma_1 \ldots \sigma_m \tau$,
- $\sigma_{\mathrm{ren}_{TG}} = \sigma_1 \ldots \sigma_m \tau_{\mathrm{ren}_{TG}}$, *and*
- $\mathrm{bR}_{TG}(\mathbf{ev}(\hat{t}))\tau_{\mathrm{ren}_{TG}} \to^*_{\bigcup_{s\in\mathbf{con}(\hat{t})} \mathrm{ren}_{TG}(\mathbf{rl}(s))} \mathrm{bR}_{TG}(q')$

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

Proof. The lemma is proven by induction. As induction relation the lexicographic combination of the reduction length of $t\sigma \to_{\mathsf{H}}^* q$ and the edge relation in TG after removing all outgoing edges of ***Eval***-nodes is used. This relation is well-founded, since all paths starting in a node that has an incoming instantiation edge either traverse an ***Eval***-node or a leaf with an error term after traversing only ***Case***- or ***TyCase***-nodes. Therefore, the resulting relation is acyclic.

If $t\sigma = q$, then we can choose $q' = \mathbf{ev}(t)\sigma$ and obtain from lemma 6.18 that $q = t\sigma \Rightarrow_{\mathsf{H}}^* \mathbf{ev}(t)\sigma = q'$. Thus, by definition 6.6, the following holds:

$$\mathrm{bR}_{TG}(\mathbf{ev}(t))\sigma_{\mathrm{ren}_{TG}} = \mathrm{bR}_{TG}(\mathbf{ev}(t)\sigma) \to^0_{\bigcup_{s\in\mathbf{con}(t)} \mathrm{ren}_{TG}(\mathbf{rl}(s))} \mathrm{bR}_{TG}(q')$$

This shows (a) for the case where $t\sigma = q$. For (b) nothing has to be shown, since it is only of relevance when $t\sigma \neq q$.

In the remainder it will be assumed that $t\sigma \neq q$. As the reduction $t\sigma \to_{\mathsf{H}}^* q$ is necessary, it holds that the head of q is a constructor.

Case Analysis is performed according to the expansion rule applied to generate t's children. It should be noted that for every child $\tilde{t}$ of t, $\tilde{t} \notin \mathsf{PU}_{TG}$. If this was not the case, then also $t \in \mathsf{PU}_{TG}$ would hold, which was ruled out by assumption.

Leaf

If t is a leaf, then t is either an error term, a constructor, or a variable x. Neither an error term nor a constructor can be reduced with $\to_{\mathsf{H}}$, therefore it holds that $t\sigma = q$ which contradicts the assumption.

In the case of a variable x it still holds that $t\sigma = \sigma(x) = q$. Because σ is evaluated enough, $\mathrm{drop}(t\sigma) = \mathrm{drop}(x\sigma) = x\sigma_{\mathrm{drop}} = t\sigma_{\mathrm{drop}} \to_{\mathsf{H}}^* q'$ for some term q' with $\mathrm{undrop}(q') = q$. In order to show that $t\sigma = q$, a more general claim is proven: if for arbitrary terms s, p, and p', $s \to_{\mathsf{H}}^* p$ is a necessary reduction and $\mathrm{drop}(s) \to_{\mathsf{H}}^* p'$ with $\mathrm{undrop}(p') = p$, then $s = p$. From this result $t\sigma = q$ follows by setting $s = t\sigma$, $p = q$, and $p' = q'$. We show the claim by structural induction on s:

- If s has functional type, then $s = p$ follows from 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 for all $1 \leq i \leq n$. From the definition of drop follows $\mathrm{drop}(s) = (c\ \mathrm{drop}(s_1) \dots \mathrm{drop}(s_n)) \to_{\mathsf{H}}^* (c\ p'_1 \dots p'_n) = p'$, and the definition of undrop gives us $p = (c\ p_1 \dots p_n) = \mathrm{undrop}(p') = (c\ \mathrm{undrop}(p'_1) \dots \mathrm{undrop}(p'_n))$ for some terms p'_i. Therefore $\mathrm{drop}(s_i) \to_{\mathsf{H}}^* p'_i$ and $\mathrm{undrop}(p'_i) = p_i$ hold, making the induction hypothesis applicable for every pair s_i, p_i. This yields $s_i = p_i$ and hence $s = p$.
- In all other cases $\mathrm{drop}(s) = x_s$. Since $\mathrm{drop}(s) \to_{\mathsf{H}}^* p'$, it must hold that $p' = x_s$. Therefore $p = \mathrm{undrop}(p') = \mathrm{undrop}(x_s) = s$.

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 $t\sigma$ starts with this evaluation step, the reduction looks as follows: $t\sigma \to_{\mathsf{H}} \tilde{t}\sigma \to_{\mathsf{H}}^* q$. The reduction $\tilde{t}\sigma \to_{\mathsf{H}}^* q$ is shorter than the reduction $t\sigma \to_{\mathsf{H}}^* q$. Because of this reduction being necessary, $\tilde{t}\sigma \to_{\mathsf{H}}^* q$ is necessary, too. Furthermore, since σ is evaluated enough, $t\sigma_{\mathrm{drop}} \to_{\mathsf{H}} \tilde{t}\sigma_{\mathrm{drop}}$, and hence σ is evaluated enough in $\tilde{t}\sigma \to_{\mathsf{H}}^* q$ as well. Thus the induction hypotheses for (a) and (b) can be used.

From the induction hypothesis for (a) follows

$$\mathrm{bR}_{TG}(\mathbf{ev}(\tilde{t}))\sigma_{\mathrm{ren}_{TG}} \to^*_{\bigcup_{s \in \mathbf{con}(\tilde{t})} \mathrm{ren}_{TG}(\mathbf{rl}(s))} \mathrm{bR}_{TG}(q')$$

for some term q' with $q \Rightarrow_{\mathsf{H}}^* q'$. Since $\mathbf{ev}(t) = \mathbf{ev}(\tilde{t})$ and $\mathbf{con}(t) = \mathbf{con}(\tilde{t})$, (a) has been shown.

To show (b) we distinguish two cases: In case $\tilde{t}$ is neither a ***Case***-, nor a ***TyCase***-, nor an ***Eval***-node, then the path from t to $\tilde{t}$ is a rule path and (b) follows directly from the induction hypothesis for (a).

In the other case, i.e., when $\tilde{t}$ is a ***Case***-, a ***TyCase***-, or an ***Eval***-node, then the head of $\tilde{t}$ must be defined. As the head of q is a constructor, we have $\tilde{t}\sigma \neq q$. From the induction hypothesis for (b) it is known that a rule path from $\tilde{t}$ to some $\hat{t}$ exists, which satisfies the conditions in (b). This rule path remains a rule path when prepending the edge from t to $\tilde{t}$ to it, thus (b) is proven.

Case
If t is a ***Case***-node, then $\mathbf{ev}(t) = t$, $\mathbf{con}(t) = \{t\}$, and $t|_{\mathbf{e}(t)} = x$ for some variable x. Because of σ being evaluated enough, $\sigma(x)$ 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 the node $t\delta$ where $\delta = [x/(c\ x_1 \dots x_n)]$ with fresh variables $x_1, \dots, x_n$. Let σ' be like σ, but on $x_1, \dots, x_n$ we define $\sigma'(x_i) = t_i$ for $1 \leq i \leq n$. Then $\sigma = \delta\sigma'$ and thus $t\sigma = t\delta\sigma' \to_{\mathsf{H}}^* q$ is a necessary reduction. We obtain $t\delta\sigma'_{\mathrm{drop}} = t\delta\sigma_{\mathrm{drop}}[x_1/\mathrm{drop}(t_1), \dots, x_n/drop(t_n)] = t\sigma_{\mathrm{drop}}$. From this and the precondition that σ was evaluated enough in the reduction $t\sigma \to_{\mathsf{H}}^* q$, it follows that σ' is evaluated enough in the reduction $t\delta\sigma' \to_{\mathsf{H}}^* q$.

If the node $t\delta$ is neither a ***Case***-, a ***TyCase***-, nor an ***Eval***-node, the path from t to $t\delta$ is a rule path. Then (b) directly follows from the induction hypothesis for (a).

Otherwise $t\delta$ is a ***Case***-, a ***TyCase***-, or an ***Eval***-node. Because the node $t\delta$ is a child of t and the reduction $t\delta\sigma' \to_{\mathsf{H}}^* q$ has the same length as $t\sigma \to_{\mathsf{H}}^* q$, the

induction hypothesis for (b) implies the existence of a rule path from $t\delta$ to some $\hat{t}$ labelled with substitutions $\sigma_1, \ldots, \sigma_m$ such that $\sigma' = \sigma_1 \ldots \sigma_m\tau$, $\sigma'_{\mathrm{ren}_{TG}} = \sigma_1 \ldots \sigma_m\tau_{\mathrm{ren}_{TG}}$, and $\mathrm{bR}_{TG}(\mathbf{ev}(\hat{t}))\tau_{\mathrm{ren}_{TG}} \to^*_{\bigcup_{s\in\mathbf{con}(\hat{t})} \mathrm{ren}_{TG}(\mathbf{rl}(s))} \mathrm{bR}_{TG}(q')$ for some substitution τ and some term q' with $q \Rightarrow^*_{\mathsf{H}} q'$. This rule path remains a rule path when prepended with the edge from t to $t\delta$. This new rule path is labelled with $\delta, \sigma_1, \ldots, \sigma_m$ such that $\sigma = \delta\sigma' = \delta\sigma_1 \ldots \sigma_m\tau$ and $\sigma_{\mathrm{ren}_{TG}} = \delta_{\mathrm{ren}_{TG}}\sigma_1 \ldots \sigma_m\tau_{\mathrm{ren}_{TG}} = \delta\sigma_1 \ldots \sigma_m\tau_{\mathrm{ren}_{TG}}$ which proves (b).

Now the rule $\mathrm{bR}_{TG}(t)\delta\sigma_1 \ldots \sigma_m \to \mathrm{bR}_{TG}(\mathbf{ev}(\hat{t}))$ is contained in $\mathrm{ren}_{TG}(\mathbf{rl}(t))$ and it holds that $\mathbf{rl}(s) \subseteq \mathbf{rl}(t)$ for all $s \in \mathbf{con}(\hat{t})$. This enables us to prove (a):

$$\begin{array}{lll} \mathrm{bR}_{TG}(\mathbf{ev}(t))\sigma_{\mathrm{ren}_{TG}} & = & \mathrm{bR}_{TG}(t)\sigma_{\mathrm{ren}_{TG}} \\ & = & \mathrm{bR}_{TG}(t)\delta\sigma_1 \ldots \sigma_m\tau_{\mathrm{ren}_{TG}} \\ & \to_{\mathrm{ren}_{TG}(\mathbf{rl}(t))} & \mathrm{bR}_{TG}(\mathbf{ev}(\hat{t}))\tau_{\mathrm{ren}_{TG}} \\ & \to^*_{\mathrm{ren}_{TG}(\mathbf{rl}(t))} & \mathrm{bR}_{TG}(q') \end{array}$$

TyCase

If $\underline{cs}\Rightarrow t$ is a ***TyCase***-node, then $\mathbf{ev}(t) = t$ and $\mathbf{con}(t) = \{t\}$. Moreover, $t|_{\mathbf{e}(t)}$ is a term $(g\ s_1 \ldots s_k)$ of type $\rho'[a]$ for a defined function symbol g and terms $s_1 \ldots s_k$. Since $t\sigma \neq q$, there must be an instance $(C\ \rho[b_1/\rho_1, \ldots, b_n/\rho_n])$ of a class $(C\ a)$ whose definition of g is used in the necessary reduction $t\sigma \to^*_{\mathsf{H}} q$.

One of the children of t is $\mathrm{reduce}(\underline{cs}[a/\rho])\Rightarrow t$, where $\rho = (T\ b_1 \ldots b_n)$ for some type constructor T of arity n. Let $\delta = [a/\rho]$ and let σ' be like σ, but on the fresh variables $b_1, \ldots, b_n$ we define $\sigma'(b_i) = \rho_i$ for $1 \leq i \leq n$, yielding $\sigma = \delta\sigma'$. Since $t\sigma \to^*_{\mathsf{H}} q$ is a necessary reduction, the reduction $t\delta\sigma' \to^*_{\mathsf{H}} q$ is also necessary. Also σ' is evaluated enough in the reduction $t\delta\sigma' \to^*_{\mathsf{H}} q$, as σ is evaluated enough in the reduction of $t\sigma \to^*_{\mathsf{H}} q$: For the type variable a, it holds that $\sigma(a) = \rho[b_1/\rho_1, \ldots, b_n/\rho_n]$, thus $drop(\sigma(a)) = T\ drop(\rho_1) \ldots drop(\rho_n) = T\ drop(\sigma'(b_1)) \ldots drop(\sigma'(b_n)) = drop(\sigma'(\rho))$.

If $\mathrm{reduce}(\underline{cs}[a/\rho])\Rightarrow t$ is neither a ***Case***-, nor a ***TyCase***-, nor an ***Eval***-node, the induction hypothesis for (a) directly proves (b). This holds because the path from $\underline{cs}\Rightarrow t$ to $\mathrm{reduce}(\underline{cs}[a/\rho])\Rightarrow t$ is a rule path, the reduction has the same length, and $\mathrm{reduce}(\underline{cs}[a/\rho])\Rightarrow t$ is a child of $\underline{cs}\Rightarrow t$.

Otherwise, $\mathrm{reduce}(\underline{cs}[a/\rho])\Rightarrow t$ is a ***Case***-, a ***TyCase***-, or an ***Eval***-node. Since $\mathrm{reduce}(\underline{cs}[a/\rho])\Rightarrow t$ is a child of $(t, \underline{cs})$ and since the reduction has the same length, the induction hypothesis for (b) implies the existence of a rule path from $\mathrm{reduce}(\underline{cs}[a/\rho])\Rightarrow t$ to some term $\hat{t}$ which is labelled with substitutions $\sigma_1, \ldots, \sigma_m$, such that $\sigma' = \sigma_1 \ldots \sigma_m\tau$, $\sigma'_{\mathrm{ren}_{TG}} = \sigma_1 \ldots \sigma_m\tau_{\mathrm{ren}_{TG}}$ and we have a reduction $\mathrm{bR}_{TG}(\mathbf{ev}(\hat{t}))\tau_{\mathrm{ren}_{TG}} \to^*_{\bigcup_{s\in\mathbf{con}(\hat{t})} \mathrm{ren}_{TG}(\mathbf{rl}(s))} \mathrm{bR}_{TG}(q')$ for some term q' with $q \Rightarrow^*_{\mathsf{H}} q'$. This rule path can be extended by prepending $\underline{cs}\Rightarrow t$, proving (b).

Thus, the rule $\mathrm{bR}_{TG}(t)\delta\sigma_1 \ldots \sigma_m \to \mathrm{bR}_{TG}(\mathbf{ev}(\hat{t}))$ is contained in the rules $\mathrm{ren}_{TG}(\mathbf{rl}(t))$. Furthermore, for all $s \in \mathbf{con}(\hat{t})$ it holds that $\mathrm{ren}_{TG}(\mathbf{rl}(s)) \subseteq \mathrm{ren}_{TG}(\mathbf{rl}(t))$. Hence, (a) can now be proven:

$$\begin{array}{lll} \mathrm{bR}_{TG}(\mathbf{ev}(t))\sigma_{\mathrm{ren}_{TG}} & = & \mathrm{bR}_{TG}(t)\sigma_{\mathrm{ren}_{TG}} \\ & = & \mathrm{bR}_{TG}(t)\delta\sigma'_{\mathrm{ren}_{TG}} \\ & = & \mathrm{bR}_{TG}(t)\delta\sigma_1 \ldots \sigma_m\tau_{\mathrm{ren}_{TG}} \\ & \to_{\mathrm{ren}_{TG}(\mathbf{rl}(t))} & \mathrm{bR}_{TG}(\mathbf{ev}(\hat{t}))\tau_{\mathrm{ren}_{TG}} \\ & \to^*_{\mathrm{ren}_{TG}(\mathbf{rl}(t))} & \mathrm{bR}_{TG}(q') \end{array}$$

VarExp
If t is a ***VarExp***-node then $t\sigma$ has a functional type. Therefore $t\sigma = q$ which contradicts the assumption.

ParSplit, where head(t) is a constructor
This implies $t = (c\ t_1 \dots t_n)$ and $q = (c\ q_1 \dots q_n)$ with $t_i\sigma \to_{\mathsf{H}}^* q_i$ for all i. Because of $t\sigma \neq q$ and the fact that the original reduction is necessary, the arity of c must be n. Therefore, all reductions $t_i\sigma \to_{\mathsf{H}}^* q_i$ are necessary. Moreover, $t\sigma_{\text{drop}} = (c\ t_1\sigma_{\text{drop}} \dots t_n\sigma_{\text{drop}}) \to_{\mathsf{H}}^* (c\ p_1 \dots p_n)$ where $\text{undrop}(p_i) = q_i$ for all i. Thus, σ is evaluated enough in every reduction $t_i\sigma \to_{\mathsf{H}}^* q_i$ as $t_i\sigma_{\text{drop}} \to_{\mathsf{H}}^* p_i$.

It holds that $\mathbf{ev}(t) = (c\ \mathbf{ev}(t_1) \dots \mathbf{ev}(t_n))$ and $\mathbf{con}(t) = \mathbf{con}(t_1) \cup \dots \cup \mathbf{con}(t_n)$. The reductions $t_i\sigma \to_{\mathsf{H}}^* q_i$ have at most the same length as the reduction $t\sigma \to_{\mathsf{H}}^* q$ and since the t_i are children of t, the induction hypothesis implies

$$\text{bR}_{TG}(\mathbf{ev}(t_i))\sigma_{\text{ren}_{TG}} \to^*_{\bigcup_{s\in\mathbf{con}(t_i)} \text{ren}_{TG}(\mathbf{rl}(s))} \text{bR}_{TG}(q_i')$$

for terms q_i' with $q_i \Rightarrow_{\mathsf{H}}^* q_i'$ for all i. Defining $q' = (c\ q_1' \dots q_n')$ enables us to show (a):

$$\begin{array}{lcl}
\text{bR}_{TG}(\mathbf{ev}(t))\sigma_{\text{ren}_{TG}} & = & \text{bR}_{TG}(c\ \mathbf{ev}(t_1) \dots \mathbf{ev}(t_n))\sigma_{\text{ren}_{TG}} \\
& = & (c\ \text{bR}_{TG}(\mathbf{ev}(t_1)) \dots \text{bR}_{TG}(\mathbf{ev}(t_n)))\sigma_{\text{ren}_{TG}} \\
& = & c\ \text{bR}_{TG}(\mathbf{ev}(t_1))\sigma_{\text{ren}_{TG}} \dots \text{bR}_{TG}(\mathbf{ev}(t_n))\sigma_{\text{ren}_{TG}} \\
& \to^*_{\bigcup_{s\in\mathbf{con}(t)} \text{ren}_{TG}(\mathbf{rl}(s))} & c\ \text{bR}_{TG}(q_1') \dots \text{bR}_{TG}(q_n') \\
& = & \text{bR}_{TG}(q')
\end{array}$$

It holds that $q \Rightarrow_{\mathsf{H}}^* q'$, since $q = (c\ q_1 \dots q_n)$ and $q_i \Rightarrow_{\mathsf{H}}^* q_i'$ for all i.

ParSplit, where head(t) is a variable
In this case, it holds that $t \in \mathsf{PU}_{TG}$, which contradicts the assumption.

Ins
Since t is an ***Ins***-node, $t = \tilde{t}[x_1/t_1, \dots, x_n/t_n, b_1/\rho_1, \dots b_v/\rho_v]$. If $\tilde{t} = (x\ y)$ then $\mathbf{ev}(t)$ is a fresh variable and therefore the above case applies. Otherwise, $\tilde{t}$ is an ***Eval***- or ***Case***-node. Without loss of generality it is assumed that $x_1, \dots, x_n, b_1, \dots, b_v$ are fresh variables that neither occur in t nor in the domain of σ. Then $t\sigma = \tilde{t}\sigma[x_1/t_1\sigma, \dots, x_n/t_n\sigma, b_1/\rho_1\sigma, \dots, b_v/\rho_v\sigma] \to_{\mathsf{H}}^* q$. It should be noted that for every subterm t_i, $t_i \notin \mathsf{PU}_{TG}$ holds. This is, because if some $t_i \in \mathsf{PU}_{TG}$, then also $t \in \mathsf{PU}_{TG}$ would hold, which is ruled out by assumption.

Instead of the above reduction, we start with first evaluating the subterms $t_i\sigma$ "as much as ever needed in the reduction $t\sigma \to_{\mathsf{H}}^* q$". This way, each $t_i\sigma$ is evaluated to a term s_i. For a precise definition of s_i consider the reduction $\tilde{t}\sigma[x_1/t_1\sigma, \dots, x_n/t_n\sigma, b_1/\rho_1\sigma, \dots, b_v/\rho_v\sigma] \to_{\mathsf{H}}^* q$. Initially, we choose $s_i = t_i\sigma$. If the above reduction is still possible for the term, where the subterms have been reduced to the constructors occuring in them, i.e., for the term $\tilde{t}\sigma[x_1/\text{drop}(s_1), \dots, x_n/\text{drop}(s_n), b_1/\rho_1\sigma, \dots, b_v/\rho_v\sigma]$ (yielding a term p with $\text{undrop}(p) = q$), then we have found our final term s_i. In this case the substitution $\sigma[x_1/s_1, \dots, x_n/s_n, b_1/\rho_1\sigma, \dots, b_v/\rho_v\sigma]$ is evaluated enough in this reduction. Otherwise, there must be some point in the evaluation of the term $\tilde{t}\sigma[x_1/\text{drop}(s_1), \dots, x_n/\text{drop}(s_n), b_1/\rho_1\sigma, \dots, b_v/\rho_v\sigma]$ where it gets stuck,

because a variable x_r has to be evaluated that was introduced by applying drop on some s_i. Then s_i is replaced by evaluating it further; more precisely a constructor c is required for the evaluation to continue. As the original reduction was able to continue, r must be reducible to a term of the form $(c\ l_1 \dots l_k)$. Therefore, by replacing the subterm r of s_i by $(c\ l_1 \dots l_k)$, the evaluation can continue beyond the point where it 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 have $\mathrm{drop}(s_i) = C[\mathrm{drop}(c\ l_1 \dots l_k)] = C[c\ \mathrm{drop}(l_1) \dots \mathrm{drop}(l_k)]$, i.e., the required constructor c now is present. In this way, the s_i are redefined until all necessary constructors are present. By construction, the reductions $t_i\sigma \to_{\mathsf{H}}^* s_i$ are necessary. Furthermore, as σ was evaluated enough in the reduction $t\sigma \to_{\mathsf{H}}^* q$ and as this reduction includes all reductions $t_i\sigma \to_{\mathsf{H}}^* s_i$, the substitution σ is evaluated enough in the reductions $t_i\sigma \to_{\mathsf{H}}^* s_i$, too.

The length of a reduction $t_i\sigma \to_{\mathsf{H}}^* s_i$ is at most the same as the length of the reduction $t\sigma \to_{\mathsf{H}}^* q$, for all $1 \leq i \leq n$. Thus the induction hypothesis can be applied, since all t_i are children of t. This way also the necessary reduction $\tilde{t}\sigma[x_1/s_1, \dots, x_n/s_n, b_1/\rho_1\sigma, \dots, b_v/\rho_v\sigma] \to_{\mathsf{H}}^* \tilde{q}$ for a term $\tilde{q}$ with $q \Rightarrow_{\mathsf{H}}^* \tilde{q}$ is obtained, which has at most the same length as the original reduction $t\sigma \to_{\mathsf{H}}^* q$. The substitution $\sigma[x_1/s_1, \dots, x_n/s_n, b_1/\rho_1\sigma, \dots, b_v/\rho_v\sigma]$ is evaluated enough by construction.

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

$$(*) \quad \mathrm{bR}_{TG}(\mathbf{ev}(t_i))\sigma_{\mathrm{ren}_{TG}} \to^*_{\bigcup_{s \in \mathbf{con}(t_i)} \mathrm{ren}_{TG}(\mathbf{rl}(s))} \mathrm{bR}_{TG}(s_i')$$

for some terms s_i' with $s_i \Rightarrow_{\mathsf{H}}^* s_i'$. Moreover, when reducing the terms s_i to the terms s_i', the above properties are not destroyed, i.e., for the substitution $\sigma' = \sigma[x_1/s_1', \dots, x_n/s_n', b_1/\rho_1\sigma, \dots, b_v/\rho_v\sigma]$ we still have $\tilde{t}\sigma' \to_{\mathsf{H}}^* q''$ for a term q'' with $q \Rightarrow_{\mathsf{H}}^* q''$. 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.

The reduction $\tilde{t}\sigma' \to_{\mathsf{H}}^* q''$ has a length greater than zero, since $\mathrm{head}(q) = \mathrm{head}(q'')$ is a constructor and $\mathrm{head}(\tilde{t})$ is defined. As $\tilde{t}$ is an ***Eval***- or a ***Case***-node, the induction hypothesis (b) can be used for the child $\tilde{t}$ of t. We obtain a node $\hat{t}$ and a rule path from $\tilde{t}$ to $\hat{t}$ labelled with substitutions $\sigma_1', \dots, \sigma_m'$ where $\sigma' = \sigma_1' \dots \sigma_m'\tau$, $\sigma'_{\mathrm{ren}_{TG}} = \sigma_1' \dots \sigma_m'\tau_{\mathrm{ren}_{TG}}$, and

$$(**) \quad \mathrm{bR}_{TG}(\mathbf{ev}(\hat{t}))\tau_{\mathrm{ren}_{TG}} \to^*_{\bigcup_{s \in \mathbf{con}(\hat{t})} \mathrm{ren}_{TG}(\mathbf{rl}(s))} \mathrm{bR}_{TG}(q')$$

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

Because of the rule path, the rule $\mathrm{bR}_{TG}(\tilde{t})\sigma_1' \dots \sigma_m' \to \mathrm{bR}_{TG}(\mathbf{ev}(\hat{t}))$ is included in $\mathbf{rl}(\tilde{t})$. As $\mathbf{rl}(s) \subseteq \mathbf{rl}(\tilde{t})$ for all $s \in \mathbf{con}(\tilde{t})$, the statement (a) can now

be proven:

$$
\begin{array}{lll}
& \mathrm{bR}_{TG}(\mathbf{ev}(t))\sigma_{\mathrm{ren}_{TG}} & \\
= & \mathrm{bR}_{TG}(\tilde{t}[x_1/\mathbf{ev}(t_1),\ldots,x_n/\mathbf{ev}(t_n)])\sigma_{\mathrm{ren}_{TG}} & \\
= & f_{\tilde{t}}\ (\mathrm{bR}_{TG}(\mathbf{ev}(t_1))\sigma_{\mathrm{ren}_{TG}})\ldots(\mathrm{bR}_{TG}(\mathbf{ev}(t_n))\sigma_{\mathrm{ren}_{TG}}) & \\
 & \quad \rho_1\sigma_{\mathrm{ren}_{TG}}\cdots\rho_v\sigma_{\mathrm{ren}_{TG}} & \\
\to^*_{\bigcup_{s\in\mathbf{con}(t_1)\cup\ldots\cup\mathbf{con}(t_n)}\mathrm{ren}_{TG}(\mathbf{rl}(s))} & f_{\tilde{t}}\ \mathrm{bR}_{TG}(s'_1)\ldots\mathrm{bR}_{TG}(s'_n) & \text{by (*)} \\
 & \quad \rho_1\sigma_{\mathrm{ren}_{TG}}\cdots\rho_v\sigma_{\mathrm{ren}_{TG}} & \\
= & (f_{\tilde{t}}\ x_1\ldots x_n\ b_1\ldots b_v)\sigma'_{\mathrm{ren}_{TG}} & \\
= & \mathrm{bR}_{TG}(\tilde{t})\sigma'_{\mathrm{ren}_{TG}} & \\
= & \mathrm{bR}_{TG}(\tilde{t})\sigma'_1\ldots\sigma'_m\tau_{\mathrm{ren}_{TG}} & \\
\to_{\mathrm{ren}_{TG}(\mathbf{rl}(\tilde{t}))} & \mathrm{bR}_{TG}(\mathbf{ev}(\hat{t}))\tau_{\mathrm{ren}_{TG}} & \\
\to^*_{\mathrm{ren}_{TG}(\mathbf{rl}(\tilde{t}))} & \mathrm{bR}_{TG}(q') & \text{by (**)}
\end{array}
$$

As desired, we have $q \Rightarrow^*_{\mathsf{H}} q'' \Rightarrow^*_{\mathsf{H}} q'$.

□

Since H-termination allows for the addition of H-terminating terms to terms with functional types, showing the absence of infinite $\to_{\mathsf{H}}$ evaluations does not show H-termination. Thus, we first introduce a different relation which is allowed to append arbitrary terms and can strip off constructors.

Definition 6.23 ($\hookrightarrow_{\mathsf{H}}$, [GSSKT06])**.** *We define $s \hookrightarrow_{\mathsf{H}} t$, iff*

(a) $s \to_{\mathsf{H}} t$,

(b) $s = (f\ s_1\ldots s_m)$ for a defined function symbol f of arity n, where $m < n$ and $t = (f\ s_1\ldots s_m\ t')$ for an H-terminating term t', or

(c) $s = (c\ s_1\ldots s_n)$ for a constructor c and $t = s_i$ for some $1 \le i \le n$.

Corollary 6.24 (Properties of $\hookrightarrow_{\mathsf{H}}$, [GSSKT06])**.** *For the relation $\hookrightarrow_{\mathsf{H}}$, the following two properties hold:*

(1) A ground term is not H-terminating iff it starts an infinite $\hookrightarrow_{\mathsf{H}}$-reduction.

(2) If a non-ground term t is H-terminating, then t is terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$.

The properties of the above corollary should be easy to see from definition 2.7. However, it should be noted that the other direction of property (2) does not hold, i.e., a non-ground term can be terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$, but not H-terminating. This is shown in the following example:

Example 6.25 (Termination w.r.t. $\hookrightarrow_{\mathsf{H}}$ does not imply H-termination)**.** *Consider the following Haskell program, which was already presented in [GSSKT06], and the start term* `nonterm False x`*.*

```
nonterm True  x = True
nonterm False x = nonterm (x True) x
```

It can be seen that the start term is terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$, since after the evaluation to `nonterm (x True) x`*, no further reduction is possible. However, if one instantiates the start term with the substitution $\sigma = [\mathtt{x}/\mathtt{not}]$, then an infinite evaluation exists:*

$$\begin{array}{lcl} \texttt{(nonterm False x)}\sigma & = & \texttt{nonterm False not} \\ & \to_{\mathsf{H}} & \texttt{nonterm (not True) not} \\ & \to_{\mathsf{H}} & \texttt{nonterm False not} \\ & \to_{\mathsf{H}} & \texttt{...} \end{array}$$

After having introduced the relation $\hookrightarrow_{\mathsf{H}}$, we can now prove a version of lemma 20 from [GSSKT06] where renaming and type classes are introduced.

Lemma 6.26 (Properties of **dp** when renaming is used)**.** *Let TG be a Termination Graph and let s be a node in TG. 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 $t = \tilde{t}[x_1/t_1, \ldots, x_n/t_n, b_1/\rho_1, \ldots, b_v/\rho_v]$ labelled with $\sigma_1, \ldots, \sigma_m$ and an instantiation edge from t to a node $\tilde{t}$ such that*

- *$\sigma = \sigma_1 \ldots \sigma_m\tau$ for some substitution τ, such that $t\tau$ is not $\hookrightarrow_{\mathsf{H}}$-terminating*
- *$\sigma_{\mathrm{ren}_{TG}} = \sigma_1 \ldots \sigma_m\tau_{\mathrm{ren}_{TG}}$*
- *$\mathrm{bR}_{TG}(\mathbf{ev}(t))\tau_{\mathrm{ren}_{TG}} \xrightarrow{>\varepsilon}{}^{*}_{\mathcal{R}_{\mathrm{ren}_{TG}}} \mathrm{bR}_{TG}(\tilde{t})\mu_{\mathrm{ren}_{TG}}$ for the set of rules $\mathcal{R}_{\mathrm{ren}_{TG}} = \bigcup_{s \in \mathbf{con}(t_1) \cup \ldots \cup \mathbf{con}(t_n)} \mathrm{ren}_{TG}(\mathbf{rl}(s))$ such that $\tilde{t}\mu$ starts an infinite $\hookrightarrow_{\mathsf{H}}$-reduction, 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 proven by induction on the edge relation in the graph obtained from TG by removing all instantiation edges. This means that one can assume the lemma to hold for all children of a term s, except for those that are reachable only via instantiation edges. Case analysis is performed according to the expansion rule applied to s.

Leaf
If s is a leaf, then either s is an error term, a constructor, or a variable x. In either of 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. Therefore $s\sigma$ cannot start an infinite $\hookrightarrow_{\mathsf{H}}$-reduction, which contradicts the assumption.

Eval
If s is an ***Eval***-node with child $\tilde{s}$, then $s \to_{\mathsf{H}} \tilde{s}$ and therefore the infinite $\hookrightarrow_{\mathsf{H}}$-reduction has to start with $s\sigma \hookrightarrow_{\mathsf{H}} \tilde{s}\sigma$. Therefore, $\tilde{s}\sigma$ is also not terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$ and σ is evaluated enough in this infinite $\hookrightarrow_{\mathsf{H}}$-reduction. Since $\tilde{s}$ is a child of s, the induction hypothesis implies the lemma.

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

One of the children of s is $s\delta$ with $\delta = [x/(c\ x_1 \ldots x_n)]$ for fresh variables $x_1, \ldots, x_n$. Let σ' be like σ, but let $\sigma'(x_i) = s_i$ for all i. Then $\sigma = \delta\sigma'$, which implies that $s\sigma = s\delta\sigma'$ also starts an infinite $\hookrightarrow_{\mathsf{H}}$-reduction. The same

argument as in the case of ***Case***-nodes in the proof of Lemma 6.22 shows that σ' is evaluated enough in the infinite $\hookrightarrow_{\mathsf{H}}$-reduction of $s\delta\sigma'$.

Because of $s\delta$ being a child of s and because $\sigma_{\mathrm{ren}_{TG}} = \delta_{\mathrm{ren}_{TG}}\sigma'_{\mathrm{ren}_{TG}} = \delta\sigma'_{\mathrm{ren}_{TG}}$, the lemma holds via the induction hypothesis.

TyCase

If $\underline{cs}\Rightarrow s$ is a ***TyCase***-node, then $s|_{\mathbf{e}(s)}$ is a term $(g\ s_1 \ldots s_m)$ of type $\rho'[a]$. Since we have an infinite $\hookrightarrow_{\mathsf{H}}$-reduction, an instance $(C\ \rho[b_1/\rho_1, \ldots, b_n/\rho_n])$ of a class $(C\ a)$ introduced by σ exists, which defines the rules of g.

One of the children of $\underline{cs}\Rightarrow s$ is $\mathrm{reduce}(\underline{cs}[a/\rho])\Rightarrow s$, where the type variables $b_1, \ldots, b_n$ are fresh. Let σ' be like σ, but let $\sigma'(b_i) = \rho_i$ for $1 \leq i \leq n$. Then $\sigma = \delta\sigma'$ for $\delta = [a/\rho]$, and therefore $s\sigma = s\delta\sigma'$ starts an infinite $\hookrightarrow_{\mathsf{H}}$-reduction. The same argument as in the proof of Lemma 6.22 shows that σ' is evaluated enough in the infinite $\hookrightarrow_{\mathsf{H}}$-reduction of $s\delta\sigma'$.

Since $\mathrm{reduce}(\underline{cs}[a/\rho])\Rightarrow s$ is a child of $\underline{cs}\Rightarrow s$, the lemma follows from 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'. Because an H-terminating term is also terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$, s' cannot start an infinite $\hookrightarrow_{\mathsf{H}}$-reduction. It is therefore safe to assume that s' is evaluated "as much as ever needed" in the infinite reduction of $s\sigma\ s'$, cf. the proof of lemma 6.22 in the case of ***Ins***-nodes. (one could also replace s' by its normal form w.r.t. $\rightarrow_{\mathsf{H}}$)

Let $(s\ x)$ be the child of s. By extending σ to have $\sigma(x) = s'$, $(s\ x)\sigma$ starts an infinite $\hookrightarrow_{\mathsf{H}}$-reduction. By construction, σ remains evaluated enough in the infinite reduction of $(s\ x)\sigma$ and instantiates all variables with terms that terminate w.r.t. $\hookrightarrow_{\mathsf{H}}$. Thus, the lemma follows from the induction hypothesis, since $(s\ x)$ is the child of s.

ParSplit

In this case it holds that $s = (c\ s_1 \ldots s_n)$ 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 either. Since s_i is a child of s, the lemma follows from the induction hypothesis.

Ins

Now $s = \tilde{s}[x_1/s_1, \ldots, x_n/s_n, b_1/\rho_1, \ldots, b_v/\rho_v]$ and the children of s are $s_1, \ldots, s_n$ and $\tilde{s}$.

First, the case is regarded where a s_i exists such that $s_i\sigma$ starts an infinite $\hookrightarrow_{\mathsf{H}}$-reduction. In this case the induction hypothesis may be applied and the lemma follows from it. It should be noted that if $\tilde{s} = (x\ y)$, then this case always applies. The reason is the same as in the ***ParSplit***-case where the head is a variable: If both $s_1\sigma$ and $s_2\sigma$ terminate w.r.t. $\hookrightarrow_{\mathsf{H}}$, then this is also the case for $(s_1\sigma\ s_2\sigma) = s\sigma$.

Thus, in the following it is assumed that all s_i terminate w.r.t. $\hookrightarrow_{\mathsf{H}}$ and $s \neq (x\ y)$. Without loss of generality, we assume that $m \leq n$ exists, such that for all $1 \leq i \leq m$, we have $s_i \notin \mathsf{PU}_{TG}$, and for $m < j \leq n$ we have $s_j \in \mathsf{PU}_{TG}$. Then, $\mathbf{ev}(s) = \tilde{s}[x_1/\mathbf{ev}(s_1), \ldots, x_m/\mathbf{ev}(s_m), x_{m+1}/y_{m+1}, \ldots, x_n/y_n]$, where $y_{m+1}, \ldots, y_n$ are fresh variables. Let $t = s$, $t_k = s_k$ for $1 \leq k \leq n$, $\tilde{t} = \tilde{s}$,

and $\tau = \sigma$. Without loss of generality it is assumed that $x_1, \ldots, x_n, b_1, \ldots, b_v$ are fresh variables not occuring in t or in the domain of $\tau = \sigma$. Then $s\sigma = t\tau = \tilde{t}\tau[x_1/t_1\tau, \ldots, x_n/t_n\tau, b_1/\rho_1\tau, \ldots, b_v/\rho_v\tau]$ starts an infinite $\hookrightarrow_{\mathsf{H}}$-reduction. Furthermore, we still have an infinite $\hookrightarrow_{\mathsf{H}}$-reduction for any term that results from $\tilde{t}\tau[x_1/t_1\tau, \ldots, x_n/t_n\tau, b_1/\rho_1\tau, \ldots, b_v/\rho_v\tau]$ by first reducting $t_i\tau$ "as much as ever needed in the infinite reduction", cf. the proof of Lemma 6.22 in the case of ***Ins***-nodes. This way, every $t_i\tau$ reduces to a term q_i.

By construction, the reduction $t_i\tau \to_{\mathsf{H}}^* q_i$ is necessary and τ is evaluated enough in this reduction. Hence we can apply Lemma 6.22 (a) in the case of $1 \leq i \leq m$, which gives us $\mathrm{bR}_{TG}(\mathbf{ev}(t_i))\tau_{\mathrm{ren}_{TG}} \to^*_{\bigcup_{s \in \mathbf{con}(t_i)} \mathrm{ren}_{TG}(\mathbf{rl}(s))} \mathrm{bR}_{TG}(q_i')$ for some term q_i' with $q_i \Rightarrow_{\mathsf{H}}^* q_i'$. For $m < i \leq n$, we define $q_i' = q_i$ and extend τ by $\tau(y_i) = q_i'$, because then $\mathrm{bR}_{TG}(\mathbf{ev}(s_i))\tau_{\mathrm{ren}_{TG}} = y_i\tau_{\mathrm{ren}_{TG}} = \mathrm{bR}_{TG}(q_i')$ for a term q_i' which satisfies $q_i \Rightarrow_{\mathsf{H}}^* q_i'$.

Let μ be like τ, but on the variables $x_1, \ldots, x_n$ we define $\mu(x_i) = q_i'$ and on the type variables $b_1, \ldots, b_v$ of $\tilde{t}$ we define $\mu(b_j) = \rho_j\tau$. Then:

$$
\begin{array}{ll}
& \mathrm{bR}_{TG}(\mathbf{ev}(t))\tau_{\mathrm{ren}_{TG}} \\
= & \mathrm{bR}_{TG}(\tilde{t}[x_1/\mathbf{ev}(t_1), \ldots, x_m/\mathbf{ev}(t_m), x_{m+1}/y_{m+1}, \ldots, x_n/y_n])\tau_{\mathrm{ren}_{TG}} \\
= & f_{\tilde{t}}\ \mathrm{bR}_{TG}(\mathbf{ev}(t_1))\tau_{\mathrm{ren}_{TG}} \ldots \mathrm{bR}_{TG}(\mathbf{ev}(t_m))\tau_{\mathrm{ren}_{TG}} \\
& \qquad\qquad y_{m+1}\tau_{\mathrm{ren}_{TG}}\ \cdots\ y_n\tau_{\mathrm{ren}_{TG}}\ \rho_1\tau_{\mathrm{ren}_{TG}}\ \cdots\ \rho_v\tau_{\mathrm{ren}_{TG}} \\
\xrightarrow{>\varepsilon}{}^{*}_{\bigcup_{s \in \mathbf{con}(t_1) \cup \ldots \cup \mathbf{con}(t_n)} \mathrm{ren}_{TG}(\mathbf{rl}(s))} & f_{\tilde{t}}\ \mathrm{bR}_{TG}(q_1') \ldots \mathrm{bR}_{TG}(q_n') \\
& \qquad\qquad\qquad\qquad \rho_1\tau_{\mathrm{ren}_{TG}}\ \cdots\ \rho_v\tau_{\mathrm{ren}_{TG}} \\
= & (f_{\tilde{t}}\ x_1\ \ldots\ x_n\ b_1\ \ldots\ b_v)\mu_{\mathrm{ren}_{TG}} \\
= & \mathrm{bR}_{TG}(\tilde{t})\mu_{\mathrm{ren}_{TG}}
\end{array}
$$

Furthermore, $\tilde{t}\mu$ starts an infinite $\hookrightarrow_{\mathsf{H}}$-reduction, where μ is evaluated enough by construction and terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$.

□

Now theorem 6.16 can be proven using the above lemma:

Proof of theorem 6.16. This theorem is shown indirectly. Thus it has to be shown that if a non-H-terminating term exists in TG, then there exists an SCC G' and a DP problem $\mathbf{dpRen}_{G'} = (\mathcal{P}_{ren}, \emptyset, \mathcal{R}_{ren}, \mathsf{a})$ such that there is an infinite reduction of the form

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

This implies there is also an infinite reduction of the form

$$s_1 \to_{\mathcal{P}_{ren}^\sharp} t_1 \to^*_{\mathcal{R}_{ren}} s_2 \to_{\mathcal{P}_{ren}^\sharp} t_2 \to^*_{\mathcal{R}_{ren}} \ldots$$

Let t be a non-H-terminating term. Then a substitution σ exists such that $t\sigma$ is a non-H-terminating ground term and where σ instantiates variables only with H-terminating terms. Therefore, it can be assumed that σ is a normal substitution, i.e., $\sigma(x)$ is a normal form w.r.t. $\to_{\mathsf{H}}$. 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. $\hookrightarrow_{\mathsf{H}}$, $\sigma(x)$ is terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$ for all variables x.

Since $t\sigma$ is a non-H-terminating ground term, it also is not terminating w.r.t. $\hookrightarrow_{\mathsf{H}}$, i.e., it starts an infinite $\hookrightarrow_{\mathsf{H}}$-reduction. By lemma 6.26, there is an infinite

path in TG. Since TG only has a finite number of nodes, the infinite path must end in some SCC G'. Only the infinite tail of this path is regarded further, which only traverses nodes in G'. Thus, there is 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 $i \in \mathbb{N}$:

- 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 $\mathcal{P}_{ren}$ contains the rule $\mathrm{bR}_{TG}(s^i)\sigma^i_1 \ldots \sigma^i_{m_i} \to \mathrm{bR}_{TG}(\mathbf{ev}(t^i))$
- $s^i\sigma^i$ 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
- $\sigma^i_{\mathrm{ren}_{TG}} = \sigma^i_1 \ldots \sigma^i_{m_i}\tau^i_{\mathrm{ren}_{TG}}$
- $\mathrm{bR}_{TG}(\mathbf{ev}(t^i))\tau^i_{\mathrm{ren}_{TG}} \xrightarrow{>\varepsilon}{}^*_{\mathcal{R}_{ren}} \mathrm{bR}_{TG}(s^{i+1})\sigma^{i+1}_{\mathrm{ren}_{TG}}$

Putting this together, we find an infinite chain:

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

□

Thus, we have proven that for a Termination Graph which was built using the expansion rules of definition 4.12 all nodes are H-terminating, if the renamed and corrected DP problems created from it are finite, which means that no infinite chain exists.

6.3 Examples for the strength of Renaming

Theorem 6.16 tells us that when adding ***TyCase***-nodes and applying renaming in order to create DP problems, we still have the property that if all DP problems are finite, then all evaluations of instances of the start term w.r.t. the given Haskell program are finite, as well. A question is, whether this added complexity results in more strength.

As presented in the beginning of this chapter, the idea of renaming is the separation of different cycles. Thus, the example 6.1 shall be considered again to demonstrate that the intended separation is achieved. There, it is the case that the evaluation of a Haskell term has to leave one cycle in order to get into another cycle. However, these cycles are embedded in one SCC, therefore they are contained in the same DP problem. The renaming will assign different names, which makes such cycles distinct and thus, the call structure of the problem is preserved in the created DP problem.

Example 6.27 (Separation of cycles by renaming)**.** *We want to reconsider example 6.1 to show that the intended separation of cycles by assigning different names to different nodes indeed gives the proposed results.*

The presented renaming assigns different names to the nodes that start the different implementations of the function `terminate`*. This happens, because of the fact that no instantiation edge exists between the two cycles. Thus, two*

different names are assigned, leading to the separation of the functions in the resulting DP problem.

$$
\begin{array}{lcll}
\mathcal{P} = \{ & new\_f(S(x), Z, Nats) & \to & new\_f(Z, Z, Nats) \\
& new\_f(S(Z), S(y), Bool) & \to & new\_f(Z, Z, Bool) \\
& new\_f(S(x), S(S(y0)), Nats) & \to & new\_terminate0(S(x), y0) \\
& new\_f(S(S(x0)), S(y), Bool) & \to & new\_terminate(x0, S(y)) \\
& new\_terminate0(x, S(y0)) & \to & new\_terminate0(S(x), y0) \\
& new\_terminate0(x, Z) & \to & new\_f(Z, Z, Nats) \\
& new\_terminate(S(x0), y) & \to & new\_terminate(x0, S(y)) \\
& new\_terminate(Z, y) & \to & new\_f(Z, Z, Bool) \quad \} \\
\mathcal{R} = \emptyset &&&
\end{array}
$$

In contrast to the DP problem created without renaming, this DP problem is finite, since in it the two different cycles for the different instances of `terminate` *are separated into the two different functions new_terminate*0 *and new_terminate, like it was proposed above. Finiteness of this DP problem can thus be easily shown with the Size-Change processor [TG05], for example.*

Another example that shows the improvements due to renaming is the previously presented example 3.1 for the start term `take u (from m)`. There, the predecessor of a natural number was computed explicitly, which makes termination analysis rather hard. However, if renaming is used, this can be solved rather easily, as shown in the following example.

Example 6.28 (Easier DP problems due to renaming)**.** *We want to consider termination analysis of the start term* `take u (from m)` *again. Please see example 3.1 for the program and figure 3.1 for the Termination Graph.*

If we create the DP problem for the SCC of `take` *without renaming, it will be the following:*

$$
\begin{array}{lcll}
\mathcal{P} = \{ & take(S(u0), from(m)) & \to & take(p(S(u0)), from(S(m))) \quad \} \\
\\
\mathcal{R} = \{ & p(S(Z)) & \to & Z \\
& p(S(S(u00))) & \to & S(p(S(u00))) \quad \}
\end{array}
$$

The above DP problem is too hard for many DP processors. This is, because one has to find out that the rules for p decrement their argument. However, this is only done in the case the end of the number is reached, where a constructor S is removed. A possibility to show this are polynomials with negative coefficients [HM07], where a possible interpretation of the function symbols is the following:

$$
\begin{array}{lcl}
\mathrm{Pol}(take)(x_1, x_2) & = & x_1 \\
\mathrm{Pol}(from)(x_1) & = & 0 \\
\mathrm{Pol}(S)(x_1) & = & 1 + x_1 \\
\mathrm{Pol}(p)(x_1) & = & \max\{0, x_1 - 1\}
\end{array}
$$

Then we have ordered the Dependency Pair strictly and can delete it. But searching for such orders is a computationally complex process. This complexity can be avoided if renaming is used. From the renamed Termination Graph shown in figure 6.3, the following renamed DP problem for the SCC of `take` *is created, as shown in example 6.10.*

$$\begin{array}{lcll} \mathcal{P} & = & \{ & new\_take(S(u0), m) \rightarrow new\_take(new\_p(u0), S(m)) \quad \} \\ \\ \mathcal{R} & = & \{ & new\_p(Z) \rightarrow Z \\ & & & new\_p(S(u00) \rightarrow S(new\_p(u00)) \quad \} \end{array}$$

This DP problem is much easier, because the recursive calls of new_take and of new_p decrement the number of S constructors they contain. This is also found out by the Size-Change processor [TG05], for example.

The reason, why this happens, is that the function `p` *always demands one* `S` *constructor. This constructor is already present in the* ***Ins****-node G containing the term* `take (p (S u0)) (from (S m))`*, therefore this constructor does not influence the rule that is applied to the subterm* `p (S u0)` *in node I and can be left out. The renaming only collects the variables of node I, and therefore the constructor is left out in the renamed term for this node.*

Chapter 7

Innermost Termination Analysis

Innermost termination analysis is far more powerful than regular termination analysis, which was already stated in [AG00]. This is due to the fact that for innermost termination a lot of modularity results exist which do not hold for the general termination case. Furthermore, there are a lot of processors available in the DP framework that are only applicable for innermost termination termination analysis, but not for the general case. One example for such a processor is the Bounded Increase processor [GTSSK07] which allows to prove termination of problems that terminate because eventually an upper boundary is reached for some argument. This class of problems could not be solved automatically before. In Haskell, such problems arise, for example, in case of the widely used arithmetic sequences, e.g., `[`n`..`m`]`. This is illustrated in the following example.

Example 7.1 (Arithmetic Sequences in Haskell)**.** *In this example, the arithmetic sequences of the form* `[n..m]` *shall be examined.*

This expression will be translated into the expression `enumFromTo n m` *by the definition of the semantics of these arithmetic sequences [Jon03, Section 3.10]. The function* `enumFromTo` *is a class member of the class* `Enum`*, where the default implementation of this class member, that is used for most types in the class* `Enum`*, is mapping it to the function* `numericEnumFromTo`*. This function and the functions that it uses, are defined in the Prelude. Below we give a slightly simplified version of these rules.*

```
numericEnumFromTo     :: Int -> Int -> [Int]
numericEnumFromTo n m = takeWhile (<= m) (numericEnumFrom n)

numericEnumFrom       :: Int -> [Int]
numericEnumFrom n     = n : (numericEnumFrom (n+1))

takeWhile             :: (a -> Bool) -> [a] -> [a]
takeWhile p []        = []
takeWhile p (x:xs)    | p x       = x : takeWhile p xs
                      | otherwise = []
```

As can be seen, the function `numericEnumFromTo` *works by taking a prefix of the infinite list of natural numbers starting at* `n`*, until the upper boundary* `m` *is reached. Note that we will reach this boundary, since the list elements are strictly increasing. This will also be found out by the mentioned processor for Bounded Increase.*

The above example shows that innermost termination enables us to greatly improve termination analysis for Haskell programs. Therefore, we will now show, why this counterintuitive property holds. The intuition is, that we already performed all lazy evaluation steps in the Termination Graph. Where we did not succeed with the lazy evaluation strategy, we do a kind of innermost termination analysis, since for the right-hand sides of Dependency Pairs, which result from ***Ins***-nodes, we regard all subterm children further, showing their termination.

In order to get innermost DP problems, we will first show that considering *minimal* chains suffices. Then, we will show that from these minimal DP problems, we can switch to minimal innermost DP problems, using the already existing technique of the Modular Non-Overlap Check [GTSK05b].

7.1 Towards Innermost: Minimal Chains

Termination analysis for DP problems is more powerful when only minimal chains have to be considered, instead of having to consider all possible chains. Thus, it would be an improvement in strength, if it would suffice to show absence of *minimal* chains, instead of arbitrary chains, for the DP problems created from a Termination Graph. This section will show that this property holds.

In order to be able to talk about the SCCs that correspond to a DP problem created from a Termination Graph, the following definition creates a mapping from a TRS to the corresponding SCCs in the graph.

Definition 7.2 (functions SCC and $SCCs$)**.** *Let TG be a Termination Graph, $(\mathcal{P}, \emptyset, \mathcal{R}, \mathsf{a})$ be a DP problem created from it.*

The result of the function $SCC(\mathcal{P})$ is the SCC in the graph TG the TRS $\mathcal{P}$ was read from.

The result of the function $SCCs(\mathcal{R})$ is the set of SCCs the TRS $\mathcal{R}$ was read from.

Then it holds that every non-terminating evaluation in the rules of a DP problem read from TG implies the existence of an SCC for these rules. This is formalized in the following lemma:

Lemma 7.3. *Let TG be a Termination Graph, $(\mathcal{P}, \emptyset, \mathcal{R}, \mathsf{a})$ be a DP problem created from it.*

If an infinite reduction $t_0 \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \ldots$ exists, then $SCCs(\mathcal{R}) \neq \emptyset$.

Proof. Assume there exists an infinite reduction $t_0 \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \ldots$.

Since the TRS $\mathcal{R}$ is finite, there exist $l = f(s_1, \ldots, s_n) \to r \in \mathcal{R}$ and an infinite set $I = \{i_1, i_2, \ldots\} \subseteq \mathbb{N}$ with $i_j < i_{j+1} \;\; \forall j \in \mathbb{N}$, such that

$$t_{i_j} = C_j[l\sigma_j]_{\pi_j} \text{ and } t_{i_j+1} = C_j[r\sigma_j]_{\pi_j}$$

for some contexts C_j, substitutions σ_j, and positions $\pi_j \in Occ(C_j)$.

Thus, the defined function symbol f occurs infinitely often. Renaming assigns different names to different nodes that start with a defined function symbol, except for the case of ***Ins***-nodes. This implies that an ***Ins***-node must exist, as otherwise there would be only non-recursive functions. Since the node, whose fresh name is f, can be reached again, there must be a cycle in the graph. Therefore, an SCC exists which contains this cycle, i.e., we have found an element of $SCCs(\mathcal{R})$, which implies that $SCCs(\mathcal{R}) \neq \emptyset$.

□

To show the main theorem, we will use an ordering on SCCs in a Termination Graph. The idea of this ordering is, that an SCC G_1 is smaller than another SCC G_2, if from the smaller SCC G_1, rules in the other SCC G_2 are created. This ordering is not necessarily irreflexive, i.e., it is not certain that an element is larger than itself in the ordering. This could happen, because an SCC might also contain the nodes from which rules have to be created for the DP paths in that SCC. This is illustrated in the following example.

Example 7.4 (Creation of rules from the same SCC as the Dependency Pairs)**.** *We construct the Termination Graph for the start term* `f x y` *for the following Haskell program.*

```
data Nats = Z | S Nats

f :: Nats -> Nats -> Nats
f (S x) y = f (f (g x) y) y

g :: Nats -> Nats
g Z = S Z
g (S x) = g x
```

As can be seen in the Termination Graph for the given start term, that is shown in figure 7.1, we have two SCCs in the Termination Graph. The first one consists of the nodes A,C,E,F and the other consists of the nodes G,I,K. The path from node A to node E is a DP Path, therefore rules must be created for its subterms. As subterm, we have the node F. Because this is an ***Ins****-node, the rules that are created from this node start in node A. Therefore, we have the same SCC for the rules as for the Dependency Pairs. In this situation we do not want that an SCC is bigger than itself, because then we could not use this ordering as an induction relation, because it would not be well-founded.*

However, the SCC consisting of the nodes G,I,K shall be smaller than the SCC A,C,E,F. This is, because we have that the SCC G,I,K is below the SCC A,C,E,F and cannot reach it.

To simplify the ordering, we use an ordering that does not consider whether rules have to be created or not. So we are sure that the intended order based on the creation of rules is contained in the order $\succ_{TG}$. This order is made irreflexive by demanding different SCCs.

Definition 7.5 ($\succ_{TG}$)**.** *Let G_1, G_2 be two SCCs in a Termination Graph TG.*

$G_1 \succ_{TG} G_2$ iff there exist a node $n_1 \in G_1$ and a node $n_2 \in G_2$ such that a path exists from n_1 to n_2 and $G_1 \neq G_2$.

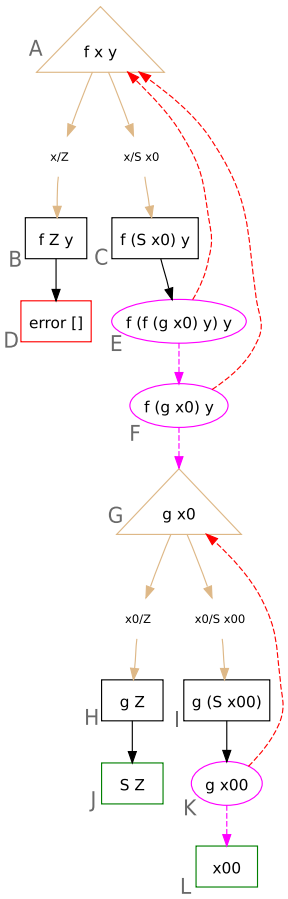

Figure 7.1: Termination Graph for `f x y`, illustrating the order on SCCs

Since we made the relation irreflexive, we can show that this is indeed a well-founded order. Thereby, it is usable as an induction relation.

Lemma 7.6. *The order $\succ_{TG}$ is well-founded.*

Proof. Let TG be a Termination Graph. Since it is finite, only finitely many SCCs exist. Let $S = \{G_1, \ldots, G_n\}$ be the set of SCCs, where $n \in \mathbb{N}$.

Assume, we have an infinite sequence $G'_1, G'_2, \ldots$ where $G'_i \succ_{TG} G'_{i+1}$. Because S is finite, there must be a $G_k \in S$ such that an infinite set $I = \{i_j \mid j \in \mathbb{N},\ G'_{i_j} = G_k\} \subseteq \mathbb{N}$ exists. Since the path relation is transitive, the order $\succ_{TG}$ is transitive, too. This implies that $G_k \succ_{TG} G_k$ holds. But this would imply $G_k \neq G_k$, yielding a contradiction.

□

Now the main theorem of this section shall be proven, which states that it suffices to consider only minimal chains. Before, all chains had to be considered, represented by the flag `a` for DP problems.

This flag can now be replaced by the flag `m`, yielding easier problems for the DP-framework backend. This property holds, because we create DP problems

for all SCCs in the Termination Graph. This is, because we do not know, whether a rule will be evaluated or not. Thereby, we could switch to a kind of "innermost" evaluation for ***Ins***-nodes, since these are the nodes from which the creation of rules started. The claim is that we either have already proven termination of the rules, or a DP problem exists for an SCC of the rules, such that we will find an infinite chain in that DP problem.

Theorem 7.7 (Minimal chains)**.** *For the set of DP problems created from a Termination Graph, the following holds:*

There exists an infinite DP problem $(\mathcal{P}, \emptyset, \mathcal{R}, \mathsf{a})$ *iff there exists an infinite* minimal $(\mathcal{P}', \emptyset, \mathcal{R}')$*-chain for some DP problem* $(\mathcal{P}', \emptyset, \mathcal{R}', \mathsf{a})$ *in the set of DP problems.*

Proof. Since every minimal chain is also a chain, this direction holds trivially.

For the other direction, we start with an infinite DP problem $(\mathcal{P}, \emptyset, \mathcal{R}, \mathsf{a})$. According to [GTSK05b], the DP problem $(\mathcal{P}, \emptyset, \mathcal{R}, \mathsf{a})$ is infinite iff either of the following holds:

- There exists an infinite $(\mathcal{P}, \emptyset, \mathcal{R})$-chain, or
- $\mathcal{R}$ is not terminating.

Case 1: $\mathcal{R}$ is terminating

This implies the existence of an infinite $(\mathcal{P}, \emptyset, \mathcal{R})$–chain, i.e., there exists an infinite sequence

$$s_1 \to t_1, s_2 \to t_2, \ldots$$

from $\mathcal{P}$ (with $\mathcal{V}(s_i) \cap \mathcal{V}(s_{i+1}) = \emptyset \ \ \forall i \in \mathbb{N}$) and a substitution σ over an infinite domain such that $t_i\sigma \to_{\mathcal{R}}^* s_{i+1}\sigma$. Since $\mathcal{R}$ is terminating, especially all $t_i\sigma$ are terminating w.r.t. $\to_{\mathcal{R}}$. Thus, this chain is a minimal chain, which proves the theorem.

Case 2: $\mathcal{R}$ is not terminating

This implies the existence of an infinite reduction $t_0 \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \ldots$

Case 2.1: $SCCs(\mathcal{R}) = \emptyset$

As shown in lemma 7.3, this case cannot occur.

Case 2.2: $SCC(\mathcal{P}) \succ_{TG} G'$ for some $G' \in SCCs(\mathcal{R})$ and the DP problem created for G' is infinite

In this case, the theorem inductively holds, since $\succ_{TG}$ is well-founded.

Case 2.3: For all $G' \in SCCs(\mathcal{R})$ with $SCC(\mathcal{P}) \succ_{TG} G'$, the DP problem created for G' is not infinite

For this case, an infinite minimal chain shall be constructed.

Since $\mathcal{R}$ is not terminating, we have $SCCs(\mathcal{R}) \ni SCC(\mathcal{P})$, since all sub-SCCs of $SCC(\mathcal{P})$ are not infinite by assumption.

Without loss of generality we can assume that the first term t_0 is a node on $SCC(\mathcal{P})$ with an incoming instantiation edge. This holds since the only possibility to have an infinite reduction is by traversing nodes on $SCC(\mathcal{P})$, as all DP problems created for sub-SCCs are not infinite by assumption.

Starting in t_0, we follow the path of the infinite reduction. When reaching an ***Ins***-node, all children that are not connected via its instantiation node are considered. If one of them starts an infinite reduction, then this infinite reduction will be the guide for building the minimal chain and no Dependency Pair is added to the sequence. Otherwise, if all children not connected via the instantiation edge only start finite evaluations, the Dependency Pair $\mathrm{bR}_{TG}(t_0)\ \sigma_1 \dots \sigma_m \to \mathrm{bR}_{TG}(\mathbf{ev}(t))$ is added to the sequence, where t is the term of the ***Ins***-node and $\sigma_1, \dots, \sigma_m$ are the substitutions the edges on the path were labelled with. Since all subterms are terminating by assumption, this Dependency Pair does not violate the minimality. Building the sequence is then continued from the node $\tilde{t}$, which is the node connected to t via the instantiation edge. The term t_0 is now being updated with $\tilde{t}$.

Since this sequence always respects the minimality condition, the whole sequence is minimal. As Dependency Pairs are created for every DP path, and the nodes t_0 and t are connected by such a path (t_0 has an incoming instantiation edge in $SCC(\mathcal{P})$, t has an outgoing instantiation edge in $SCC(\mathcal{P})$), $\mathcal{P}$ contains all Dependency Pairs in this sequence. Therefore, an infinite minimal chain exists.

□

An example that illustrates the above theorem is given below.

Example 7.8 (Minimal chain exists for an infinite Haskell program)**.** *We want to analyze termination of the start term* `f x` *in the following Haskell program:*

```
f :: Bool -> Bool
f x = f (f x)
```

This start term is clearly not H*-terminating, because an infinite evaluation exists:*

$$\begin{aligned} \texttt{f x} &\to_{\mathsf{H}} \texttt{f (f x)} \\ &\to_{\mathsf{H}} \texttt{f (f (f x))} \\ &\to_{\mathsf{H}} \texttt{...} \end{aligned}$$

Now, we want to consider the DP problem created from the Termination Graph for this start term, which is shown in figure 7.2.

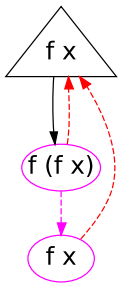

Figure 7.2: Termination Graph for `f x`, illustrating minimal chains

The DP problem for the only SCC in the Termination Graph is:

$$\begin{array}{rcl} \mathcal{P} & = & \{ \begin{array}{lcl} F(x) & \to & F(f(x)) \\ F(x) & \to & F(x) \end{array} \} \\ \\ \mathcal{R} & = & \{ \begin{array}{lcl} f(x) & \to & f(f(x)) \end{array} \} \end{array}$$

As can be seen, we also have an infinite chain that stacks up the function f:

$$\begin{array}{lll} F(x) & \to_{\mathcal{P}} & F(f(x)) \\ & \to_{\mathcal{P}} & F(f(f(x))) \\ & \to_{\mathcal{P}} & \ldots \end{array}$$

*However, this is not a minimal chain, since the subterm $f(x)$ is not terminating. But since we consider the children of **Ins**-nodes as well, we also follow these and create Dependency Pairs for them. This generates the second Dependency Pair, which does not correspond to a Haskell evaluation. Using this Dependency Pair, another infinite chain exists, which simply is $F(x) \to_{\mathcal{P}} F(x) \to_{\mathcal{P}} \ldots$. This is an infinite* minimal *chain.*

7.2 Switching to Innermost

The previous section showed that it suffices to only consider minimal chains. This is one of the necessary preconditions for switching to innermost termination analysis, i.e., setting $\mathcal{Q} = \mathcal{R}$. A DP processor that does this is the Modular Non-Overlap Check processor which is presented in [GTSK05b]. This section will show that this processor can always be applied, and hence innermost DP problems can directly be generated.

Theorem 7.9 (Innermost termination analysis suffices)**.** *Let TG be a Termination Graph, let $(\mathcal{P}, \emptyset, \mathcal{R}, \mathsf{m})$ be a DP problem created from it.*

Then this DP problem can be replaced with $(\mathcal{P}, \mathcal{R}, \mathcal{R}, \mathsf{m})$, i.e., only innermost termination analysis needs to be performed.

Proof. This will be proven using theorem 32 of [GTSK05b] which introduces and proves soundness and completeness of the Modular Non-Overlap Check processor.

In order for this processor to be applicable to a DP problem $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, f)$, four properties have to be fulfilled:

1. for all $s \to t \in \mathcal{P}$, non-variable subterms of s do not unify with left-hand sides of rules from $\mathcal{R}$ (after variable renaming), and
2. $\xrightarrow{\mathcal{Q}}_{\mathcal{R}}$ is locally confluent, and
3. $\mathcal{Q} \subseteq \mathcal{R}$, and
4. $f = \mathsf{m}$.

If these properties are fulfilled, then the result of the processor is the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{R}, \mathsf{m})$.

Let $(\mathcal{P}, \emptyset, \mathcal{R}, \mathsf{m})$ be a DP problem created from TG, i.e., $\mathcal{Q} = \emptyset$. Thus, the properties 3 and 4 are fulfilled trivially.

Property 1 is fulfilled, since the left-hand side of a DP problem created from a Termination Graph is always of the form $(f\ x_1 \dots x_n\ b_1 \dots b_v)\ \sigma_1 \dots \sigma_m$, where every σ_i is of the form $[y/(c_i\ z_1 \dots z_k)]$ where every c_i is either a data- or a type-constructor and $z_1, \dots, z_k$ are fresh variable. Therefore, no left-hand side of a rule can unify with a non-variable subterm, since every left-hand side starts with a defined symbol.

As a last step, it has to be shown that property 3 is satisfied, i.e., that $\xrightarrow{\mathcal{Q}}_{\mathcal{R}} = \rightarrow_{\mathcal{R}}$ is locally confluent. First is shall be noted that also every left-hand side of a rule is of the above form, i.e., there are no defined symbols in subterms of the left-hand side.

Assume that there are two rules $l_1 \rightarrow r_1 \in \mathcal{R}$ and $l_2 \rightarrow r_2 \in \mathcal{R}$, where a most general unifier μ of l_1 and l_2 exists. This means that the defined function symbol, the rules start with, must be the same. Therefore, the rule paths that led to these rules must start in the same node, as otherwise renaming would have assigned different function symbols.

For the left-hand sides of the rules, let $l_1 = (f\ x_1 \dots x_n\ b_1 \dots b_v)\ \sigma_{1,1} \dots \sigma_{1,m_1}$ and let $l_2 = (f\ x_1 \dots x_n\ b_1 \dots b_v)\ \sigma_{2,1} \dots \sigma_{2,m_2}$. If there exists an $i \in \mathbb{N}$ such that $\sigma_{1,i} \neq \sigma_{2,i}$ and for all $j < i$ it holds that $\sigma_{1,j} = \sigma_{2,j}$, then $\sigma_{1,i}$ and $\sigma_{2,i}$ must have resulted from a ***Case***- or a ***TyCase***-node. Because of $\sigma_{1,i} \neq \sigma_{2,i}$, a variable y exists, for which $\sigma_{1,i}(y) \neq y \neq \sigma_{2,i}(y)$ and $\sigma_{1,i}(y) \neq \sigma_{2,i}(y)$ holds. Since both heads of $\sigma_{1,i}(y)$ and $\sigma_{2,i}(y)$ are constructors, this would lead to a clash failure in unification, i.e., this contradicts the assumption that $l_1\mu = l_2\mu$. Therefore, this case cannot occur.

Otherwise, it holds that $\sigma_{1,i} = \sigma_{2,i}$ for all $1 \leq i \leq \min\{m_1, m_2\}$. If $m_1 < m_2$, then the rule path π_1 that led to the rule $l_1 \rightarrow r_1$ is a prefix of the rule path π_2 that led to $l_2 \rightarrow r_2$. Thus, the node that r_1 was created from must not be a ***Case***-, a ***TyCase***-, or an ***Eval***-node, since π_1 ended in that node. As π_2 is a rule path, all nodes, except for the node r_2 was created from, must be ***Case***-, ***TyCase***-, or ***Eval***-nodes. Since π_1 is a real prefix of π_2, the node r_1 was created from cannot be the node r_2 was created from. Thus, the node r_1 was created from must be a ***Case***-, ***TyCase***-, or ***Eval***-node, in contradiction to the above. Hence, this case cannot occur, either. Analogously, $m_2 > m_1$ can never occur. Therefore, $m_1 = m_2$ can be concluded, which entails that the rule paths must be the same, since they must end in the first node that is neither a ***Case***-, nor a ***TyCase***-, nor an ***Eval***-node. Hence, the left- and right-hand sides of the rules must be the same, i.e., $l_1 \rightarrow r_1 = l_2 \rightarrow r_2$.

From the above, it can be concluded that $\mathcal{R}$ is non-overlapping and therefore no critical pairs exist. Thus, the relation $\rightarrow_{\mathcal{R}}$ is locally confluent.

□

Another interesting question is, whether it would be possible to set $\mathcal{Q}$ to $\{f(x_1, \dots, x_n) \rightarrow \dots \mid f \text{ defined symbol}, n = arity(f)\}$. Then, the created DP problems must only be regarded as constructor rewrite systems, i.e., one has to only consider chains where every subterm of a left-hand side of a Dependency Pair consists only of constructors and variables. Unfortunately, this is not the case, as the correctness of the Haskell termination analysis would be destroyed, as illustrated in the following example.

Example 7.10. *Consider the following Haskell program:*

```
f :: Int -> Int
f x = f (div x 0)
```

with the start term `f x` *(where* `div` *is the standard division function, i.e., especially it holds that* `div _ 0 = error []`*). As can easily be seen, the start term is not* H*-terminating, since an infinite evaluation of a ground instance exists.*

The above program would result in the following DP problem:

$$\begin{array}{lcl} \mathcal{P} & = & \{F(x) \to F(div0(x))\} \\ \mathcal{R} & = & \{div0(Pos(x0)) \to error([]),\ div0(Neg(x0)) \to error([])\} \end{array}$$

When considering $\mathcal{Q} = \mathcal{R}$, then we find that an infinite $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-chain exists: Let $\sigma = [x/error([])]$, then $F(x)\sigma = F(error([])) \to_{\mathcal{P}} F(div0(error([])))$ where the term $F(div0(error([])))$ is normal w.r.t. $\mathcal{R}$ and therefore it is also normal w.r.t. $\mathcal{Q}$. This term starts the following infinite reduction, where every term is normal w.r.t. $\mathcal{Q}$:

$$F(div0(error([]))) \to_{\mathcal{P}} F(div0(div0(error([])))) \to_{\mathcal{P}} \ldots$$

However, if $\mathcal{Q}$ is set to $\{div0(x) \to \ldots\}$, then the term $F(div0(error([])))$ is not normal w.r.t. $\mathcal{Q}$, but it cannot be further reduced using the rules in $\mathcal{R}$. Therefore, this is incorrect, as every chain contains this term (really they must contain a term of the form $F(div0(g(s_1, \ldots, s_n)))$ where $g \notin \{Pos, Neg\}$, since DP problems do not regard types).

In order to fix this, one would have to add the set $\mathcal{R}_{error}$ to $\mathcal{R}$, where $\mathcal{R}_{error} = \{f(x_1, \ldots, x_{i-1}, error(y), x_{i+1}, \ldots, x_n) \to error(y) \mid$ y is a fresh variable, f is a defined symbol in $\mathcal{R}$, $arity(f) = n$, $1 \leq i \leq n\}$. Then an error term could no longer block the chain. But this increases the size of $\mathcal{R}$ a lot, and the rules in $\mathcal{R}$ could be merged into $\mathcal{P}$ by the Narrowing processor [AG00, GTSKF03], creating a large set of Dependency Pairs that would have to be ordered. Therefore, setting $\mathcal{Q}$ to these more general terms is not used.

Chapter 8

Evaluation of the Improvements

In order to assess whether the presented improvements are useful in practice, we compared an AProVE version without the described improvements, which we will call $\mathsf{AProVE}_{\text{PLAIN}}$, to a version where all the above improvements where enabled, which shall be identified with $\mathsf{AProVE}_{\text{FULL}}$. Thereby, we measure the improvements presented in Chapters 6 and 7. The DP framework configuration for both versions was the strategy for the termination competition in 2007. Please note that some processors were not applicable during the runs of $\mathsf{AProVE}_{\text{PLAIN}}$, since here no minimal or innermost information was available.

In order to assess the effects of renaming, we furthermore ran over the set of examples another version of AProVE, where renaming was used and created first-order DP problems, but these were not restricted to minimal chains and hence were neither restricted to innermost chains. Also, no type classes were introduced into the created terms, i.e., the addition of the type variables in the renaming was not made. This version of AProVE is called $\mathsf{AProVE}_{\text{REN}}$ in the following. This comparison thus measure the effects of the improvements presented in Chapter 6, without the improvements of Chapter 7.

We ran all versions on a test corpus consisting of 1281 examples which were all extracted from libraries shipped with the Hugs interpreter [JP99]. The libraries we tested were `FiniteMap`, `List`, `Maybe`, `Monad`, `Prelude`, and `Queue`. Here, we tried to prove start terms for every exported function. For such a function, we tried the general version, and where applicable, a version of the start term where the type classes had been instantiated once with all instances for that type class. It is worth noting that our `Prelude` implementation differs from the Hugs implementation, as it contains quite a few native functions that are not implemented in Haskell, but are implemented on the processor. For these, a pure Haskell implementation has been developed to model the function as closely as possible. These integer operations assume that the numbers for an `Int` are not bounded, i.e., the predefined data type `Int` behaves just like the predefined data type `Integer`.

A result can either be a **YES**, in which case the start term could successfully be shown H-terminating, **MAYBE**, where no proof could be found, or **Timeout**, when no proof could be found within 5 minutes. It should be mentioned

that showing 100 % of the start terms as terminating cannot be reached, since some of the start terms are non-terminating.

Version	YES	MAYBE	Timeout
AProVE$_{\text{PLAIN}}$	726 (56.67 %)	104 (8.11 %)	451 (35.20 %)
AProVE$_{\text{REN}}$	973 (75.95 %)	86 (6.71 %)	222 (17.33 %)
AProVE$_{\text{FULL}}$	1008 (78.68 %)	68 (5.30 %)	205 (16.00 %)

Table 8.1: Overall results of AProVE$_{\text{PLAIN}}$, AProVE$_{\text{REN}}$ and AProVE$_{\text{FULL}}$

For the described set of modules and start terms, the total results presented in table 8.1 were obtained. As can be seen in those results, the improvements resulted in a total of 21.8 % more examples that could be proven terminating. Therefore, it is conjectured that the presented improvements are of practical value. Especially, the renaming seems to have a great impact onto the results, since with it enabled we already have an increase of almost 19.2 % more examples which could be proven terminating.

When looking closer at the modules, it is interesting in which modules these gains have been made. Table 8.2 shows the numbers broken down according to the modules.

Module	Version	YES	MAYBE	Timeout
`FiniteMap`	AProVE$_{\text{PLAIN}}$	116 (36.13 %)	16 (4.98 %)	189 (58.87 %)
	AProVE$_{\text{REN}}$	233 (72.58 %)	0 (0.00 %)	88 (27.41 %)
	AProVE$_{\text{FULL}}$	258 (80.37 %)	0 (0.00 %)	63 (19.62 %)
`List`	AProVE$_{\text{PLAIN}}$	64 (36.78 %)	29 (16.66 %)	81 (46.55 %)
	AProVE$_{\text{REN}}$	165 (94.82 %)	5 (2.87 %)	4 (2.29 %)
	AProVE$_{\text{FULL}}$	168 (96.56 %)	4 (2.29 %)	2 (1.14 %)
`Maybe`	AProVE$_{\text{PLAIN}}$	9 (100.00 %)	0 (0.00 %)	0 (0.00 %)
	AProVE$_{\text{REN}}$	9 (100.00 %)	0 (0.00 %)	0 (0.00 %)
	AProVE$_{\text{FULL}}$	9 (100.00 %)	0 (0.00 %)	0 (0.00 %)
`Monad`	AProVE$_{\text{PLAIN}}$	68 (85.00 %)	11 (13.75 %)	1 (1.25 %)
	AProVE$_{\text{REN}}$	69 (86.25 %)	11 (13.75 %)	0 (0.00 %)
	AProVE$_{\text{FULL}}$	69 (86.25 %)	11 (13.75 %)	0 (0.00 %)
`Prelude`	AProVE$_{\text{PLAIN}}$	464 (67.05 %)	48 (6.93 %)	180 (26.01 %)
	AProVE$_{\text{REN}}$	492 (71.09 %)	70 (10.11 %)	130 (18.78 %)
	AProVE$_{\text{FULL}}$	499 (72.10 %)	53 (7.65 %)	140 (20.23 %)
`Queue`	AProVE$_{\text{PLAIN}}$	5 (100.00 %)	0 (0.00 %)	0 (0.00 %)
	AProVE$_{\text{REN}}$	5 (100.00 %)	0 (0.00 %)	0 (0.00 %)
	AProVE$_{\text{FULL}}$	5 (100.00 %)	0 (0.00 %)	0 (0.00 %)

Table 8.2: Comparison of AProVE$_{\text{PLAIN}}$, AProVE$_{\text{REN}}$, and AProVE$_{\text{FULL}}$ by modules

In these results, one sees especially that for the modules `FiniteMap` and `List` the number of proven start terms has more than doubled. This can be explained by the high amount of higher order functions for these structures. Here, usually the provided function is applied to each element of the structure consecutively, i.e., the output of this application does not influence the termination behavior. However, such problems cannot be transformed using the A-Transformation [GTSK05a], i.e., they cannot be converted into first-order

DP problems. Then, proving termination is a much harder problem, as can be seen in the results.

Furthermore, more than 5 % have been gained in the Prelude. This is again mostly due to the renaming generating first order terms, but also due to the minimal innermost strategy that suffices: Examples such as `gcd`, which calculates the greatest common divisor using Euclid's algorithm, can now be proven with the processor for Bounded Increase [GTSSK07]. This processor was not applicable before.

Chapter 9

Lazy-Termination Analysis

Since Haskell is a lazy-evaluating language, terms which are not H-terminating are still of interest, since they can contribute to an H-terminating evaluation by generating a sequence of constructors. An example of a lazy-terminating term is `repeat Z`, where the function `repeat` is defined as `repeat x = x : repeat x`. Here, a sequence of list constructors is built which could, for example, be consumed by a `take` function.

A lazy-terminating function could also build infinite constructor sequences for terms that are arguments of another infinite constructor sequence, such as `repeat (repeat Z)`, which is an infinite list of infinite lists filled with `Z`. Such terms shall still be called lazy-terminating. However, terms such as `repeat bot`, where `bot` is defined as `bot = bot`, shall not be lazy-terminating, i.e., we demand constructor terms for every argument of a constructor. Still, this last example might be useful for a program; if only the list structure is of interest, but not the contained elements. This notion of lazy termination will not be handled by our approach, since here only certain arguments of certain constructors are being regarded. Thus, an automatic selection of these would be hard, especially for user-defined data types.

Lazy-Termination analysis might also help in reusing termination proofs. If it could for example be found out that a function only requires a lazy-terminating argument to be H-terminating, then the idea could be to reuse the H-termination proof of that function and show Lazy-Termination of the argument.

In the following, Lazy-Termination shall be defined formally. This definition follows the definition of [PSS97], but extends it to functional types. For these, we allow the application to an arbitrary lazy-terminating term. Then this new application must also be lazy-terminating in order to call the function lazy-terminating. So for example, the function `repeat` is lazy-terminating, since for every lazy-terminating argument t the term `repeat` t is lazy-terminating, as well.

Definition 9.1 (lazy-terminating)**.** *Every ground Haskell expression is 0-lazy-terminating. A ground Haskell expression t is n-lazy-terminating iff either t is* H*-terminating, or $t \to_{\mathsf{H}}^{*} (c\ t_1 \dots t_n)$ for a constructor c and every t_i is $(n-1)$-lazy-terminating, or t has a functional type and $(t\ s)$ is n-lazy-terminating for every n-lazy-terminating term s.*

A ground Haskell expression is lazy-terminating iff it is n-lazy-terminating for every $n \in \mathbb{N}$.

A Haskell expression t is lazy-terminating, iff for every substitution σ that instantiates the variables in t with expressions of correct types, $t\sigma$ is lazy-terminating.

In order to show Lazy-Termination of a term, we want to reuse the analysis for H-Termination. In order to do this, a class instance is generated for every data type of the Haskell program. This instance consumes every constructor up to a certain depth.

9.1 Generating Instances for Lazy-Termination

Whether a Haskell expression is lazy-terminating or not can be reduced to H-termination analysis, which was already mentioned in [PSS97] for terms having non-functional types. We follow the approach presented there, but use a class `LazyTermination` to integrate the lazy termination analysis into Haskell. This class is defined in the following.

Definition 9.2 (`LazyTermination` class)**.** *The following class is added as a predefined, not exported class to the Prelude:*

```
class LazyTermination a where
  lazyTerminating :: Nats -> a -> Bool
```

where `Nats` *is the definition of natural numbers in Peano notation. These are defined as*

```
data Nats = Z | S Nats
```

Then, for every type an instance of this class is generated which consumes the data constructors of that type. This shall be illustrated in the following example.

Example 9.3 (Lazy Termination instance)**.** *Consider the type*

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

For this type, the following instance, which is used for Lazy Termination analysis, is generated:

```
instance LazyTermination a => LazyTermination (Tree a) where
  lazyTerminating  Z     _           = True
  lazyTerminating (S n) (Leaf x)     = lazyTerminating n x
  lazyTerminating (S n) (Node x l r) = (lazyTerminating n x)
            && (lazyTerminating n l) && (lazyTerminating n r)
```

The problem is that also in the case of Lazy Termination analysis, an extension with fresh variables is needed, but variables are currently fixed to represent H-terminating terms. This semantics shall not be changed, so another mechanism is needed to handle functional types. For this purpose, a function is used that non-deterministically introduces arbitrary H-terminating terms on right-hand sides, namely the function `terminator`. This function is added to the

Prelude, and has the special semantics during evaluation that it will be evaluated to a fresh variable. In order to represent arbitrary lazy-terminating terms, the class definition of `LazyTermination` is extended by a new defined function `lazyGenerator`.

Definition 9.4 (`LazyTermination` class with member `lazyGenerator`)**.** *The class* `LazyTermination`*, which was defined in definition 9.2, is extended to the following definition:*

```
class LazyTermination a where
  lazyTerminating :: Nats -> a -> Bool
  lazyGenerator   :: a
```

The instance presented in example 9.3 is extended by the following definition of `lazyGenerator`.

Example 9.5 (Extension of example 9.3 to `lazyGenerator`)**.** *The function* `lazyTerminating` *remains as presented above, so only the definition of the additional function* `lazyGenerator` *is presented here.*

```
instance LazyGenerator a => LazyGenerator (Tree a) where
  lazyTerminating ...

  lazyGenerator
    | terminator = Leaf lazyGenerator
    | otherwise  = Node lazyGenerator lazyGenerator lazyGenerator
```

As can be seen, the function `terminator` *is used to introduce non-determinism in the evaluation. Here, it is used as a* `Bool` *which then determines which constructor shall be created.*

In order to also handle terms having a functional type, an instance of the class `LazyTermination` for the constructor `->` must be provided. This instance is defined as follows:

Definition 9.6 (Handling of functional types)**.** *The following instance is added to the Prelude as a non-exported instance of the class* `LazyTermination`*.*

```
instance (LazyTermination a, LazyTermination b)
  => LazyTermination (a -> b) where
    lazyTerminating  Z    _ = True
    lazyTerminating (S n) f
      = lazyTerminating (S n) (f lazyGenerator)

    lazyGenerator x
      = (lazyTerminating terminator x) `seq` lazyGenerator
```

For the function `lazyTerminating`, we see that a function is $(n+1)$-lazy-terminating, if the result of the function applied to a lazy terminating argument, which is generated by the function `lazyGenerator`, is also $(n+1)$-lazy-terminating.

The result of the function `lazyGenerator` must be a function that is lazy-terminating if its argument is lazy-terminating. The requirement for the argument to be lazy terminating is the first part before the `` `seq` ``, which means that

the argument must be m-lazy-terminating for all possible values of m. Then the function returns a lazy terminating term, which is the `lazyGenerator` after the `‘seq‘`.

For every other type, the instances are generated according to their definition. These derived instances require a term with a data constructor as head symbol, and evaluate the arguments further, where one `S` constructor less is used for the arguments. This corresponds to the definition of lazy termination, where a term starting with a constructor is said to be $(n+1)$-lazy terminating, iff its arguments are n-lazy terminating.

Definition 9.7 (Generated instances of `LazyTermination` for data types)**.** *Let*

$$\texttt{data}\ T\ a_1\ \ldots\ a_m\ \texttt{=}\ C_1\ \rho_{1,1}\ \ldots\ \rho_{1,k_1}\ \texttt{|}\ \ldots\ \texttt{|}\ C_l\ \rho_{l,1}\ \ldots\ \rho_{l,k_l}$$

be a type declaration in a Haskell program, where $a_1, \ldots, a_m$ are type variables, $C_1, \ldots, C_l$ are the data constructors of this type, and where $\rho_{i,j}$ is a type for all i, j.

Then for the type T the following instance of the class `LazyTermination` *is generated:*

```
instance (LazyTermination a_1, ..., LazyTermination a_m)
  => LazyTermination (T a_1 ... a_m) where
    lazyTerminating  Z     _              = True
    lazyTerminating (S n) (C_1 x1 ... xk_1)
      = (lazyTerminating n x1) && ... && (lazyTerminating n xk_1)
    ⋮
    lazyTerminating (S n) (C_l x1 ... xk_l)
      = (lazyTerminating n x1) && ... && (lazyTerminating n xk_l)

    lazyGenerator
      | terminator = C_1 lazyGenerator_1 ... lazyGenerator_k_1
      | terminator = C_2 lazyGenerator_1 ... lazyGenerator_k_2
      ⋮
      | terminator = C_l-1 lazyGenerator_1 ... lazyGenerator_k_l-1
      | otherwise  = C_l lazyGenerator_1 ... lazyGenerator_k_l
```

where the indices of the `lazyGenerator` *arguments shall only indicate the number of arguments. If there is a constructor C_i that has no arguments, then the right-hand side of the function* `lazyTerminating` *that consumes this constructor is set to be* `True`*.*

These generated instances will then be used to reduce the analysis of lazy-termination to the analysis of H-termination. The correctness of this approach is shown in the next section.

9.2 Reduction of Lazy-Termination to H-Termination

When a start term t is to be checked for lazy termination, the instances of the class `LazyTermination` for all types are added. Then, the start term is replaced

by `lazyTerminating n` t, which will be analyzed for H-termination. We will show that from the H-termination of this new start term, lazy-termination of the initial start term has been shown.

In order to show the main theorem, we first need a new relation that is able to also append new lazy terminating arguments.

Definition 9.8 ($\rightsquigarrow_{\mathsf{H}}$)**.** *For two terms s and t, we have $s \rightsquigarrow_{\mathsf{H}} t$ iff*

- $s \to_{\mathsf{H}} t$*, or*
- $t = (s\ u)$ *for an arbitrary lazy-terminating term u, if s has a functional type.*

This relation is defined in such a way that the definition of lazy termination is modelled with it. This can be seen from the definition of lazy termination, and is formalized in the following.

Corollary 9.9 (Properties of $\rightsquigarrow_{\mathsf{H}}$)**.** *For the relation $\rightsquigarrow_{\mathsf{H}}$, it holds that a term t is $(n+1)$-lazy-terminating, iff either t is H-terminating, or $t \rightsquigarrow_{\mathsf{H}}^* (c\ t_1 \dots t_n)$ for a constructor c and all t_i are n-lazy-terminating.*

The above relation is used in order to show that the class `LazyTermination` does indeed help in proving lazy-termination of a start term. This is done by showing that if `lazyTerminating n` t is H-terminating, then lazy-termination of t can be concluded. This is proven in the following theorem.

Theorem 9.10 (H-termination of `lazyTerminating` proves Lazy-Termination)**.** *Let t be a Haskell term, let TG be a Termination Graph for the start term `lazyTerminating n` t.*

The term t is lazy-terminating, if `lazyTerminating n` t *is H-terminating.*

Proof. Assume, the term t is not lazy-terminating, but `lazyTerminating n` t is H-terminating.

In case t is not a ground term, we know that a ground instance of t exists, that is not lazy-terminating. Thus, it suffices to only consider the case where t is a ground term, since this shows the theorem for non-ground terms, as well.

Because t is not lazy-terminating, t is not H-terminating, either. Furthermore, there must be a $m \in \mathbb{N}$, for which t is not $(m+1)$-lazy-terminating, i.e., there exists a term t', a context s, and a position $\pi \in Occ(s)$, such that $t \rightsquigarrow_{\mathsf{H}}^* s[t']_\pi$, $t' \not\rightsquigarrow_{\mathsf{H}}^* (c\ t'_1 \dots t'_l)$ for any constructor c of arity l, $\pi = i_1 \dots i_m$ with $i_j \in \mathbb{N}$ for all $1 \leq j \leq m$, and for all $\pi' \in Occ(s)$ with either $|\pi'| < |\pi|$ or where $\pi' = i_1 \dots i_{m-1}\ i$ with $i < i_m$, we have $s|_{\pi'} = (c'\ s'_1 \dots s'_k)$ for a constructor c'.

The term t' is not H-terminating, either. This holds, because it cannot be the case that $t' \rightsquigarrow_{\mathsf{H}}^* e$ for an error term e. Otherwise, this term would make the term t H-terminating, which it is not. Furthermore, t' does not reach any weak head normal form by assumption, therefore its evaluation must continue. Thus, only cases (a) and (b) from definition 2.7 can apply, which correspond to Haskell evaluation and argument saturation.

For the start term `lazyTerminating` $(m+1)$ `t`, the reduction with $\to_{\mathsf{H}}$ starts as follows (where *rest* is some Haskell term, and the natural numbers are converted to their representation using `Z` and `S`):

$$\texttt{lazyTerminating}\ (m+1)\ t \to_{\mathsf{H}}^* \texttt{lazyTerminating 1}\ t'\ \texttt{\&\&}\ \mathit{rest}$$

As seen in definition 9.6, whenever a function type is encountered, it is supplied with another arbitrary lazy-terminating argument. Since the arity of every type is finite, every functional type will be reduced to a non-functional type. Thus, the analysis of functional types is reduced to that of non-functional types.

By construction, all rules for `lazyTermination` for non-functional types require a constructor on the second position, when the first argument contains the representation of a number that is greater than zero. Since we have the representation of 1 as the first argument, and no constructor can be reached starting with the term t', the evaluation position for the reached term is always at least the second position of the term starting with `lazyTerminating`. Therefore, an infinite evaluation of `lazyTerminating n` t exists, contradicting the assumption that this term was H-terminating.

□

9.3 Examples for Lazy-Termination

After we have shown the correctness of our approach, we want to present examples to illustrate how Lazy-Termination of a start term is proven.

First, we are coming back to the example from the beginning of this chapter. It shall now be shown that the start term `repeat (x :: Nats)` is indeed lazy-terminating.

Example 9.11 (Lazy Termination analysis of `repeat`)**.** *First, the instances of the class* `LazyTermination` *are generated for the data types* `Nats` *and for the predefined lists. Please note that the second data type definition is not present in the final program, but it is only included to show the constructors of the predefined lists.*

```
data Nats = Z  | S Nats
data [a]  = [] | a : [a]

instance LazyTermination Nats where
  lazyTerminating  Z     _    = True
  lazyTerminating (S n)  Z    = True
  lazyTerminating (S n) (S x) = lazyTerminating n x

  lazyGenerator | terminator = Z
                | otherwise  = S lazyGenerator


instance LazyTermination a => LazyTermination [a] where
  lazyTerminating  Z    _      = True
  lazyTerminating (S n) []     = True
  lazyTerminating (S n) (x:xs) = (lazyTerminating n x) &&
                                 (lazyTerminating n xs)

  lazyGenerator | terminator = []
                | otherwise  = lazyGenerator : lazyGenerator
```

These instances are added to the program, and the start term is replaced by `lazyTerminating n (repeat (x :: Nats))`*. Then the Termination Graph is developed for this start term, which is shown in figure 9.1.*

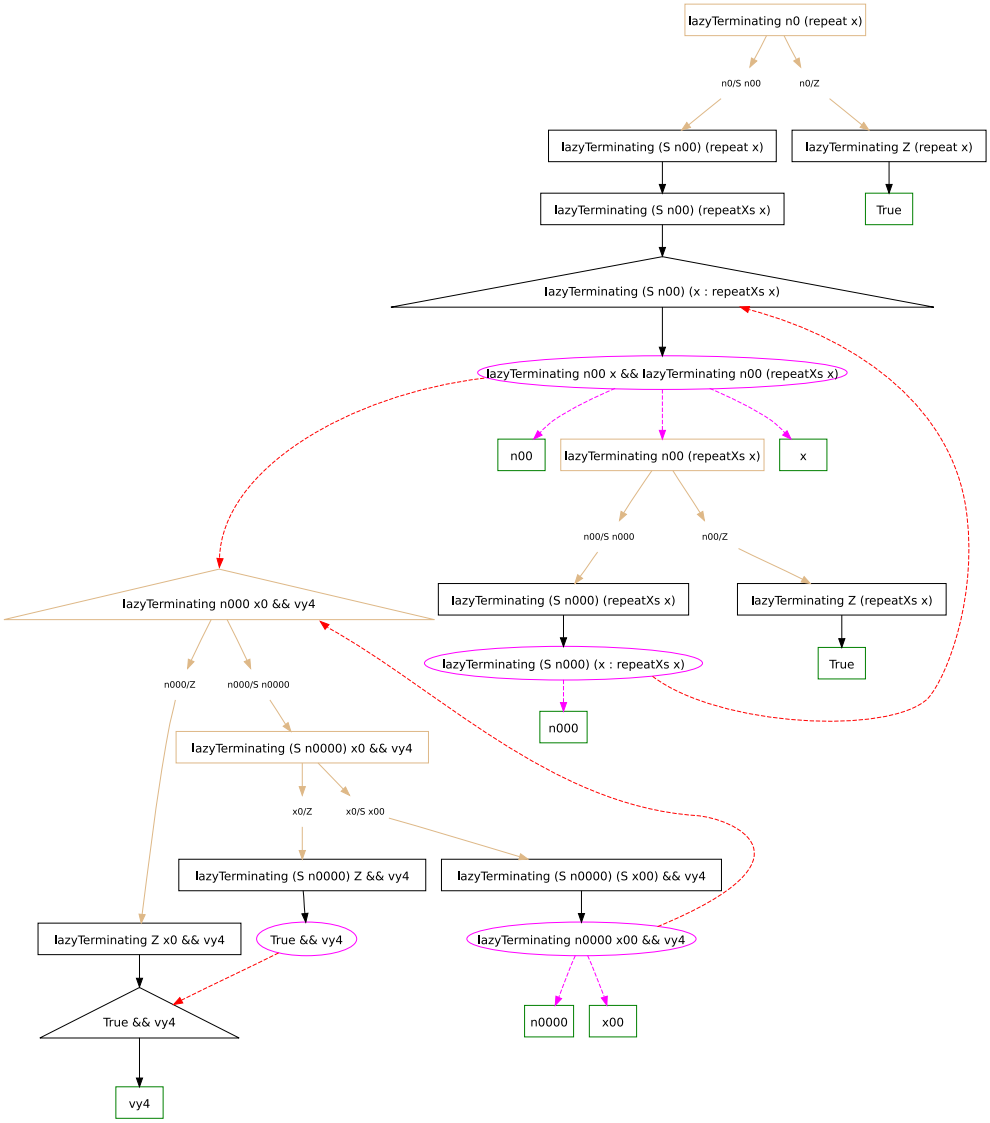

Figure 9.1: Termination Graph for Lazy-Termination analysis of start term `repeat (x::Nats)`

As can be seen, two SCCs exist in the Termination Graph. For these, the following DP problems are created (after applying renaming and correction):

$$\begin{aligned} \mathcal{P} &= \{\ new\_asAs(S(n0000), S(x00), vy4) \rightarrow new\_asAs(n0000, x00, vy4)\ \} \\ \mathcal{R} &= \emptyset \end{aligned}$$

and

$$\begin{array}{lcl} \mathcal{P} & = & \{\ new\_lT(S(n000), x) \to new\_lT(n000, x)\ \} \\ \mathcal{R} & = & \emptyset \end{array}$$

Both of these DP problems are finite, as can be shown by the Size-Change-Principle [TG05], for example. Therefore, we can conclude that the initial start term was lazy-terminating.

Please note that our approach to show lazy-termination is different from the approach presented in [PSS97]. There, a ***ParSplit***-node with a constructor head is required in all cycles. This is not the case for our approach, which allows to show lazy-termination of functions that only sometimes generate a constructor, but this case is always reached. Such a Haskell program is presented in the following example.

Example 9.12 (Not every cycle contains a constructor split)**.** *We want to show lazy termination of the start term* `h m n` *for the following Haskell program.*

```
data Nats   = Z   | S Nats
data List a = Nil | Cons a (List a)

h :: Nats -> Nats -> List Nats
h  Z    n = Cons n (h n (S n))
h (S m) n = h m n
```

As we can see, the function `h` *will decrement its first argument until it reaches* `Z`*, then it produces the constructor* `Cons` *and continues with the infinite list construction. Therefore, this is a lazy terminating program. The output that is generated by this function is similar to the output of the function* `from` *shown in example 2.1, i.e., for the start term* `h m n`*, the infinite list of natural numbers starting at* `n` *is generated.*

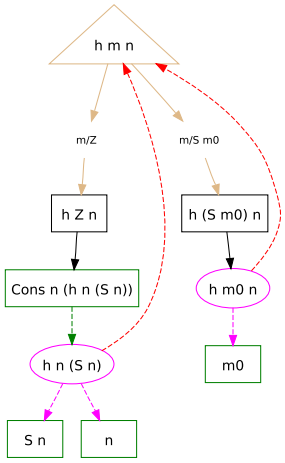

Figure 9.2: Termination Graph for `h m n`, where there is not a ***ParSplit***-node in every cycle

When looking at the Termination Graph with the start term `h m n`*, shown in figure 9.2, we see that not every cycle contains a* ***ParSplit****-node with a constructor symbol. Therefore, the approach presented in [PSS97] cannot prove this start term to be lazy terminating.*

In our approach, the start term `lazyTerminating x (h m n)` *must be shown* H*-terminating. This is reduced to proving finiteness of the following two DP problems:*

$$\begin{array}{lcll} \mathcal{P} & = & \{ & new\_asAs(S(x0), S(n0), vx3) \quad \rightarrow \quad new\_asAs(x0, n0, vx3) \quad \} \\ \mathcal{R} & = & \emptyset & \end{array}$$

and

$$\begin{array}{lcll} \mathcal{P} & = & \{ & new\_lT(S(x0), Z, n) \quad \rightarrow \quad new\_lT(x0, n, S(n)) \\ & & & new\_lT(S(x0), S(m0), n) \quad \rightarrow \quad new\_lT(S(x0), m0, n) \quad \} \\ \mathcal{R} & = & \emptyset & \end{array}$$

Here, the first DP problem corresponds to the lazy termination analysis of the generated argument `n` *of the list constructor* `Cons`*. The second DP problem is the more interesting DP problem, where the function* `h` *is contained in. Here, we see that for the argument Z, we have a decrease in the first argument, i.e., we have found a constructor head. In the second Dependency Pair, this argument remains unchanged. However, we see that the argument $S(m0)$ is decremented to $m0$. This corresponds to the second rule for the function* `h`*, where we decrement its first argument until we reach* `Z`*. Finiteness of these DP problems can be shown by the Size-Change processor [TG05], for example.*

As we have seen in this chapter, there are start terms that contribute to a finite computation, although these terms are not terminating. Such a function can be used as an argument of another function, to generate a continuous stream of items. However, non-termination is often not desired and is considered an error. Therefore, we will have a look at how we can prove start terms to be non-terminating in the next chapter.

Chapter 10

Non-Termination Analysis for Haskell

As it was said in the introduction, a programmer often makes small mistakes. In order to assist the programmer in finding such mistakes, it is of great help to identify programs that are definitely not terminating by giving a counterexample. If this is the case, there exists an infinite reduction sequence w.r.t. $\to_{\mathsf{H}}$, or a result having a functional type can be provided with enough H-terminating arguments such that an infinite reduction sequence exists. So the definition of *non-*H*-terminating* terms is the inverse of H-terminating terms, which was introduced in definition 2.7.

Definition 10.1 (Non-H-Termination)**.** *The set of* non-H-terminating *ground terms t is defined as the largest set, such that:*

(a) t starts an infinite reduction $t \to_{\mathsf{H}} \ldots$,

(b) if $t \to_{\mathsf{H}}^{*} (f\ t_1 \ldots t_n)$ for a defined function symbol f, $n < arity(f)$, and the term t' is H*-terminating, then $(f\ t_1 \ldots t_n\ t')$ is non-*H*-terminating, or*

*(c) if $t \to_{\mathsf{H}}^{*} (C\ t_1 \ldots t_n)$ for a constructor C, then a t_i is non-*H*-terminating.*

A term t is non-H-terminating*, iff $t\sigma$ is non-*H*-terminating for a substitution σ with* H*-terminating ground terms of correct types. Such a substitution may introduce new defined functions with arbitrary defining equations.*

Based on this definition of non-H-terminating terms, this chapter describes a method to automatically prove non-H-termination of start terms. First, we motivate the choice for allowing evaluations inside arguments of a constructor in an arbitrary order, instead of a left-to-right fashion. Second, we will take a look at the Termination Graphs and determine which DP paths can be used for Non-Termination analysis. Thereafter, we will solve the problem that class constraints can restrict types in such a way, that no type exists that could start an infinite evaluation of a ground term, which is required in definition 10.1. Last, we show that an infinite chain in one of these restricted DP problems implies a non-H-terminating ground instance of our start term.

10.1 Motivation for allowing evaluation inside any argument of a constructor

As could be seen in the definition of non-H-terminating terms, it is not required to evaluate terms from left to right. Why this was chosen shall be illustrated in the following example.

Example 10.2 (Non-Termination based on skipping evaluation position)**.** *Consider the following Haskell program:*

```
data D = Z | S D | C D D

f (S x) (S y) = C (f x (S (S y))) (f (S (S x)) y)
```

and the start term `f x y`*.*

When `data D` *is extended with* `deriving Show`*, then the evaluation of any instance of the start term will terminate with an error, as the derived instance of* `Show` *will evaluate the subterms of* `C` *from left to right, leading to a term where no* `S` *occurs on the first position of* `f` *anymore. However, the function* `f` *is not* H*-terminating, as shown in the following reduction sequence:*

$$
\begin{array}{lll}
t & = & \texttt{f (S (S Z)) (S Z)} \\
 & \to_{\mathsf{H}} (\pi = \epsilon) & \texttt{C (f (S Z) (S (S Z))) (f (S (S Z)) Z)} \\
 & \to_{\mathsf{H}} (\pi = 1) & \texttt{C (C (f Z (S (S (S Z)))) } \underbrace{\texttt{(f (S (S Z)) (S Z))}}_{=t} \texttt{)} \ \ldots
\end{array}
$$

As $\to_{\mathsf{H}}$ *does not require the leftmost position to be evaluated, the term* t *could be evaluated infinitely often.*

The reasoning, why evaluation on arbitrary positions is allowed, is that one could always generate a corresponding instance of `Show`*. For the above example, one possible instance would be the following:*

```
instance Show D where
        show Z = "Z"
        show (S x) = "S" ++ show x
        show (C x y) = "C" ++ show' x ++ show' y
                where   show' Z = "Z"
                        show' (S x) = "S" ++ show x
                        show' (C x y) = "C" ++ show y ++ show x
```

When the start term t *would be evaluated with this instance of* `Show`*, the evaluation of a Haskell interpreter would also loop infinitely often, outputting an infinite sequence of "*`C`*"s.*

As can be seen in the above example, the instance of the class `Show` determines whether a term has an infinite evaluation sequence. An instance for `Show` that always finds infinite sequences which result from skipping evaluation positions can be constructed for every start term with a corresponding Haskell program. This is shown in the following lemma.

Lemma 10.3 (`Show` instances for Non-Termination)**.** *For every Haskell Program* HP *and every start term* t*, there exists a function* $\texttt{show}_{HP}$*, such that the evaluation of* $\texttt{show}_{HP}$ `t` *will be infinite if there exists an infinite evaluation of the start term* t *w.r.t.* $\to_{\mathsf{H}}$ *and that is finite if* t *is* H*-terminating.*

Proof. Let HP be a Haskell program, and let TM be the Turing Machine representation of this program. TM always exists, since Haskell is a Turing-complete language.

TM can be encoded in Haskell again, forming a Haskell program which will be denoted by HP_{TM}. The program HP_{TM} is then imported into the original program HP, and the $\texttt{show}_{HP}$ function is generated as follows:

First, the term passed as argument is converted to the internal representation used in HP_{TM}. On this representation, the evaluation is then performed. Whenever an `error`-term is reached as an argument of a constructor, the $\texttt{show}_{HP}$ function replaces this term by the representation of a new constructor `Error` and continues the evaluation.

If the term t is H-terminating, then it holds that after a finite number of steps a normal form is reached. This normal form must also be reached in the program HP_{TM}. Therefore, none of the rules of HP_{TM} are applicable anymore and hence we have that $\texttt{show}_{HP}\ t$ has only a finite evaluation.

The other case is shown indirectly. Assume that all reduction sequences of a start term t w.r.t. $\to_{\mathsf{H}}$ are finite, but the evaluation of $\texttt{show}_{HP}\ t$ is infinite.

This implies the existence of a normal form $t\downarrow_{\mathsf{H}}$ of t w.r.t. $\to_{\mathsf{H}}$. This is due to the fact that $\to_{\mathsf{H}}$ is non-overlapping, as always the first rule is applied and the two cases in the definition of $\to_{\mathsf{H}}$ are divided into whether the term starts with a defined function symbol or a constructor. Therefore, the Turing Machine representation must reach its representation of $t\downarrow_{\mathsf{H}}$, which must also be a normal form in the program HP_{TM}. This contradicts the assumption that the evaluation of $\texttt{show}_{HP}\ t$ is infinite.

□

This means that evaluation could occur on any position of a constructor, which is why the definition of $\to_{\mathsf{H}}$ was chosen in the presented way. Please note that the `Show` class was only chosen for motivation, since there are predefined instances of `Show` for most predefined data types, which have a fixed order of evaluation. But for such terms, a context of the above form could still allow for an infinite evaluation.

10.2 Termination Graph for Non-Termination

The problem with Non-Termination Analysis for Haskell is that due to the lazy evaluation strategy, terms that are not terminating when called directly, are terminating when put in the right context. This can be seen in example 2.1 for the start term `take u (from m)`. Here, the subterm `from m` is not H-terminating, but the function `take` only considers a finite prefix of the generated infinite list.

In order to handle this problem, only so called "Top-Cycles" are considered. Intuitively, these are cycles in a Termination Graph which are not inside other defined functions. Therefore, these terms will always be evaluated completely, i.e., there is no context around them that might stop their evaluation.

Definition 10.4 (Top-Cycle)**.** *Let TG be a Termination Graph for a start term t.*

A cycle G' in TG is called a Top-Cycle*, if there exists a path p from t to a node on G', there is no node s in G' such that $s \in \mathsf{U}_{TG}$, for every child s' of an **Ins**-node in G' it holds that $s' \notin \mathsf{PU}_{TG}$, every path in G' does not*

*traverse children of an **Ins**-node, except for the instantiation edge, and there is no **Ins**-node in p.*

A Top-Cycle never traverses the subterm children of ***Ins***-nodes. This is, because it is not known whether a subterm will be evaluated. When we analyze H-termination, then we consider the worst case, i.e., we assume that this term has to be evaluated. This was used in section 7.1, where it allowed us to switch to minimal chains. However, for Non-Termination analysis the worst case is that this is an unneeded argument. Thus, we cannot deduce from a non-H-terminating argument that the start term was non-H-terminating.

As can be seen in the definition of Top-Cycles, there must not be a child of an ***Ins***-node that is a predecessor of a ***ParSplit***-node with variable head. This is, because otherwise we would have a free variable on a right-hand side of a Dependency Pair, which would have to be filtered away. But filtering is not complete, i.e., it does not guarantee that an infinite chain in the filtered DP problem is also an infinite chain in the unfiltered DP problem. An example for this is given below.

Example 10.5 (Filtering introduces infinite chains)**.** *Consider the following Haskell program with the start term* `f g x`*.*

```
data Nats = Z | S Nats

f :: (Nats -> Nats) -> Nats -> Bool
f g (S x) = True
f g  Z    = f g (S (g Z))
```

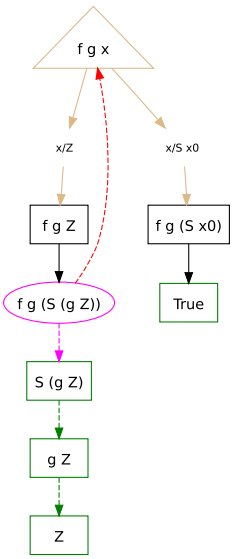

Figure 10.1: Termination Graph for `f g x`, showing a ***ParSplit***-node with variable head as child of an ***Ins***-node

As we can see in the Termination Graph for this start term, which is shown in figure 10.1, the term `g Z` *corresponds to a **ParSplit**-node with variable head*

in the Termination Graph. Therefore, a fresh variable would be introduced in the Dependency Pair, which would look as follows:

$$new\_f(g, Z) \to new\_f(g, y)$$

If we filter away the second argument of the function new_f, then we have an infinite chain. However, the original program is terminating, since the constructor S will be introduced on the second argument of `f`*, regardless of the result of* `g Z`*, making the first rule of* `f` *applicable.*

Furthermore, no ***Ins***-node must exist on the path leading to a Top-Cycle. This rules out generalizations, where the instantiated term was terminating, but there is a case in the generalized term such that it is non-terminating.

Example 10.6 (No ***Ins***-node must exist on the path to a Top-Cycle)**.** *We want to consider the start term* `le1 Z` *for the following Haskell program:*

```
le1 (S Z) = True
le1 x = le1 (S x)
```

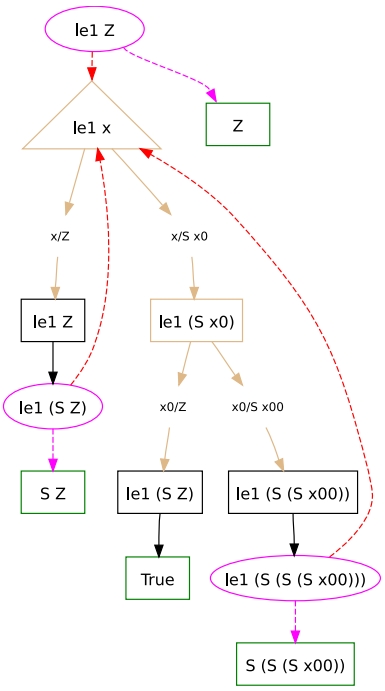

Figure 10.2: Termination Graph for start term `le1 Z`, illustrating why no ***Ins***-node must be on the path to a non-terminating SCC

The Termination Graph for this example is shown in figure 10.2. In this Termination Graph, an instantiation edge is drawn from `le1 Z` *to the term* `le1 x`*. If we called the cycle from* `le1 x` *to* `le1 (S (S (S x00)))` *labelled with $[x/S(S(x00))]$ a Top-Cycle, then an infinite chain exists in the created*

DP problem which is shown below, although the start term was H*-terminating.*

$$\begin{array}{lcll} \mathcal{P} & = & \{ & new\_le1(Z) \to new\_le1(S(Z)) \\ & & & new\_le1(S(S(x00))) \to new\_le1(S(S(S(x00)))) \quad \} \\ \mathcal{R} & = & \emptyset & \end{array}$$

For the shown DP problem, the following infinite chain exists:

$$\begin{array}{rcl} new\_le1(S(S(Z))) & \to_{\mathcal{P}} & new\_le1(S(S(S(Z)))) \\ & \to_{\mathcal{P}} & new\_le1(S(S(S(S(Z))))) \\ & \to_{\mathcal{P}} & \ldots \end{array}$$

The problem is that the ***Ins****-node in the path from the start term to the term* `le1 x` *corresponds to a generalization. In the start-term it was fixed that the evaluation must start with the argument* `Z`*. This information was dropped during the introduction of the new node* `le1 x`*, thus also terms with more than one* `S`*-constructor are considered.*

The restriction to paths without ***Ins***-nodes does not apply to DP paths on cycles. This is, because all information of an ***Ins***-node will be encoded into the right-hand side of a Dependency Pair. Thus, every following Dependency Pair in a chain must fulfill the requirements imposed by this right-hand side; therefore, the information is carried over to the next Dependency Pair.

Another restriction for Top-Cycles is that no ***ParSplit***-nodes with variable head must occur. This is due to the referential transparency required by Haskell functions and is shown by a counterexample.

Example 10.7 (Counterexample when ***ParSplit***-nodes with variable head are allowed)**.** *For the following Haskell program, the start term* `f x y` *shall be analyzed.*

```
f :: Bool -> (Bool -> Bool -> Bool) -> Bool
f False h = h (f True h) (error [])
f True  h = h (error []) (f False h)
```

As can be seen, the call to the function argument `h` *of* `f` *is called with an* `error`*-term on the second position in the first case, whereas the* `error`*-term occurs on the first position in the second case.*

If one disregards the `error`*, the created DP problem would be*

$$\begin{array}{lcll} \mathcal{P} & = & \{ & new\_f(False, h) \to new\_f(True, h) \\ & & & new\_f(True, h) \to new\_f(False, h) \quad \} \\ \mathcal{R} & = & \emptyset & \end{array}$$

This is an infinite DP problem, since an infinite chain exists.

When a ground start term is entered, the argument `h` *must be set to a single function. Thus, in order to require an infinite evaluation, the function that is inserted for* `h` *must demand a constructor on the second argument, as this is required to get the recursive call from* `f False h` *to* `f True h`*. On the other hand, the function* `h` *must evaluate its first argument, since this is required for the other recursive call of* `f`*. Therefore, since this Haskell function is fixed, it must evaluate an* `error`*-term eventually. Thus, there exists no infinite Haskell evaluation sequence, since every evaluation will stop with an error.*

Please note that ***ParSplit***-nodes with variable head are allowed on the path leading to an SCC. This is, because this path has to be evaluated only once; therefore, an instance that does not run into `error`-cases exists.

10.3 Basic Instances and DP problems

In the definition of non-H-terminating terms, a ground instance of a start term containing variables is required, such that it is non-H-terminating. Thus, we also face the problem to decide whether for a given start term a ground instance exists. This is different in the case of termination analysis, where we would show the absence of infinite reductions for start terms that actually do not exist. An example for this is given below.

Example 10.8 (Non-existent infinite reduction)**.** *Non-termination of the following Haskell program shall be analyzed for the start term* `f x`*.*

```
class A a where
    f :: a -> Bool

class B a where

instance B a => A [a] where
    f xs = f xs

instance A Bool where
    f x = x
```

For the given start term, the class constraint `A a` *exists, since the class member* `f` *is being used. Obviously, an infinite evaluation exists for the list instance of the class* `A`*. But no ground instance of the start term exists, since then an instance of the class* `B` *would have to exist for the elements of the list.*

The variables inside a term stand for arbitrary H-terminating terms, by definition of (non-)H-termination. Because of polymorphic types in Haskell, these do not impose a problem, since one can always instantiate any term with `undefined`. For type variables without constraints, there is no problem, since these type variables may be instantiated with any type. However, when class constraints exist for a type variable, then a type must be used in the start term, such that all class constraints are satisfied. Therefore, we have to guarantee the existence of a *basic instance*, such that it is non-H-terminating. Please note that this was already implicit in the definition of non-H-termination, as we required the existence of a ground substitution with correct types, which means that all class constraints can be satisfied. This is due to the fact that the Haskell type checker does not allow unresolved constraints in start terms.

Example 10.9 (Basic Instances)**.** *We consider the classes and instances presented in example 2.2.*

The class constraint `Addition (List Nats)` *is a basic instance, because we have the instance* `Addition Nats` *which enables us to build the instance* `Addition (List Nats)`*, as the requirement for the list instance that the type of the elements must also be in the type class* `Addition` *is met.*

For the class constraint `Addition a` *it holds that this is not a basic instance, because it cannot be decided. Therefore, it is not allowed to use such a constraint when entering a term into a Haskell interpreter. However, this class constraint has a basic instance, if we replace the type variable* `a` *by the type* `Nats`*, for example.*

When we consider the class constraint `Multiplication a`*, then this is neither a basic instance, nor does it have a basic instance. This holds, because there are no instances for this type class in the program.*

To formally define a basic instance, we have to check whether all class constraints can be resolved. This check is performed by the function reduce, which was introduced in definition 4.2.

Definition 10.10 (Basic Instance)**.** *For a Haskell program HP, a term $\underline{cs} \Rightarrow t$ is a* basic instance*, iff* $\text{reduce}(\underline{cs}) = \emptyset$*.*

A term $\underline{cs} \Rightarrow t$ is said to have a basic instance, iff a type substitution σ exists, such that $\text{reduce}(\underline{cs}\sigma) = \emptyset$*.*

A basic instance can always be instantiated further to a ground instance, since every subterm has either a fixed- or an unrestricted type. Therefore, such terms can be replaced by as many constructors as needed (for fixed types), and then all further subterms thereof can be replaced by, e.g., `undefined`.

The existence of basic instances that allow infinite reductions is checked by two means. First, it is ensured for every Top-Cycle that it is reachable from a basic instance of the start term. Then, the instance information for every DP path is encoded into the terms that are created. Thus, we distinguish between the restricted type variables that occur inside an SCC that was built from Top-Cycles and those which do not occur in the current SCC. These will be called *dependent* and *independent* type constraints, respectively.

Example 10.11 (Dependent and Independent Class Constraints)**.** *The following Haskell program shall be considered for the start term* `f x (g y)`*.*

```
data Nats = Z | S Nats

class A a
class B a

instance B Nats

f :: A a => a -> b -> b
f x y = y

g :: B a => a -> a
g x = g x
```

For the given start term, we construct the Termination Graph shown in figure 10.3. As can be seen, we have a Top-Cycle in which the term `g y` *loops back to itself. In this Top-Cycle we have the class constraint* `B b`*, which is therefore called a* dependent *class constraint. This class constraint has a basic instance, because of the instance* `B Nats`*.*

However, the node containing the start term has two class constraints: On the one hand the dependent class constraint `B b` *and on the other hand the class constraint* `A a`*, which is dropped due to the implementation of the function* `f`*. This latter class constraint* `A a` *is called an* independent *class constraint. This is, because it is not linked to the SCC in any way. It should be noted that this class constraint does not have a basic instance, because no instances exist for*

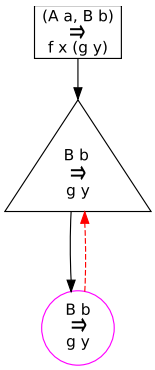

Figure 10.3: Termination Graph for `f x (g y)` showing dependent and independent class constraints

the class `A`*. Therefore, the given start term does not have any instances and hence it is* H*-terminating.*

Next, the notion of dependent and independent class constraints shall be defined formally.

Definition 10.12 (Dependent and Independent Class Constraints)**.** *Let TG be a Termination Graph, let G' be the subgraph of all Top-Cycles for an SCC of TG, and let $\underline{cs}_{G'}$ be the set of class constraints that occur in G'.*

We say that a class constraint $C\ \rho$ is independent *of G', iff $\mathcal{V}(\rho) \cap \mathcal{V}(\underline{cs}_{G'}) = \emptyset$. Otherwise, $C\ \rho$ is called a* dependent *class constraint.*

For the independent class constraints, any basic instance of these class constraints can be used, since they are not used recursively. Thus, a simple search for a basic instance suffices. For the dependent class constraints, it is necessary to also consider the recursive structure in the SCC, therefore the class constraints are encoded into the DP problem, as stated above. Here, we do not only have to consider the class constraints that are present at the ***Ins***-node, but we must also consider all dependent class constraints on a DP path. Why this is the case is illustrated in the example below.

Example 10.13 (Collection of class constraints)**.** *We want to analyze the start term* `f x y` *for the following Haskell program.*

```
data Nats = Z | S Nats
data A a => D a = J | W a

class A a
instance A Nats

f :: D a -> D b -> Nats
f J y = Z
f x y = f y y
```

For this start term, the Termination Graph shown in figure 10.4 is built which contains exactly one DP Path from node A to node E. For this DP path,

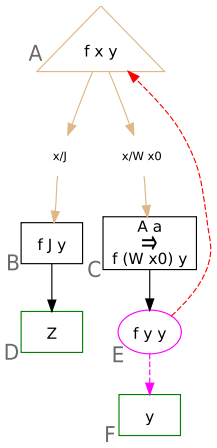

Figure 10.4: Termination Graph for `f x y`, showing the collection of class constraints

we see that at node C a new class constraint is introduced. This happens, because this class constraint is bound to the type variable a in the definition of data constructor `W`*. When now the* ***Eval****-step in node C results in node E, this class constraint does no longer exist, since in node E the type* `a` *does not occur anymore.*

When now the class constraints are collected, this class constraint must be considered. This is, because in order to run into this case, a ground term with the data constructor `W` *must be used. Then, the class constraint* `A a` *of this data constructor must also be fulfilled.*

To collect the dependent constraints of a path, the function collectCCs is used. This function is defined relative to a subgraph G of TG which determines which class constraints shall be collected, i.e., which class constraints are currently dependent class constraints.

Definition 10.14 (collectCCs)**.** *Let TG be a Termination Graph, let t' to t be a path in TG not crossing instantiation edges, and let G be a subgraph of TG.*

We define the set $\text{collectCCs}_G(t', t)$ *recursively along the path, by looking at a current node* $\underline{cs_s} \Rightarrow s$*, where we start with* $\underline{cs_s} \Rightarrow s = t'$*.*

If we have $\underline{cs_s} \Rightarrow s = t$*, then* $\text{collectCCs}_G(\underline{cs_s} \Rightarrow s, t) = \underline{cs_s}$*.*

Otherwise, we perform case distinction based on the expansion rule applied to the current node $\underline{cs_s} \Rightarrow s$*:*

***Eval*:** *If the child of the current node* $\underline{cs_s} \Rightarrow s$ *is* s'*, then* $\text{collectCCs}_G(\underline{cs_s} \Rightarrow s, t) = \underline{cs_s} \cup \text{collectCCs}_G(s', t)$*.*

VarExp*:** *In case* $\underline{cs_s} \Rightarrow s$ *is a* ***VarExp*-node with the child* $\underline{cs_s} \Rightarrow (s\ x)$*, then we have* $\text{collectCCs}_G(\underline{cs_s} \Rightarrow s, t) = \text{collectCCs}_G(\underline{cs_s} \Rightarrow (s\ x), t)$*.*

Case*:** *For a* ***Case*-node, only the child s' is considered further which is on the path from* $\underline{cs_s} \Rightarrow s$ *to t, i.e.,* $\text{collectCCs}_G(\underline{cs_s} \Rightarrow s, t) = \text{collectCCs}_G(s', t)$*.*

TyCase*:** *When* $\underline{cs_s} \Rightarrow s$ *is a* ***TyCase*-node, then those class constraints that contain the type variable a on which the type case was performed are removed. Let s' be the child that lies on the path from* $\underline{cs_s} \Rightarrow s$ *to t. Then*

$\text{collectCCs}_G(\underline{cs}_s \Rightarrow s, t) = \text{collectCCs}_G(s', t)$. *Here, the class constraints are removed in the child node s', due to the construction of the Termination Graph.*

***ParSplit*:** *At a **ParSplit**-node, only those class constraints are collected, where the children are in G. Let $\{\underline{cs}_1 \Rightarrow s_1, \ldots, \underline{cs}_n \Rightarrow s_n\} \subseteq \mathbf{ch}(\underline{cs}_s \Rightarrow s)$ be the children of $\underline{cs}_s \Rightarrow s$ that are in G, and let $\underline{cs}_j \Rightarrow s_j$ be the child on the path from $\underline{cs}_s \Rightarrow s$ to t. Then* $\text{collectCCs}_G(\underline{cs}_s \Rightarrow s, t) = \bigcup_{1 \leq i \leq n} \underline{cs}_i \cup \text{collectCCs}_G(\underline{cs}_j \Rightarrow s_j, t)$.

***Ins*:** *Since $\underline{cs}_s \Rightarrow s \neq t$, there must be a child of the **Ins**-node not connected via the instantiation edge, such that it is on the path to t, since the path does not contain instantiation edges. Let s' be this child on the path to t, then* $\text{collectCCs}_G(\underline{cs}_s \Rightarrow s, t) = \text{collectCCs}_G(s', t)$.

For the collected class constraints, it holds that a basic instance thereof can propagated backwards along paths in a Termination Graph. For this to work, the independent class constraints that are dropped at ***ParSplit***-nodes must also have a basic instance.

Lemma 10.15 (Basic instances exist along paths)**.** *Let TG be a Termination Graph, let G be a subgraph of TG, and let $\underline{cs}_{s'} \Rightarrow s'$ to $\underline{cs}_s \Rightarrow s$ be a path in TG which does not traverse instantiation edges, where a basic instance exists for* $\text{collectCCs}_G(\underline{cs}_{s'} \Rightarrow s', \underline{cs}_s \Rightarrow s)$ *and for all independent constraints of **ParSplit**-nodes.*

Then a basic instance exists for $\underline{cs}_{s'} \Rightarrow s'$.

Proof. The lemma is shown inductively over the length of the path from s' to s. This is a well-founded order, since by disallowing instantiation edges, all paths in TG have only finite length.

If the path from $\underline{cs}_{s'} \Rightarrow s'$ to $\underline{cs}_s \Rightarrow s$ has length zero, then a basic instance for $\underline{cs}_{s'} \Rightarrow s'$ exists by requirements on $\underline{cs}_s \Rightarrow s$.

Otherwise, a predecessor $\underline{cs}_t \Rightarrow t$ of $\underline{cs}_s \Rightarrow s$ exists. Case analysis is performed based on the expansion rule applied to $\underline{cs}_t \Rightarrow t$:

Eval*:** Since at an ***Eval-node, all class constraints of the ***Eval***-node are pushed downwards, these class constraints have a basic instance. Formally, this is because $\underline{cs}_t \subseteq \text{collectCCs}_G(\underline{cs}_{s'} \Rightarrow s', \underline{cs}_s \Rightarrow s)$ and a basic instance exists for $\underline{cs}_s$. Thus, for the path from $\underline{cs}_{s'} \Rightarrow s'$ to $\underline{cs}_t \Rightarrow t$, the claim inductively holds, since this path is shorter than the path from $\underline{cs}_{s'} \Rightarrow s'$ to $\underline{cs}_s \Rightarrow s$.

Case*:** For a ***Case-node, $\underline{cs}_t \subseteq \underline{cs}_s \subseteq \text{collectCCs}_G(\underline{cs}_{s'} \Rightarrow s', \underline{cs}_s \Rightarrow s)$ holds. Hence, a basic instance exists for $\underline{cs}_t \Rightarrow t$. Based on this basic instance, the induction hypothesis proves the claim.

VarExp*:** If $\underline{cs}_t \Rightarrow t$ is a ***VarExp-node, then its only child is $\underline{cs}_t \Rightarrow (t\ x) = \underline{cs}_s \Rightarrow s$ for a fresh variable x. Therefore, a basic instance exists for $\underline{cs}_t$ and therefore also for $\underline{cs}_{s'}$.

***Ins*:** Because the path does not traverse instantiation edges, the child $\underline{cs}_s \Rightarrow s$ must be a subterm of the node $\underline{cs}_t \Rightarrow t$. Because $\underline{cs}_s$ has a basic instance, and because $\underline{cs}_t \subseteq \underline{cs}_s \subseteq \text{collectCCs}_G(\underline{cs}_{s'} \Rightarrow s', \underline{cs}_s \Rightarrow s)$, we also have a basic instance of $\underline{cs}_t$ and thus inductively also of $\underline{cs}_{s'}$.

TyCase*:** When $\underline{cs_t} \Rightarrow t$ is a ***TyCase-node, then we have that for the child node $\underline{cs_s} \Rightarrow s = (\underline{cs_t} \Rightarrow t)\delta$ on the path, the class constraints of this node have the form $\underline{cs}_s = (\underline{cs}_t \setminus \{C_1\ \rho_1[a], \ldots, C_m\ \rho_m[a]\}) \cup \{C_{m+1}\ \rho_{m+1}, \ldots, C_n\ \rho_n\}$, where $\delta = [a/(T\ a_1 \ldots a_k)]$, $a, a_1, \ldots, a_k$ are type variables, T is a type constructor, $C_1\ \rho_1, \ldots, C_n\ \rho_n$ are class constraints, and $m \leq n$. This especially means that for the constraints $\{C_{m+1}\ \rho_{m+1}, \ldots, C_n\ \rho_n\}$ basic instances exist. The constraints $\{C_1\ \rho_1[a], \ldots, C_m\ \rho_m[a]\}$ could only be removed because of instances for the types $\rho_i[a/(T\ a_1 \ldots a_k)]$. Since all their preconditions are contained in $\{C_{m+1}\ \rho_{m+1}, \ldots, C_n\ \rho_n\}$, basic instances also exist for $\{C_1\ \rho_1[a], \ldots, C_m\ \rho_m[a]\}$, which inductively proves the claim, since we have a shorter path.

ParSplit*:** In case $\underline{cs_t} \Rightarrow t$ is a ***ParSplit-node, we have to distinguish between the dependent and independent class constraints of this node. Let $\underline{cs}_t = \{C_1\ \rho_1, \ldots, C_m\ \rho_m\} \cup \{C_{m+1}\ \rho_{m+1}, \ldots, C_n\ \rho_n\}$, where $m \leq n$, for all $1 \leq i \leq m$ we have that $C_i\ \rho_i$ is an independent class constraint, and for all $m < i \leq n$, $C_i\ \rho_i$ is a dependent class constraint.

For the independent class constraints, a basic instance exists by assumption, i.e., a substitution σ' exists, such that $\{C_1\ \rho_1, \ldots, C_m\ \rho_m\}\sigma'$ is a basic instance.

For the dependent class constraints, we have $\{C_{m+1}\ \rho_{m+1}, \ldots, C_n\ \rho_n\} \subseteq \underline{cs}_s \subseteq \text{collectCCs}_G(\underline{cs_{s'}} \Rightarrow s', \underline{cs_s} \Rightarrow s)$. Thus, the induction hypothesis gives us the existence of a substitution σ'', such that $\{C_{m+1}\ \rho_{m+1}, \ldots, C_n\ \rho_n\}\sigma''$ is a basic instance. Since the type variables of the dependent- and the type variables of the independent class constraints are disjoint, for the substitution $\sigma = \sigma'\sigma''$ it holds that $\underline{cs}_t\sigma$ is a basic instance. Therefore, the claim inductively holds, since the path to $\underline{cs_t} \Rightarrow t$ is shorter.

□

Using the collected class constraints, we can build the DP problems that will be used for the Non-Termination analysis of Haskell. For these, we want the property that an infinite chain corresponds to an infinite Haskell evaluation of a ground instance of the start term, which is why our idea is to encode this property into the DP problems. In order to be able to resolve class constraints, rules for the reduction of class constraints must also be added. Since we do not want to add rules for every instance in the program, we define the *usable instances* as the set of instances that might possibly be used.

Definition 10.16 (Usable Instances)**.** *Let HP be a Haskell program, let $\underline{cs}$ be a set of class constraints.*

The set usableIns($\underline{cs}$) *of* usable instances *for $\underline{cs}$ is defined as the smallest set such that*

- *For every instance* $I =$ `(C`$_1$ `a`$_{i_1}$`, ..., C`$_m$ `a`$_{i_m}$`) => C (T a`$_1$ `... a`$_n$`)` *in HP, where a class constraint* `C` ρ *exists in $\underline{cs}$, we have* $I \in$ usableIns($\underline{cs}$).
- *If* `(C`$_1$ `a`$_{i_1}$`, ..., C`$_m$ `a`$_{i_m}$`) => C (T a`$_1$ `... a`$_n$`)` $\in$ usableIns($\underline{cs}$), *then* usableIns($\{$`C`$_j$ `a`$_{i_j}\}$) $\subseteq$ usableIns($\underline{cs}$) *for every* $1 \leq j \leq m$.

So for example, if we regard the example 2.2 and the set of class constraints $\underline{cs}_1 = \{\texttt{Multiplication a}\}$, we have that $\text{usableIns}(\underline{cs}_1) = \emptyset$, because no instance for `Multiplication` exists in the example program. However, for the set of class constraints $\underline{cs}_2 = \{\texttt{Addition a}\}$, we have $\text{usableIns}(\underline{cs}_2) = \{\texttt{Addition Nats}, \texttt{Addition a => Addition (List a)}\}$, as these are the instances that exist for `Addition`, and the only further class constraint is for the class `Addition`, whose instances are already included.

To encode the check for existence of basic instances for a set of class constraints into the terms, the function introCCs is defined. For a set of class constraints, it appends a term starting with the defined symbol ccCheck and having a list of these class constraints as arguments to the defined function symbols in a given term. The idea is to define the rules for ccCheck in such a way that they represent the reduction of class constraints.

Definition 10.17 (introCCs)**.** *Let $\underline{cs} = \{C_1\ \rho_1, \ldots, C_m\ \rho_m\}$ be a set of class constraints, let t be a term.*

The term $\text{introCCs}_{\underline{cs}}(t)$ *is defined as:*

- *If $t = f(t_1, \ldots, t_n)$, where f is a defined symbol and $arity(f) = n$, then*

$$\begin{aligned}\text{introCCs}_{\underline{cs}}(t) = f(&\text{introCCs}_{\underline{cs}}(t_1), \ldots, \text{introCCs}_{\underline{cs}}(t_n),\\ &\text{ccCheck}([C_1(\rho_1), \ldots, C_m(\rho_m)]))\end{aligned}$$

- *If $t = c(t_1, \ldots, t_n)$, where c is a constructor or a variable, then*

$$\text{introCCs}_{\underline{cs}}(t) = c(\text{introCCs}_{\underline{cs}}(t_1), \ldots, \text{introCCs}_{\underline{cs}}(t_n))$$

Please note that the list representation above is only used for readability. What is really to be produced is a term containing only the constructors `:` *and* `[]`*. Another thing to note is that the types $\rho_1, \ldots, \rho_m$ must be represented in applicative form, as already discussed in chapter 6.*

The following example shall illustrate, how introCCs adds class constraints to the defined symbols in the resulting DP problems.

Example 10.18 (introCCs)**.** *Consider example 2.2 again. We want to consider the case, where the set of class constraints $\underline{cs} = \{\texttt{Addition a}\}$ is to be inserted into the term* `Cons (plus x y) (plus xs ys)`*, which appears on the right-hand side of the instance for lists. Let the type variable be a, then the term with the introduced type variables is constructed as shown in the following.*

$$\begin{aligned}&\text{introCCs}_{\underline{cs}}(Cons(plus(x, y, a), plus(xs, ys, a)))\\ &\quad= Cons(\text{introCCs}_{\underline{cs}}(plus(x, y, a)), \text{introCCs}_{\underline{cs}}(plus(xs, ys, a)))\\ &\quad= Cons(\\ &\qquad\qquad plus(x, y, a, \text{ccCheck}(\texttt{Addition}(a) : [])),\\ &\qquad\qquad plus(xs, ys, a, \text{ccCheck}(\texttt{Addition}(a) : []))\\ &\qquad)\end{aligned}$$

The idea is that before a chain can be built using the result of this rule application, one has to show first that in the above example, an instance of `Addition` a exists. How this is guaranteed is shown later, when the DP problems for Non-Termination analysis are defined.

The above example shows that we have to provide means to check whether a set of class constraints has a basic instance. This is, because there might be rules that allow for a chain, but the paths these rules represent cannot be entered, because the corresponding instances do not exist. Therefore, we first define the function that reads rules suitable for Non-Termination analysis.

Here, also the rules for ccCheck are defined. These mimic the reduction of class constraints based on the instance definitions of the usable instances.

Definition 10.19 (Rules $\mathbf{rl}^{NT}$ for Non-Termination analysis)**.** *Let TG be a Termination Graph, let s be a node in it.*

We define $\mathbf{rl}^{NT}(s)$ *to be the smallest set such that*

- *If* $f(s_1, \ldots, s_n)\sigma_1 \ldots \sigma_m \to r \in \mathbf{rl}(s)$ *and* $\underline{cs} = \text{collectCCs}_{TG}(t, r)$*, then* $f(s_1, \ldots, s_n, [])\sigma_1 \ldots \sigma_m \to \text{introCCs}_{\underline{cs}}(r) \in \mathbf{rl}^{NT}(s)$*,*

  where for every ***ParSplit****-node u on the path from* $f(s_1, \ldots, s_n)$ *to r a basic instance of the independent class constraints of u exists,*

- *If* $q \in \mathbf{con}(s)$*, then* $\mathbf{rl}^{NT}(q) \subseteq \mathbf{rl}^{NT}(s)$*,*

- $\text{ccCheck}([]) \to [] \in \mathbf{rl}^{NT}(s)$*, and*

- *If for an instance* $I =$ `(C`$_1$ `a`$_{i_1}$`, ..., C`$_m$ `a`$_{i_m}$`) => C (T a`$_1$ `... a`$_n$`)`*, it holds that* $I \in \text{usableIns}(\text{collectCCs}_{TG}(t, r))$ *for a rule* $t \to r \in \mathbf{rl}(s)$*, then* $\text{ccCheck}(C(T(a_1, \ldots, a_n)) : ccs) \to \text{ccCheck}([C_1(a_{i_1}), \ldots, C_m(a_{i_m})] +\!+ ccs) \in \mathbf{rl}^{NT}(s)$*.*

Here again, the notation for the lists and the append operator are only used to increase readability. Really, the terms are built using the constructors `:` *and* `[]` *only, as well.*

An example shall demonstrate, how the introduced terms starting with ccCheck guarantee the existence of basic instances for those rules.

Example 10.20 (ccCheck)**.** *Consider again example 2.2, containing the rule* `plus (Cons x xs) (Cons y ys) = Cons (plus x y) (plus xs ys)`*.*

For this rule, we have the collected class constraints $\underline{cs} = \{$`Addition a`$\}$ *which have usable instances for the data type* `Nats` *and for the data type* `List a`*. Thus, we would have the following rules for* ccCheck*:*

$$\begin{array}{lcl} \text{ccCheck}([]) & \to & [] \\ \text{ccCheck}(\texttt{Addition}(\texttt{Nats}) : ccs) & \to & \text{ccCheck}(ccs) \\ \text{ccCheck}(\texttt{Addition}(\texttt{app}(\texttt{List}, a)) : ccs) & \to & \text{ccCheck}(\texttt{Addition}(a) : ccs) \end{array}$$

If we have a ground term, where the type of a is instantiated with `List Nats`*, then the subterm* $\text{ccCheck}(\texttt{Addition}(\texttt{app}(\texttt{List},\texttt{Nats})) : [])$ *would be contained. The above rules can reduce this term to the empty list, which means that a basic instance exists for this class constraint.*

$$\begin{array}{lcl} \text{ccCheck}(\texttt{Addition}(\texttt{app}(\texttt{List},\texttt{Nats})) : []) & \to & \text{ccCheck}(\texttt{Addition}(\texttt{Nats}) : []) \\ & \to & \text{ccCheck}([]) \\ & \to & [] \end{array}$$

The above definition of the rules guarantees that a rule can only be evaluated, if the corresponding instance exists. For this to work also for the next rule that might be applicable, it must be the case that any term starting with the defined function ccCheck is evaluated as far as possible. How this is achieved can be seen in the next definition, when the DP problems containing these rules are constructed. Furthermore, these must also ensure that the dependent class constraints on their DP paths exist. This is done by encoding the class constraints into the Dependency Pairs, too.

Definition 10.21 (DP problems $\mathbf{dpRen}^{NT}_{G'}$ for Non-Termination analysis)**.** *Let TG be a Termination Graph, let G' be the subgraph of all Top-Cycles of a SCC in TG.*

We define $\mathbf{dpRen}^{NT}_{G'} = (\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathtt{a})$, where $\mathcal{P}$ and $\mathcal{R}$ are the smallest sets, such that:

- *If a DP path exists from s to t in G' labelled with substitutions $\sigma_1, \ldots, \sigma_m$, where $\mathrm{bR}_{TG}(s) = f(s_1, \ldots, s_n)$, $\mathrm{bR}_{TG}(\mathbf{ev}(t)) = g(t_1, \ldots, t_n)$, and where $\mathrm{collectCCs}_{G'}(s,t) = \underline{cs}$, then*

  $f(s_1, \ldots, s_n, [])\sigma_1 \ldots \sigma_m \to \mathrm{introCCs}_{\underline{cs}}(g(t_1, \ldots, t_n)) \in \mathcal{P}$,

  *where for every **ParSplit**-node on the DP path, a basic instance of the independent class constraints exists,*

- $\mathbf{rl}^{NT}(q) \subseteq \mathcal{R}$, *for all* $q \in \mathbf{con}(t_1) \cup \ldots \cup \mathbf{con}(t_n)$,

- *If for an instance* $I =$ `(C`$_1$ `a`$_{i_1}$`, ..., C`$_m$ `a`$_{i_m}$`) => C (T a`$_1$ `... a`$_n$`)`, *it holds that* $I \in \mathrm{usableIns}(\mathrm{collectCCs}_{G'}(s,t))$ *for a DP path from s to t in G', then*

  $\mathrm{ccCheck}(C(T(a_1, \ldots, a_n)) : ccs) \to \mathrm{ccCheck}([C_1(a_{i_1}), \ldots, C_m(a_{i_m})] +\!+ ccs) \in \mathcal{R}$.

Here, the list representation above is again only used to ease readability. Really, the representation with the constructors `:` *and* `[]` *is built.*

In the returned DP problem, the set $\mathcal{Q}$ is set to be $\{\mathrm{ccCheck}(x) \to \ldots\}$, *thereby forcing evaluation of* ccCheck *to a normal form.*

By including the defined function ccCheck in the set $\mathcal{Q}$, we have that for every reduction, it must first be checked whether the class constraints are fulfilled. This is, because every term starting with ccCheck must be reduced to the empty list in order to not be reducible w.r.t. $\mathcal{Q}$. Thereby, only such reductions are valid, where a basic instance of the class constraints exists. Why this is the case shall be illustrated in the following example.

Example 10.22 (Why $\mathcal{Q}$ contains $\mathrm{ccCheck}(x)$)**.** *We want to analyze non-*H*-termination of the start term* `h x y z` *for the following Haskell program:*

```
class A a
data Nats = Z | S Nats
data A a => D a = J | W a

g :: D a -> D b -> Nats
g J y = Z
g x y = S (g y y)
```

```
h :: Nats -> D a -> D b -> Nats
h (S n) y z = h (g y z) y z
```

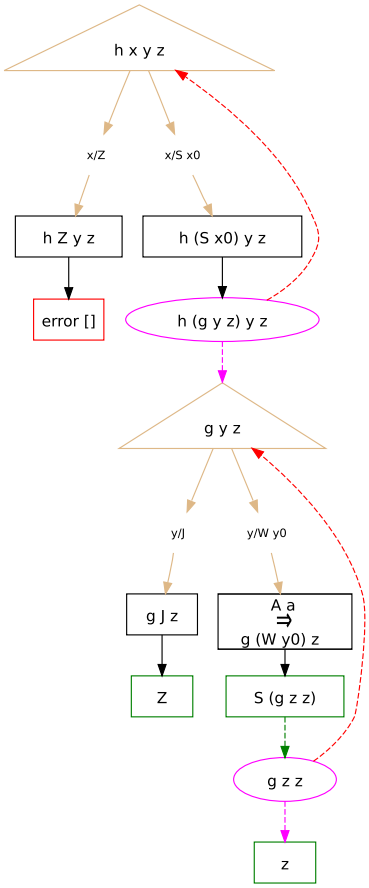

Figure 10.5: Termination Graph for start term `h x y z`, which shows why $\mathcal{Q}$ must contain ccCheck(x)

From the Termination Graph for this start term, which is shown in figure 10.5, the following DP problem for Non-Termination analysis is generated.

$$
\begin{array}{lll}
\mathcal{P} & = & \{ \\
 & & \quad new\_h(S(n), y, z, a, b, []) \rightarrow new\_h(new\_g(y, z, a, b, []), y, z, []) \\
 & & \}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{R} & = & \{ \\
 & & \quad new\_g(J, y, a, b, []) \rightarrow Z \\
 & & \quad new\_g(W(x0), y, a, b, []) \rightarrow S(new\_g(y, y, b, b, \text{ccCheck}(A(a) : []))) \\
 & & \quad \text{ccCheck}([]) \rightarrow [] \\
 & & \}
\end{array}
$$

In the above DP problem, there are no further rules for ccCheck *since no instances of the class* `A` *exist. But if* $\mathcal{Q} = \emptyset$*, then an infinite chain could be built:*

$$\begin{aligned} & new\_h(S(n), W(y), z, a, b, []) \\ \to_{\mathcal{P}} \quad & new\_h(new\_g(W(y), z, a, b, []), W(y), z, a, b, []) \\ \to_{\mathcal{R}} \quad & new\_h(S(new\_g(z, z, b, b, \text{ccCheck}(A(a) : []))), W(y), z, a, b, []) \\ \to_{\mathcal{P}} \quad & new\_h(new\_g(W(y), z, a, b, []), W(y), z, a, b, []) \\ \to_{\mathcal{R}} \quad & \dots \end{aligned}$$

As can be seen, the term in the second line and the last term are equal, which then results in an infinite chain.

This infinite chain only exists, because the reduction of g needs to be done only once, which directly generates the required constructor S. However, this rule cannot be applied, since the class constraint $(A\ a)$ *of the constructor W cannot be fulfilled. Hence, this class constraint must be reduced fully first, which is why* $\mathcal{Q}$ *contains the left-hand side* $\text{ccCheck}(x)$*. Then, the evaluation of the outer redex starting with new_h is blocked until the term* $\text{ccCheck}(A(a) : [])$ *could be evaluated further to a term not reducible w.r.t.* $\mathcal{Q}$*.*

For a set of class constraints $\underline{cs}$, we will write $\text{ccCheck}(\underline{cs})$ in the following, where really the list representation as shown in definition 10.17 is meant. Then a reduction of $\text{ccCheck}(\underline{cs})$ to the empty list constructor [] shows that a basic instance exists for the class constraints $\underline{cs}$, which is shown in the following lemma.

Lemma 10.23 (ccCheck implies existence of a basic instance)**.** *Let TG be a Termination Graph, let* G' *be an SCC in it consisting only of Top-Cycles, let* $\mathbf{dpRen}^{NT}_{G'} = (\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathtt{a})$*, and let* $\underline{cs}$ *be a set of class constraints.*

If a reduction $\text{ccCheck}(\underline{cs}) \to^*_{\mathcal{R}} []$ *exists, then* $\underline{cs}$ *is a basic instance.*

Proof. Assume that $\underline{cs}$ is not a basic instance, but $\text{ccCheck}(\underline{cs}) \to^*_{\mathcal{R}} []$.

Since $\underline{cs}$ is not a basic instance, there must be a class constraint $C\ \rho$ in the reduce-steps of $\underline{cs}$, such that no instance exists for it, i.e., for all instances $cx \Rightarrow C\ (T\ a_1 \dots a_n)$ in the Haskell program, $(T\ a_1 \dots a_n)$ does not match ρ. By construction, no rule can match the term $\text{ccCheck}(C(\rho) : ccs)$, i.e. this is a normal form. Thus, since $\mathcal{R}$ is non-overlapping, the reduction $\text{ccCheck}(\underline{cs}) \to^*_{\mathcal{R}} []$ cannot exist, which contradicts the assumption.

□

These reductions are part of any chain, since every term starting with ccCheck must first be reduced to the empty list. Thus, we will now show that the existence of a chain ensures the existence of basic instances for all used terms.

Lemma 10.24 (Chains imply existence of basic instances)**.** *Let TG be a Termination Graph, let* G' *be an SCC in TG consisting only of Top-Cycles, let* $\mathbf{dpRen}^{NT}_{G'} = (\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathtt{a})$ *be the DP problem created from* G'*, and let basic instances exist for all independent class constraints of* G'*, and for all independent class constraints that occur on rule paths used to construct* $\mathcal{R}$*.*

If $s_1 \to t_1, s_2 \to t_2 \in \mathcal{P}$ *is a chain, then a substitution* σ *exists, such that* $s_1\sigma$ *is a basic instance.*

Proof. Since $s_1 \to t_1, s_2 \to t_2$ is a chain, it must hold that substitutions σ_1 and σ_2 exist, such that $t_1\sigma_1 \to^*_{\mathcal{R}} s_2\sigma_2$, i.e., especially for the subterm $\text{ccCheck}(\underline{cs}_{t_1}) \lhd t_1$, it must hold that $\text{ccCheck}(\underline{cs}_{t_1}\sigma_1) \to^*_{\mathcal{R}} []$, and therefore, by lemma 10.23, the class constraints of $t_1\sigma_1$ are a basic instance.

Furthermore, for every proper subterm $t' = g(u_1, \ldots u_k, \text{ccCheck}(\underline{cs}_{t'}))$ of t_1, where g is a defined symbol, it must hold that $\text{ccCheck}(\underline{cs}_{t'})\sigma_1 \to^*_{\mathcal{R}} []$, because by having $\mathcal{Q} = \{\text{ccCheck}(x) \to \ldots\}$, a chain only exists if the defined function ccCheck can be removed. Thus, for every subterm t', $t'\sigma_1$ is a basic instance by lemma 10.23.

Especially, all independent class constraints on the rule paths used to reduce t' have a basic instance. Thus, for every term used on these rule paths, a basic instance exists by lemma 10.15, since every right-hand side of a rule contains a term starting with ccCheck, which must be reduced first due to $\mathcal{Q} = \{\text{ccCheck}(x) \to \ldots\}$. Therefore, lemma 10.23 guarantees the existence of a basic instance of these right-hand sides, and thus also for the left-hand sides of those rules.

Putting this together, we get that $t_1\sigma_1$ is a basic instance, since for every subterm and the used rules, a basic instance exists.

Since the DP path from s_1 to t_1 does not traverse instantiation edges, lemma 10.15 gives us the existence of a basic instance of s_1, i.e., a substitution σ exists, such that $s_1\sigma$ is a basic instance.

□

From the first term of a chain, a basic instance of the start term can be found in a similar manner. This is done again by following the path from the start node to the first node from which the first Dependency Pair was read. The difference is, that now also for ***Eval***-nodes it has to be made sure that a corresponding basic instance exists.

Lemma 10.25 (Existence of a chain implies existence of a start term)**.** *Let TG be a Termination Graph for a start term s, let G' be an SCC in TG consisting only of Top-Cycles, let $\mathbf{dpRen}^{NT}_{G'} = (\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathsf{a})$ be the DP problem created from G'.*

*If $s_1 \to t_1, s_2 \to t_2 \in \mathcal{P}$ is a chain and for every **ParSplit**- and every **Eval**-node on the path from s to s_1 a basic instance exists for the independent class constraints, then a substitution σ exists, such that $s\sigma$ is a basic instance.*

Proof. From lemma 10.24, it follows that a basic instance $s_1\sigma'$ exists, for some substitution σ'.

Let G be the path from s to s_1 in TG. This path does not contain ***Ins***-nodes and therefore cannot traverse instantiation edges, by requirement on Top Cycles.

For the set $\text{collectCCs}_{G'}(s, s_1)$ it holds that $\text{collectCCs}_{G'}(s, s_1) = \underline{cs}_{s_1} \uplus \{C_1\ \rho_1, \ldots, C_k\ \rho_k\}$, where $C_1\ \rho_1, \ldots, C_k\ \rho_k$ are independent class constraints. For a class constraint to become an independent class constraint, it must be stripped off at an ***Eval***- or a ***ParSplit***-node. Since for these, basic instances exist by assumption, also a basic instance exists for $\text{collectCCs}_{G'}(s, s_1)$.

Now lemma 10.15 is applicable, yielding a basic instance for s.

□

Therefore, only such Dependency Pairs and rules are included in a generated DP problem, where we are sure that for some basic instances, these can be reached. In order to prove existence of some basic instance of the independent class constraints, we use a fixed depth limit. If we cannot prove the existence of a basic instance, the Dependency Pair or the rule is left out. This only has an influence on the strength of the approach, since every infinite chain in a reduced DP problem is also an infinite chain in the DP problem, where no Dependency Pairs or rules were removed.

Lemma 10.26 (Removal of Dependency Pairs and rules is complete)**.** *For a DP problem* $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathsf{a})$*, an infinite* $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$*-chain exists, if for a DP problem* $(\mathcal{P}', \mathcal{Q}, \mathcal{R}', \mathsf{a})$ *with* $\mathcal{P}' \subseteq \mathcal{P}$ *and* $\mathcal{R}' \subseteq \mathcal{R}$ *an infinite* $(\mathcal{P}', \mathcal{Q}, \mathcal{R}')$*-chain exists.*

Proof. Let $s_1 \to t_1, s_2 \to t_2, \ldots$ be an infinite $(\mathcal{P}', \mathcal{Q}, \mathcal{R}')$-chain. Since $\mathcal{P}' \subseteq \mathcal{P}$ and $\mathcal{R}' \subseteq \mathcal{R}$, this is also a $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-chain.

□

This last lemma enables us to restrict ourselves to those Dependency Pairs and rules, where we know that at least some instance exists. Therefore, it is first checked whether basic instances exist, such that an SCC can be reached at all, and whether ***Ins***- and ***ParSplit***-nodes in the SCC have some arbitrary basic instance. This implies the existence of basic instances for the independent class constraints of these nodes. Therefore, we only have to ensure existence of the dependent class constraints, which is integrated in the search for an infinite chain by construction of the DP problems.

10.4 Infinite Chains imply Non-Termination

The Non-Termination analysis shall again make use of the DP Framework as a backend. The problem is that infinite DP problems cannot easily be mapped onto infinite Haskell evaluations. This is due to the fact that a DP problem is already infinite if the set of rules allows for infinite evaluations [GTSK05b]. An example for this is given below:

Example 10.27 (Infinite DP problems do not imply infinite Haskell evaluations)**.** *For the Haskell program*

```
f (S x) y = f x (g x)
g x = g x
```

and the start term `f x' y`*, the following DP problem is created:*

$$\begin{array}{lcl} \mathcal{P} & = & \{ \quad new\_f(S(x), y) \quad \to \quad new\_f(x, new\_g(x)) \quad \} \\ \mathcal{R} & = & \{ \quad new\_g(x) \quad \to \quad new\_g(x) \quad \} \end{array}$$

As can be seen, the TRS $\mathcal{R}$ *is not terminating. This already suffices to call this DP problem infinite, whereas the start term* `f x' y` *is* H*-terminating for all* H*-terminating instantiations of* `x'` *and* `y`*.*

Therefore, we have to require an infinite chain in a DP problem that resulted from a Top-Cycle, in order to have proven Non-Termination. Please note that we cannot restrict the analysis to minimal chains, since we do not consider

subterms in separate DP problems anymore. However, this is not a drawback, since this restriction makes Non-Termination analysis harder, due to the fact that for proving the existence of a minimal infinite chain, one must also show that the instantiated right-hand sides of the Dependency Pairs in the chain are terminating with respect to the rules. Thus, for Non-Termination analysis, removing restrictions makes this analysis easier, because infinite minimal chains are also infinite chains.

Theorem 10.28 (Non-Termination of Haskell)**.** *Let TG be a Termination Graph for a start term t.*

The start term t is non-H*-terminating, if there exists a DP problem* $\mathbf{dpRen}^{NT}_{G'}$ *created only from Top-Cycles in TG which allows for an infinite chain.*

In order to show theorem 10.28, one has to show that infinite chains correspond to infinite Haskell evaluations. Thus it has to be shown first that for DP problems created from a Termination Graph one can reduce the outermost position first. This will be done in lemma 10.30. However, we will use the notion of parallel-disjoint reduction there, which shall be introduced first. This definition goes back to [Hue80].

Definition 10.29 (Parallel-Disjoint Evaluation $\rightarrow^{||}$, [Hue80])**.** *Let* $\mathcal{R}$ *be a TRS.*

For two terms s, t *it holds that* $s \rightarrow^{||}_{\mathcal{R}} t$*, iff for all positions* $\pi_1, \ldots, \pi_n \in Occ(s)$ *where* $\pi_i \perp \pi_j$ *for all* $i \neq j$*, we have* $s|_{\pi_i} \rightarrow_{\mathcal{R}} t|_{\pi_i}$ *for every* $1 \leq i \leq n$*.*

This definition is used to show that outermost evaluation can be done first. In the following, the reductions for the defined symbol ccCheck are disregarded, since they are only needed to ensure the existence of basic instances. We assume the existence of basic instances for all used terms, because otherwise the chain would not exist, due to either a reduction of a term starting with ccCheck not being reducible to the empty list, or a Dependency Pair in the chain would have been removed due to independent class constraints for which no basic instance exists.

Lemma 10.30 (Outermost evaluation w.r.t. $\rightarrow_{\mathcal{R}}$ can be done first)**.** *Let TG be a Termination Graph for a start term t, let* $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathsf{a})$ *be a DP problem created only from Top-Cycles in TG, and let* $l_1 \rightarrow r_1, l_2 \rightarrow r_2 \in \mathcal{R}$*.*

If $t \rightarrow_{l_1 \rightarrow r_1, \pi_1} t_1 \rightarrow_{l_2 \rightarrow r_2, \pi_2} t_2$*,* $t \rightarrow_{l_2 \rightarrow r_2, \pi_2} t[r_2\sigma_2]_{\pi_2}$ *for a substitution* σ_2 *such that* $l_2\sigma_2 = t|_{\pi_2}$*, and* $\pi_1 = \pi_2\pi$ *for some* π*, then* $r_2\sigma_2 \rightarrow^{||}_{l_1 \rightarrow r_1} t_2|_{\pi_2}$*.*

Proof. Please note that in the following, curly braces will be used for substitutions, as the square brackets are used for replacement at a specific position.

For the term t it must hold that $t = t[l_2\sigma_2]_{\pi_2}$ for a matcher σ_2. It also must hold that $t = t[l_1\sigma_1]_{\pi_1}$ for a matcher σ_1, and since $\pi_1 = \pi_2\pi$, the term t must have the following form: $t = t[l_2\sigma_2[l_1\sigma_1]_{\pi}]_{\pi_2}$.

Since $l_2 \rightarrow r_2$ is applicable at position π_2 of t, the term l_2 must be a variable x at position π_l which is defined as the maximal position $\pi' \in Occ(l_2)$, for which $\pi = \pi'\pi''$ for some $\pi'' \in Occ(l_1\sigma_1|_{\pi'})$. This is the case, because subterms on left-hand sides of rules in $\mathcal{R}$ consist only of constructors or variables.

Therefore, $\sigma_2(x) = C[l_1\sigma_1]_{\pi''}$ for some context C. Let $\sigma_2' = \sigma_2$, except that $\sigma_2'(x) = x$. Then $\sigma_2 = \sigma_2'\{x/C[l_1\sigma_1]_{\pi''}\}$.

Thus, the evaluation in the original order looks as follows:

$$\begin{array}{lll} t = t[l_2\sigma_2[l_1\sigma_1]_\pi]_{\pi_2} & \to_{l_1\to r_1,\pi_1} & t[l_2\sigma_2[r_1\sigma_1]_\pi]_{\pi_2} = t_1 \\ & \to_{l_2\to r_2,\pi_2} & t[r_2\sigma_2'\{x/C[r_1\sigma_1]_{\pi''}\}]_{\pi_2} = t_2 \end{array}$$

When first applying $l_2 \to r_2$, the following reduction will take place:

$$t = t[l_2\sigma_2[l_1\sigma_1]_\pi]_{\pi_2} \quad \to_{l_2\to r_2,\pi_2} \quad t[r_2\sigma_2'\{x/C[l_1\sigma_1]_{\pi''}\}]_{\pi_2}$$

Since all occurrences of x in $r_2\sigma_2'$ are on orthogonal positions, it holds that

$$r_2\sigma_2 = r_2\sigma_2'\{x/C[l_1\sigma_1]_{\pi''}\} \quad \to^{||}_{l_1\to r_1} \quad r_2\sigma_2'\{x/C[r_1\sigma_1]_{\pi''}\}$$

As can be seen, the term $r_2\sigma_2'\{x/C[r_1\sigma_1]_{\pi''}\}$ is equal to the term $t_2|_{\pi_2}$.

□

Next, it will be shown that redexes are conserved when a term is reduced using a Dependency Pair that was created from a Termination Graph.

Lemma 10.31 (Reducibility carries over $\mathcal{P}$-steps)**.** *Let $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathsf{a})$ be a DP problem created from Top-Cycles in a Termination Graph and let $s \to t \in \mathcal{P}$.*

If $s\tau \to^*_{\mathcal{R}} s\sigma$ for some substitutions τ and σ, then $t\tau \to^*_{\mathcal{R}} t\sigma$.

Proof. Since every subterm of s consists only of constructors and variables, every redex in $s\tau$ must be on a position π_i where $s|_{\pi_i} \in \mathcal{V}$. Thus, for all $1 \le i \le n$, we have $\tau(x_i) \to^*_{\mathcal{R}} s\sigma|_{\pi_i}$.

Thus, for the reduction $s\tau \to^*_{\mathcal{R}} s\sigma$, it holds that $s\tau = (s[x_1]_{\pi_1} \dots [x_n]_{\pi_n})\tau = s\tau[x_1\tau]_{\pi_1} \dots [x_n\tau]_{\pi_n} \to^*_{\mathcal{R}} s\tau[s\sigma|_{\pi_1}]_{\pi_1} \dots [s\sigma|_{\pi_n}]_{\pi_n} = s\sigma$.

For the substitution τ', where $\tau'(x) = \begin{cases} x, & \text{if } x = x_i \text{ for some } x_i \\ \tau(x), & \text{otherwise} \end{cases}$, we then have $s\sigma = s\tau[s\sigma|_{\pi_1}]_{\pi_1} \dots [s\sigma|_{\pi_n}]_{\pi_n} = s\tau'\{x_1/s\sigma|_{\pi_1}, \dots, x_n/s\sigma|_{\pi_n}\}$, which tells us that $\sigma = \tau'\{x_1/s\sigma|_{\pi_1}, \dots, x_n/s\sigma|_{\pi_n}\}$.

As $s \to t \in \mathcal{P}$, we get $s\tau \to_{\mathcal{P}} t\tau$ and $\mathcal{V}(t) \subseteq \mathcal{V}(s)$. Therefore, $t|_{\pi_j'}\tau \to s\sigma|_{\pi_i}$ for every position $\pi_j' \in Occ(t)$, where $t|_{\pi_j'} = x_{i_j}$ for some $1 \le i_j \le n$. Thus $t\tau = t\tau'\{x_1/\tau(x_1), \dots, x_n/\tau(x_n)\} \to^*_{\mathcal{R}} t\tau'\{x_1/s\sigma|_{\pi_1}, \dots, x_n/s\sigma|_{\pi_n}\} = t\sigma$, which proves the lemma.

□

The next step, that is needed in the proof of theorem 10.28, is to show that given a chain, the reductions from one Dependency Pair to the next are also possible with the Haskell evaluation strategy.

Lemma 10.32 ($\xrightarrow{\mathsf{H}}_{\mathcal{R}}$ suffices for term reachability)**.** *Let $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathsf{a})$ be a DP problem that was created from a Termination Graph TG using Top-Cycles only, let s be a left-hand side in $\mathcal{P}$, let t be a term from a right-hand side of $\mathcal{P}$, and let σ', σ be two substitutions.*

If $t\sigma' \to^*_{\mathcal{R}} s\sigma$, then a substitution τ exists, such that $t\sigma' \xrightarrow{\mathsf{H}}{}^*_{\mathcal{R}} s\tau$ and $s\tau \to^*_{\mathcal{R}} s\sigma$.

Proof. This lemma is proven inductively. As induction relation, the lexicographic combination of whether s matches $t\sigma'$, where in case of a match we have a descent in this order, and over the length of the reduction $t\sigma' \to^{||}{}^*_{\mathcal{R}} s\sigma$ is

used. This is a well-founded order, because both orders which are combined are well-founded.

Case 1: s matches $t\sigma'$

Then a substitution μ exists, such that $s\mu = t\sigma'$. By setting $\tau = \mu$, we get $t\sigma' \xrightarrow{\mathsf{H}}{}^{0}_{\mathcal{R}} s\tau$ and $s\tau = t\sigma' \to^*_{\mathcal{R}} s\sigma$.

Case 2: s does not match $t\sigma'$

In this case, a position $\pi = \mathbf{e}(t\sigma')$ exists, where $\pi \neq \epsilon$. This position must also be a position of s: Since s is a left-hand side in $\mathcal{P}$, all subterms consist only of constructors and variables. Therefore, since s does not match $t\sigma'$, the term $t\sigma'|_\pi$ must be a term of the form $f(t_1, \ldots, t_n)\nu$, where a rule $f(t_1, \ldots, t_n) \to r$ exists in $\mathcal{R}$, the term $s|_\pi$ has the form $C(s_1, \ldots, s_m)$, where C is a constructor, and ν is some substitution. This must be the case, since a variable at $s|_\pi$ would not have blocked the matching, and it must hold that $t\sigma'|_\pi \to^*_{\mathcal{R}} s|_\pi$, because π is the leftmost outermost redex.

Since $\mathbf{e}(t\sigma') = \pi$, the rule application respects the Haskell evaluation strategy, i.e., $t\sigma' \xrightarrow{\mathsf{H}}_{\mathcal{R},\pi} t\sigma'[r\nu]_\pi$. By lemma 10.30, this evaluation can be done first without losing the reachability of $s\sigma$. Therefore, $t\sigma'[r\nu]_\pi \to^*_{\mathcal{R}} s\sigma$ and the reduction $t\sigma'[r\nu]_\pi \to^{||}_{\mathcal{R}}{}^{*} s\sigma$ is shorter than the reduction $t\sigma' \to^{||}_{\mathcal{R}}{}^{*} s\sigma$, because the same rule must be applied in the reduction $t\sigma' \to^*_{\mathcal{R}} s\sigma$. The induction hypothesis for this reduction yields a substitution τ, where $t\sigma'[r\nu]_\pi \xrightarrow{\mathsf{H}}{}^{*}_{\mathcal{R}} s\tau$ and $s\tau \to^*_{\mathcal{R}} s\sigma$. This proves the claim, since $t\sigma' \xrightarrow{\mathsf{H}}_{\mathcal{R}} t\sigma'[r\nu]_\pi \xrightarrow{\mathsf{H}}{}^{*}_{\mathcal{R}} s\tau$.

□

Now the above lemma enables us to show the main theorem, which states that for an infinite chain, an infinite Haskell evaluation of an instance of the start term exists.

Proof of theorem 10.28. Let $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathsf{a})$ be a DP problem created from Top-Cycles in a Termination Graph TG for the start term t, where an infinite chain $s_1 \to t_1, s_2 \to t_2, \ldots$ exists.

Thus, a substitution σ exists, such that for every $i \in \mathbb{N}$: $t_i\sigma \to^*_{\mathcal{R}} s_{i+1}\sigma$. Without loss of generality, we can assume that the sets of variables in two consecutive Dependency Pairs in the chain are disjoint, i.e., for all $i \in \mathbb{N}$ it holds that $\mathcal{V}(s_i) \cap \mathcal{V}(s_{i+1}) = \emptyset$. Then, the substitution σ can be split into an infinite sequence of substitutions σ_i, such that $s_i\sigma = s_i\sigma_i$, which implies $t_i\sigma = t_i\sigma_i$.

Because only Top-Cycles are considered, a term containing $s_1\sigma_1$ on its evaluation position can be reached by a finite number of Haskell evaluation steps from an instance $t\sigma_t$ of the start term. This term can be reduced by a finite number of Haskell evaluation steps to a term containing the subterm $t_1\sigma_1$ on a position which is reducible w.r.t. $\to_{\mathsf{H}}$.

As the preconditions of lemma 10.32 are fulfilled (because $t_1\sigma_1 \to^*_{\mathcal{R}} s_2\sigma_2$), a substitution τ_2 exists, such that $t_1\sigma_1 \xrightarrow{\mathsf{H}}{}^{*}_{\mathcal{R}} s_2\tau_2$, where $s_2\tau_2 \to^*_{\mathcal{R}} s_2\sigma_2$.

For every $i \geq 2$, we have that $s_i\tau_i \xrightarrow{\mathsf{H}}_{\mathcal{P}} t_i\tau_i$ and thus $t_i\tau_i \to^*_{\mathcal{R}} t_i\sigma_i$ by lemma 10.31. This enables another application of Lemma 10.32 starting with

$t_i\tau_i$, since $t_i\tau_i \to_{\mathcal{R}}^* t_i\sigma_i \to_{\mathcal{R}}^* s_{i+1}\sigma_{i+1}$. Thereby, we get $t_i\tau_i \xrightarrow{\mathsf{H}}_{\mathcal{R}}^* s_{i+1}\tau_{i+1}$ for some substitution τ_{i+1}.

This means that an infinite evaluation sequence which respects the Haskell evaluation strategy exists:

$$t\sigma_t \to_{\mathsf{H}}^* s_1\sigma \xrightarrow{\mathsf{H}}_{\mathcal{P}} t_1\sigma \xrightarrow{\mathsf{H}}_{\mathcal{R}}^* s_2\tau_2 \xrightarrow{\mathsf{H}}_{\mathcal{P}} t_2\tau_2 \xrightarrow{\mathsf{H}}_{\mathcal{R}}^* s_3\tau_3 \ldots$$

Every $\xrightarrow{\mathsf{H}}_{\mathcal{P}}$-step comprises at least one ***Eval***-node, which corresponds to a $\to_{\mathsf{H}}$-step, i.e., these correspond to case (a) of the definition of Non-H-Termination (definition 10.1). Furthermore, any ***ParSplit***-node with a constructor as head symbol corresponds to case (c) of that definition. The ***Case***- and ***TyCase***-nodes correspond to the instantiation of the start term. The ***VarExp***-nodes, which are contained in a DP path, correspond to case (b) of the definition of Non-H-Termination. Last, the case of ***Ins***-nodes shall be considered. Since these are contained within the Top-Cycles, there is a sequence of substitutions, such that its cases are used in the infinite chain. This must be reachable from the right-hand side of a Dependency Pair, since it occurs on the infinite chain. Thus, the Haskell evaluation must follow this path, since this path is determined by the subterms of the right-hand side of the previous Dependency Pair, i.e., the subterms of the ***Ins***-node. Therefore, this also corresponds to the Haskell evaluation strategy, since the defined function symbol of the ***Ins***-node occurs on the outermost position, as only Top-Cycles are considered.

Therefore, an instance of the start term exists which is non-H-terminating. □

The following example shall demonstrate that the encoding of type information into the terms is vital for proving non-termination of terms. This is, because terms might be terminating because of type information only.

Example 10.33 (Termination based on types only)**.** *For the following Haskell program, the start term* `base` *is to be analyzed.*

```
class Base a where
        base :: a

instance Base Bool where
        base = True

instance Base a => Base [a] where
        base = [base]
```

As can be seen, the function `base` *makes a recursive call to* `base` *again in the case of lists. So the created DP problem would look as follows (when leaving out the type classes), if no types would be contained in the terms:*

$$\begin{array}{lcl} \mathcal{P} & = & \{ \quad new\_base \quad \to \quad new\_base \quad \} \\ \mathcal{R} & = & \emptyset \end{array}$$

This DP problem clearly contains an infinite chain. However, the original program is terminating. This is due to the fact that the type enforced by the context determines the next `Base` *instance that is to be used in the recursive call of the list instance. Since this type is finite and cannot be changed, eventually*

the call will be made to the instance `Base Bool`*. This can be seen, when types are included in the terms:*

$$\begin{array}{lcl} \mathcal{P} & = & \{ \quad new\_base(\mathsf{app}([], a)) \quad \rightarrow \quad new\_base(a) \quad \} \\ \mathcal{R} & = & \emptyset \end{array}$$

This DP problem does not contain an infinite chain, as in every recursive call the size of the argument is decreased. This can be shown by the Size-Change processor [TG05], for example.

Last, an example shall be presented, where the Non-Termination analysis will find out that the given start term is indeed not H-terminating for the accompanying Haskell program.

Example 10.34 (Non-H-terminating function `div`)**.** *We want to analyze the start term* `div m n` *in the following Haskell program:*

```
data Nats = Z | S Nats

minus  Z     _    = Z
minus  x     Z    = x
minus (S x) (S y) = minus x y

div Z _ = Z
div x y = S (div (minus x y) y)
```

As can be seen, this program calculates the division of two natural numbers by first testing whether the dividend is Zero. If this is the case, the result is Zero, as well. Otherwise, it subtracts the divisor from the dividend and adds one to the result. Here, the ceiling function is applied to the result, i.e., `div` m n `=` $\lceil \frac{m}{n} \rceil$.

For this program, the Termination Graph shown in figure 10.6 is constructed. As we can observe, there is only one Top-Cycle in the Termination Graph, namely the cycle for `div`*. Thus, the created DP problem for Non-Termination is the following:*

$$\mathcal{P} \; = \; \{ \quad new\_div(S(m0), n, []) \quad \rightarrow \quad new\_div(new\_minus(S(m0), n, []), n, []) \quad \}$$

$$\begin{array}{lcll} \mathcal{R} \; = \; \{ & new\_minus(m0, Z, []) & \rightarrow & S(m0) \\ & new\_minus(m0, S(n0), []) & \rightarrow & new\_minus0(m0, n0, []) \\ & new\_minus0(m0, Z, []) & \rightarrow & m0 \\ & new\_minus0(Z, S(n00), []) & \rightarrow & Z \\ & new\_minus0(S(m00), S(n00), []) & \rightarrow & new\_minus0(m00, n00, []) \quad \} \end{array}$$

When this DP problem is processed by the Non-Termination processor, which is described in [GTSK05a], then it returns "no" due to the following infinite chain:

$$\mathbf{new\_div}(\mathbf{S}(\mathbf{m0'}), \mathbf{Z}, []) \quad \rightarrow_{\mathcal{P}} \quad \mathbf{new\_div}(\mathbf{new\_minus}(\mathbf{m0'}, \mathbf{Z}, []), \mathbf{Z}, [])$$

with rule $new\_div(S(m0), n, []) \rightarrow new\_div(new\_minus(m0, n, []), n, [])$
and matcher $[n/Z, m0/m0']$

$$\rightarrow_{\mathcal{R}} \quad \mathbf{new\_div}(\mathbf{S}(\mathbf{m0'}), \mathbf{Z}, [])$$

with rule $new\_minus(m0'', Z, []) \rightarrow S(m0'')$ *at position* 1
and matcher $[m0''/m0']$

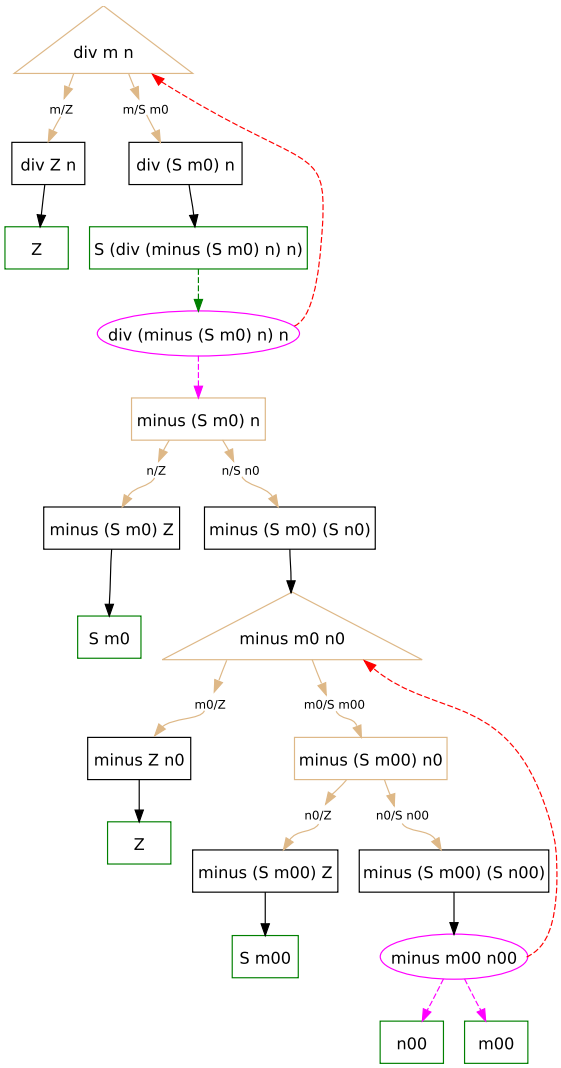

Figure 10.6: Termination Graph for the non-H-terminating start term `div m n`

As can be seen, the term `div (S m0') Z` *can be rewritten in two steps to itself again. This is, because we forgot to test for a division by zero in the above program. In this case, the subtraction will not modify the first argument of* `div`*, and therefore, the evaluation of this instance of the start term will not terminate.*

As this chapter showed, for a restricted set of examples, non-H-termination of start terms can be proven. However, this approach has its drawbacks. Because only Top-Cycles are considered, Non-Termination of terms that require an ***Ins***-node, such as `from Z` for example, cannot be shown as non-H-terminating, since in order to built a cycle, a generalization via an instantiation edge must be performed. Furthermore, the function `seq`, which is defined in the standard

Prelude and enforces evaluation of the first argument to its weak head normal form, and then continues with the second argument, is a problem. When this function is used, then the subterms of it would have to be regarded further, which is not allowed in our Top-Cycles. However, this function is often used in order to decrease the amount of laziness in Haskell programs, such as for example in the case of integer additions.

Thus, a lot of further work remains to be done to develop a component showing Non-Termination of Haskell programs that is comparable in strength to the termination proving component.

Chapter 11

Conclusion and Outlook

In this thesis, various improvements of the Haskell termination analysis have been presented that were all implemented in the automatic termination prover AProVE [GTSKF04, GSKT06]. After a short introduction into the preliminaries and the previous Haskell Termination approach in Chapters 2 and 3, it has been shown in Chapter 4 how to include type classes into the formalism presented in [GSSKT06], and how to get from DP problems with free variables in both the Dependency Pairs and the rules to such DP problems, where only the Dependency Pairs contain free variables.

Chapter 5 presented a way to reduce Haskell programs to only those components that are possibly used in the evaluation of the start term. This speeds up the whole analysis, since it is not required to consider every rule and every data type in the transformations to simpler Haskell and in the construction of the Termination Graph.

The next improvement, that was developed as a part of this thesis, is the renaming which was presented in Chapter 6. Here, different cycles in the Termination Graph are assigned different names, which cares for the separation of cycles in the created DP problem. Furthermore, it allows to always generate first-order terms, which speeds up the creation of the estimated Dependency Graph a lot and also makes the standard orderings work better. Finally, it also includes type classes in the created terms, which makes the resulting rules non-overlapping.

The renaming enabled us to show in Chapter 7 that instead of proving the absence of arbitrary chains in the DP problems, it suffices to consider only minimal innermost chains. This makes the termination analysis easier, and allows for more techniques developed for Term Rewriting to be applicable to the DP problems.

In Chapter 8, we have evaluated these improvements on a set of examples taken from widely used libraries that ship with the Hugs interpreter [JP99]. On these examples, we could witness a large improvement in strength, since we increase the number of examples that were shown to be terminating from 56.67 % to 76.78 %. The largest improvement was due to the renaming. But still, the introduction of minimality and of innermost evaluation yielded further examples that could be shown terminating. Among these examples there are such that are relevant in pratice, but could not be shown with any other technique that was applicable without having minimality and innermost chains.

Also, it was shown in Chapter 9 how lazy-termination, a property that only makes sense in a lazy evaluating language like Haskell, can be reduced to termination analysis. This based on previous work in [PSS97], but now also considers functional types.

Last, Chapter 10 presented a newly developed approach for showing non-termination of Haskell start terms. This aids programmers in finding errors, since the produced counterexample can be traced in order to find and correct errors.

Especially the presented approach for non-termination analysis is rather weak. This is due to the lazy evaluation strategy that is employed by Haskell. Furthermore, the Haskell libraries often use forced strictness to calculate numbers, for example. These hinder our approach, since one has to generalize these calculations which makes our approach fail. Thus, it would be interesting to investigate how to handle such evaluations to a weak-head normal form of the first argument by the predefined Haskell function `seq`. An idea would be to disregard the first argument and considering only the second argument of this function. Then, Non-Termination of the program would follow from an infinite chain, because either the first argument cannot be evaluated to a weak-head normal form, or an infinite evaluation exists using the second argument of `seq`.

Another interesting topic is interaction with the environment. At the moment, the IO Monad in Haskell is not handled very well: All calculations with it are either marked as terminating with unknown result or even marked as non-terminating. It would be promising to make some assumptions about the environment, such as for example that all inputs are finite, and then reconsider these functions. It seems as if under such circumstances, one can better handle this very basic pattern of Haskell programs.

Furthermore, in the case of termination analysis we make a very rigid assumption: If we identify a term as an instance of another term, we assume that all subterms must be shown terminating, since these might be evaluated at some other point. It would be interesting, if one can show that an argument will never be evaluated. Then, termination of this argument would not need to be shown. An approach to do this would be to do an analysis on the termination graph and follow those paths, where a term was matched to a variable.

The above might seem, as if a strictness analysis [Nöc93, SSPS95] could help. However, strictness analysis is only concerned with whether the weak-head normal form of a term can be precomputed without changing the termination behavior. This is not what helps in H-termination proofs, because there we would be interested in finding out that a term is not considered at all or only up to a certain constructor depth. In the latter case, one could then switch to an analysis for n-lazy-termination, where n is the depth that is never exceeded.

For the Non-Termination analysis, strictness analysis seems much better suited. However, it is unclear how to use it for this purpose, since strictness only means that a reduction to a weak-head normal form is required by a context. This does not help in extending the Top-Cycles to subterms of other terms, because in order to infer Non-Termination, we need full evaluation of a term.

Also, the generated Dependency Pairs are filtered in the presented approach. This is a rather harsh approach, since it removed quite a lot of information that was present, and disallows us to show non-H-termination for such terms. Here, an adaption of the approach for Prolog [SKGST07] seems interesting, since this approach also handles fresh variables. This approach needs constructor

rewriting, which was already discussed in section 7.2, but was not used in this thesis since it seemed to entail more complex DP problems. This might change however, if fresh variables could be used, in exchange for more complicated DP problems.

Bibliography

[AG00] Thomas Arts and Jürgen Giesl. Termination of Term Rewriting Using Dependency Pairs. *Theoretical Computer Science*, 236(1–2):133–178, 2000.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK, 1998.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In *Proceedings of the 18th conference on Computer Aided Verification (CAV '06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 415–418, Seattle, USA, 2006.

[CT99] Michael Codish and Cohavit Taboch. A semantic basis for termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.

[GSKT06] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR '06)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 281–286, Seattle, USA, 2006.

[GSSKT06] Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp, and René Thiemann. Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages. In *Proceedings of the 17th International Conference on Rewriting Techniques and Applications (RTA '06)*, volume 4098 of *Lecture Notes in Computer Science*, pages 297–312, Seattle, USA, 2006.

[GTSK05a] Jürgen Giesl, René Thiemann, and P. Schneider-Kamp. Proving and Disproving Termination of Higher-Order Functions. In *Proceedings of the 5th International Workshop on Frontiers of Combining Systems (FroCoS '05)*, volume 3717 of *Lecture Notes in Artificial Intelligence*, pages 216–231, Vienna, Austria, 2005.

[GTSK05b] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '04)*, volume 3452 of *Lecture Notes in Computer Science*, pages 301–331, Montevideo, Uruguay, 2005.

[GTSKF03] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Improving Dependency Pairs. In *Proceedings of the 10th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '03)*, volume 2850 of *Lecture Notes in Artificial Intelligence*, pages 165–179, Almaty, Kazakhstan, 2003.

[GTSKF04] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated Termination Proofs with AProVE. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA-04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220, Aachen, Germany, 2004.

[GTSSK07] Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp. Proving Termination by Bounded Increase. In *Proceedings of the 21st International Conference on Automated Deduction (CADE '07)*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 443–459, Bremen, Germany, 2007.

[Has04] Christian Haselbach. Transformation Techniques to Verify Imperative and Functional Programs. Diploma thesis, RWTH Aachen, Germany, 2004.

[HM07] Nao Hirokawa and Aart Middeldorp. Tyrolean Termination Tool: Techniques and Features. *Information and Computation*, 205(4):474–511, 2007.

[Hue80] Gérard Huet. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *Journal of the ACM*, 27(4):797–821, 1980.

[Jon03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. Also available from `http://www.haskell.org/definition`.

[JP99] Mark P. Jones and John C. Peterson. The Hugs 98 user manual, 1999. Available from `http://www.haskell.org/hugs`.

[Käu05] Christian Käunicke. Automatic Termination Analysis of Logic Programs. Diploma thesis, RWTH Aachen, Germany, 2005.

[MB05] Fred Mesnard and Roberto Bagnara. cTI: A Constraint-Based Termination Inference Tool for ISO-Prolog. *Theory and Practice of Logic Programming*, 5(1&2):243–257, 2005.

[MZ07] Claude Marché and Hans Zantema. The Termination Competition. In *Proceedings of the 18th International Conference on Rewriting Techniques and Applications (RTA '07)*, volume 4533 of *Lecture Notes in Computer Science*, pages 303–313, Paris, France, 2007. See also `http://www.lri.fr/~marche/termination-competition`.

[Nöc93] Eric Nöcker. Srictness Analysis using Abstract Reduction. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 255–265, Copenhagen, Denmark, 1993. ACM Press.

[PSS97] Sven Eric Panitz and Manfred Schmidt-Schauss. TEA: Automatically proving termination of programs in a non-strict higher order functional language. In *Proceedings of the 4th International Symposium on Static Analysis (SAS '97)*, volume 1302 of *Lecture Notes in Computer Science*, pages 345–360, Paris, France, 1997.

[SKGST07] Peter Schneider-Kamp, Jürgen Giesl, Alexander Serebrenik, and René Thiemann. Automated Termination Analysis for Logic Programs by Term Rewriting. In *Proceedings of the 16th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR '06)*, volume 4407 of *Lecture Notes in Computer Science*, pages 177–193, Venice, Italy, 2007.

[Son07] Matthias Sondermann. Automatische Terminierungsanalyse für imperative Programme. Diploma thesis, RWTH Aachen, Germany, 2007.

[SSPS95] Manfred Schmidt-Schauß, Sven Eric Panitz, and Marko Schütz. Strictness Analysis by Abstract Reduction using a Tableau Calculus. In *Proceedings of the Second International Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 348–365, Glasgow, UK, 1995.

[Swi05] Stephan Swiderski. Terminierungsanalyse von Haskellprogrammen. Diploma thesis, RWTH Aachen, Germany, 2005.

[TG05] René Thiemann and Jürgen Giesl. The Size-Change Principle and Dependency Pairs for Termination of Term Rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 16(4):229–270, 2005.

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

List of Figures

Index