Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

On Termination by Dependency Pairs and Termination of First-Order Functional Specifications in PVS

Ariane Alves Almeida

Tese apresentada como requisito parcial para
conclusão do Doutorado em Informática

Orientador
Prof. Dr. Mauricio Ayala-Rincón

Coorientador
Dr. César Augusto Muñoz

Brasília
2021

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

On Termination by Dependency Pairs and Termination of First-Order Functional Specifications in PVS

Ariane Alves Almeida

Tese apresentada como requisito parcial para
conclusão do Doutorado em Informática

Prof. Dr. Mauricio Ayala-Rincón (Orientador)
CIC e MAT/UnB

Dr. César Augusto Muñoz (Coorientador)
(NASA LaRC)

Dr. René Thiemann
Universität Innsbrück

Dr. Mariano Miguel Moscato
NIA/NASA LaRC FM Group

Dr. Edward Hermann Haeusler
PUC-Rio

Dr. Vander Ramos Alves
CIC/UnB

Prof.a Dr.a Genaína Nunes Rodrigues
Coordenadora do Programa de Pós-graduação em Informática

Brasília, 09 de Julho de 2021

Dedication

This thesis is dedicated to my parents and my husband.
They always supported and encouraged me in this endeavor.
Without the presence, love, and understanding of any of them,
I would not have been able to accomplish any of this.
The merit of this thesis is also theirs.

Agradecimentos

I thank my advisor, Professor Dr. Maurício Ayala-Rincón, for all the time and effort invested in guiding me in this work (and in previous ones) and for always demanding the best from his students. Your support and constant availability were fundamental for the conclusion of this stage of my academic life.

Also in the role of guiding me in this work, I thank Dr. César A. Muñoz, my co-advisor. From our first interactions, always being very helpful and considerate with any collaboration I needed.

I also really appreciate all contributions given by the referees who evaluated this document and its presentation, Dr. René Thiemann, Dr. Mariano Moscato, Dr. Edward Hermann Haeusler, Dr. Vander Ramos and Dr. Flávio de Moura. Every comment and suggestion really enriched the work.

I am also grateful for the collaboration we were able to make between Prof. Mauricio, Dr. César Muñoz, Dr. Mariano M. Moscato and me, which allowed me to work twice in loco with them at National Institute of Aerospace (NIA/NASA LaRC), which was of great value to my personal and academic growth.

Still on this subject, I also thank the NIA/NASA LaRC for allowing these visits, both with all the bureaucratic part and for the financial support. I am very grateful for all the help and welcome I received from the entire NIA/NASA team, the researchers I worked with, and also their families, who made me feel at home during my stay in the US.

I also thank the University of Brasília, in particular the Department of Informatics (CIC), for the wonderful work environment where we all are surrounded by encouraging people. Everyone helped me somehow, from the guards, cleaning and secretariat staff, professors and my laboratory and study friends, who always helped me think of new perspectives in research, to review works and face this journey from beginning to end.

This work was supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), through access to the Portal de Periódicos and a PhD scholarship through the Programa de Demanda Social. The work also had the support of the Fundação de Apoio a Pesquisa do Distrito Federal (FAPDF) for participation in relevant scientific events and for the collaboration with the NIA/NASA LaRC team.

Resumo

Embora indecidível, terminação é uma propriedade muito importante relacionada à correção de objetos computacionais. Ela garante que para cada entrada, não haverá execução infinita, ou seja, a execução deve parar e fornecer algum resultado. Esta propriedade permite, por exemplo, raciocinar sobre a correção de programas, uma vez que garantir que alguma propriedade seja válida para cada saída depende da obtenção de uma saída a ser verificada sempre que uma entrada for fornecida. Ao longo dos anos, várias estratégias de semidecisão foram usadas para abordar esse problema e raciocinar sobre ele. No contexto de Programas Funcionais (FPs), por exemplo, a análise pode ser feita por meio dos Gráficos de Contexto de Chamada, e no contexto de Sistemas de Reescrita de Termos (TRSs), podem ser usados Pares Dependentes.

Este trabalho formaliza o Critério de Terminação por Pares Dependentes (Mais Internos), um critério muito conhecido para análise de terminação de TRSs, no assistente de prova PVS. PVS é um assistente de prova com uma linguagem de especificação funcional que permite lógica de ordem superior e realiza provas seguindo o Cálculo de Sequentes de Gentzen. Também são apresentados vários Critérios de terminação formalizados para FPs em uma linguagem simplificada que modela especificações em PVS (chamada `PVS0`) e a formalização da equivalência entre eles, permitindo automação de provas de terminação de especificações de funções recursivas de primeira ordem em PVS. O trabalho também discute a possibilidade de navegar entre os critérios de terminação para TRSs e FPs com o objetivo de aprimorar a automação para verificar terminação.

Palavras-chave: Termination, formalization, functional programs, term rewriting systems

Abstract

Although undecidable, termination is a very important property related to the correctness of computational objects. It ensures that for every input, there will be no infinite execution, i.e., the execution must stop and provide some result. This property allows, for instance, reason about correctness of programs, since to guarantee that some property hold for every output depends on obtaining an output to be analysed whenever an input is provided. Through the years, several semi-decision strategies have been used to address such problem and reason over it. In the context of Functional Programs (FPs), for instance, the analysis can be done through the Calling Context Graphs, and in the context of Term Rewriting Systems (TRSs), Dependency Pairs can be used.

This work formalizes the (Innermost) Dependency Pairs Termination Criterion, a very well-known criteria for analyzing termination of TRSs, in the proof assistant PVS. PVS is a proof assistant with a functional specification language that allows higher order logic and performs proofs following the Gentzen Calculus of Sequents. It is also presented several termination criteria formalized for FPs in a simplified language modeling PVS specifications (called `PVS0`) and the formalization of the equivalence between them, allowing automation for proving termination of recursive functions first-order specification in PVS. The work also discusses the possibility of navigating between the termination criteria for TRSs to FPs aiming to improve automation of verification for termination.

Keywords: Termination, formalization, functional programs, term rewriting systems

Contents

1 Introduction **1**
1.1 Work Organization . . . 6

2 Background **7**
2.1 Functional Programs . . . 7
2.2 Term Rewriting Systems . . . 13

3 Termination Criteria **19**
3.1 Ranking Functions . . . 20
3.1.1 Termination in the Prototype Verification System . . . 23
3.2 The Size-Change Principle and Calling Context Graphs . . . 25
3.3 Dependency Pairs . . . 28

4 Specification of DPs for TRSs and Termination Criteria for `PVS0` **33**
4.1 Extension of TRS with DPs . . . 33
4.2 `PVS0` . . . 38
4.2.1 Semantic termination of PVS0 . . . 42
4.2.2 Specification of Ranking Functions for PVS0 . . . 45
4.2.3 Specification of Size-Change based technologies . . . 46

5 Formalization of termination by Dependency Pairs **51**
5.1 Necessity for the Innermost Dependency Pairs Termination Criterion . . . 52
5.2 Sufficiency for the Innermost Dependency Pairs Termination Criterion . . . 55
5.2.1 Existence of *mint* Subterms . . . 57
5.2.2 Non-root Innermost Normalization of *mint* Terms . . . 57
5.2.3 Existence of DPs . . . 62
5.2.4 Construction of Chained DPs . . . 63
5.2.5 Construction of the Infinite Innermost Dependency Chain . . . 65
5.3 Formalization of DP termination for other rewriting relations . . . 66
5.4 Library - `TRS` Theory Summary . . . 67

5.5 Related work: other formalizations of DPs 74

6 Formalization of Termination Criteria in PVS0 76

6.1 Equivalence between semantic criteria 76

6.2 Equivalence between TCC termination and semantic termination 78

6.3 Equivalence between TCC and SCP technologies 80

6.4 NASA PVS Library - `PVS0` Theory Summary 83

6.5 NASA PVS Library - `CCG` Theory Summary 86

7 Connecting FP and TRS Termination Criteria 88

7.1 CC versus DP 88

7.2 Evaluation versus Derivation 92

7.3 Using the Dependency Pairs Termination Criterion for `PVS0` Programs . . . 93

7.4 Related work 98

7.4.1 A translation to orthogonal TRSs 98

8 Conclusion and Future Work 102

Bibliography 106

Chapter 1

Introduction

Termination is a very important property related to the correctness of computational objects, since it can guarantee that some output will eventually be provided to any input. Once termination is guaranteed, it can be verified if the output of a program or the reaction of a process is correct. Thus, developing mechanisms to verify termination of computational objects is of great interest, leading to the development of various approaches to prove and disprove termination, enabling certification of termination.

However, termination of programs and processes is a well-known undecidable problem, closely related to the undecidability of the Halting Problem [Tur37]. In general, the undecidability of termination in a specific computational setting is proved by building reductions of known computational models with known undecidable Halting Problem to that setting. For instance, by reducing Turing Machines into *Term Rewriting Systems* (for short, TRSs), it can be proved the undecidability of termination for TRSs [HL78], a formal framework to reason about functional programs. Nevertheless, syntactic and semantic restrictions, data structures, and heuristics can lead to solutions for subclasses of undecidable problems such as termination.

Through the years, several methods to verify termination have been applied and further developed to provide some automation to the process of checking termination for different computational models, such as the λ-calculus, the rewriting systems and the functional programs and others.

In the context of rewriting, termination can be checked by local analysis on the rules, such as simplification orderings that include, for instance, lexicographic, multisets, recursive paths, ranking function into ordinals [Der79, DM79a, Der82, Der87, YKS15, TSSY20, Sch14]. Also, more sophisticated methods can be used, such as the Dependency Pairs Termination Criterion [Art96, AG97, AG98, AG00]. This criterion provides a robust mechanism to analyze the termination of TRSs. Instead of locally checking the decreasingness of rewrite rules, it is checked the decreasingness of the fragments of rewrite rules headed by symbols that allow application of other rules. Indeed, a Dependency Pair consists of the left-hand side (*lhs*) of a rewrite rule and a subterm of the right-hand side (*rhs*) of the rule headed by *defined symbols*. Thus, a Dependency Pair expresses the dependency of a function in other calls of any function, which could lead to an infinite derivation.

In the context of functional programs, local and general approaches are used to check termination, such as:

- *Ranking functions* [Tur49], which are measures that must decrease over the arguments of each possible (recursive) function call (data exchange point).
- Size-Change Principle [LJBA01], which aims to state the termination of a program for any input by showing that an infinite computation would lead to an infinite decrease over a well-founded order.
- The Calling Context Graph criterion [MV06] automates the analysis of the termination of recursive calls in a similar way to the Size-Change Principle. This approach creates a graph whose vertices are Call Contexts (CC), which bring the information of the parameters and the conditions, which lead to a recursive call in the definition of another function. The idea is then to deduce the termination verifying that the infinite execution of a program generates an infinite sequence of CCs, where the current parameters of each context are related to the current parameters of the following context. This verification is done by providing a decreasing measure through every cycle of the obtained graph.

The development of approaches for verifying termination, such as the ones mentioned above, along with advances in theorem proving, enables the formal verification of algorithms used in several applications to ensure their correctness. Formal verification can be aided by proof assistants, allowing formalization of properties of algorithms, such as termination, in a systematic way. For instance, in Coq [BC04], termination of well-typed functions is guaranteed by the Calculus of Inductive Constructions on which the prover is based. Termination of recursive functions are checked in ACL2 [KMM00] by incorporated syntactic conditions. In Isabelle, lexicographic order is used by default to try stating termination, requiring a well-founded order to be manually defined when the default technique fails. In the Prototype Verification System (PVS) [ORS92] a measuring function stating decrease over the formal and actual parameters of recursive functions must always be provided by the user.

In particular, PVS is an interactive theorem prover based on classical higher-order logic extensively used at NASA to verify safety-critical algorithms of autonomous unmanned systems[1], which usually are specified as recursive functions whose computations must be

[1] `https://shemesh.larc.nasa.gov/fm/pvs/`

terminating. Thus, to provide ways to automate the verification of termination in PVS specification is of great interest. This prover, such as others, uses decreasing measures as its semantics of termination. In PVS the *ranking functions* must be provided by the user as part of each recursive definition, and the decreasingness requirements are implemented by the so-called *termination Type Correctness Conditions* (termination TCCs, for short). Termination TCCs are *proof obligations* built by static analysis over the recursive definitions, stating that the measure of the actual parameters of each recursive call strictly decreases regarding the measure of the formal parameters.

Take, for instance, the PVS definition of a recursive function `occurrences` over list of naturals that counts how many times a given natural x occurs in a list l:

$$
\begin{array}{l}
\texttt{occurrences}(l)(x) = \\
\quad \texttt{IF null?}(l) \texttt{ THEN} \\
\qquad 0 \\
\quad \texttt{ELSIF car}(l) = x \texttt{ THEN} \\
\qquad 1 + \texttt{occurrences } (\texttt{cdr}(l))(x) \\
\quad \texttt{ELSE} \\
\qquad \texttt{occurrences } (\texttt{cdr}(l))(x)
\end{array}
$$

This function has recursive calls on the `ELSIF` and `ELSE` branches, both with the same argument $\texttt{cdr}(l)$. The measure $\texttt{length}(l)$ can be provided by the user and can easily be proven strictly decreasing on the argument of these calls ($\texttt{cdr}(l)$) regarding the input l.

For other functions, to provide such measure can be more challenging; analysing such functions and their termination is a point of interest since the '70s. Some of the functions to illustrate this are the ones with nested calls as the McCarthy 91 function [MM69, MP70] and those to generate the so-called Meta-Fibonacci sequences [Vaj89, chapter XII], such as the Hofstadter [Hof79] and Conway [Mal91] sequences.

McCarthy 91	**Hofstadter**	**Conway**
$\texttt{f91}(x) =$	$\texttt{h}(x) =$	$\texttt{c}(x) =$
$\quad \texttt{IF } x > 100 \texttt{ THEN}$	$\quad \texttt{IF } x = 1 \texttt{ OR } x = 2 \texttt{ THEN}$	$\quad \texttt{IF } x \leq 2 \texttt{ THEN}$
$\qquad x - 10$	$\qquad 1$	$\qquad 1$
$\quad \texttt{ELSE}$	$\quad \texttt{ELSE}$	$\quad \texttt{ELSE}$
$\qquad \texttt{f91}(\texttt{f91}(x + 11))$	$\qquad \texttt{h}(x - \texttt{h}(x-1)) +$	$\qquad \texttt{c}(\texttt{c}(x-1)) +$
	$\qquad \texttt{h}(x - \texttt{h}(x-2))$	$\qquad \texttt{c}(x - \texttt{c}(x-1))$

Notice however that such measure can be defined, for instance, by observing the behaviour of the function. Whenever the input n is such that $n > 100$, no recursive step is performed, then the measure can be any $c|c \leq 100$ and the decrease holds. If $n \leq 100$, the execution of the inner call leads to k nested calls adding 11 each time until it exceeds 100, where the range of ten for numbers greater than two gives a pattern allowing to obtain the amount of nested recursive steps before start executions without recursion as $\texttt{f91}(x) = \texttt{f91}^{k+1}(\texttt{f91}(x + 11 * k))$ where $k = \lfloor \frac{(100-x)}{11} \rfloor$:

Range	Expression $\texttt{f91}^k(x)$
$0 \leq n \leq 1$	$\texttt{f91}^{9+1}(n + 99)$
$2 \leq n \leq 12$	$\texttt{f91}^{8+1}(n + 88)$
$\vdots$	$\vdots$
$79 \leq n \leq 89$	$\texttt{f91}^{1+1}(n + 11)$
$90 \leq n \leq 100$	$\texttt{f91}^{1+0}(n + 0)$

Then the value obtained will be some $y > 100$, which will decrease the amount of nested calls by one and produce results decreased by 10. This will lead to a behavior of obtaining results greater than 100 and smaller or equal to 100 alternately when the input is $y|90 \leq y \leq 100$. The number of steeps in this scenario can be obtained by observing the execution of the function:

Execution	Steps
$\texttt{f91}(100) = \texttt{f91}(\texttt{f91}(111)) = \texttt{f91}(101) = 91$	3
$\texttt{f91}(99) = \texttt{f91}(\texttt{f91}(110)) = \texttt{f91}(100) \cdots$	5
$\vdots$	$\vdots$
$\texttt{f91}(91) = \texttt{f91}(\texttt{f91}(102)) = \texttt{f91}(92) \cdots$	21
$\texttt{f91}(90) = \texttt{f91}(\texttt{f91}(101)) = \texttt{f91}(91) \cdots$	23

Then, for every $90 \leq y \leq 100$ the number of steps in the execution of `f91` is $(101 - y) * 2 + 1$. And, since by this analysis it is possible to state that $\texttt{f91}(x) = \texttt{f91}^k(x) = \texttt{f91}^k(\texttt{f91}(y))$, that for all $90 \leq y \leq 100$, $\texttt{f91}(y) = 91$ and that $\mathit{Steps}(\texttt{f91}(91)) = 21$, the final measure can be given as:

$$
\begin{array}{l}
\mathit{Steps}(\texttt{f91}, x) = \\
\quad \texttt{IF } (x > 100) \texttt{ THEN} \\
\qquad 1 \\
\quad \texttt{ELSE} \\
\qquad k + [((101 - (x + k * 11)) * 2 + 1)] + k * 21
\end{array}
$$

Such measure can be required when specifying the `f91` function. For instance, PVS will generate the following termination TCCs for this function, where μ is a measuring function that must be provided in the moment of the specification, which may not be such an easy task, as seen in the previous discussion.:

$$\begin{array}{l}\forall(n : nat) : \\ \quad \neg(n > 100) \Rightarrow \\ \qquad \mu(n + 11) < \mu(n)\end{array} \quad \text{and} \quad \begin{array}{l}\forall(n : nat, \\ \quad v : [\{z : nat|\mu(z) < \mu(n)\} \to nat]) : \\ \qquad \neg(n > 100) \Rightarrow \\ \qquad \mu(v(n + 11)) < \mu(n)\end{array}$$

To reason over properties of specifications of first-order functions in PVS, a simplified functional language model named `PVS0`[2] was defined. The `PVS0` language is Turing-Complete, and can model first order PVS functions with input and output of same type. Thus the simplified language of `PVS0` is adequate to show evidence of the correctness of applying any sound termination criteria to check first-order PVS specifications. The verification of equivalence between different termination criteria provides a propitious setting to automate termination analysis. Furthermore, proofs of properties such as termination for PVS specifications are easier to provide in `PVS0`, since there are less elements to be considered in this language.

The operational semantics of `PVS0`, just as of several functional languages, is defined by eager evaluation. Such semantics can be modeled by the innermost normalization of TRSs. There are several termination approaches for TRSs, from very syntactic to more flexible and robust ones, such as the Dependency Pairs Criterion, which can be applied for innermost reduction. If such criterion could be used to reason over PVS specifications, it would be valuable to automate verifying their termination.

This work provides a formalization of the Dependency Pairs Termination Criterion for innermost reduction in PVS. This formalization enriches the NASA PVS Library `TRS` (a library on Term Rewriting Systems) with a new termination criterion, which allows the possibility of investigate the possibility of using such a robust criterion to improve the efforts of automating termination of PVS specifications. For such use, there must be an equivalence between `PVS0` specifications and TRSs and also between the criterion for TRSs and the criteria for functional programs formalized in `PVS0`. The discussion on how these equivalences could be reached is also initiated in this work.

[2] Theory `PVS0` is available in the PVS nasalib `https://shemesh.larc.nasa.gov/fm/pvs/PVS-library/` and presented in [MARM$^+$21].

1.1 Work Organization

The document presents its main contributions in Chapters 4 and 5, which discuss the formalization of Dependency Pairs Termination in the PVS proof assistant. Also, it collaborates with the NASA LaRC Formal Methods Group and other researchers from the Universidade de Brasília in results for the `PVS0` language presented in Chapter 6. The presentation of these criteria and the contributions over them follow the structure below:

- Chapter 2 presents the background necessary, including functional programs and term rewriting systems, their essential elements, operational semantics, and termination definitions.
- Chapter 3 discusses termination criteria for functional programs as Ranking Functions [Tur49], Size-Change Termination Principle [LJBA01] and, Calling Context Graphs [MV06]. Also, this chapter discusses the notion of Dependency Pairs Termination for term rewriting systems [AG97]. The former criteria are formalized for the `PVS0` functional language [MARM$^+$21], and the latter is described in further chapters in this work.
- Chapter 4 describes the specification of Dependency Pairs developed in this work that extends the PVS theory for term rewriting systems, called `TRS`. Besides, it describes the specification of termination criteria for the `PVS0` language described in the previous chapter.
- Chapter 5 presents the formalization of the Dependency Pairs Termination Criterion for the innermost rewriting relation. Also, it explains how this formalization is extended for the ordinary rewriting relation and for the Q-restricted rewriting relation as initially done in [ST10]. These results and details on how this formalization extended the NASA PVS `TRS` library are published in [AAAR20].
- Chapter 6 presents the formalization in PVS of the termination criteria mentioned in Chapter 3 for `PVS0`.
- Chapter 7 speculates about the application of the Dependency Pairs Termination Criterion and its formalization in PVS to deal with automation of termination of functional programs, as specified for the `PVS0` language and discuss its relation with the work from [KST$^+$11], which aims the same goal for functions specified in Isabelle.
- Finally, Chapter 8 concludes and suggests future work.

Chapter 2

Background

Two computational models are overviewed in this work: Functional Programs and Term Rewriting Systems. Assuming familiarity with both models, some basic notions regarding them and their necessary elements to reason over termination are recalled.

2.1 Functional Programs

Functional programming is a well-known programming paradigm based on the pure application of functions to input values. Thus, this approach is very close to mathematical functions and allows the analysis of several properties in a more direct way.

In this work, a simplified model of first-order functional programs will be considered. Let *FName*, *Const*, and *Var* be countable (in)finite sets of function symbols, constant symbols, and variable names, respectively. Additionally, let *Op* be a finite set of operator symbols that are interpreted as terminating built-in operators. Each $f \in \mathit{FName}$ or $op \in \mathit{Op}$ is said to be of arity n if it is defined or interpreted using n formal parameters $(x_0, ..., x_n \in \mathit{Var})$. The set of well-formed expressions *Exp* has the following grammar:

$$e := c \mid x \mid ite(e, e, e) \mid op(e, ..., e) \mid f(e, ..., e)$$

Where $c \in \mathit{Const}$, $x \in \mathit{Var}$, $op \in \mathit{Op}$, and $f \in \mathit{FName}$; arguments of f and op are expressions given consistently with their arities. The `if-then-else` branching instruction is denoted by *ite* and has three arguments: the guard, the `then`, and the `else` expressions.

The definition of an n-ary function $f \in \mathit{FName}$, uses the syntax $f(x_0, .., x_n) := e_f$, where the expression e_f is the *body* of the function. Variables occurring in e_f are restricted to belong to the formal parameters of the definition. A *Functional Program* (FP) P is a sequence of non-mutually recursive function definitions.

Example 2.1.1 (Expression for the Ackermann's function). *Let* $=, +, -,$ *and* $0, 1$ *be, respectively, binary operators and constants with their usual interpretation. Also, let* n *and* m *be variable names. The Ackermann function over naturals is defined as:*

$$
\begin{aligned}
ack(m,n) := ite(\ & = (m,0),\\
& +(n,1),\\
& ite(\ = (n,0),\\
& \qquad ack(-(m,1),1),\\
& \qquad ack(-(m,1), ack(m, -(n,1)))
\end{aligned}
$$

To analyze the behavior of functions, it is necessary to get through its body (sub) expression(s). This is achieved by the path position of each subexpression, given in Definition 2.1.1.

Definition 2.1.1 (Positions of expressions). *The set of positions of an expression* e*, denoted as* $Pos(e)$*, is the set of sequences or "*paths*" of naturals, where* λ *denotes the empty sequence, recursively defined as:*

$$
Pos(e) := \begin{cases}
\{\lambda\} & \textit{if } e = c \in Const \textit{ or } e = x \in Var;\\
\{\lambda\} \cup \bigcup_{i=0}^{2} \{i\pi \mid \pi \in Pos(e_i)\} & \textit{if } e = ite(e_0, e_1, e_2);\\
\{\lambda\} \cup \bigcup_{i=0}^{k} \{i\pi \mid \pi \in Pos(e_i)\} & \textit{if } e = op(e_0, ..., e_k);\\
\{\lambda\} \cup \bigcup_{i=0}^{m} \{i\pi \mid \pi \in Pos(e_i)\} & \textit{if } e = f(e_0, ..., e_m).
\end{cases}
$$

For $\pi \in Pos(e)$*,* $e|_\pi$ *denotes the subexpression of* e *at path* π*.*

Example 2.1.2 (Continuing Example 2.1.1). *The expression* $-(m,1)$ *occurs at the positions* 210 *and* 220 *in the body of the definition of ack, for short, denoted as* e_{ack}*; that is* $e_{ack}|_{210} = e_{ack}|_{220} = -(m,1)$.

The functions usually refer to other functions or even themselves throughout their body during their execution/evaluation. Such *function calls* require special attention in the analysis of the existence of possible loops.

Definition 2.1.2 (Function Calls). *Let* $f(x_0, \ldots, x_{n_f}) := e_f$ *be a function definition and* $\pi \in Pos(e_f)$ *such that the* $e_f|_\pi = g(e_0, \ldots, e_{n_g})$*, where* $g \in FName$*. This is called a* function call *and is denoted as* $f(x_0, \ldots, x_{n_f}) \xrightarrow{\pi} g(e_0, \ldots, e_{n_g})$*, or for brevity* $f \xrightarrow{\pi} g$*, or even simply as* π*, when no confusion arises. The expressions* $e_0, \ldots, e_{n_g}$ *are called the* actual parameters *of the function call.*

It is expected that function calls reference functions that are defined inside the program. The actual parameters of these function calls are analyzed through eager evaluation. Thus, for the evaluation, it is necessary to keep track of the actual parameters of the function calls (Definition 2.1.3).

Definition 2.1.3 (State Transition)**.** *Consider a program P with function definitions $f(x_0, \ldots x_{n_f}) := e_f$, $g(y_0, \ldots, y_{n_g}) := e_g$. A* state *is a pair (f, δ), where δ is a map from the formal parameters of f to expressions. If there is a function call in e_f, $f(x_0, \ldots, x_{n_f}) \xrightarrow{\pi} g(e_0, \ldots, e_{n_g})$, then for the mapping δ' on the formal parameters of g, defined as $\delta'(y_i) \mapsto e_i\delta$, for $i = 0, \ldots, n_g$, we say that $(f, \delta) \xrightarrow{\pi} (g, \delta')$ is a* state transition*. A* state transition sequence *is a sequence of states such that consecutive states form state transitions.*

Example 2.1.3 (Continuing Example 2.1.2)**.** *The Ackermann function given in Example 2.1.1 has three recursive calls: one at position 21 ($ack(-(m, 1), 1)$); another one at position 22 ($ack(-(m, 1), ack(m, -(n, 1)))$); and a last one at position 221 ($ack(m, -(n, 1))$). Take as first state $(ack, (u, v))$, where (u, v) abbreviates the assignment $\beta = \{m/u, n/v\}$, for expressions u and v. Then, one has the state transition sequence below associated to nested calls 221, 21, 22:*

$$
\begin{aligned}
&(ack, (u, v)) \xrightarrow{221} (ack, (u, -(v, 1))) \xrightarrow{21} (ack, (-(u, 1), 1)) \xrightarrow{22} \\
&(ack, (-(-(u, 1), 1), ack(-(u, 1), -(1, 1))))
\end{aligned}
$$

Below there is an infinite state transition sequence for the program ack:

$$
(ack, (u, v)) \xrightarrow{221} (ack, (u, -(v, 1))) \xrightarrow{221} (ack, (u, -(-(v, 1), 1))) \xrightarrow{221} \cdots
$$

Since function calls happen in specific paths of function definitions, one can determine the expressions for the guards of branching instructions that encompass the function call by analyzing the prefixes of the path of a function call, as defined below.

Definition 2.1.4 (Calling Conditions)**.** *Let e be an expression and $\pi \in Pos(e)$. The* calling conditions *for π in e are recursively defined as:*

$$
CConds(\pi, e) := \begin{cases}
true & \text{if } \pi = \lambda; \\
CConds(\pi', e_j) & \text{if } e = op(..., e_j, ...) \text{ and } \pi = j \cdot \pi'; \\
CConds(\pi', e_j) & \text{if } e = f(..., e_j, ...) \text{ and } \pi = j \cdot \pi'; \\
CConds(\pi', e_0) & \text{if } e = ite(e_0, e_1, e_2) \text{ and } \pi = 0 \cdot \pi'; \\
e_0 \wedge CConds(\pi', e_1) & \text{if } e = ite(e_0, e_1, e_2) \text{ and } \pi = 1 \cdot \pi'; \\
\neg e_0 \wedge CConds(\pi', e_2) & \text{if } e = ite(e_0, e_1, e_2) \text{ and } \pi = 2 \cdot \pi'.
\end{cases}
$$

Each expression (e_i or $\neg e_i$) conjugated in calling conditions *$CConds(\pi, e)$ is a* condition*. For an expression $f(x_0, ..., x_n) := e_f$ and $\pi \in Pos(e_f)$, $CConds(\pi, e_f)$ are called the* calling conditions *of the expression $e_f|_\pi$. In particular, for an expression $f(x_0, ..., x_n) := e_f$*

which is the body of a program P and $\pi \in Pos(e_f)$, $CConds(\pi, e_f)$ are called the calling conditions *of the program P.*

Example 2.1.4 (Continuing Example 2.1.3). *For Ackermann, one has that:*

$$\begin{array}{lcl} CConds(21, e_{ack}) & := & \neg(=(m,0)) \wedge =(n,0) \\ CConds(221, e_{ack}) & := & \neg(=(m,0)) \wedge \neg(=(n,0)) \end{array}$$

For the operational semantics of FPs, let Val be the set of values whereupon the program is defined. Val is enriched with a special additional distinguished and fresh value $\diamondsuit$ that represents "none", and the boolean values $TRUE$ and $FALSE$. There is an assignment $\beta : Var \to Val$ and a mapping $\mathcal{I}$ that interprets the primitive operators as total built-in functions. Also, in this context $\mathcal{I}$ interprets constants (as zero-ary operators).

The semantic evaluation of expressions in a definition e_f of a program, given an interpretation $\mathcal{I}$ and an assignment β for the formal parameters of the definition of f, is given by the function $\chi(e, \beta, n)$, for $n \in \mathbb{N}$, below. This function returns a value whenever the evaluation is possible, allowing at most n nested function calls, and $\diamondsuit$ otherwise.

Definition 2.1.5 (Semantic Evaluation). *Let P be a program with an assignment β as formal parameters, and e be any expression. For $n \in \mathbb{N}$, an expression e is evaluated recursively as $\chi(e, \beta, n) :=$*

$$\begin{cases} \diamondsuit & \textit{if } n = 0; \ \textit{otherwise:} \\ \mathcal{I}(c) & \textit{if } e = c \in Val; \\ \beta(x) & \textit{if } e = x \in Var; \\ \diamondsuit & \textit{if } e = ite(e_0, e_1, e_2) \ \textit{and} \ \chi(e_0, \beta, n) = \diamondsuit; \\ \chi(e_1, \beta, n) & \textit{if } e = ite(e_0, e_1, e_2) \ \textit{and} \ \chi(e_0, \beta, n) = TRUE; \\ \chi(e_2, \beta, n) & \textit{if } e = ite(e_0, e_1, e_2) \ \textit{and} \ \chi(e_0, \beta, n) = FALSE; \\ \mathcal{I}(op)(\chi(e_0, \beta, n), & \textit{if } e = op(e_0, \ldots, e_k) \ \textit{and} \ \forall (0 \leq i \leq k) : \quad \chi(e_i, \beta, n) \neq \diamondsuit; \\ \qquad \ldots, & \\ \qquad \chi(e_k, \beta, n)) & \\ \diamondsuit & \textit{if } e = op(e_0, \ldots, e_k) \ \textit{and} \ \exists (0 \leq i \leq k) : \chi(e_i, \beta, n) = \diamondsuit; \\ \chi(e_g, e_g, \beta', n-1) & \textit{if } e = g(e_0, \ldots, e_m), \ \textit{where for each formal parameter } y_i \textit{ of } g : \\ & \qquad\qquad\qquad \beta'(y_i) := \chi(e_i, \beta, n) \neq \diamondsuit; \\ \diamondsuit & \textit{if } e = g(e_0, \ldots, e_m) \ \textit{and} \ \exists (0 \leq i \leq m) : \chi(e_i, \beta, n) = \diamondsuit. \end{cases}$$

Semantic evaluation allows one to state semantic termination as below.

Definition 2.1.6 (Semantic Termination). *Given a program P and a function body e_f for a function f in P, e_f is said to be terminating by semantic evaluation, denoted as $\mathcal{T}_\chi(e_f)$, if $\forall(\beta) : \exists(n) : \chi(e_f, \beta, n) \neq \diamondsuit$. P is said to be terminating, denoted as $\mathcal{T}_\chi(P)$, whenever $\mathcal{T}_\chi(e_f)$, for all e_f in P.*

Regarding semantic evaluation, for a given assignment, a function call will only be performed if the conditions on its path hold. Furthermore, the values of the actual parameters of the function call are determined during the evaluation. Thus, the state transitions (Definition 2.1.3) must include adequate instantiations for the formal parameters of the calls, as given in Definition 2.1.7.

Definition 2.1.7 (Nested Calls)**.** *Consider a function call* $f(x_0,\ldots,x_{n_f}) \xrightarrow{\pi} g(e_0,\ldots,e_{n_g})$ *of a program* P*. Let* β *be an assignment from the formal parameters of* f *to* Val*. If there exists* $n \geq 0$*, such that if for each condition* e_c *in* $CConds(\pi, e_f)$*,* $\chi(e_c,\beta,n) = TRUE$ *and for each actual parameter* e_i*, for* $i = 0\ldots n_g$*, of the nested call,* $\chi(e_i,\beta,n) \neq \Diamond$*, then* $(f,\beta) \xrightarrow{\pi,n} (g,\beta')$ *is called an* evaluated state transition *or a* nested call*, where the assignment* β' *is defined from the formal parameters of* g*,* $y_0,\ldots,y_{n_g}$*, such that* $\beta'(y_i) \mapsto \chi(e_i,\beta,n)$*. Additionally, a* feasible sequence of nested calls *is a sequence of nested calls such that, for every two consecutive nested calls of the sequence, the second state of a call and the first state of the next call are over the same function definition and assignment, i.e.* $(f_i,\beta) \xrightarrow{\pi_i,n} (f_j,\beta'),(f_j,\beta') \xrightarrow{\pi_j,n} (f_k,\beta'')$*, that can be abbreviated as* $(f_i,\beta) \xrightarrow{\pi_i,n} (f_j,\beta') \xrightarrow{\pi_j,n} (f_k,\beta'')$*.*

Notice that a sequence of nested calls is indeed a sequence of feasible state transitions related to some possible program execution.

Example 2.1.5 (Sequence of Nested Calls for Ackermann)**.** *Continuing Example 2.1.4, for the first state transition sequence, below it is given a related sequence of nested calls, where assignments from the formal parameters of* e_{ack}*,* m,n *are abbreviated as pairs of naturals.*

$$(ack,(3,1)) \xrightarrow{221,7} (ack,(3,0)) \xrightarrow{21,6} (ack,(2,1)) \xrightarrow{22,5} (ack,(1,3))$$

Notice however that the second (infinite) sequence in Example 2.1.4 is not feasible, since $e_{ack}|_{221} = ack(m,-(n,1))$*, and for every initial assignment from the formal parameters of Ackermann to* $\mathbb{N}$*, say* β_0*, since the condition* $\neg = (\beta_i(n),0)$ *belongs to* $CConds(221,e_{ack})$*, and no possible infinite assignments* β_i*,* $i > 0$*, exist such that* $\beta_i(n) \mapsto \chi(e_{ack},\neg = (n,0),\beta_{i-1},k_i) = TRUE$*, for some* $k_i \in \mathbb{N}$*:*

$$(ack,\beta_0) \xrightarrow{221,k_0} (ack,\beta_1) \xrightarrow{221,k_1} (ack,\beta_2) \xrightarrow{221,k_2} \cdots$$

Another notion of semantic termination can then be stated regarding the nested calls of a program as below.

Definition 2.1.8 (Termination by Finite Nested Calls)**.** *A program* P *is said to be terminating by finite nested calls, denoted as* $\mathcal{T}_\nu(P)$*, if there exist no infinite sequences of assignments* β_i*, and functions* f_i *in the definition of* P*, and* n*, with* $i,n \in \mathbb{N}$*, such that*

$$(f_0,\beta_0) \xrightarrow{\pi_1,n} (f_1,\beta_1) \xrightarrow{\pi_2,n} \cdots \xrightarrow{\pi_i,n} (f_i,\beta_i) \xrightarrow{\pi_{i+1},n} \cdots$$

It is possible to state the equivalence between the two notions of semantic termination, given in Definitions 2.1.6 and 2.1.8, through the result of Lemmas 2.1.1 and 2.1.2 below.

Lemma 2.1.1 (Evaluation to $\diamondsuit$ produces infinite nested calls)**.** *Let f be a function defined in a program P. If for all $n \in \mathbb{N}$, $\chi(e, \beta, n) = \diamondsuit$, then, for some f_0 in the expression e there is a sequence of infinite nested calls $(f_0, \beta_0) \xrightarrow{\pi_1,n} (f_1, \beta_1) \xrightarrow{\pi_2,n} (f_2, \beta_2) \xrightarrow{\pi_3,n} \cdots$.*

Proof. It will be checked in general that there should exist a call at some position π of the expression e of some function g, $e|_\pi = g(e_1, \dots, e_{n_g})$, such that for all conditions c in $\mathit{CConds}(\pi, e)$, $\chi(e_f, c, \beta, n) = \mathit{TRUE}$ for some $n \in \mathbb{N}$, and $\chi(e_f, g(e_0, ..., e_{n_g}), \beta, n) = \diamondsuit$, for all $n \in \mathbb{N}$. This is proved by induction in e.

- If $e = x$ or $e = c$, then the assumption that for all $n \in \mathbb{N}$, $\chi(e, \beta, n) = \diamondsuit$ does not hold.
- If $e = \mathit{ite}(e_0, e_1, e_2)$, the call may happen in e_0, e_1 or e_2. If for some n, $\chi(e_f, e_0, \beta, n) \neq \diamondsuit$, there are two cases to consider. If $\chi(e_f, e_0, \beta, n) = \mathit{TRUE}$, then for all $n \in \mathbb{N}$, $\chi(e_f, e_1, \beta, n) = \diamondsuit$, and by induction hypothesis, there is a call at position π of e_1 such that for all conditions c in $\mathit{CConds}(\pi, e_1)$, $\chi(e_f, c, \beta, n) = \mathit{TRUE}$ for some $n \in \mathbb{N}$, and $\chi(e_f, e_1|_\pi, \beta, n) = \diamondsuit$, for all $n \in \mathbb{N}$. If $\chi(e_f, e_0, \beta, n) = \mathit{FALSE}$, induction is similarly applied to e_2. Otherwise, the call that generates $\diamondsuit$ happens in the condition itself since for all $n \in \mathbb{N}$, $\chi(e_f, e_0, \beta, n) = \diamondsuit$, and by induction hypothesis, there is a call at position π of e_0 such that for all conditions c in $\mathit{CConds}(\pi, e_0)$, $\chi(e_f, c, \beta, n) = \mathit{TRUE}$ for some $n \in \mathbb{N}$, and $\chi(e_f, e_0|_\pi, \beta, n) = \diamondsuit$, for all $n \in \mathbb{N}$. In the three cases, $i = 0, 1$ and 2, consider that $e|_{i\pi} = g(e_1, \dots, e_{n_g})$.
- If $e = \mathit{op}(e_0, ..., e_m)$, for some $0 \le i \le m$, it should happen that $\chi(e_f, e_i, \beta, n) = \diamondsuit$, for all $n \in \mathbb{N}$. Thus, by induction hypothesis, there is a call at position π of e_i such that for all conditions c in $\mathit{CConds}(\pi, e_i)$, $\chi(e_f, c, \beta, n) = \mathit{TRUE}$ for some $n \in \mathbb{N}$, and $\chi(e_f, e_i|_\pi, \beta, n) = \diamondsuit$, consider that $e|_{i\pi} = g(e_1, \dots, e_{n_g})$.
- If $e = g(e_0, ..., e_{n_g})$, then if for all $0 \le i \le n_g$, there is some $n \in \mathbb{N}$, such that $\chi(e_f, e_i, \beta, n) \neq \diamondsuit$, the call that produces $\diamondsuit$ is this call itself. Otherwise, if for some $0 \le i \le n$, $\chi(e_i, \beta, n) = \diamondsuit$, for all $n \in \mathbb{N}$, then induction is applied on e_i, and it should exist a π in e_i, such that for all conditions c in $\mathit{CConds}(\pi, e_i)$, $\chi(e_f, c, \beta, n) = \mathit{TRUE}$ for some $n \in \mathbb{N}$, and $\chi(e_f, e_i|_\pi, \beta, n) = \diamondsuit$ for all $n \in \mathbb{N}$.

Thus, in all relevant cases, there is a call at some position π of e, $e|_\pi = g(e_1, \dots, e_{n_g})$. And according to the semantic evaluation, $\chi(e_f, g(e_0, ..., e_{n_g}), \beta, n) = \chi(e_g, e_g, \beta', n-1) = \diamondsuit$, for all $n \ge 1$, where for m large enough, the assignment β' is given by $\beta'(y_i) := \chi(e_f, e_i, \beta, m) \neq \diamondsuit$, for all formal parameters y_i of g. Let $f_0 = g$ and $\beta_0 = \beta'$; proceeding in the same manner for $\chi(e_{f_0}, e_{f_0}, \beta_0, n)$, a nested call of the form $(f_0, \beta_0) \xrightarrow{\pi_1,n} (f_1, \beta_1)$, for which once again one will have that $\chi(e_{f_1}, e_{f_1}, \beta_1, n) = \Diamond$, for all $n \in \mathbb{N}$. In this manner, the infinite sequence of nested calls is built. $\square$

Lemma 2.1.2 (Semantic Termination Equivalence)**.** *Given a program P, $\mathcal{T}_\chi(P)$ iff $\mathcal{T}_\nu(P)$.*

Proof. ($\Leftarrow$) By contraposition: if not $\mathcal{T}_\chi(P)$, then there exist an f_0 defined in P and an assignment β on the formal parameters of e_{f_0} such that for all $k \in \mathbb{N}$, $\chi(e_f, e_{f_0}, \beta_0, k) = \Diamond$; by Lemma 2.1.1, this implies the existence of an infinite sequence of nested calls $(f_0, \beta_0) \xrightarrow{\pi_1} (f_1, \beta_1) \xrightarrow{\pi_2}$ .... Thus, by Definition 2.1.8, not $\mathcal{T}_\nu(P)$.

($\Rightarrow$) Assuming $\mathcal{T}_\chi(P)$ it will be proved by induction that there is no possible infinite sequence of nested calls initiated from any evaluation of some function f_0 in P with assignment β_0: $(f_0, \beta_0) \xrightarrow{\pi_1,k} \cdots$. Notice that this sequence is originated by an evaluation of the form $\chi(e_{f_0}, e_{f_0}, \beta_0, k)$. The lexicographic order on the pair k and the size of the subexpression of e_{f_0} being evaluated (in general at position π of e_{f_0}) is the used measure. Assume a minimal pair of k and e_{f_0} have been chosen such that it initiates an infinite sequence of nested calls with assignment β. Starting from this evaluation, some function call at position π' in e_{f_0} of the form $g(e_0, \ldots, e_{n_g})$ is performed under assignment β. Thus, we have three possibilities to build an infinite sequence of nested calls. If π is a position in some argument, say e_i, of this call, the evaluation $\chi(e_{f_0}, e_i, \beta, k)$ would give rise to an infinite sequence, but this contradicts the minimality assumption. Another possibility is that π happens in some condition c in $\mathit{CConds}(\pi', e_{f_0})$; thus, the evaluation $\chi(e_{f_0}, c, \beta, k)$ gives rise to the infinite sequence of nested calls, but again this is a contradiction by minimality assumption. Then, the sole remaining possibility is that the infinite sequence of nested calls starts at position π itself, further by the evaluation $\chi(e_g, e_g, \beta', k-1)$, where β' is defined as $\beta'(y_i) \mapsto \chi_s(e_{f_0}, e_i, \beta, k)$, for all $0 \le i \le n_g$: $(f_0, \beta_0) \xrightarrow{\pi,k} (g, \beta') \cdots$, but this is not possible by minimality assumption.

$\square$

2.2 Term Rewriting Systems

This section extends the background section of the paper [AAAR20]. Notations are compatible with those given in textbooks on rewriting [BN98, BKB+03].

The logical framework of Term Rewriting Systems is a well-known computational model to reason over functional programs. Overall, TRSs consist of pairs of elements (terms) which are related by a binary relation. Given any relation R, the notations R^+ and R^* denote, respectively, its transitive and reflexive-transitive closure. The relation R^* between two terms will be referred to as *derivation*. For a relation R and an element

s, if there exists t such that $s\ R\ t$ holds, then s is said to be R-reducible, otherwise, it is said to be in R-normal form, denoted by $nf(R)(s)$.

The standard notation for terms, subterms, and positions will be followed in this work (e.g. [BN98]).

Definition 2.2.1 (Term)**.** *Given a countable set of variables V and a signature Σ, a term $t \in T(\Sigma, V)$ is given as a variable or as a function symbol g applied to a tuple of terms of length given by the arity of g according to the signature Σ. The set $T(\Sigma, V)$ is given by the terms freely generated from a set V according to a signature Σ.*

Remark 1. *The symbol* `root` *is used as a special operator that returns the root function symbol of application terms, which is automatically created when the datatype for terms is specified.*

As in functional programs, the analysis over TRSs often relies on the structure of its elements, such as the positions within terms (given as sequences of naturals, as usual) and the subterms at such positions.

Definition 2.2.2 (Positions of terms)**.** *The set of positions of a term t, denoted as $Pos(t)$ includes the* root *position that is the empty sequence, denoted as λ, and if t is an application, say $g(t_1, \ldots, t_n)$, all positions of the form $\{i\pi \mid 0 \leq i < n, \pi \in Pos(t_i)\}$.*

Definition 2.2.3 (Subterm and Replacement)**.** *Given a term s and a position $\pi \in Pos(s)$, the* subterm *of s at position π is denoted as $s|_\pi$. The subterm relation is denoted by $\trianglerighteq$: $s \trianglerighteq s'$, if there exists $\pi \in Pos(s)$ such that $s' = s|_\pi$. If such given position π is such that $\pi \neq \lambda$, s' is called a* proper subterm *of s, which is denoted as $s \triangleright s'$. Notation $s[\pi \leftarrow t]$ is used to denote the term resulting from replacing the subterm $s|_\pi$ of s by the term t.*

Example 2.2.1 (Subterm and Replacement)**.** *The term $t = g(x, g(y, y))$ has the following subterms at respective positions, where only the first one is not a proper subterm:*

$$\begin{array}{lcl} t|_\lambda & = & g(x, f(y, y)) \\ t|_0 & = & x \\ t|_1 & = & g(y, y) \\ t|_{10} & = & y \\ t|_{11} & = & y \end{array}$$

And the term $g(g(x, x), g(y, y))$ is obtained by replacing $g(x, g(y, y))[0 \leftarrow g(x, x)]$

Definition 2.2.4 (Rewrite rule)**.** *A* rewrite rule *is an ordered pair of terms, l and r, called respectively the left-hand side (lhs for short) and the right-hand side (rhs for short), denoted by $l \to r$, such that $l \notin V$ and $Var(r) \subseteq Var(l)$.*

Definition 2.2.5 (Term Rewriting System (TRS)). *Given a countable set of variables V and a signature Σ, a TRS E is a set of rewrite rules that are ordered pairs of terms in $T(\Sigma, V)$.*

Example 2.2.2 (TRS). *Three rules below conform a TRS for the Ackermann function, where s and 0 are the usual constructors for naturals.*

$$\begin{aligned} &a(0, y) \to s(y) \\ &a(s(x), 0) \to a(x, s(0)) \\ &a(s(x), s(y)) \to a(x, a(s(x), y)) \end{aligned}$$

The relations between terms define the operational semantics of TRSs, given by the application of rewrite rules to terms to obtain *reductions*.

Definition 2.2.6 (Reduction and Normal Form). *Given a TRS E, and terms s and t, there is a* reduction *from s to t at position $\pi \in Pos(s)$, denoted as $s \xrightarrow{\pi}_E t$ (or just $s \xrightarrow{\pi} t$ if E is clear from context), if there exist some rule $l \to r \in E$ and some substitution σ such that $l\sigma = s|_\pi$ and $t = s[\pi \leftarrow r\sigma]$. The term s is then* reducible at position π.

If no specific position is given, but there exists some position $\pi \in Pos(s)$ and term a t such that $s \xrightarrow{\pi} t$, the term s is said to be reducible, *and whenever t is given, the term s is said to* reduce *to t, denoted as $s \to_E t$. Since reduction is a relation, a term that is not reducible is in* normal form.

Example 2.2.3. *Considering the TRS for Ackermann in Example 2.2.2, one has, in general, that terms of the form $a(0, s^k(0))$ reduce into $s^{k+1}(0)$, and terms of the form $a(s(0), s^k(0))$ derive into $s^{k+2}(0)$, for $k > 0$, where s^k abbreviates k applications of s. Also, terms of the form $0, s(0)$, etc are normal forms.*

There are scenarios that require not only the pattern matching used in reductions but also to solve more general problems, such as the equational problems discussed in Chapter 7. For such purposes the *narrowing* relation can be used [KK96].

Definition 2.2.7 (Narrowing). *Given a TRS E, and terms s and t, s* narrows *to t at position π, denoted as $s \rightsquigarrow_E t$, if π is a nonvariable position and there exist some rule $l \to r \in E$ and* most general unifier *σ of $s|_\pi$ and l such that $t = s\sigma[\pi \leftarrow r\sigma]$. When details such as the position, the rule and/or the position used are required, the notation can include them in this order and them it may be written as $s \rightsquigarrow_{[\pi, l \to r, \sigma]} t$. A* narrowing derivation *$s \rightsquigarrow^*_\sigma t$ is a sequence of narrowings steps $s \rightsquigarrow^*_{\sigma_1} s' \rightsquigarrow^*_{\sigma_2} \cdots \rightsquigarrow^*_{\sigma_n} t$, where the* solution *$\sigma$ is given as $\sigma = \sigma_n \cdots \sigma_2\sigma_1$.*

Example 2.2.4 (Narrowing). *Add the following rules to the TRS for Ackermann in Example 2.2.2:*

$$\stackrel{?}{=}(x, x) \to \top$$

Narrowing the expression $\stackrel{?}{=}(\stackrel{?}{=}(a(x,y), s(s(0))), \top)$ *search for solutions to the equational question* $\stackrel{?}{=}(a(x,y), s(s(0)))$*:*

- $\stackrel{?}{=}(\stackrel{?}{=}(a(x,y), s(s(0))), \top) \rightsquigarrow_{[\{x/0\}]} \stackrel{?}{=}(\stackrel{?}{=}(s(y), s(s(0))), \top) \rightsquigarrow_{[\{y/s(0)\}]} \stackrel{?}{=}(\top, \top) \rightsquigarrow \top$, *that correspond to the solution* $\{x/0, y/s(0)\}$*;*
- $\stackrel{?}{=}(\stackrel{?}{=}(a(x,y), s(s(0))), \top) \rightsquigarrow_{[\{x/s(x'), y/0\}]} \stackrel{?}{=}(\stackrel{?}{=}(a(x', s(0)), s(s(0))), \top) \rightsquigarrow_{[\{x'/0\}]} \stackrel{?}{=}(\stackrel{?}{=}(s(s(0)), s(s(0))), \top) \rightsquigarrow \stackrel{?}{=}(\top, \top) \rightsquigarrow \top$, *that correspond to the solution* $\{x/s(0), y/0\}$*;*
- $\stackrel{?}{=}(\stackrel{?}{=}(a(x,y), s(s(0))), \top) \rightsquigarrow_{[\{x/s(x'), y/0\}]} \stackrel{?}{=}(\stackrel{?}{=}(a(x', s(0)), s(s(0))), \top) \rightsquigarrow_{[\{x'/s(x'')\}]} \stackrel{?}{=}(\stackrel{?}{=}(a(x'', a(s(x''), 0))), s(s(0))), \top) \rightsquigarrow_{[\{x''/0\}]} \stackrel{?}{=}(\stackrel{?}{=}(s(a(s(0), 0)), s(s(0))), \top) \rightsquigarrow \stackrel{?}{=}(\stackrel{?}{=}(s(s(s(0))), s(s(0))), \top)$, *which gives no solution;*
- *Other narrowing derivations are possible, but they will not produce solutions.*

In some specific implementations, such as the one used in this work to deal with chains of *Dependency Pairs*, it is interesting to avoid reductions at the root position of terms. For this, one uses the *non-root reduction* relation.

Definition 2.2.8 (Non-root Reduction). *Denoted by* $\stackrel{>\lambda}{\longrightarrow}$ *, the* non-root reduction *relation is induced by a TRS E and relates terms s and t whenever* $s \stackrel{\pi}{\to} t$ *for some* $\pi \in Pos(s)$ *such that* $\pi \neq \lambda$*.*

Example 2.2.5 (Non-root Reduction). *Considering the TRS for Ackermann in Example 2.2.2 and the term* $a(s(s(0)), s(a(s(0), 0)))$*:*

$a(s(s(0)), s(a(s(0), 0))) \stackrel{10}{\longrightarrow} a(s(s(0)), s(a(0, s(0))))$ *is a non-root reduction;*
$a(s(s(0)), s(a(s(0), 0))) \stackrel{\lambda}{\to} a(s(0), a(s(s(0)), a(s(0), 0))$ *is a reduction, but is not a non-root reduction.*

The positions where the reductions occur also lead to reductions strategies that allow, for instance, to reason over such reductions and relate them with other strategies. This is the case of the *innermost reduction*, a strategy that has an operational semantic close to the one of the eager evaluation of functional programs.

Definition 2.2.9 ((Non-root) Innermost Reduction)**.** *A term s is said to be* innermost reducible at position $\pi \in Pos(s)$ *if* $nf(\xrightarrow{>\lambda})(s|_\pi)$ *and* $s \xrightarrow{\pi}_E t$ *for some term t; this is denoted as* $s \xrightarrow{\pi}_{in} t$*.*

If no specific position is given, but there exists some position $\pi \in Pos(s)$ and term t such that $s \xrightarrow{\pi}_{in} t$, s is said to be innermost reducible*, and whenever t is given, s is said to* innermost reduce *to t; denoted as $s \rightarrow_{in} t$.*

Whenever the innermost reduction takes place at a position $\pi \neq \lambda$, one has a so-called non-root innermost reduction*, denoted as* $\xrightarrow{>\lambda}_{in}$*.*

Example 2.2.6 (Non-root innermost reduction for Ackermann)**.** *Considering the TRS for Ackermann in Example 2.2.2 and the term $a(0, a(s(0), s^k(0)))$, one has that:*

$$a(0, a(s(0), s^k(0))) \xrightarrow{1}_{in} a(0, a(0, a(s(0), s^{k-1}(0))))$$

The innermost reduction also allows one to obtain results regarding the termination of TRSs with specific properties, such as Orthogonal TRSs (which are the TRSs that model functional programs, [BN98]), where innermost termination and termination are equivalent [Gra96].

Another relation useful to this work is regarding the *descendants* of a term through a given relation. This relation is relevant, for instance, when analyzing a given derivation and it is required to know exactly which term gave rise to this specific derivation.

Definition 2.2.10 (Rewriting Restricted to Descendants)**.** *The* reduction relation restricted to (descendants of) a term t *is induced by pairs of terms u, v derived from t such that there are derivations $t \rightarrow^* u \rightarrow v$. The notation used is $u \underset{t}{\rightarrow} v$. Similarly to the previous rewriting relations, it can be explicit the exact position $\pi \in Pos(u)$ where the reduction took place, and then the notation $\underset{t}{\xrightarrow{\pi}}$ is used. Analogous notation applies to innermost and non-root reductions.*

Example 2.2.7 (Rewriting Restricted to Descendants for Ackermann)**.** *Considering the TRS for Ackermann in Example 2.2.2 and terms*

$$\begin{aligned}
&t = a(0, a(s(0), s^k(0)),\\
&u = a(0, a(0, a(s(0), s^{k-1}(0))))\\
&v = s(a(0, a(s(0), s^{k-1}(0))))
\end{aligned}$$

One has that $u \underset{t}{\rightarrow} v$, since

$$a(0, a(s(0), s^k(0)) \rightarrow a(0, a(0, a(s(0), s^{k-1}(0)))) \rightarrow s(a(0, a(s(0), s^{k-1}(0))))$$

This work focuses on verifying techniques for termination analysis to provide means to automate termination analysis correctly for functional programs. Since TRSs are suitable to reason over such programs, the notion of termination for such systems is a key property.

Definition 2.2.11 ((Innermost) Terminating TRSs and terms)**.** *A TRS E is said to be* (innermost) terminating *if it has no infinite (innermost) derivations.*

A term s is (innermost) terminating *if no infinite (innermost) derivation starts with it. Otherwise, the term s is* (innermost) non-terminating *denoted as $\uparrow(s)$ (or $\uparrow_{in}(s)$). Whenever a term s is non-terminating, but all its proper subterms are terminating, one says the term is* minimal non-terminating *(*mnt *for short, denoted by $\twoheaduparrow(s)$), and for innermost termination one says* minimal innermost non-terminating *(*mint *for short, denoted by $\twoheaduparrow_{in}(s)$).*

Example 2.2.8 ((Innermost)Terminating TRS and terms)**.** *Consider the TRS below:*

$$\begin{aligned} &f(a) \to f(a) \\ &f(b) \to b \\ &a \to b \end{aligned}$$

This TRS is innermost terminating, but not terminating. For instance, the term $f(f(a))$ is innermost terminating:

$$f(f(a)) \to_{in} f(f(b)) \to_{in} f(b) \to_{in} b$$

But, in general, this term is non-terminating:

$$f(f(a)) \to f(f(a)) \to f(f(a)) \to f(f(a)) \to \cdots$$

And the subterm $f(a)$ is mnt*, since every proper subterm of it, i.e., a is terminating:*

$$a \to b$$

Chapter 3

Termination Criteria

Termination is a relevant computational property for all computational models, such as Turing Machines, λ-calculus, term rewriting systems, programming language models, etc. This chapter briefly discusses the termination property and termination analysis criteria focusing on the formalizations discussed in this work.

There are several approaches to deal with termination analysis. In the literature, these approaches usually are separated into two ways to deal with the analysis: local analysis, commonly referred to as logical relation or syntactic approaches, and the semantic analysis, that deal with checking the possible execution flow of a program.

Some approaches using local analysis are well-known for checking termination of Term Rewriting Systems, for instance, where simplification orderings are used to check that the rules of such systems have some decrease from the left to the right-hand side of each rule in the system. Such techniques include analysis via multisets, path ordering, etc [Der79, DM79a, Der82, Der87, YKS15, TSSY20] and can be used to check termination of λ-calculus, such as shown in [Ned94] and formalized in [DX07].

As previously mentioned, termination, in general, is an undecidable property, but it can be decidable in specific settings, such as when dealing with the Simply-Typed λ-calculus, where typability is decidable and typability implies strong normalization. As for the general case, where intersection types are used to state strong normalization in general, the undecidability remains [Hin92].

Computational models that provide more expressiveness than functional models also require to analyze termination to ensure some properties. For instance, the Process Calculus (π-calculus) is used to model concurrent computation, where it is desired to ensure that the processes eventually provide some output. This is ensured, for instance, by a combination of conditions based on types and the syntax of the system or linear logic, or by defining well-founded order that decreases along with reductions [San06, YBH04, DS04].

3.1 Ranking Functions

The termination of a program might be guaranteed by ensuring that there is a *decrease* over the data being exchanged during its execution. For different paradigms, the points to measure such data exchange can be specified differently; for instance, for imperative programs they might be given by program instructions, by redexes for term rewriting systems, etc. In particular, for functional specifications the points to measure data exchange are function calls, so that the verification of *decrease* is simplified to verification of the *decrease* of the arguments used as inputs for the execution of a function regarding the actual parameters used in other function calls generated during the execution. This criterion, also known as *ranking functions*, is frequently used and can be traced to [Tur49]. The approach proposed by Turing aims to verify large routines by enriching the instructions of interest with assertions over the variables that can be checked individually, reducing the effort of verifying a program. Additionally, some *measurement* of the variables must be provided that should be shown continually decreasing through execution.

A related practical approach was further proposed [Flo67]. The inputs and outputs of program commands are enriched with assertions (Floyd-Hoare first-order well-known pre and postconditions) so that if the precondition holds and the command is executed, the postcondition must hold. To verify termination, these assertions must be enriched with *decrease* assertions that are built using a well-founded ordering according to some *measurement* function on the inputs and outputs of the program commands.

Recursive programs can also follow this approach [BM79]. The *measurement* required to guarantee *decrease* must then be provided over the actual parameters (arguments) of every *possible* function call regarding the formal parameters of a recursive function. The conditions to *effectively* execute the function call are given by the guards of branching instructions that lead to the function call. This criterion is used in several proof assistants to check the totality of recursive functions. In the case of PVS, the user should provide a measure function for each recursive function, and the necessary decrease conditions are built during type checking as so-called "termination-subtype" *Type Correctness Conditions* (this is the nomenclature used in [OSRSC99], here the abbreviation *termination TCCs* will be used). Then, for the recursive functions, it is expected as a type requirement that formal and actual parameters of each recursive call fulfill this decrease criterion, whenever guards of branching instruction leading to the recursive call hold.

In the remainder of this section, a general overview of the ranking function criterion for recursive functions is given. The first necessary notion identifies the *contexts* associated with function *calls*.

Definition 3.1.1 (Calling Context (CC))**.** *Let* $f(x_0, ..., x_{n_f}) \xrightarrow{\pi} g(e_1, \ldots, e_{n_g})$ *be a function call in a program* P*. The triple* $\langle f(x_0, ..., x_{n_f}), CConds(\pi, e_f), g(e_1, \ldots, e_{n_g})\rangle$ *is called a* Calling Context *(for short, CC) of* P*. The set of all* CCs *of* P *is denoted as* $CCs(P)$*.*

Notice that a CC of the form $\langle f(x_0, ..., x_{n_f}), CConds(\pi, e_f), g(e_1, \ldots, e_{n_g})\rangle$ might be obtained just from the pair $\langle f, \pi\rangle$, since the position π suffices to identify the function call of g in the body of the function f; to make it explicit, the called function is used to ease readability. When referring to a CC, originated by a function call of a function g at position π of e_f, such that $e_f|_\pi$ is rooted by the function symbol g and $conds = CConds(\pi, e_f)$, the notation will be simplified as $\langle f, conds, g\rangle$.

Example 3.1.1 (CCs for Ackermann)**.** *There are three* CCs *for Ackermann in Example 2.1.1:*

$$\begin{aligned}
&\langle ack(m, n),\ \neg(=(m, 0)) \wedge = (n, 0),\ ack(-(m, 1), 1)\rangle \\
&\langle ack(m, n),\ \neg(=(m, 0)) \wedge \neg(=(n, 0)),\ ack(-(m, 1), ack(m, -(n, 1)))\rangle \\
&\langle ack(m, n),\ \neg(=(m, 0)) \wedge \neg(=(n, 0)),\ ack(m, -(n, 1))\rangle
\end{aligned}$$

Notice that for a CC $\langle f(x_0, ..., x_{n_f}), CConds(\pi, e_f), g(e_1, \ldots, e_{n_g})\rangle$, given an assignment β on the parameters of f, the function call $g(e_1, \ldots, e_{n_g})\beta$ will be performed only if the conditions in $CConds(\pi, e_f)$ instantiated with β hold.

Definition 3.1.2 (Measuring Function)**.** *A* measuring function μ_f *is defined from tuples of values assigned to the parameters of a function* f *to a well-founded set* $\mathcal{M}$*. For a given function* f *defined in a program* P *and an assignment* β *on its parameters,* $\mu_f(\beta)$ *will denote the measure of the tuple given by the parameters of* f *instantiated by* β*:* $\mu_f(x_1\beta, \ldots, x_{n_f}\beta)$*. Measures are extended to expressions with free variables in the parameters of* β*, whenever they can be evaluated under the assignment* β *and this is denoted as* $\mu_f(e_1\beta, \ldots, e_{n_f}\beta)$*.*

Indeed, each measure function corresponds to a collection of measures for each function symbol in the program. But here, for simplicity, it will be assumed that each measure adapts to the parameters of each function in P.

Example 3.1.2. *A possible measuring function (on the well-founded set* $\mathbb{N}$*) for the Ackermann function is the lexicographical ordering* $lex(m, n)$ *built over the ordering of naturals.*

Providing the proper comparison between the measures of all assignments of the formal parameters and the corresponding measures of instantiated tuples of actual parameters of recursive calls, one obtains the notion of TCC termination.

Definition 3.1.3 (TCC Termination)**.** *A given program P is said to be* TCC terminating *if there exists a well-founded order $\succ$ and measure functions μ_f, for all functions f defined in P, over a well-founded set $\mathcal{M}$ such that*

$$\forall(\langle f, conds, g\rangle \in CCs(P)) : conds \quad \Rightarrow \quad \mu_f(x_1, \ldots, x_{n_f}) \succ \mu_g(e_1, \ldots, e_{n_g})$$

Such assertions are called termination TCCs *and the notation $\mathcal{T}_\varsigma(P)$ is used when P is* TCC terminating.

Notice that since all e_f sub expressions depend only on the formal parameters of f, the termination TCC generated by a CC related to a pair $\langle\pi\rangle$, where $e_f|_\pi = g(e_1, \ldots, e_{n_g})$, can be written as:

$$\forall(x_1, \ldots, x_{n_f}) : CConds(\pi, e_f) \Rightarrow \mu_f(x_1, \ldots, x_{n_f}) \succ \mu_g(e_1, \ldots, e_{n_g})$$

or also, if β ranges over assignments on the parameters of f, as:

$$\forall(\beta) : CConds(\pi, e_f\beta) \Rightarrow \mu_f(x_1\beta, \ldots, x_{n_f}\beta) \succ \mu_g(e_1\beta, \ldots, e_{n_g}\beta)$$

Notice also that programming languages where mutual recursion is not allowed, as PVS, usually perform termination analysis on each function definition separately, assuming that all other called functions in the definition of a function are previously shown to be terminating. Thus, it is enough to analyze just the CCs related to recursive calls, so that for verifying termination of a function, a sole measure function on the parameters of the function is necessary.

Example 3.1.3 (TCCs Termination for Ackermann)**.** *Using the well-founded ordering $>$ on naturals and the lexicographic measure on pairs of naturals, it can be proved that $\mathcal{T}_\varsigma(ack)$; indeed, the three termination TCCs (related to the three $CCs(ack)$) below are easily provable.*

$$\begin{array}{l}
\forall(m, n) : \neg(= (m, 0)) \wedge = (n, 0) \Rightarrow lex(m, n) > lex(m - 1, 1) \\
\forall(m, n) : \neg(= (m, 0)) \wedge \neg(= (n, 0)) \Rightarrow lex(m, n) > lex(m - 1, ack(m, n - 1)) \\
\forall(m, n) : \neg(= (m, 0)) \wedge \neg(= (n, 0)) \Rightarrow lex(m, n) > lex(m, n - 1)
\end{array}$$

Lemma 3.1.1 (TCC Termination Equivalence)**.** *For a given program P, $\mathcal{T}_\varsigma(P)$ iff $\mathcal{T}_\nu(P)$.*

Proof. ($\Rightarrow$) Assuming not $\mathcal{T}_\nu(P)$, there exists an infinite sequence of nested calls of the form

$$(f_0, \beta_0) \xrightarrow{\pi_1, n} (f_1, \beta_1) \xrightarrow{\pi_2, n} (f_2, \beta_2) \xrightarrow{\pi_3, n} \ldots$$

Then, for each nested call in this sequence, $(f_i, \beta_i) \xrightarrow{\pi_{i+1}} (f_{i+1}, \beta_{i+1})$, there is a corresponding calling context $\langle f_i, CConds(\pi_{i+1}, e_{f_i}), f_{i+1}\rangle$ such that the conditions evaluate to $TRUE$: for all c in $CConds(\pi_{i+1}, e_{f_i})$, $\chi(e_{f_i}, c, \beta_i, n) = TRUE$, and also each formal parameter of f_{i+1} and each actual parameter, say y_j and e_j for $i = 1, .., n_{f_{i+1}}$, respectively, are such that $\chi(e_{f_i}, e_j, \beta_i, n) = y_j\beta_{i+1}$. Assuming, $\mathcal{T}_\varsigma(P)$, there exist measuring functions μ_{f_i} for each f_i in P and a well-founded order $\succ$ over a well-founded set (Definition 3.1.3) such that for each associated termination TCC, instantiated with β_i and β_{i+1}, one has $CConds(\pi_{i+1}, e_{f_i})\beta_i \Rightarrow \mu_{f_i}(\beta_i) \succ \mu_{f_{i+1}}(\beta_{i+1})$, which gives $\mu_{f_i}(\beta_i) \succ \mu_{f_{i+1}}(\beta_{i+1})$, since the condition holds. From this, it is obtained an infinite decreasing sequence for $\succ$ contradicting its well-foundedness.

($\Leftarrow$) Assuming $\mathcal{T}_\nu(P)$, it holds that every possible sequence of nested calls for P is finite such as below:

$$(f_0, \beta_0) \xrightarrow{\pi_1,k_1} (f_1, \beta_1) \xrightarrow{\pi_2,k_2} \cdots \xrightarrow{\pi_n,k_n} (f_n, \beta_n)$$

Let us consider the sequences originated by evaluation of a function f defined in P under assignment β and consider as measure, say μ_f, for this assignment the maximum length of a possible sequence of nested calls generated by it. Then for every CC $\langle f, CConds(\pi, e_f), g\rangle$ and any β such that the conditions hold, the associated nested call $(f, \beta) \xrightarrow{\pi,k} (g, \beta')$ imply that $\mu_f(\beta) \succ \mu_g(\beta')$. Therefore, f is TCC terminating. □

3.1.1 Termination in the Prototype Verification System

PVS is an interactive theorem prover based on classical higher-order logic. The PVS specification language is strongly-typed and supports several typing features including predicate sub-typing, dependent types, inductive data types, and parametric theories. The expressiveness of the PVS type system prevents its type-checking procedure from being decidable. Hence, the type-checker generates TCCs as proof *"obligations"*, that have to be separately proved in order for the type checking process to be considered complete. In practice, the system includes several predefined proof strategies able to automatically discharge most of the TCCs. One goal of formalizing termination criteria is indeed to provide means to enrich such strategies and allow automation of proofs for TCCs regarding termination.

In PVS, a recursive function f of type [A→B] is shown to be tota by providing a *measure function* μ of type [A→T], where T is an arbitrary type, the measuring type, where a *well-founded relation* $<$ exists (over elements in T). The termination TCCs produced by PVS for a recursive function f guarantee that the measuring function strictly decreases with respect to the well-founded relation at every recursive call of f. These

TCCs correspond to the *Ranking Functions Criterion* that is widely used in verification systems to check termination of recursive program.

Example 3.1.4. *In PVS, the Ackermann function on two arguments can be defined as follows.*

Specification 3.1: PVS Specification for Ackermann

$\texttt{ackermann}(m, n : \texttt{nat}) : \texttt{RECURSIVE nat} =$
$\quad \texttt{IF } m = 0 \texttt{ THEN } n + 1$
$\quad \texttt{ELSIF } n = 0 \texttt{ THEN ackermann}(m - 1, 1)$
$\quad \texttt{ELSE ackermann}(m - 1, \texttt{ackermann}(m, n - 1))$
$\quad \texttt{MEASURE lex2}(m, n)\texttt{BY } <$

In this case, the type of the function is $[[\texttt{nat} \times \texttt{nat}] \to \texttt{nat}]$*, the measuring type is chosen as the type* `ordinal`*, and the measuring function is the* `lex2` *function that maps a pair of naturals* (m, n) *into the ordinal number* $m\omega + n$*, and the well-founded relation "*$<$*" on ordinals. PVS generates the following TCCs:*

Specification 3.2: TCCs for Ackermann in PVS

$\texttt{ackermann\_TCC1} : \texttt{OBLIGATION}$
$\quad \forall (m, n : \texttt{nat}) : \qquad n = 0 \land m \neq 0 \Rightarrow m - 1 \geq 0$

$\texttt{ackermann\_TCC2} : \texttt{OBLIGATION}$
$\quad \forall (m, n : \texttt{nat}) :$
$\qquad n = 0 \land m \neq 0 \Rightarrow \texttt{lex2}(m - 1, 1) < \texttt{lex2}(m, n)$

$\texttt{ackermann\_TCC3} : \texttt{OBLIGATION}$
$\quad \forall (m, n : \texttt{nat}) :$
$\qquad n \neq 0 \land m \neq 0 \Rightarrow m - 1 > 0$

$\texttt{ackermann\_TCC4} : \texttt{OBLIGATION}$
$\quad \forall (m, n : \texttt{nat}) :$
$\qquad n \neq 0 \land m \neq 0 \Rightarrow n - 1 > 0$

$\texttt{ackermann\_TCC5} : \texttt{OBLIGATION}$
$\quad \forall (m, n : \texttt{nat}) :$
$\qquad n \neq 0 \land m \neq 0 \Rightarrow \texttt{lex2}(m, n - 1) < \texttt{lex2}(m, n)$

$\texttt{ackermann\_TCC6} : \texttt{OBLIGATION}$
$\quad \forall (m, n : \texttt{nat}, f : [\{z : [\texttt{nat} \times \texttt{nat}] | \texttt{lex2}(z`1, z`2) < \texttt{lex2}(m, n)\} \to \texttt{nat}]) :$
$\qquad n \neq 0 \land m \neq 0 \Rightarrow \texttt{lex2}(m - 1, f(m, n - 1)) < \texttt{lex2}(m, n)$

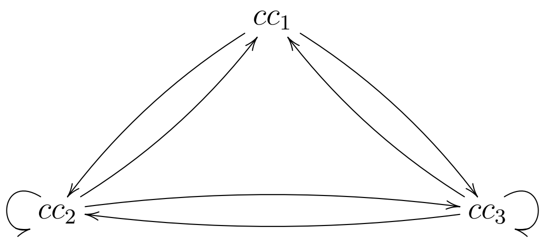
Proof obligations `ackermann_TCC1`, `ackermann_TCC3`, *and* `ackermann_TCC4` *are generated by PVS only to guarantee that the type of the arguments is indeed* `nat`*. The other proof obligations, namely* `ackermann_TCC2`, `ackermann_TCC5` *and* `ackermann_TCC6` *are the ones of interest, i.e., the termination conditions generated by PVS from the three recursive calls used in the definition. Providing the measure* `lex2` *in the specification all termination TCCs are automatically discharged by PVS. Hence, the PVS semantics guarantees that, since all TCCs are properly discharged, the function* `ackermann` *is well-defined in all inputs.*

3.2 The Size-Change Principle and Calling Context Graphs

The Size-Change Principle (SCP) introduced in [LJBA01] aims to state program termination by verifying that "every possibly infinite sequence of data exchanging causes an infinite descent over some data values regarding a well-founded relation". Thus, since such an infinite decrease is impossible, it is enough to state that there are no infinite sequences of data exchanging as in Definition 2.1.8. The SCP analysis is implemented by the Calling Context Graph (CCGs) technique introduced by [MV06]. The analyzed data values are usually the actual parameters of function calls, but the vertices of a CCG are not labeled just with the parameters, they are labeled with the CCs of the program given in Definition 3.1.1. The edges of the CCG connect pairs of vertices that for some input might give rise to a state transition (for which the conditions for the calls hold). Thus, the edges are given whenever the two adjacent vertices form a nested call.

Definition 3.2.1 (Graph of Calling Contexts)**.** *The Graph of Calling Contexts for a program P is the (over-approximated) digraph $G = (CCs(P), E)$, whose edges are pairs of vertices in $CCs(P)$ related by nested calls; that is, there is an edge $(\langle f, conds, g\rangle, \langle g, conds', h\rangle)$ in E iff there is a nested call $(f, \beta) \xrightarrow{\pi,n} (g, \beta')$, where π is the position of the function call related to the CC $\langle f, conds, g\rangle$ (i.e., $conds = CConds(\pi, e_f)$ and $e_f|_\pi$ is the call of the function g) and for all condition $e_c \in conds'$, $\chi(e_g, e_c, \beta', n) = TRUE$.*

Example 3.2.1 (Graph of Calling Contexts for Ackermann)**.** *Let cc_1, cc_2 and cc_3 be, respectively, the first, second and third CCs for e_{ack} in Example 3.1.1. Then, the Graph* *of Calling Contexts for Ackermann is depicted below.*



Notice that for any nested call $(ack, \beta) \xrightarrow{21,k} (ack, \beta')$*, since the condition* $= (n, 0) \in CConds(21, e_{ack})$ *holds for* β*, one has that* β' *maps* $n \mapsto 1$*, which implies that for all* $k > 0$*,* $\chi(e_{ack}, = (n, 0), \beta', k) = FALSE$*. Therefore, there is no edge from* cc_1 *to itself.*

Notice that termination by finite nested calls (Definition 2.1.8) can be stated as the non-existence of infinite sequences of CCs associated with feasible sequences of nested calls. In a graph of calling contexts, however, the existence of a nested call is checked only locally. Thus, every feasible sequence of nested calls is a path in the graph. But not every path in the graph corresponds to a feasible sequence of nested calls, since every element of the sequence must satisfy the definition of nested call.

To state that an infinite sequence of ("feasible") CCs produces an infinite decrease over a well-founded order, the relation between CCs along the paths of the Graph of Calling Contexts should be analyzed. Paths of interest for this analysis are *circuits* which are paths that end with a CC adjacent to the starting calling context; thus corresponding to possible execution loops. For establishing a decrease along paths, the structure of a Graph of Calling Contexts is enriched with a family of measure functions whose outputs will label every vertex of the graph.

Definition 3.2.2 (Family of Measures)**.** *A family of measures for a program* P *is a finite set of measure functions* $M = \{\mu_1, \ldots, \mu_k\}$ *over a well-founded set* $\mathcal{M}$*, on the parameters of functions in a program* P*.*

Example 3.2.2. *A family of measures for Ackermann in Example 2.1.1 might contain measures such as* $\mu_1(m, n) := m$*,* $\mu_2(m, n) := n$*,* $\mu_3(m, n) := lex(m, n)$*, etc. Since the program specifies a sole binary function, these measures work without any necessary adaptation to the parameters of all CCs.*

Definition 3.2.3 (Calling Context Graph - CCG)**.** *Let* $(CCs(P), E)$ *and* M *be a Graph of Calling Contexts and a family of measures for a given program* P*. Then,* $(CCs(P), E, M)$ *is a* Calling Context Graph *(CCG) for* P*.*

It can occur that one of the measures in the given family of measures does not provide a decrease over some parameters after each function call. Thus, allowing different measures brings an advantage, since it is possible to obtain the required decrease along paths in the graph by consistently comparing these measures along the edges of paths in the CCG.

Definition 3.2.4 (Measure Comparison Function)**.** *Let $(CCs(P), E, M)$ be a CCG for a program P, where M consists of measure functions over a well-founded set $\mathcal{M}$ with a well-founded order $\succ$. Over an edge $e = (\langle f, conds, g\rangle, \langle g, conds', h\rangle) \in E$ and pairs of measures $\mu_i, \mu_j \in M$, a* measure comparison function*, given as $\phi(\mu_i, \mu_j, e)$, is defined as:*

$$\phi(\mu_i, \mu_j, e) := \begin{cases} >, & \text{if } \forall(\beta) : conds\beta \Rightarrow \mu_i(x_1\beta, \ldots, x_{n_f}\beta) \succ \mu_j(y_1\beta', \ldots, y_{n_g}\beta'); \\ \geq, & \text{if } \forall(\beta) : conds\beta \Rightarrow \mu_i(x_1\beta', \ldots, x_{n_f}\beta) \succsim \mu_j(y_1\beta', \ldots, y_{n_g}\beta'); \\ \times, & \text{otherwise.} \end{cases}$$

Notice that for an edge $(cc1, cc2)$, the vertex $cc1$ brings already the function definition over which the second measure function is defined; i.e., $\phi(\mu_i, \mu_j, e)$ depends only on $cc1$. Thus, for short, the simplified notation $\phi_{i,j}(cc1)$ will be used.

The decrease along paths in the CCG of a program can then be verified through the application of the measure comparison function. Such function will allow combinations of different measures along with the graph.

Example 3.2.3 (Continuing Example 3.2.2)**.** *Notice that along all paths in the graph of Calling Contexts for Ackermann (Example 3.2.1), μ_3 decreases in each edge, while μ_1 and μ_2 only decrease in some edges of the graph; for instance μ_2 decreases along (cc_3, cc_3), but not along (cc_3, cc_2). Nevertheless, just using measures μ_1 and μ_2 decreasement along circuits $cc1, cc3, cc2$ and $cc3$ of the CCG for Ackermann can be guaranteed since for the former $\phi_{1,2}(cc1) =$ "$\geq$"; $\phi_{2,2}(cc3) =$ "$>$"; $\phi_{1,1}(cc2) =$ "$>$" and for the latter $\phi_{2,2}(cc3) =$ "$>$"; thus, the first and second parameters of Ackermann strictly decrease along the former and the latter circuit, respectively.*

Definition 3.2.5 (Measure Combination)**.** *Let $(CCs(P), E, M)$ be a CCG for a program P as in Definition 3.2.4 and $cc_1, cc_2, \ldots$ be a path in G, a* measure combination *for p is a sequence $\mu_1, \mu_2, \ldots$ of measure functions in M with the same length of the path, where for every i such that $1 \leq i$ and $i + 1$ bounded by the length of the sequence $\phi(\mu_i, \mu_{i+1}, cc_i) \neq$ "$\times$"; it is said to be a* decreasing *measure combination, if in addition, there exists some j with $1 < j$ and $j + 1$ bounded by the length of the sequence, such that $\phi(\mu_j, \mu_{j+1}, cc_j) =$ "$>$".*

Since the paths of the graph represent possible executions of the program, possible execution loops are given by the circuits of the graph. Thus, it is possible to state termination by analyzing these circuits and provide measure combinations for them.

Definition 3.2.6 (CCG Termination). *Let P be a program. Then P is said to be CCG terminating, denoted by $\mathcal{T}_\varrho(P)$, if there exists a finite family of measures M such that for every circuit of the graph $(CCs(P), E, M)$, there exists a decreasing measure combination starting and ending with the same measure.*

Example 3.2.4 (Continuing Example 3.2.3). *The measure combinations μ_1, μ_2, μ_1 and μ_2 are decreasing, respectively, for the circuits $cc1, cc3, cc2$ and $cc3$ of the CCG for the Ackermann function.*

Equivalence between the notion of termination by finite sequences of nested calls and the CCG Termination criterion is then established.

Lemma 3.2.1 (CCG Termination Equivalence). *For a given program P, $\mathcal{T}_\varrho(P)$ iff $\mathcal{T}_\nu(P)$.*

Proof. ($\Rightarrow$) Assume $\mathcal{T}_\varrho(P)$. Every circuit of a CCG G for P represents a sequence of CCs $cc_1, cc_2, \ldots, cc_n$ where n is the length of the circuit, thus, a sequence of nested calls. The decreasing measure combination for each circuit has the same measure function for the first and last vertice; thus, the result of the measure combination dictates the behavior of some specific parameters through the execution of P. Furthermore, this measure combination is decreasing; thus, if the sequence of nested calls represented by (possible repetitions of) this circuit were infinite, there would be infinitely occurrences of a strict decrease for a given parameter regarding a well-founded order, reaching a contradiction.

($\Leftarrow$) Assuming $\mathcal{T}_\nu(P)$, by Lemma 3.1.1 TCC termination holds. Then, there exist measure functions μ_f for each function f in P that decrease regarding a well-founded order $\succ$ for every calling context. Let the CCG $(CCs(P), E, M)$, where the family of measure functions M is a singleton consisting of μ built from these measures μ_f adapted accordingly to the parameters of each function f in P. Then, the decreasing measure combination for each circuit of the graph is a sequence of μ's of the length of the circuit, which would be decreasing (along each step of the circuit). Thus, f (and also P) is CCG terminating. $\square$

3.3 Dependency Pairs

The termination analysis for rewriting systems aims to verify the non-existence of infinite reduction steps (derivations) for every term over which the reduction relation is applied. To do this, the Dependency Pairs Termination Criterion, proposed in [AG97], analyzes the possible reductions in a term resulting from a previous reduction, i.e., those that can arise from defined symbols on the *rhs*'s of rules. Thus, it analyzes the *defined symbols* of a TRS E, i.e., the set given by $D_E = \{g \mid \exists (l \to r \in E) : \texttt{root}(l) = g\}$.

Definition 3.3.1 (Dependency Pairs (DPs))**.** *Let E be a TRS. The set of* Dependency Pairs *for E is given as*

$$DP(E) = \{\langle l, t\rangle \mid l \to r \in E \ \wedge \ r \trianglerighteq t \ \wedge \ \mathtt{root}(t) \in D_E\}$$

Example 3.3.1 (Dependency Pairs)**.** *The DPs for the TRS in Example 2.2.2 are:*

$\langle a(s(x), 0), a(x, s(0))\rangle$, *from the second rule;*

$\langle a(s(x), s(y)), a(x, a(s(x), y))\rangle$, *from the third rule at the root position of the rhs;*

$\langle a(s(x), s(y)), a(s(x), y)\rangle$, *from the third rule at position* 1 *of the rhs.*

Standard definitions of DPs substitute defined symbols by new *tuple symbols* (or *marked symbols*), i.e., symbols that are not interpreted as function symbols, to avoid (innermost) reductions at the root positions, which is required for the analysis of termination. Using such tuple symbols (or marked defined symbols) is convenient when using polynomial interpretations, since it allows given different interpretations to the defined symbols and their associated tuple symbols (e.g., [AG00], [TG03]). For the main purpose of this work, and for relating the DP Criterion with other termination criteria (available in the PVS theory `PVS0`), the flexibility allowed by tuple symbols would not be required. In the current formalization, instead of extending the language with such tuple symbols, DPs are built with unmarked symbols of the original signature. Reductions at the root position are avoided through the restriction to non-root (innermost) derivations. This choice will be made clearer in Chapter 4, but it avoids the need to extend the signature of the TRSs.

Each DP represents the possibility of a future reduction after one (innermost) reduction step. However, distinct rewriting redexes can appear in terms after (possibly) several (innermost) reduction steps, which can also give rise to another possible reduction, producing a *Dependency Chain.*

Definition 3.3.2 (Dependency Chain)**.** *A* dependency chain *for a TRS E, E-chain, is a finite or infinite sequence of DPs $\langle s_1, t_1\rangle, \langle s_2, t_2\rangle \ldots$ for which there exists a substitution σ such that $t_i\sigma \xrightarrow{>\lambda}{}^* s_{i+1}\sigma$, for every i below the length of the sequence, after renaming the variables of pairs with disjoint new variables.*

Example 3.3.2 (Dependency Chain)**.** *A dependency chain built using the second DP in the Example 3.3.1 is given by:*

$$\langle a(s(x), s(y)), a(x, a(s(x), y))\rangle, \langle a(s(x), s(y)), a(x, a(s(x), y))\rangle$$

since $a(s(0), a(s^2(0), 0)) \xrightarrow{>\lambda}{}^* a(s(0), s(a(s(0), 0)))$.

Similarly, the notion of *Innermost Dependency Chain* is given:

Definition 3.3.3 (Innermost Dependency Chain)**.** *An* innermost dependency chain *to a TRS E, E-in-chain, is a finite or infinite sequence of DPs $\langle s_1, t_1\rangle\langle s_2, t_2\rangle \ldots$ for which there exists a substitution σ such that, for every i below the length of the sequence, $t_i\sigma \xrightarrow{>\lambda}{}^{*}_{in} s_{i+1}\sigma$ and $nf(\xrightarrow{>\lambda})(s_i)$, after renaming the variables of pairs with disjoint new variables.*

Definition 3.3.4 ((Innermost) Dependency Pairs Termination Criterion)**.** *Termination of a TRS E by the DP Termination Criterion, (denoted as $\mathcal{T}_{DP}(E)$) and shortened as DP Criterion) is then defined as the absence of infinite dependency chains (cf., Theorems 3.2 and 4 of [AG97]) regarding E. Similar definition is used for innermost rewriting which is denoted by $\mathcal{T}_{DP_{in}}(E)$).*

Indeed, to show the absence of such infinite chains is a more flexible criterion than showing decreasing after each reduction step. It is possible to provide a simpler measure that eventually decreases after some steps of reduction, but that never increases.

Example 3.3.3 (DP Termination for GCD)**.** *Consider the following TRS for obtaining the greater common divisor (GCD) between naturals (not simultaneously null). For simplicity, the addition symbol is considered built-in.*

$$
\begin{aligned}
&gcd(0, s(y)) \to s(y)\\
&gcd(s(x), 0) \to s(x)\\
&gcd(s(x+y), s(x)) \to gcd(y, s(x))\\
&gcd(s(x), s(x+s(y))) \to gcd(s(x+s(y)), s(x))
\end{aligned}
$$

To show Noetherianity of this TRS it suffices to use a lexicographic ordering for the arguments of gcd in the lhs's and rhs's of the rules. However the simple addition of the arguments of gcdcan also be used. The TRS has the following DPs:

$$
\begin{aligned}
&\langle gcd(s(x+y), s(x)),\ gcd(y, s(x))\rangle \text{ from the third rule}\\
&\langle gcd(s(x), s(x+s(y))),\ gcd(s(x+s(y)), s(x))\rangle \text{ from the fourth rule}
\end{aligned}
$$

This measure decreases for the lhs and rhs of the first DP but remains the same for this comparison with the second DP.

Notice that consecutive DPs in a dependency chain may be only of the forms below.

- $\langle gcd(s(x+y), s(x)), gcd(y, s(x))\rangle\langle gcd(s(x+y_1), s(x)), gcd(y_1, s(x))\rangle$, *for* $y = s(x+y_1)$;
- $\langle gcd(s(x+y), s(x)), gcd(y, s(x))\rangle$ $\langle gcd(s(x_1), s(x_1+s(y_1))), gcd(s(x_1+s(y_1)), s(x_1))\rangle$, *for* $y = s(x_1)$ *and* $x = x_1+s(y_1)$;
- $\langle gcd(s(x), s(x+s(y))), gcd(s(x+s(y)), s(x))\rangle$ $\langle gcd(s(x+y_1), s(x)), gcd(y_1, s(x))\rangle$, *for* $y_1 = s(y)$

Additionally, notice that the second DP can not be followed by itself in a dependency chain because this would require consecutive DPs of the form $\langle gcd(s(x), s(x + s(y))), gcd(s(x + s(y)), s(x))\rangle\langle gcd(s(x_1), s(x_1 + s(y_1))), gcd(s(x_1 + s(y_1)), s(x_1))\rangle$, *for* $x = x_1 + s(y_1)$ *and* $x_1 = x + s(y)$, *which is not possible.*

Noetherianity is then concluded noticing that for any possible consecutive DPs in a chain, this measure decreases for the lhs of the first and the rhs of the second DP.

- *for the first case,* $y_1 + s(x) < s(x + s(x + y_1)) + s(x)$*;*
- *for the second case,* $s(x) + y < s(x + y) + s(x)$*;*
- *for the third case:* $s(y) + s(x) < s(x) + s(x + s(y))$*.*

Thus, no infinite dependency chain can be obtained.

A generalization of rewriting, allowing to restrict the redexes to terms in normal form for a specific set of rewriting rules can be used to subsume the cases of ordinary and innermost reductions. This generalization, introduced in [ST10], is given by the relation Q-restricted in Definition 3.3.5.

Definition 3.3.5 (Q-restricted relation)**.** *For TRSs* E *and* Q*, the* Q*-restricted relation, denoted as* $\xrightarrow{Q}_E$ *, is defined as* $s \xrightarrow{Q}_E t$ *iff* $s \to_E t$ *at some position* π *such that proper subterms of* $s|_\pi$ *are normal regarding* Q*; In this work, this relation also has its variants for the reduction in a specific position (*$\xrightarrow{Q;\pi}_E$ *) and the non-root case (*$\xrightarrow{Q;\pi\neq\emptyset}_E$ *).*

Note that the relations $\xrightarrow{\emptyset}_E$ and $\xrightarrow{E}_E$ correspond respectively to the ordinary and the innermost reduction relations [GTSK05b]. This allows to use one single formalization which can be used both the ordinary and innermost reduction relations.

In addition to the ordinary ($\xrightarrow{\emptyset}_E$) and innermost cases ($\xrightarrow{E}_E$), the Q-restricted relation also allows us to deal with interesting examples in which rewriting is applied in a modular manner, such as using rules which evaluation lead a boolean value (and the rules used by them) used in guards of branching expressions. Then the rules regarding the guards can be analyzed separately.

Example 3.3.4 (A TRS for Arithmetic)**.** *This TRS considers some rules to deal with the logic operators that are omitted for simplicity. For instance, transformations from conditions such as* $\neg(=(x, y))$ *into inequalities* $\leq$ *or* $>$ *(e.g.,* $x < y \vee y > x$*).*

R_1	$\leq(0,x) \rightarrow \top$	R_8	$+(s(x),y) \rightarrow s(+(x,y))$
R_2	$>(s(x),s(y)) \rightarrow >(x,y)$	R_9	$>(s(x),x) \rightarrow \top$
R_3	$\leq(s(x),s(y)) \rightarrow \leq(x,y)$	R_{10}	$\leq(x,+(x,y)) \rightarrow \top$
R_4	$>(s(x),0) \rightarrow \top$	R_{11}	$>(s(+(x,y),y)) \rightarrow \top$
R_5	$\stackrel{?}{=}(x,x) \rightarrow \top$	R_{12}	$-(x,x) \rightarrow 0$
R_6	$\top \wedge \top \rightarrow \top$	R_{13}	$-(x,0) \rightarrow x$
R_7	$+(0,x) \rightarrow x$	R_{14}	$-(s(x),s(y)) \rightarrow -(x,y)$

This TRS can be separated into two subset of rules, a subset Q regarding only arithmetic relations, successor operator "s" and the logical connective $\wedge$, and a subset E containing the rules involving other arithmetic operators, such as $+$ and $-$. These two subsets conform two separated TRSs which can have their desired properties analyzed separately.

Q	E
$R_1 \leq (0,x) \rightarrow \top$	$R_7 + (0,x) \rightarrow x$
$R_2 > (s(x),s(y)) \rightarrow > (x,y)$	$R_8 + (s(x),y) \rightarrow s(+(x,y))$
$R_3 \leq (s(x),s(y)) \rightarrow \leq (x,y)$	$R_{10} \leq (x,+(x,y)) \rightarrow \top$
$R_4 > (s(x),0) \rightarrow \top$	$R_{11} > (s(+(x,y),y)) \rightarrow \top$
$R_5 \stackrel{?}{=} (x,x) \rightarrow \top$	$R_{12} - (x,x) \rightarrow 0$
$R_6 \top \wedge \top \rightarrow \top$	$R_{13} - (x,0) \rightarrow x$
$R_9 > (s(x),x) \rightarrow \top$	$R_{14} - (s(x),s(y)) \rightarrow -(x,y)$

In order to ensure that there are no infinite dependency chains for a TRS, it is enough to provide a well-founded weakly monotonic ordering closed under substitution over the rules and DPs of the TRS. With such ordering there must be a strict decrease over the *lhs* and *rhs* of each DP as long as there is no increase from the *lhs* to the *rhs* of every rule of the TRS (cf., Theorem 4.1 of [AG97]). This analysis allows the development of automated analysis to state termination of TRSs.

Other than being a very flexible and powerful criterion, several refinements have been developed to automate analysis of termination by DPs, such as (innermost) Dependency Graphs and means to approximate them, Argument Filtering, Narrowing of DPs and Usable Rules, and the Subterm Criterion [AG00, HM03, HM04].

Chapter 4

Specification of DPs for TRSs and Termination Criteria for `PVS0`

This Chapter presents details on the specifications of the PVS libraries `TRS` and `PVS0`. Initially, the extension with DPs of the TRS library is discussed and then, the specification of termination criteria for the functional language `PVS0`. When no confusion arises, some typing notation will be omitted.

4.1 Extension of TRS with DPs

This section presents the extension of the PVS term rewriting library `TRS` with termination criteria based on the notion of DPs that were published in [AAAR20]. This library is a development that already contains the basic elements of abstract reduction systems and TRS, such as reducibility, confluence and Noetherianity regarding a given relation, notions of subterms and replacement, etc. Furthermore, this theory embraces several elaborate formalizations regarding such systems, such as the confluence of abstract reduction systems (see [GAR08]), the Critical Pair Theorem (see [GAR10]) and orthogonal TRSs and their confluence (see [ROGAR17]).

Terms in the library `TRS` are specified in theory `term.pvs` as a datatype with three parameters: nonempty types for variables and function symbols, and the arity function of these symbols. Terms are either variables or applications built as function symbols with a sequence of terms of length equal to its arity. The predicate `app?` holds for application terms and, as previously mentioned, the operator `root` extracts the root function symbol of an application.

The theory `rewrite_rules.pvs` specifies rewrite rules as pairs of terms following the Definition 2.2.1 and the notion of a set of defined symbols for a set of rewrite rules E (i.e., D_E used in Section 3.3) given as predicate `defined?` in Specification 4.1.

Specification 4.1: Predicate for defined symbols.

$\texttt{defined?}(E)(d) = \exists (e \in E) : \texttt{root}(\mathit{lhs}(e)) = d$

Basic elements and results were imported in this formalization, such as aforementioned terms, rules and predicates to represent pertinence of positions of a term (`positonsOF` found in theory `positions.pvs`), functions to obtain the subterm of a specific position (`subtermOF` in theory `subterm.pvs`), the replacement operation (`replaceTerm` in theory `replacement.pvs`) and so on. However, specification of some general definitions regarding TRS's required to specify DPs and formalization of several properties were missing and filled in as part of this work. Some of these new basic notions and results were included either in existing theories, such as the notion of non-root reduction ($\xrightarrow{>\lambda}$) specified in theory `reduction.pvs`, or in new complementary basic theories to deal with specialized reductions, such as `innermost_reduction.pvs` and `restricted_reduction.pvs`, where the relations $\rightarrow_{in}$ and $\underset{t}{\rightarrow}$ (Definitions 2.2.9 and 2.2.10) are found.

Furthermore, the new basic definitions, such as `non_root_reduction?` ($\xrightarrow{>\lambda}$ of Definition 2.2.8) are, mostly, specializations of previously existing ones, such as the notions presented by predicates `reduction_fix?` and `reduction?` (see Specification 4.2), which respectively specify the predicates for relations $\xrightarrow{\pi}$ and $\rightarrow$ given by Definition 2.2.6 (in theory `reduction.pvs`).

Specification 4.2: Predicates for the $\xrightarrow{\pi}$, $\rightarrow$ and $\xrightarrow{>\lambda}$ relations.

$\texttt{reduction\_fix?}(E)(s, t, (\pi \in Pos(s))) =$
$\quad \exists (e \in E, \sigma) : s|_{\pi} = \mathit{lhs}(e)\sigma \wedge t = s[\pi \leftarrow \mathit{rhs}(e)\sigma]$

$\texttt{reduction?}(E)(s, t) =$
$\quad \exists (\pi \in Pos(s)) : \texttt{reduction\_fix?}(E)(s, t, \pi)$

$\texttt{non\_root\_reduction?}(E)(s, t) =$
$\quad \exists (\pi \in Pos(s) |\ \pi \neq \lambda) : \texttt{reduction\_fix?}(E)(s, t, \pi)$

Notice that such relations are specified as predicates over pairs of terms in a Curried way, a discipline followed through the whole `TRS` library that allows one to rely on, for instance, parameterizable definitions and properties provided for arbitrary abstract reductions systems, such as closures of relations (in theory `relations_closure.pvs`), reducibility and normalization (in theory `ars_terminology.pvs`), Noetherianity (in theory `Noetherian.pvs`), etc.

The new required relations given in Definition 2.2.9 are available in theory `innermost_reduction.pvs` and are specified as the predicates `innermost_reduction_fix?` ($\xrightarrow{\pi}_{in}$), `innermost_reduction?` ($\rightarrow_{in}$) and `non_root_innermost_reduction?` ($\xrightarrow{>\lambda}_{in}$) in Specification 4.3.

Specification 4.3: Predicates for the $\xrightarrow{\pi}_{in}$, $\rightarrow_{in}$ and $\xrightarrow{>\lambda}_{in}$ relations.

---

innermost_reduction_fix?$(E)(s, t, (\pi \in Pos(s))) =$
  is_normal_form?(non_root_reduction?$(E))(s|_{\pi}) \wedge$
  reduction_fix?$(E)(s, t, \pi)$

innermost_reduction?$(E)(s, t) =$
  $\exists(\pi \in Pos(s))$: innermost_reduction_fix?$(E)(s, t, \pi)$

non_root_innermost_reduction?$(E)(s, t) =$
  $\exists(\pi \in Pos(s)|\ \pi \neq \lambda)$:
    is_normal_form?(non_root_reduction?$(E))(s|_{\pi}) \wedge$
    reduction_fix?$(E)(s, t, \pi)$

The notion of $\underset{s}{\rightarrow}$ in Definition 2.2.10 is given in Specification 4.4 as `rest?` for any binary relation R in theory `restricted_reduction.pvs`. A specialization of restricted relations for term rewriting is given by `arg_rest?` also in Specification 4.4, allowing to fix the argument where innermost reductions can take place between given descendants of a term s (i.e., relation $\underset{s}{\xrightarrow{\pi}}_{in}$), which is specified in theory `innermost_reduction.pvs`. The function `first(`π`)` returns the first element of the sequence of naturals given by the position π.

Specification 4.4: Predicates for the $\underset{s}{\rightarrow}$ and $\underset{s}{\xrightarrow{\pi}}_{in}$ relations.

---

rest?$(R, s)(u, v) =$
  $(s\, R^*\, u) \wedge (u\, R\, v)$

arg_rest?$(E)(s)(k)(u, v) =$
  rest?$(\xrightarrow{>\lambda}_{in}, s)(u, v) \quad \wedge$
  $\exists(\pi \in Pos(s)|\ \pi \neq \lambda) : first(\pi) = k \wedge$
    innermost_reduction_fix?$(E)(u, v, \pi)$

Notice that this restriction on non-root innermost rewriting follows the previously mentioned discipline of Currying and modularity of `TRS`, allowing generic application of rewriting predicates and their properties over general rewriting relations. In this specification, i.e., `arg_rest?`, the predicate `rest?` receives as parameter the relation $\xrightarrow{>\lambda}_{in}$, i.e., the terms related by this relation are only those that, besides having an innermost reduction, are also derivations from a given term s.

In theory `dependency_pairs.pvs` are specified the notion of DP given in Definition 3.3.1 (as predicate `dep_pair?` in Specification 4.5) and its termination criterion given in Definition 3.3.4. As previously mentioned, instead of extending the language with tuple symbols, DPs are specified with the same language of the given signature, and thus DPs chained through non-root (innermost) derivations.

Specification 4.5: Predicate for DPs as pairs of terms.

$\texttt{dep\_pair?}(E)(s,t) =$
  $\texttt{app?}(t) \wedge$
  $\texttt{defined?}(E)(f(t)) \wedge$
  $\exists(e \in E) : lhs(e) = s \wedge$
  $\exists(\pi \in Pos(rhs(e))) : rhs(e)|_\pi = t$

This specification of DPs follows the standard theoretical approach straightforwardly. However, it is stated over two existential quantifiers that, throughout the proofs, would bring several difficulties about which rule and position had created the DP being analyzed. This is because, due to the PVS proof calculus, whenever these existential quantifiers appear in the antecedent of a sequent in a proof, their Skolemization leads to some arbitrary rule and position being chosen, making it difficult to construct derivations of terms associated with chained DPs. It is easy to see that different *rhs* positions, and even different rules can produce identical DPs; take, for instance, the TRS below, where $\langle h(x,y), g(x,y)\rangle$ can be built in three different manners.

$$\{h(x,y) \to h(g(x,y), g(g(x,y),y), \quad h(x,y) \to g(x,y), \quad g(x,y) \to y\}$$

To discriminate how DPs are extracted from the rewrite rules and to circumvent the difficulties of existential quantifiers, an alternative notion of DP is provided in Specification 4.6 as predicate `dep_pair_alt?`.

Specification 4.6: Predicate for DPs as a pair of rule and position at its *rhs*.

$\texttt{dep\_pair\_alt?}(E)(e,\pi) =$
  $e \in E \wedge$
  $\pi \in Pos(rhs(e)) \wedge$
  $\texttt{app?}(rhs(e)|_\pi) \wedge$
  $\texttt{defined?}(E)(f(rhs(e)|_\pi)$

Having the rule and position that generate the DPs allows, for instance, specification of recursive functions to easily adjust and accumulate the contexts of any infinite chain of DPs in order to build the associated infinite derivations (more details are given in Section 5.1). Here, it is important to stress that for termination analysis and automation, whenever $\texttt{dep\_pair\_alt?}(E)(e,\pi)$ and $\texttt{dep\_pair\_alt?}(E)(e',\pi')$ are such that $lhs(e) = lhs(e')$ and $rhs(e)|_\pi = rhs(e')|_{\pi'}$, it is sufficient to consider only one of these DPs.

In the remainder of the discussion, these two definitions will be distinguished if necessary, and for the sake of simplicity, the first and second elements of a DP will be identified with the *lhs* of the rule and the subterm at position π of the *rhs* of the rule.

Notice that both specifications for DPs are Curried, allowing the definition of the types `dep_pair`(E) and `dep_pair_alt`(E).

To check that an infinite sequence of DPs forms an infinite (innermost) dependency chain, it is required, as given in Definitions 3.3.2 and 3.3.3, that every pair of consecutive DPs in this sequence be related through (innermost) non-root reductions, after renaming their variables, regarding some substitution. This gives rise to an imprecision since the type of substitutions does not allow infinite domains, as discussed in [Ste10]. This issue is circumvented by specifying sequences of DPs in association with sequences of substitutions. Thus, by allowing a different substitution for each DP in the sequence, it is possible to specify the notion of *(innermost) chained DPs* (predicates `inn_chained_dp?` and `chained_dp?` in Specification 4.7).

Specification 4.7: Predicates for (innermost) chained DPs.

$\texttt{chained\_dp?}(E)(dp_1, dp_2 : \texttt{dep\_pair}(E))(\sigma_1, \sigma_2) =$

$\quad dp_1' 2\sigma_1 \xrightarrow{>\lambda}{}^{*}\, dp_2' 1\sigma_2$

$\texttt{inn\_chained\_dp?}(E)(dp_1, dp_2 : \texttt{dep\_pair}(E))(\sigma_1, \sigma_2) =$

$\quad \texttt{is\_nr\_normal\_form?}(E)(dp_1' 1\sigma_1) \wedge$

$\quad \texttt{is\_nr\_normal\_form?}(E)(dp_2' 1\sigma_2) \wedge$

$\quad dp_1' 2\sigma_1 \xrightarrow{>\lambda}{}^{*}_{in}\, dp_2' 1\sigma_2$

In Specification 4.7, the elements of a DP, say dp, are projected by the operator $\_'\_$, as $dp'1$ and $dp'2$, used to project elements of tuples in PVS. Using these specifications of (innermost) chained DPs, whenever predicates `infinite_dep_chain?` and `inn_infinite_dep_chain?` in Specification 4.8 hold for a pair of a sequence of DPs and substitutions, such pair is said to be an infinite (innermost) dependency chain.

Specification 4.8: Predicates for infinite (innermost) Dependency Chains.

$\texttt{infinite\_dep\_chain?}(E)(dps : \texttt{sequence}[\texttt{dep\_pair}(E)], \sigma : \texttt{sequence}[Sub]) =$

$\quad \forall (i : nat) :$

$\qquad \texttt{chained\_dp?}(E)(dps(i), dps(i+1))(\sigma(i), \sigma(i+1))$

$\texttt{inn\_infinite\_dep\_chain?}(E)(dps : \texttt{sequence}[\texttt{dep\_pair}(E)], \sigma : \texttt{sequence}[Sub]) : bool =$

$\quad \forall (i : nat) :$

$\qquad \texttt{inn\_chained\_dp?}(E)(dps(i), dps(i+1))(\sigma(i), \sigma(i+1))$

Finally, the (innermost) DP Criterion is specified as the absence of such infinite chains in Specification 4.9, where the two first predicates specify the criterion for the standard notion of DPs (Specification 4.5), and the third and fourth ones for the alternative one (Specification 4.6). Notice that alternative DPs are translated into standard DPs in the third and fourth predicates.

Specification 4.9: Predicates for (innermost) termination for the two specifications of DPs.

$\texttt{dp\_termination?}(E) =$

$\quad\forall(dps : \texttt{sequence}[\texttt{dep\_pair}(E)], \sigma : \texttt{sequence}[Sub]) :$

$\quad\quad\neg\texttt{infinite\_dep\_chain?}(E)(dps, \sigma)$

$\texttt{inn\_dp\_termination?}(E) =$

$\quad\forall(dps : \texttt{sequence}[\texttt{dep\_pair}(E)], \sigma : \texttt{sequence}[Sub]) :$

$\quad\quad\neg\texttt{inn\_infinite\_dep\_chain?}(E)(dps, \sigma)$

$\texttt{dp\_termination\_alt?}(E) =$

$\quad\forall(dps\_alt : \texttt{sequence}[\texttt{dep\_pair\_alt}(E)], \sigma : \texttt{sequence}[Sub]) :$

$\quad\quad\texttt{LET}\ dps = \texttt{LAMBDA}(i : nat) : (lhs(dps\_alt(i)'1), rhs(dps\_alt(i)'1)|_{dps\_alt(i)'2})\texttt{IN}$

$\quad\quad\neg\texttt{infinite\_dep\_chain?}(E)(dps, \sigma)$

$\texttt{inn\_dp\_termination\_alt?}(E) =$

$\quad\forall(dps\_alt : \texttt{sequence}[\texttt{dep\_pair\_alt}(E)], \sigma : \texttt{sequence}[Sub]) :$

$\quad\quad\texttt{LET}\ dps = \texttt{LAMBDA}(i : nat) : (lhs(dps\_alt(i)'1), rhs(dps\_alt(i)'1)|_{dps\_alt(i)'2})\texttt{IN}$

$\quad\quad\neg\texttt{inn\_infinite\_dep\_chain?}(E)(dps, \sigma)$

4.2 PVS0

PVS0 is a simple first-order functional language. It is specified in such a way that it is expressive enough to specify recursive functions while reduces the cases to be analyzed when performing formal proofs. It just allows constants, a single variable symbol, unary and binary built-in operators, recursive calls and a branching instruction (`if-then-else`). Its syntax is specified as the datatype `PVS0Expr` over a non-empty type `T`, given as parameter, and its grammar is:

$$expr := \texttt{cnst}(v) \mid vr \mid \texttt{op1}_i(expr) \mid \texttt{op2}_i(expr, expr) \mid \texttt{rec}(expr) \mid \texttt{ite}(expr, expr, expr)$$

The constants are fixed elements over type `T`, i.e.: $v : \texttt{T}$. The indices i on unary and binary operators are provided to choose one of the available built-in unary and binary operators (given as lists O_1 and O_2 of well-defined functions with types `T -> T` and `[T,T] -> T`), respectively. For each `PVS0` program, it is necessary to define a constant $\perp$ of type `T` that represents the false value used in the branching instruction. These three elements (lists for unary and binary operators and false value) compose the *evaluation environment* for a program.

Example 4.2.1 (`PVS0` code for the Ackermann function). *The definition of the Ackermann function is specified below using the datatype* `PVS0Expr` *over the type* `[nat,nat]`*, where the value chosen as false is* $\langle 0, 0\rangle$ *and unary and binary operators are specified for parameters* v, v' : `[nat,nat]` *as the lists below (given in PVS):*

$$
\begin{aligned}
ack\_\texttt{op1} = \quad & (\texttt{op1}_0(v) = \texttt{[nat,nat]} = \texttt{IF } v\grave{}1 = 0 \texttt{ THEN } \langle 1,1\rangle \texttt{ELSE } \langle 0,0\rangle,\\
& \texttt{op1}_1(v) = \texttt{[nat,nat]} = \langle v\grave{}2+1, \_\rangle,\\
& \texttt{op1}_2(v) = \texttt{[nat,nat]} = \texttt{IF } v\grave{}2 = 0 \texttt{ THEN } \langle 1,1\rangle \texttt{ELSE } \langle 0,0\rangle,\\
& \texttt{op1}_3(v) = \texttt{[nat,nat]} = \texttt{IF } v\grave{}1 \neq 0 \texttt{ THEN } \langle v\grave{}1-1, 1\rangle \texttt{ELSE } v,\\
& \texttt{op1}_4(v) = \texttt{[nat,nat]} = \texttt{IF } v\grave{}2 \neq 0 \texttt{ THEN } \langle v\grave{}1, v\grave{}2-1\rangle \texttt{ELSE } v)\\
ack\_\texttt{op2} = \quad & (\texttt{op2}_0(v, v\grave{}) = \texttt{[nat,nat]} = \texttt{IF } v\grave{}1 \neq 0 \texttt{ THEN } \langle v\grave{}1-1, (v\grave{})\grave{}1\rangle \texttt{ELSE } v)
\end{aligned}
$$

$$
\begin{aligned}
ack\_pvs0 = \texttt{ite}(\; & \texttt{op1}_0(vr),\\
& \texttt{op1}_2(vr),\\
& \texttt{ite}(\texttt{op1}_1(vr),\\
& \quad \texttt{rec}(\texttt{op1}_3(vr)),\\
& \quad \texttt{rec}(\texttt{op2}_0(vr, rec(\texttt{op1}_4(vr))))))
\end{aligned}
$$

The definition, `def`, of a program consists of the evaluation environment and the expression e_f specifying the function: `def = (`$\perp$`, `O_1`, `O_2`, `e_f`)` . The type of PVS0 definitions is `PVS0` and the type of `PVS0` expressions is `PVS0Expr`. In the formalization, the variable `pvs0` is used both as a `PVS0` definition and as a `PVS0` expression. For the case of the Ackermann function, the definition is given as

$$ack\_def = (\langle 0, 0\rangle, ack\_\texttt{op1}, ack\_\texttt{op2}, ack\_pvs0)$$

Since `PVS0` expressions can have at most three arguments when they are rooted by a branching instruction (*ite*), naturals in paths are bounded by 2. The recursive predicate `valid_path`, given as $\mathcal{P}$ in Specification 4.10 specifies Definition 2.1.1 by checking if a given `path` is valid for a given PVS0 expression `expr`. Notice that paths are built as list in reverse order[1], then to follow a path in a `PVS0` expression, instead of using the usual list function *car* to select the first element of a list, the function *rac* is used to select the last element of the path. The function *rdc* is analogously defined, regarding the standard *cdr* function on lists. Functions get_arg, and get_arg1 and get_arg2, and get_cond, get_if and get_else are used, respectively, as operators to access the argument of unary and recursive expressions, the first and second arguments of binary operator expressions, and the guard, the if and the *else* branch expressions branching instructions. Additionally, the operator *get_op* is used to access the index for unary and binary operator expressions, and *get_val* to access values of variable and constant expressions. These functions are given as part of the specification of the abstract datatype `PVS0Expr`.

The function `subterm_at` in Specification 4.11 extracts the subexpression at a valid path, `path`, of a `PVS0` expression, `expr`.

Specification 4.10: Paths of `PVS0` expressions

$$
\begin{array}{llll}
\mathcal{P}(\texttt{expr})(\texttt{path}) = & \texttt{CASES} & \texttt{expr} & \texttt{OF} \\
& & \texttt{vr}: & null?(\texttt{path}), \\
& & \texttt{cnst}: & null?(\texttt{path}), \\
& & \texttt{op1}: & null?(\texttt{path}) \quad \vee \ (rac(\texttt{path}) = 0 \wedge \mathcal{P}(get\_arg(\texttt{expr}))(rdc(\texttt{path}))), \\
& & \texttt{op2}: & null?(\texttt{path}) \quad \vee \ (rac(\texttt{path}) = 0 \wedge \mathcal{P}(get\_arg1(\texttt{expr}))(rdc(\texttt{path}))), \\
& & & \qquad\qquad\qquad \vee \ (rac(\texttt{path}) = 1 \wedge \mathcal{P}(get\_arg2(\texttt{expr}))(rdc(\texttt{path}))), \\
& & ite: & null?(\texttt{path}) \quad \vee \ (rac(\texttt{path}) = 0 \wedge \mathcal{P}(get\_cond)(rdc(\texttt{path}))), \\
& & & \qquad\qquad\qquad \vee \ (rac(\texttt{path}) = 1 \wedge \mathcal{P}(get\_if)(rdc(\texttt{path}))), \\
& & & \qquad\qquad\qquad \vee \ (rac(\texttt{path}) = 2 \wedge \mathcal{P}(get\_else)(rdc(\texttt{path}))), \\
& & \texttt{rec}: & null?(\texttt{path}) \quad \vee \ (rac(\texttt{path}) = 0 \wedge \mathcal{P}(get\_arg(\texttt{expr})(rdc(\texttt{path})))
\end{array}
$$

[1]to ease formalizations and readability, since in this way the leaf subterms are at the end of the list, allowing a more intuitive construction of `path_conditions` (Specification 4.13), for instance.

Specification 4.11: Subterms at paths of PVS0 expressions

$$
\begin{array}{lll}
\texttt{subterm\_at}(\texttt{expr},\texttt{path}) = & \texttt{IF} & null?(\texttt{path})\ \texttt{THEN}\ \texttt{expr} \\
& \texttt{ELSE} & \texttt{CASES}\ \texttt{expr}\texttt{OF} \\
& & \quad \texttt{vr}: \quad \texttt{expr} \\
& & \quad \texttt{cnst}: \quad \texttt{expr} \\
& & \quad \texttt{op1}: \quad \texttt{subterm\_at}(get\_arg(\texttt{expr}), rdc(\texttt{path})), \\
& & \quad \texttt{op2}: \quad \texttt{IF}\ rac(\texttt{path}) = 0 \\
& & \qquad\qquad \texttt{THEN}\ \texttt{subterm\_at}(get\_arg1(\texttt{expr}), rdc(\texttt{path})) \\
& & \qquad\qquad \texttt{ELSE}\ \texttt{subterm\_at}(get\_arg2(\texttt{expr}), rdc(\texttt{path})) \\
& & \quad \texttt{ite}: \quad \texttt{subterm\_at}(\texttt{IF}\ rac(\texttt{path}) = 0\texttt{THEN}\ get\_cond \\
& & \qquad\qquad\qquad \texttt{ELSE IF}\ rac(\texttt{path}) = 1\texttt{THEN}\ get\_if \\
& & \qquad\qquad\qquad \texttt{ELSE}\ get\_else, rdc(\texttt{path})) \\
& & \quad \texttt{rec}: \quad \texttt{subterm\_at}(get\_arg(\texttt{expr}), rdc(\texttt{path}))
\end{array}
$$

To illustrate the previous operators, consider the expression *ack_expr* for the Ackermann function in Example 4.2.1. Both $\mathcal{P}(ack\_expr)((0,2,2))$ and $\mathcal{P}(ack\_expr)((1,2))$ hold, and both equations $\texttt{subterm\_at}(ack\_expr)((0,2,2)) = \texttt{op2}_0(\texttt{vr}, \texttt{rec}(\texttt{op1}_4(\texttt{vr}))$ and $\texttt{subterm\_at}(ack\_expr)((1,2)) = \texttt{rec}(\texttt{op1}_3(\texttt{vr}))$, also hold, but $\mathcal{P}(ack\_expr)((1,1,0))$ does not hold.

Recursive calls in a given program `pvs0`, are specified through the execution paths that lead to their evaluation in the tree representation of the program. For each recursive call in the program, the specification uses the path leading to the call both to reach the argument of the recursive call and to build the list of the conditions leading to the recursive call. These conditions have a type defined as a list of Boolean expressions `PVS0Bool[T]`, that can be interpreted as true or false (`pvs0bool` or `pvs0not`, respectively). As usual, paths in programs are paths in expressions defined as lists of naturals.

Recursive calls in a `PVS0` program are specified as *calling contexts* whose type are triples consisting of `PVS0` recursive expressions, for the recursive call itself; lists of Boolean expressions of type `PVS0Bool[T]`, for its conditions; and the `path` leading to the recursive call. The type of calling contexts is then given as:

Specification 4.12: Type for PVS0 expressions

$\texttt{PVS0Expr\_CC}$: $TYPE = [\#\ rec\_expr : (rec?),\ cnds : Conditions,\ path : Path\ \#]$

Example 4.2.2 (Calling Contexts for Ackermann in PVS0). *Considering the* `PVS0` *code for Ackermann in Example 4.2.1, the CCs for it are:*

$$
\begin{array}{lll}
(\texttt{rec}(\texttt{op1}_3(\texttt{vr})), & (\texttt{pvs0bool}(\texttt{op1}_1(\texttt{vr})), \texttt{pvs0not}(\texttt{op1}_0(\texttt{vr}))), & (1,2))\\
(\texttt{rec}(\texttt{op2}_0(\texttt{vr}, \texttt{rec}(\texttt{op1}_4(\texttt{vr})))), & (\texttt{pvs0not}(\texttt{op1}_1(\texttt{vr})), \texttt{pvs0not}(\texttt{op1}_0(\texttt{vr}))), & (2,2))\\
(\texttt{rec}(\texttt{op1}_4(\texttt{vr})), & (\texttt{pvs0not}(\texttt{op1}_1(\texttt{vr})), \texttt{pvs0not}(\texttt{op1}_0(\texttt{vr}))), & (1,0,2,2))
\end{array}
$$

The conditions of a valid path of a given `PVS0` expression (given by Definition 2.1.4), say $\mathcal{P}(\texttt{expr})(\texttt{path})$, are specified by the recursive function `path_conditions` in Specification 4.13. Notice that because of the way the list of path conditions is built, to have the conditions in the order they appear in the program, the path must be traveled in a reverse order (hence, using *car* and *cdr* to deal with the path instead of *rac* and *rdc* previously used in the other predicates/functions).

Specification 4.13: Path Conditions of `PVS0` expressions

$$
\begin{array}{l}
\texttt{path\_conditions}(\texttt{expr}, \texttt{path}) =\\
\quad \texttt{IF} \quad \mathit{null}?(\texttt{path})\texttt{THEN}\ \mathit{null}\\
\quad \texttt{ELSE} \quad \texttt{IF}\ \neg \mathit{ite}?(\texttt{subterm\_at}(\texttt{expr}, \mathit{cdr}(\texttt{path}))) \vee \mathit{car}(\texttt{path}) = 0\\
\qquad\qquad \texttt{path\_conditions}(\texttt{expr}, \mathit{cdr}(\texttt{path}))\\
\qquad\quad \texttt{ELSE}\ \texttt{IF}\ \mathit{car}(\texttt{path}) = 1\\
\qquad\qquad\quad \texttt{THEN}\ \mathit{cons}(\texttt{pvs0bool}(\mathit{get\_cond}(\texttt{subterm\_at}(\texttt{expr}, \mathit{cdr}(\texttt{path})))),\\
\qquad\qquad\qquad\qquad \texttt{path\_conditions}(\texttt{expr}, \mathit{cdr}(\texttt{path})))\\
\qquad\qquad\quad \texttt{ELSE}\ \mathit{cons}(\texttt{pvs0not}(\mathit{get\_cond}(\texttt{subterm\_at}(\texttt{expr}, \mathit{cdr}(\texttt{path})))),\\
\qquad\qquad\qquad\qquad \texttt{path\_conditions}(\texttt{expr}, \mathit{cdr}(\texttt{path})))
\end{array}
$$

For the `PVS0` program for Ackermann function in Example 4.2.1 and path $(0, 2, 2)$ the path conditions are the same as for the third calling context.

4.2.1 Semantic termination of PVS0

Semantic evaluation is given by the curried inductive predicate `semantic_rel_expr`. The first part of this currying is `semantic_rel_expr(pvs0)`, abbreviated as ε, in the representation below. The predicate ε has as parameters a triple ($\texttt{expr}$, $\texttt{v}_{in}$, $\texttt{v}_{out}$), consisting of a `PVS0` expression to be evaluated regarding the body of the `PVS0` program definition on the first part of the currying and input and output values. ε holds whenever the expression `expr` evaluates to $\texttt{v}_{out}$ with input $\texttt{v}_{in}$ using the given `PVS0` definition.

Specification 4.14: Semantic Evaluation of PVS0 expressions

$$
\begin{aligned}
\varepsilon(\texttt{expr}, \texttt{v}_{in}, \texttt{v}_{out}) = \quad & \texttt{cnst?(expr)} && \wedge\ \texttt{v}_{out} = \mathit{get\_val}(\texttt{expr}) \vee \\
& \texttt{vr?(expr)} && \wedge\ \texttt{v}_{out} = \texttt{v}_{in} \vee \\
& \texttt{op1?(expr)} && \wedge\ \exists(\texttt{v}_1) : \varepsilon(\mathit{get\_arg}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_1) \wedge \\
& && \qquad \texttt{v}_{out} = O_1(\mathit{get\_op}(\texttt{expr}))(\texttt{v}_1) \\
& \texttt{op2?(expr)} && \wedge\ \exists(\texttt{v}_1, \texttt{v}_2) : \varepsilon(\mathit{get\_arg1}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_1) \wedge \\
& && \qquad \varepsilon(\mathit{get\_arg2}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_2) \wedge \\
& && \qquad \texttt{v}_{out} = O_2(\mathit{get\_op}(\texttt{expr}))(\texttt{v}_1, \texttt{v}_2) \\
& \texttt{ite?(expr)} && \wedge\ \exists(\texttt{v}_1) : \varepsilon(\mathit{get\_cond}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_1) \wedge \\
& && \qquad (\texttt{v}_1 \neq \bot \wedge\ \ \varepsilon(\mathit{get\_if}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_{out}) \vee \\
& && \qquad\ \texttt{v}_1 = \bot \wedge\ \ \varepsilon(\mathit{get\_else}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_{out})) \vee \\
& \texttt{rec?(expr)} && \wedge\ \exists(\texttt{v}_1) : \varepsilon(\mathit{get\_arg}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_1) \wedge \\
& && \qquad \varepsilon(\texttt{pvs0`4}, \texttt{v}_1, \texttt{v}_{out})
\end{aligned}
$$

When for a given PVS0 definition def and a given input value $\texttt{v}_{in}$, it is possible to obtain an output value regarding this semantic evaluation, the definition and value are said to be determined? accordingly to the predicate specified as below, where ε abbreviates semantic_rel_expr(def`2,def`3,def`1,def`4).

Specification 4.15: Determinated definition in PVS0

$$\texttt{determined?}(\texttt{def}, \texttt{v}_{in}) = \exists(\texttt{v}_{out}) : \varepsilon(\texttt{def`4}, \texttt{v}_{in}, \texttt{v}_{out})$$

Over the predicate ε the first notion of semantic termination is specified as the predicate terminates_expr, also curried and that holds whenever it is possible to evaluate every input value. The first part of the currying is also the PVS0 definition and it is abbreviated as T_ε; the second part consists of only the PVS0 expression to be evaluated. This termination notion is then given as:

Specification 4.16: Termination by Semantic Evaluation of PVS0 expressions

$$T_\varepsilon(\texttt{expr}) = \forall(\texttt{v}_{in}) : \exists(\texttt{v}_{out}) : \varepsilon(\texttt{expr}, \texttt{v}_{in}, \texttt{v}_{out})$$

Another way of evaluating an expression is by bounding the number of nested recursive calls, as in Definition 2.1.5. For this, the recursive function eval_expr was specified. This function has type $\texttt{T} \cup \{\texttt{NONE}\}$ and is also curried using as first part the PVS0 definition. If the evaluation process gives no answer for a specific bound in the number of nested recursive calls, say n, the given output would be NONE (for short, $\diamondsuit$); for discriminating the type predicate *some?* is used. *some?*(v) holds whenever v is different from $\diamondsuit$. The first part of the currying, eval_expr(pvs0), is abbreviated as χ, and $\chi(\texttt{expr}, \texttt{v}_{in}, \texttt{n})$ is specified as:

Specification 4.17: Evaluation of PVS0 expressions by nested calls

$$
\begin{array}{lll}
\texttt{IF n} = 0 \texttt{ THEN } \Diamond & & \\
\texttt{ELSE CASES} & \texttt{exprOF} & \\
\quad \texttt{cnst}: & get\_val(\texttt{expr}) & \\
\quad \texttt{vr}: & \texttt{v}_{in} & \\
\quad \texttt{op1}: & \texttt{IF } \chi(get\_arg(\texttt{expr}), \texttt{v}_{in}, \texttt{n}) \neq \Diamond \texttt{THEN} & \\
 & \qquad O_1(get\_op(\texttt{expr}))\chi(get\_arg(\texttt{expr}), \texttt{v}_{in}, \texttt{n}) & \\
 & \texttt{ELSE } \Diamond & \\
\quad \texttt{op2}: & \texttt{IF } \chi(get\_arg1(\texttt{expr}), \texttt{v}_{in}, \texttt{n}) \neq \Diamond \wedge \chi(get\_arg2(\texttt{expr}), \texttt{v}_{in}, \texttt{n}) \neq \Diamond \texttt{THEN} & \\
 & \qquad O_2(get\_op(\texttt{expr}))(\chi(get\_arg1(\texttt{expr}), \texttt{v}_{in}, \texttt{n}), & \\
 & \qquad\qquad\qquad\qquad \chi(get\_arg2(\texttt{expr}), \texttt{v}_{in}, \texttt{n})) & \\
 & \texttt{ELSE } \Diamond & \\
\quad \texttt{ite}: & \texttt{IF } \chi(get\_cond(\texttt{expr}), \texttt{v}_{in}, \texttt{n}) \neq \Diamond \texttt{THEN} & \\
 & \quad \texttt{IF } \chi(get\_cond(\texttt{expr}), \texttt{v}_{in}, \texttt{n}) \texttt{THEN} & \\
 & \qquad \chi(get\_if(\texttt{expr}), \texttt{v}_{in}, \texttt{n}) & \\
 & \quad \texttt{ELSE } \chi(get\_else(\texttt{expr}), \texttt{v}_{in}, \texttt{n}) & \\
 & \texttt{ELSE } \Diamond & \\
\quad rec: & \texttt{IF } \chi(get\_arg(\texttt{expr}), \texttt{v}_{in}, \texttt{n}) \neq \Diamond \texttt{THEN} & \\
 & \quad \chi(\texttt{e}_f, \chi(get\_arg(\texttt{expr}), \texttt{v}_{in}, \texttt{n}), \texttt{n-1}) & \\
 & \texttt{ELSE } \Diamond &
\end{array}
$$

The predicate `eval_expr_termination` specifies a second notion of semantic termination, over the function χ, as in Definition 2.1.6, that holds for inputs, a PVS0 expression and a PVS0 program, whenever for all possible input values there is a bound of nested recursive calls that allows the evaluation giving as output a value different from $\Diamond$. The currying done in the same way as it was done for the precedent predicates and functions and it is abbreviated as T_χ, and specified as below.

Specification 4.18: Termination by Evaluation of PVS0 expressions

$$T_\chi(\texttt{expr}) = \forall(\texttt{v}_{in}) : \exists(\texttt{n}) : \chi(\texttt{expr}, \texttt{v}_{in}, \texttt{n}) \neq \Diamond$$

Several properties of these two evaluation mechanisms are stated and formalized in the PVS development, the most important being the equivalence of these two notions of semantic termination as will be seen in Subsection 6.1.

4.2.2 Specification of Ranking Functions for PVS0

To deal with the execution flow of a given `PVS0` program, it is necessary to check if the conditions of a CC hold so that the code of the associated recursive call would be indeed executed. This can be verified through the recursive predicate `eval_conds`, that is curried as for predicates and functions presented so far and it is abbreviated as $\mathcal{C}$, and the second part of the currying is the conditions of a recursive call and the input value that will make these conditions hold or not.

Specification 4.19: Evaluation of conditions of `PVS0` expressions

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{C}(\texttt{cnds}, \texttt{v}_{in}) =} \\
\texttt{IF} & \mathit{null?}(\texttt{cnds})\ \texttt{THEN}\ \mathit{true} \\
\texttt{ELSE} & \texttt{CASES}\ \mathit{car}(\texttt{cnds})\ \texttt{OF} \\
 & \texttt{expr}\ \ \mathit{bool}(\texttt{expr}) : \exists(\texttt{v}_{out}) : \varepsilon(\texttt{expr}, \texttt{v}_{in}, \texttt{v}_{out}) \wedge \bot \neq \texttt{v}_{out}, \\
 & \texttt{expr}\ \ \mathit{not}(\texttt{expr}) : \exists(\texttt{v}_{out}) : \varepsilon(\texttt{expr}, \texttt{v}_{in}, \texttt{v}_{out}) \wedge \bot = \texttt{v}_{out}, \\
 & \wedge\ \mathcal{C}(\mathit{cdr}(\texttt{cnds}), \texttt{v}_{in})
\end{array}
$$

With these predicates and the functions and predicates over a path, subexpressions and conditions in a path given at the beginning of Section 4.2, the notion of a valid calling context for a `PVS0` expression is specified as the predicate `pvs0_tcc_valid_cc`, where *cc* is a variable whose type is `PVS0Expr_CC`, that specifies Definition 3.1.1. This predicate holds for a given `cc` if its path is a valid path of the given `PVS0` expression `expr`, and the recursive call and the conditions are indeed achieved through this path.

Specification 4.20: Valid Calling Contexts of `PVS0` expressions

$$
\begin{array}{ll}
\texttt{pvs0\_tcc\_valid\_cc}(\texttt{expr})(\texttt{cc}) = & \mathcal{P}(\texttt{expr})(\texttt{cc}`\mathit{path}) \wedge \\
 & \texttt{cc}`\mathit{rec\_expr} = \texttt{subterm\_at}(\texttt{expr}, \texttt{cc}`\mathit{path}) \wedge \\
 & \texttt{cc}`\mathit{cnds} = \texttt{path\_conditions}(\texttt{expr}, \texttt{cc}`\mathit{path})
\end{array}
$$

To specify Ranking Functions for `PVS0` programs, it is necessary to provide for each program, as given by Definition 3.1.2, a measure `wfm` with a well-founded order `lt` (read as less than) over a well-founded set `MT`, a set of ordered elements, such that this measure can be proved decreasing after each recursive call of a program. The measure `wfm` is a mapping function of type `T -> MT` (type `WFM`) and it is used to compare the parameters and arguments of each recursive call with `lt`. The specification is parameterized with the well-founded set and its well-founded order, such that is possible to choose the best suitable one for each program analysis.

The natural numbers is a well-founded set for Ackermann, where lexicographic ordering over its arguments is the measure function and $<$ is an adequate well-founded order.

The predicate pvs0_ tcc_termination_pred to analyze decrease over CCs parameters and arguments, abbreviated as ς, is specified in Specification 4.21 and holds for a well-founded measure over a well-founded set whenever the values of the PVS0 program are measured and such that input values of the program are greater than the corresponding arguments for every valid CC *cc* of the program. The first curried part of ε is properly instantiated with a PVS0 program definition def.

Specification 4.21: Well-founded measure for PVS0 expressions

$$\begin{aligned}&\varsigma(\mathtt{def}, \mathtt{wfm}) = \\ &\quad \forall(\mathtt{v}_{in}, \mathtt{v}_{out}, \mathtt{cc}) : \\ &\qquad \varepsilon(get\_arg(\mathtt{cc}\grave{}rec\_expr), \mathtt{v}_{in}, \mathtt{v}_{out}) \land \mathcal{C}(\mathtt{def}, cc\grave{}cnds, \mathtt{v}_{in}) \Rightarrow \\ &\qquad \mathtt{lt}(\mathtt{wfm}(\mathtt{v}_{out}), \mathtt{wfm}(\mathtt{v}_{in}))\end{aligned}$$

Thus, the notion of TCC termination for PVS0 programs definitions, given by Definition 3.1.3, is specified by the predicate pvs0_tcc_termination (given below as T_ς) that holds if there exists a well-founded measure that satisfies ς.

Specification 4.22: TCC termination of PVS0 expressions

$$T_\varsigma(\mathtt{def}) = \exists(\mathtt{wfm}) : \varsigma(\mathtt{def}, \mathtt{wfm})$$

4.2.3 Specification of Size-Change based technologies

The Size-Change Principle is specified over a generic representation of data exchanging points of a program. Each control point must have some identification, the actual modified part of the program, and the conditions that led to it. In the specification, generic data exchanging points are similar to *calling contexts* and have a type CallingContext given in Specification 4.23.

Specification 4.23: Calling Context for generic data

$$\mathtt{CallingContext} : \mathtt{TYPE} = [\#\ \mathtt{nid} : \mathtt{NodeId},\ \mathtt{actuals} : \mathtt{Expr},\ \mathtt{cnds} : \mathtt{Condition}\ \#]$$

Regarding the type PVS0Expr_CC of CCs of PVS0 programs, described at the beginning of Section note that the generic data exchanging points or generic CCs have an identifier instead of a path (which identifies CCs in PVS0 programs); also they have actuals instead of PVS0 recursive expressions. Generic CCs and CC only coincide in which they have conditions that led to data exchanging points.

Termination accordingly to SCP would be given as the non-existence of an infinite sequence of values directly related to an infinite sequence of generic CCs through generic mechanisms for the semantic evaluation of actuals and conditions, respectively given as `sem_eval` and `cond_eval`. The feasibility of the sequence of CCs depends on instantiation with values in an associated infinite sequence. The desired relation between calling contexts and values in both sequences is given by two requirements. The first is that that the evaluation of the condition of the i^{th} CC with the i^{th} value holds. The second is that the evaluation for the actuals of the i^{th} calling context with the i^{th} value gives as a result the $(i+1)^{th}$ value. Such infinite sequences are specified in Specification 4.24 over a given sequence of CCs `ccs` and a sequence of values `vals` as the predicate `infinite_seq_ccs`.

Specification 4.24: Infinite sequence of Calling Contexts

$$\begin{array}{l}\texttt{infinite\_seq\_ccs}(\texttt{sem\_eval}, \texttt{cond\_eval})(\texttt{def}, \texttt{ccs}, \texttt{vals}) = \\ \quad \forall (i) : \texttt{cond\_eval}(\texttt{ccs}(i)`cnds, \texttt{vals}(i)) \wedge \\ \qquad \texttt{sem\_eval}(\texttt{ccs}(i)`actuals, \texttt{vals}(i), \texttt{vals}(i+1))\end{array}$$

Then, two notions of termination are specified accordingly to SCP. The first one is given in Speficification 4.25 and states the classic definition of the SCP, through the predicate `SCP`. The existential quantifier on the relation `r` used in the specification of this predicate is useful in some equivalence proofs (as will be shown in Section 6.3). However, the consequent of the implication would be always false considering a well-founded relation.

The second notion of termination by SCP is a simplified version of the previous one specified as the predicate `scp_termination?` that states that no infinite sequences of CCs and values that satisfy the predicate `infinite_seq_ccs` are possible.

Specification 4.25: The Size-Change Principle

$$\begin{array}{l}\texttt{SCP}(\texttt{sem\_eval}, \texttt{cond\_eval}) = \\ \quad \forall (\texttt{ccs}, \texttt{vals}) : \texttt{infinite\_seq\_ccs}(\texttt{sem\_eval}, \texttt{cond\_eval})(\texttt{ccs}, \texttt{vals}) \Rightarrow \\ \qquad\qquad \exists (\texttt{r}) : \forall (i) : \texttt{r}(\texttt{vals}(i+1), \texttt{vals}(i))\end{array}$$

Specification 4.26: Termination by Size-Change Principle

$$\begin{array}{l}\texttt{scp\_termination?}(\texttt{sem\_eval}, \texttt{cond\_eval}) = \\ \quad \forall (\texttt{ccs}, \texttt{vals}) : \neg\texttt{infinite\_seq\_ccs}(\texttt{sem\_eval}, \texttt{cond\_eval})(\texttt{ccs}, \texttt{vals})\end{array}$$

The well-foundedness of `r` is given according to the predicate `well_founded?` in Specification 4.27 (present in the standard prelude library of PVS), where `p` is some unary predicate over some given type. When this predicate holds, it is ensured the finiteness of chains of elements that can be compared with it.

Specification 4.27: Definition of well-foundedness

$$\texttt{well\_founded?}(\texttt{r}) = \forall(\texttt{p}) : (\exists(\texttt{y}) : \texttt{p}(\texttt{y})) \Rightarrow (\exists(\texttt{y} : (\texttt{p})) : \forall(\texttt{x} : (\texttt{p})) : \neg\texttt{r}(\texttt{x}, \texttt{y}))$$

Notice that both definitions are equivalent by the selection of a well-founded relation `r`. Also, both capture termination accordingly to SCP establishing the non-existence of feasible infinite sequences of consecutive executable (instantiated) calling contexts.

The parametrization of the SCP notion is straightforward for `PVS0` programs. The semantic evaluation `sem_eval` and the operational mechanism for evaluating conditions leading to recursive calls `eval_conds` are specified for a given `PVS0` definition `def`, respectively, as below, where the curried portion of ε and $\mathcal{C}$ are properly instantiated with the four elements that compose `def`. Then, the predicate `scp_termination_pvs0` specifies SCP termination for `PVS0` programs in Specification 4.28.

CCGs were specified as digraphs with vertices labeled by calling contexts. The structure has a type called `CCG`, which includes a digraph (`dg`) and a finite family of measure functions with type [$\texttt{T} \to \texttt{MT}$], as the type of the measure function `wfm` used for TCC termination. The family of measure functions is called here *measures* and they are given by an indexed structure `M`, with the type specified as (`FunMeasures`). Enriched digraphs are built as `make_ccg(dg,M)`.

Specification 4.28: Instantiation of Size-Change Principle for PVS0

$$\texttt{sem\_eval}(\texttt{def})(\texttt{expr}, \texttt{v}_{in}, \texttt{v}_{out}) =$$
$$\quad \varepsilon(\texttt{expr}, \texttt{v}_{in}, \texttt{v}_{out})$$

$$\texttt{cond\_eval}(\texttt{def})(\texttt{cnds}, \texttt{v}_{in}) =$$
$$\quad \mathcal{C}(\texttt{cnds}, \texttt{v}_{in})$$

$$\texttt{scp\_termination\_pvs0}(\texttt{def}) =$$
$$\quad \texttt{scp\_termination?}(\texttt{sem\_eval}(\texttt{def}), \texttt{cond\_eval}(\texttt{def}))$$

The measures of `CCG`s are used to measure possible decrease over each edge of the digraph and combined to provide the desired decrease over walks and circuits of the digraph. Walks are specified as sequences of adjacent vertices of a graph and circuits as walks with equal initial and final vertex. Combinations of measures over the edges of walks are given as sequences of naturals (indices of `M`) of the same length than the walk. These combinations represent choices of measures among program paths for measuring the values in CCs incident to edges in a walk. These combinations are specified as `measures_combination`, for short written as `mc`.

Given a walk that is a circuit, c, that a combination of measures mc provides the decreasing among this walk is specified as the predicate gt_mc?. This predicate uses the relation ge (read as greater than or equal) specified from the well-founder order lt. The predicate gt_mc? in Specification 4.29 holds whenever for all steps in the circuit, if one has an input value $\mathtt{v}_{in}$ for which the condition in the CC holds and whose actuals are evaluated as $\mathtt{v}_{out}$, the chosen measures for the steps of the circuit are such that the measure for $\mathtt{v}_{in}$ is greater than or equal to the measure for $\mathtt{v}_{out}$, and there exists (at least one) step in the circuit for which the selection of measures gives a decrease.

Specification 4.29: Decreasing measure in circuit

$$\begin{array}{l}
\mathtt{gt_mc?}(\mathtt{M},\mathtt{c})(\mathtt{mc}) = \\
\quad \forall(i,\mathtt{v}_{in},\mathtt{v}_{out}) : (\mathtt{cond_eval}(\mathtt{c}_i\grave{}\,cnds,\mathtt{v}_{in}) \wedge \mathtt{sem_eval}(\mathtt{c}_i\grave{}\,actuals,\mathtt{v}_{in},\mathtt{v}_{out})) \Rightarrow \\
\quad\quad \mathtt{ge}(\mathtt{M}(\mathtt{mc}(i))(\mathtt{v}_{in}),\mathtt{M}(\mathtt{mc}(i+1)))(\mathtt{v}_{out}) \ \wedge \\
\quad \exists(j) : \forall(\mathtt{v}_{in},\mathtt{v}_{out}) : (\mathtt{cond_eval}(\mathtt{c}_j\grave{}\,cnds,\mathtt{v}_{in}) \wedge \mathtt{sem_eval}(\mathtt{c}_j\grave{}\,actuals,\mathtt{v}_{in},\mathtt{v}_{out})) \Rightarrow \\
\quad\quad \mathtt{ge}(\mathtt{M}(\mathtt{mc}(j))(\mathtt{v}_{in}),\mathtt{M}(\mathtt{mc}(j+1)))(\mathtt{v}_{out}) \wedge \mathtt{M}(\mathtt{mc}(j))(\mathtt{v}_{in}) \neq \mathtt{M}(\mathtt{mc}(j+1))(\mathtt{v}_{out})
\end{array}$$

CCG termination is specified over a CCG, G, as the predicate ccg_termination?, where ms(G) gives the family of measures of G, c is a circuit of the graph and mc is the combination of measures. Since the analysis is done over circuits, for which the initial and final vertices coincide, the corresponding initial and final measures in the combination of measures mc must be the same.

Specification 4.30: CCG Termination

$$\mathtt{ccg\_termination?}(\mathtt{G}) = \forall(\mathtt{c}) : \exists(\mathtt{mc}) : \mathtt{gt\_mc?}(\mathtt{ms}(\mathtt{G}),\mathtt{c})(\mathtt{mc})$$

To express the flow execution of a PVS0 program definition def by a CCG, the vertices must be labeled with the calling contexts of the PVS0 program. There will be an edge from cc1 to cc2 whenever there exist input and output values such that three conditions hold: the conditions of cc1 instantiated with the input value hold; the evaluation of the actuals of cc1 instantiated with the input value results in the output value; and, the condition of cc2 instantiated with the output value hold. The predicate sound_ccg_digraph holds when a given digraph satisfies these conditions.

Specification 4.31: Sound CCG for PVS0

$$\begin{array}{ll}
\mathtt{sound_ccg_digraph}(\mathtt{def})(\mathtt{dg}) = & \forall(\mathtt{cc1},\mathtt{cc2})(\mathtt{v}_{in},\mathtt{v}_{out}) : \\
& \quad \mathcal{C}(\mathtt{def},\mathtt{cc1}\grave{}\,cnds,\mathtt{v}_{in}) \wedge \varepsilon(\mathtt{cc1}\grave{}\,actuals,\mathtt{v}_{in},\mathtt{v}_{out}) \wedge \\
& \quad \mathcal{C}(\mathtt{def},\mathtt{cc2}\grave{}\,cnds,\mathtt{v}_{out}) \Rightarrow edges(\mathtt{dg})(\mathtt{cc1},\mathtt{cc2})
\end{array}$$

Thus, CCG termination for PVS0 program definitions is defined accordingly to a sound digraph, regarding the previous description, where the evaluation mechanisms used for predicate `gt_mc?` are ε and $\mathcal{C}$. This is specified as the predicate `ccg_termination_pvs0` below, where function `make_ccg` uses the sound digraph and a family of measures to build a CCG.

Specification 4.32: CCG Termination for PVS0

$$
\begin{aligned}
&\texttt{ccg\_termination\_pvs0}(\texttt{def}) = \\
&\quad \exists(\texttt{M}, \texttt{dg} \mid \texttt{sound\_ccg\_digraph}(\texttt{def})(\texttt{dg})) : \\
&\qquad \texttt{ccg\_termination?}[\varepsilon, \mathcal{C}](\texttt{make\_ccg}(\texttt{dg}, \texttt{M}))
\end{aligned}
$$

Notice that, although there exists no decidable mechanism to eliminate unsound edges of a digraph, the predicate `ccg_termination` is specified over sound graphs, which implies that only useful edges would be considered in the analysis.

Chapter 5

Formalization of termination by Dependency Pairs

The notions of DPs, as specified in Section 4.1, are used to formalize the termination criteria by DPs for the innermost, the ordinary rewriting, and more general Q-restricted relations. The innermost relation case was formalized initially and then adapted to the ordinary rewriting case and to the Q-restricted rewriting relation. Considering in detail the specificities of the case of innermost termination, as done in this Chapter, is relevant since it captures the eager evaluation mechanism applied in the operational semantics of functional models of computations as done in `PVS0` (Section 4.2).

The full formalization of equivalence between the DP Criteria and Noetherianity is discussed for the case of the innermost reduction relation, as a result of this work, also presented in [AAAR20]. Indeed, the proof of necessity, i. e., that Noetherianity implies termination by DPs, is essentially the same for the Q-restricted, the ordinary and the innermost relations. Sufficiency, i.e., is that termination by DPs implies Noetherianity requires slightly different treatments for these three relations.

The Sufficiency demanded the higher efforts. A *constructive* approach builds infinite chains of DPs from infinite derivations through contraposition. This approach implies the introduction of new elements to the proof, such as the existence of the so-called *minimal innermost non-terminating* subterms for innermost non-terminating terms and the existence of strictly innermost normal forms for such subterms from which the existence of rules that apply at root positions of such normal forms are an obligation. The existence of such rules allows the construction of new DPs to build the infinite chain. The axiom of choice available in the prelude of PVS was used to treat the existentials. These formalizations were adjusted to obtain the more general result for termination by DPs of the

Q-restricted rewriting relation.

5.1 Necessity for the Innermost Dependency Pairs Termination Criterion

Lemma `inn_Noetherian_implies_inn_dp_termination` formalizes the necessity of the DP Criterion, which is specified in Specification 5.1 along with the specification of the `Noetherian?` predicate over a given relation, which specified as holding whenever the converse of this relation is well-founded (both `well_founded?` predicate and function `converse` follow the standard definition and are specified in the prelude file of PVS).

Specification 5.1: The `Noetherian?` predicate and the necessity lemma.

```
Noetherian?(R) =
  well_founded?(converse(R))

inn_Noetherian_implies_inn_dp_termination : LEMMA
  ∀(E) :
    Noetherian?(innermost_reduction?(E)) ⇒ inn_dp_termination?(E)
```

The formalization follows by contraposition, by building an infinite sequence of terms associated with an infinite innermost derivation from an infinite chain of DPs. These terms are built by accumulating the contexts where the reductions would take place regarding the *rhs* of the rule that generates each DP in the chain. The intuition of this formalization follows directly from the theory and is summarized in the sketch given in Figure 5.1.

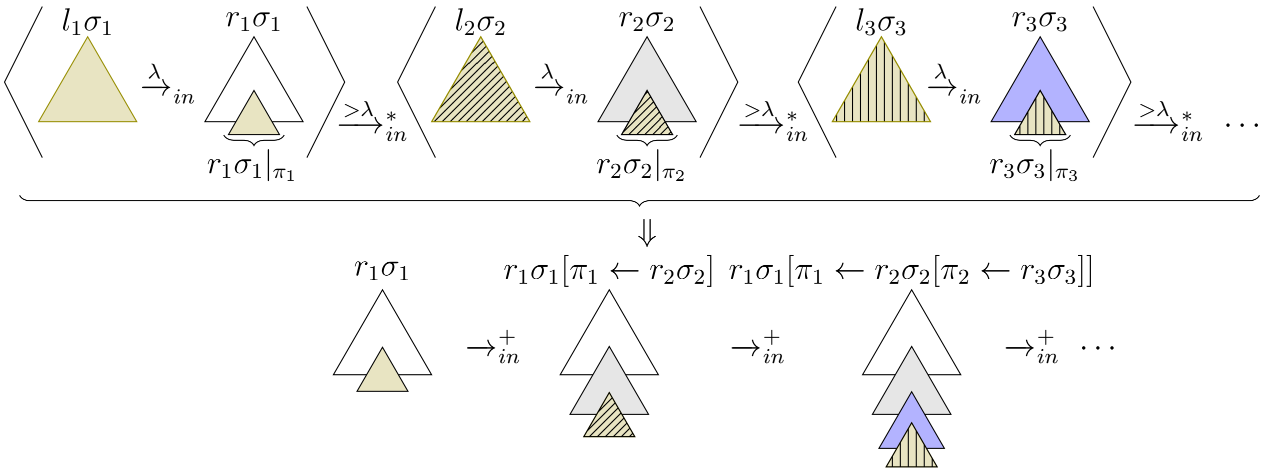


Figure 5.1: Proof sketch: building infinite innermost derivations from infinite innermost DP-chains (c.f. [AAAR20]).

Since there is a root reduction associated with each DP in the sequence, from its *lhs* to the *rhs* of the related rule, and a non-root innermost derivation to reach the *lhs* of the next DP from the *rhs* of the current DP, it is relatively simple to manipulate the rules and positions using the alternative dependency chain specification to build recursively a sequence of terms related by $\to^{+}_{in}$ through the replacement operation.

To perform this construction, the recursive function `term_pos_dps_alt` is used, taking sequences of DPs and substitutions and producing indexed pairs of term and position accumulating contexts in such a way that the terms are related by $\to^{+}_{in}$ whenever the given sequence is chained (Specification 5.2). As illustrated in Figure 5.1, if the sequence is chained, the first pair of term and position is computed as $(r_1\sigma_1, \pi_1)$; the second as $(r_1[\pi_1 \leftarrow r_2\sigma_2], \pi_1 \circ \pi_2)$; and so on. The function `term_pos_dps_alt` uses the previously obtained accumulated context (C) and replaces the *rhs* of the current DP by the *rhs* of the next DP in the sequence. Positions to perform the replacement are given by the accumulation of the positions in the alternative definition of DPs (π).

Specification 5.2: Function to accumulate contexts to build an infinite sequence of terms.

$\texttt{term\_pos\_dps\_alt}(E)(dps : \texttt{sequence}[\texttt{dep\_pair\_alt}(E)],\ \sigma : \texttt{sequence}[\texttt{Sub}], i : \texttt{nat}) :$

$\texttt{RECURSIVE}\ (C, \pi) \mid \pi \in Pos(C) =$

$\quad \texttt{IF}\ i = 0\ \texttt{THEN}$

$\qquad (rhs(dps(0)`1)\sigma(i), dps(0)`2)$

$\quad \texttt{ELSE LET}\ (C, \pi) = \texttt{term\_pos\_dps\_alt}(E)(dps, \sigma, i-1)\texttt{IN}$

$\quad (C[\pi \leftarrow rhs(dps(i)`1)\sigma(i)], \pi \circ dps(i)`2)$

$\texttt{MEASURE}\ i$

Then, an infinite sequence of terms can be built from an infinite chain given by sequences of DPs and substitutions *dps* and σ as:

Specification 5.3: Function to build the terms in an infinite derivation.

$\texttt{LAMBDA}(i : \texttt{nat}) : \texttt{term\_pos\_dps\_alt}(E)(dps, \sigma, i)`1$

Notice that the function `term_pos_dps_alt` would provide an infinite sequence of terms for any pair of infinite sequences of DPs and substitutions, disregarding if they form an infinite innermost chain or not. To prove that the generated infinite sequence indeed describes an infinite derivation for the relation $\rightarrow_{in}$, this function should be applied to a pair *dps* and σ that constitutes an infinite chain.

This is proved by showing the non-Noetherianity of $\rightarrow^+_{in}$ that relates consecutive terms generated by the function `term_pos_dps_alt`. The proof follows by induction, whereas for the induction basis it must be proved that the first term generated is related to the second by $\rightarrow^+_{in}$. `term_pos_dps_alt` builds these terms just using the first and second DPs and substitutions, say $\langle (l_1, r_1), \pi_1 \rangle$, $\langle (l_2, r_2), \pi_2 \rangle$, σ_1 and σ_2 as in Figure 5.1, in the chained input. The first term is $r_1\sigma_1$ and the second $r_1\sigma_1[\pi_1 \leftarrow r_2\sigma_2]$, which is equal to $r_1\sigma_1[\pi_1 \leftarrow l_2\sigma_2[\lambda \leftarrow r_2\sigma_2]]$. Since contiguous pairs in the sequence are innermost chained and $\xrightarrow{>\lambda}^*_{in}$ is compatible with contexts (by monotony of closures, since $\rightarrow_{in}$ is compatible with contexts and $\xrightarrow{>\lambda}_{in} \subseteq \rightarrow_{in}$), one has that $r_1\sigma_1 \xrightarrow{>\lambda}^*_{in} r_1\sigma_1[\pi_1 \leftarrow l_2\sigma_2]$. And, also by the innermost chained property, $l_2\sigma_2$ is a normal instance of the *lhs* of a rule, i.e., a single innermost reduction step can be applied only at the root position giving $r_2\sigma_2$. Since a single innermost reduction step corresponds directly to a replacement operation, and in this case at the root position, one would have one innermost reduction step $r_1\sigma_1[\pi_1 \leftarrow l_2\sigma_2] \xrightarrow{\pi_1}_{in} r_1\sigma_1[\pi_1 \leftarrow r_2\sigma_2]$. Thus, one would have $r_1\sigma_1 \rightarrow^+_{in} r_1\sigma_1[\pi_1 \leftarrow r_2\sigma_2]$. The inductive step considers analogously contiguous DPs and substitutions in the chained input, the only extra details are regarding the current term and position computed in the previous recursive step by `term_pos_dps_alt`. Notice that in the i^{th} iteration, the current term can be seen as a context C with a hole at the accumulated position, say π, filled with term $r_i|_{\pi_i}\sigma_i$. Indeed, in the induction basis the context is given by $r_1\sigma_1$ with a hole at position π_1. The term and accumulated position generated by `term_pos_dps_alt` are given as $C[\pi \leftarrow r_{i+1}\sigma_{i+1}]$ and $\pi \circ \pi_{i+1}$. Notice that this term can be seen as a context with a hole at the accumulated position filled with the term $r_{i+1}|_{\pi_{i+1}}$. Finally, observe that $C[r_i|_{\pi_i}\sigma_i] \rightarrow^+_{in} C[r_{i+1}\sigma_{i+1}]$.

Notice that this formalization is very similar to its pen-and-paper version, disregarding the specification. However, the construction of an actual function to generate each pair of accumulated context and position simplifies the inductive and constructive proof of the existence of the infinite derivation. Furthermore, proof elements that can seem too trivial must be precisely used, such as the mentioned closure of context, monotony of closures, subset properties, and properties regarding the composition of positions in replacements. For example, the last property is used in proving correctness of the *predicate subtyping* condition $\{(C, \pi) \mid \pi \in Pos(C)\}$ of the pairs built by the function `term_pos_dps_alt` (these aspects are discussed in detail in Section 5.2.4). These properties are formalized in the PVS theory `TRS` in a general manner allowing its application for arbitrary rewriting relations.

5.2 Sufficiency for the Innermost Dependency Pairs Termination Criterion

The formalization of sufficiency of the DP Criterion requires more effort and is also proven by contraposition. The core of the proof follows the idea in [AG00] to construct infinite chains from infinite innermost derivations. In an implementational level, to go from infinite derivations to infinite sequences of DPs that would create an infinite chain is challenging. Indeed, constructing the DPs requires, initially, choosing *mint* subterms from those terms leading to infinite innermost derivations; afterward, choosing non-root innermost normalized terms; and, finally, choosing instances of rules that apply at the root positions of these terms from which DPs can be constructed. All these choices are based on existential proof techniques.

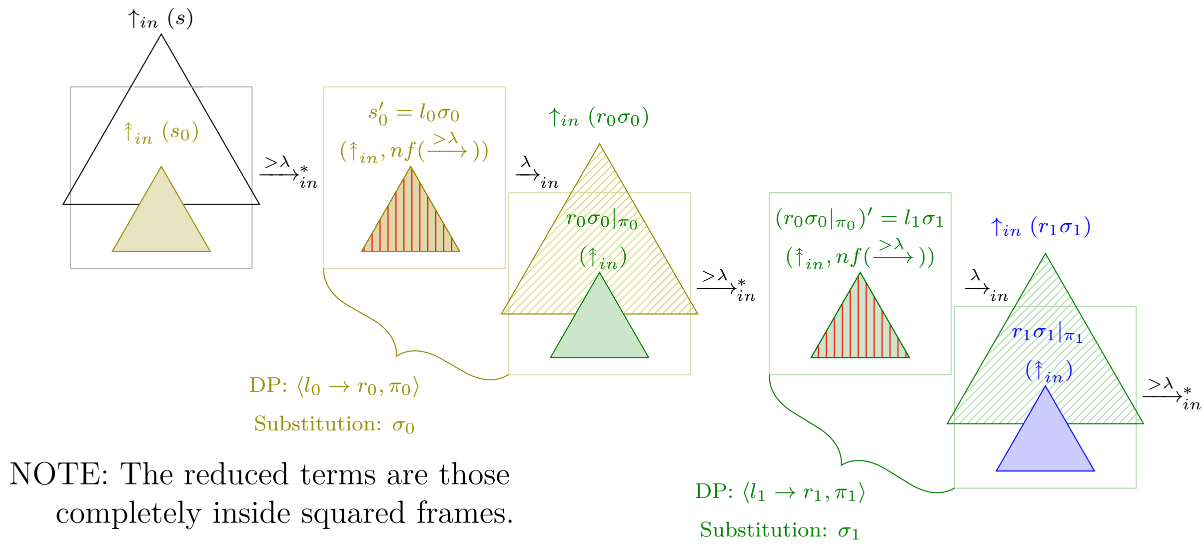


Figure 5.2: Proof sketch: building infinite innermost DP-chains from infinite innermost derivations. The two DPs created, along with their respective substitutions, form chained DPs (c.f. [AAAR20]).

Figure 5.2 illustrates the main steps of the kernel of the construction of chained DPs:

- The existence of *mint* subterms of innermost non-terminating terms is represented as the small triangles inside big ones. This part of the development is explained in Subsection 5.2.1.
- Existence of non-root innermost normalized terms obtained by derivations (through relation $\xrightarrow{>\lambda}_{in}$) from these *mint* subterms, represented as vertically striped triangles, is detailed in Subsection 5.2.2.
- Existence of DPs from rules and substitutions that reduce non innermost terminating terms which are non-root innermost normalized (small vertically striped triangles) at the root position into innermost non-terminating terms (diagonally striped triangles) through $\xrightarrow{\lambda}_{in}$. The DPs are pairs of small vertically striped and small plain triangles, that represents a minimal innermost non-terminating term. This result is explained in Subsection 5.2.3.

The last step of the construction illustrated in Figure 5.2 permits, as the first one, application of a lemma of the existence of *mint* subterms (for innermost non-terminating terms). In the last step, this result will allow constructing the required DPs.

Subsection 5.2.4 then discusses how to get adequate pairs of consecutive chained DPs and associated normal substitutions, and Subsection 5.2.5, finally, details the construction of the required chain of DPs.

5.2.1 Existence of *mint* Subterms

The *mint* property ($\Uparrow_{in}$) over terms is provided in the Specification 5.4 by predicate `minimal_innermost_non_terminating?`. Also in this box one has the specification of lemma `inn_non_terminating_has_mint`, whose formalization ensures the existence of *mint* subterms regarding innermost non-terminating terms. Although this is a simple result, the proof formalization is not as direct as it may seem to be and follows by induction on the structure of the term. The induction basis is trivial since variable terms are not reducible, so variables cannot give rise to infinite derivations. For the inductive step, whenever the term t has an empty list of arguments (that is, t is a constant), the only position it has is its root, thus, the *mint* subterm is the term itself; otherwise, either all its proper subterms are innermost terminating and then the term itself is *mint* or, by the induction hypothesis, some of its arguments is innermost non-terminating, say its *ith* argument, and then it has a *mint* subterm at some position π, thus, the *mint* subterm of t is chosen as $t|_{i\pi}$.

Specification 5.4: Predicate for specifying *mint* terms and lemma over existence of *mint* subterms in innermost non-terminating terms.

`minimal_innermost_non_terminating?`$(E)(t) =$
  $\Uparrow_{in}(t) \wedge$
  $\forall(\pi \in Pos(t) | \pi \neq \lambda) :$
    $SN_{in}(t|_{\pi}))$

`inn_non_terminating_has_mint` : `LEMMA`
  $\forall(E)(t | \Uparrow_{in}(t)) :$
    $\exists(\pi \in Pos(t))) :$
      $\Uparrow_{in}(t|_{\pi}))$

5.2.2 Non-root Innermost Normalization of *mint* Terms

The second step in the formalization proves that every *mint* term can be non-root innermost normalized (into an innermost non-terminating term). This result appears to be, as given in analytic proofs, a simple observation. By definition, every proper subterm of a *mint* term is innermost terminating, and consequently, no argument of this term may give rise to an infinite innermost derivation. However, formalizing such a result by contradiction requires several auxiliary functions and lemmas related to structural properties of such derivations that also consider positions and arguments in which each reduction step happens. These technicalities of the formalization are necessary to obtain a key result that assuming the existence of an infinite non-root innermost derivation from a *mint* term guarantees that some of its arguments begin an infinite innermost derivation, which gives the contradiction.

mint **Terms are Non-root Innermost Terminating**

For the remainder of this subsection, consider elements on Specification 5.5, where s, $seqt$ and $seqp$ are fixed term, sequences of terms and positions, respectively, associated with an infinite non-root innermost derivation on non-root innermost descendants of s, such that the n^{th} term in the sequence $seqt$ reduces into the $(n+1)^{th}$ term at position $seqp(n)$. Also, l will denote a valid argument of s (and as it will be seen, also a valid argument of any of its descendants).

Specification 5.5: Term, position, and sequences of terms and positions.

$s : \texttt{term}|app?(s)$

$l : \texttt{posnat}|l \leq \texttt{length}(args(s))$

$seqt : \texttt{sequence}[\texttt{term}]|\forall(n : \texttt{nat}) : s \xrightarrow{>\lambda}_{in} seqt(n)$

$seqp : \texttt{sequence}[\texttt{position}]|\forall(n : \texttt{nat}) :$
$\quad seqp(n) \in Pos(seqt(n)) \wedge$
$\quad seqp(n) \neq \lambda \wedge$
$\quad seqt(n) \xrightarrow{seqp(n)}_{in} seqt(n+1)$

The predicate `inf_red_arg_in_inf_nr_im_red` in Specification 5.6 holds whenever for a sequence of positions there is an infinite number of positions in the sequence starting with the same natural. For $seqp$ and l as in Specification 5.5, this predicate will be applied to state the existence of an infinite set of indices in the sequence of terms $seqt$ in which the reduction happens at the l^{th} argument. The function `args_of_pos_seq` is just used to give the argument of each position in a sequence of positions.

Specification 5.6: Function to extract the argument position from a given position in a sequence of positions where reductions take place and predicate for checking if there exist infinite reductions at a given argument position.

$\texttt{args\_of\_pos\_seq}(seq : \texttt{sequence}[\texttt{position}]|\forall(i : \texttt{nat}) : seqp(i) \neq \lambda)\,(n : \texttt{nat}) : \texttt{posnat} =$
$\quad \texttt{first}(seqp(n))$

$\texttt{inf\_red\_arg\_in\_inf\_nr\_im\_red}(seq : \texttt{sequence}[\texttt{position}]|\forall(i : \texttt{nat}) : seqp(i) \neq \lambda)$
$\qquad (i : \texttt{posnat}) =$
$\quad \texttt{is\_infinite}(\texttt{inverse\_image}(\texttt{args\_of\_pos\_seq}(seq), i))$

Then, for any l-th argument of the given term s such that the predicate `inf_red_arg_in_inf_nr_im_red`$(seqt)(l)$ holds, the function `nth_index` (Specification 5.7) provides the index of the sequence in which the $(n+1)^{th}$ reduction at argument l happens.

Specification 5.7: Function `nth_index`.

$\texttt{nth\_index}(E)(s)(seqt)(seqp)(l)(n : nat) : \texttt{nat} =$
$\quad \texttt{choose}(\{m : \texttt{nat}|\texttt{args\_of\_pos\_seq}(seqp)(m) = l \wedge$
$\quad \texttt{card}(\{k : \texttt{nat}|\texttt{args\_of\_pos\_seq}(seqp)(k) = l \wedge$
$\quad k < m\}) = n\})$

Note that the well-definedness of these functions is a consequence of the type of l that is a dependent type satisfying the predicate `inf_red_arg_in_inf_nr_im_red`, which means that reductions at the l^{th} argument happen infinitely many times. The main technical difficulty of formalizing well-definedness is related to guaranteeing non-emptiness of the argument of the built-in function `choose`. This constraint is fulfilled by the auxiliary lemma `exists_nth_in_inf_nr_im_red` in Specification 5.8.

Specification 5.8: Non-emptiness lemma for the argument positions where infinite reductions may take place.

$$
\begin{array}{l}
\texttt{exists\_nth\_in\_inf\_nr\_im\_red} : \texttt{LEMMA} \\
\quad \forall (n : \texttt{nat}) : \exists (m : \texttt{nat}) : \\
\quad\quad \texttt{args\_of\_pos\_seq}(seqp)(m) = l \wedge \\
\quad\quad \texttt{card}(\{k : \texttt{nat} | \texttt{args\_of\_pos\_seq}(seqp)(k) = l \wedge k < m\}) = n
\end{array}
$$

The formalization of this lemma follows by induction on n and, although simple, requires several auxiliary lemmas over sets. In the induction basis, since one has infinite reductions at argument l, the set of indices where such reductions take place is infinite, and thus, nonempty (by application of the PVS prelude lemma `infinite_nonempty`). Thus, it is possible to use PVS function `min` (over nonempty sets) to choose the smallest index of this set. By the definition of this `min` function, it is ensured that the set of indices smaller than this minimum in this set is empty, and thus has cardinality zero (by applying PVS prelude lemma `card_empty?`). For the inductive step, one must provide the index where one has a reduction at argument l such that it has exactly $n+1$ indices smaller than it where reductions at argument l occur. By the induction hypothesis, there exists an index m for which reduction takes place at argument l, and for which the cardinality of indices smaller than m with reductions at argument l is n. Thus, the required index is built as the minimum index bigger than m for which the reduction happens at argument l. The correctness of such indices follows similarly to the induction basis. First, since the predicate `inf_red_arg_in_inf_nr_im_red` holds, it is possible to ensure that the set of indices greater than index m for which reductions happen at argument l^{th} is infinite, which allows the application of the function `min`. Then one builds an equivalent set to the one of all indices smaller than this minimum as the addition of index m to the set of indices smaller than m (where one has reductions at argument l^{th}). This construction allows one to use another prelude lemma regarding cardinality of the addition of elements into finite sets (`card_add`) to state that the cardinality of this new set is $n+1$.

The soundness of `nth_index` follows from auxiliary properties such as its monotony and *completeness*, the latter meaning that this function covers exactly (all) the indices in which reductions happen at the l^{th} argument. The formalization of these properties follows directly from the conditions fulfilled by the natural numbers chosen as the indices in `nth_index` and prelude lemmas over cardinality of subsets (`card_subset`), since each index provided gives rise to a subset of the next one. These properties allow an easy formalization of a useful auxiliary result stating that for every index of *seqt* below `nth_index(0)` and between `nth_index(i)+1` and `nth_index(i+1)` there are no reductions in the l^{th} argument (lemma `argument_protected_in_non_nth_index`). And then it is possible to ensure that there are only finitely many non-root innermost reductions regarding a term with *mint* property, which is stated in Specification 5.9 as the lemma `mint_is_nr_inn_terminating`.

Specification 5.9: Lemma for non-root innermost termination of *mint* terms.

`mint_is_nr_inn_terminating` : LEMMA
  $\Uparrow_{in}(s) \Rightarrow \texttt{Noetherian?}(\xrightarrow[s]{>\lambda}_{in}))$

This proof follows by contraposition, by assuming the non-Noetherianity of the $\xrightarrow[s]{>\lambda}_{in}$ relation and building then an infinite derivation for some argument of s, as illustrated in Figure 5.3. Thus, initially one would have an infinite sequence *seqt* of descendants of term s where each one is related to the next one by one step of non-root reduction. From this sequence, since there is a finite number of possible arguments where the reductions can take place and infinitely many reductions taking place in non-root positions, i.e., argument positions, one uses the pigeonhole principle to ensure that there exists some argument position l that satisfies the predicate `inf_red_arg_in_inf_nr_im_red`. This allows the use of function `nth_index` to extract exactly the index of the sequence where such reduction occurs. Then the required infinite derivation is built in two steps. First, since one has, by definition, that $s \xrightarrow{>\lambda} seqt(0)$, this leads to a finite sequence of reduced terms that will be used. Given that every argument of a term innermost reduces at the root position to the argument of a reduced term by non-root reductions (lemma `non_root_rtc_reduction_of_argument`), the subterms of each element of this derivation at the chosen argument position are used to the first portion of the infinite sequence. Finally, the function `nth_index` is used to extract from sequence *seqt* those indices where reductions occur in the selected argument, keeping this argument intact whenever the reduction does not occur in such indices (result given in lemma `argument_protected_in_non_nth_index`). Then, for each term obtained by a reduction on the l-th argument on this (now infinite) derivation, its subterm at argument l is used to build the second and final portion of the infinite sequence.

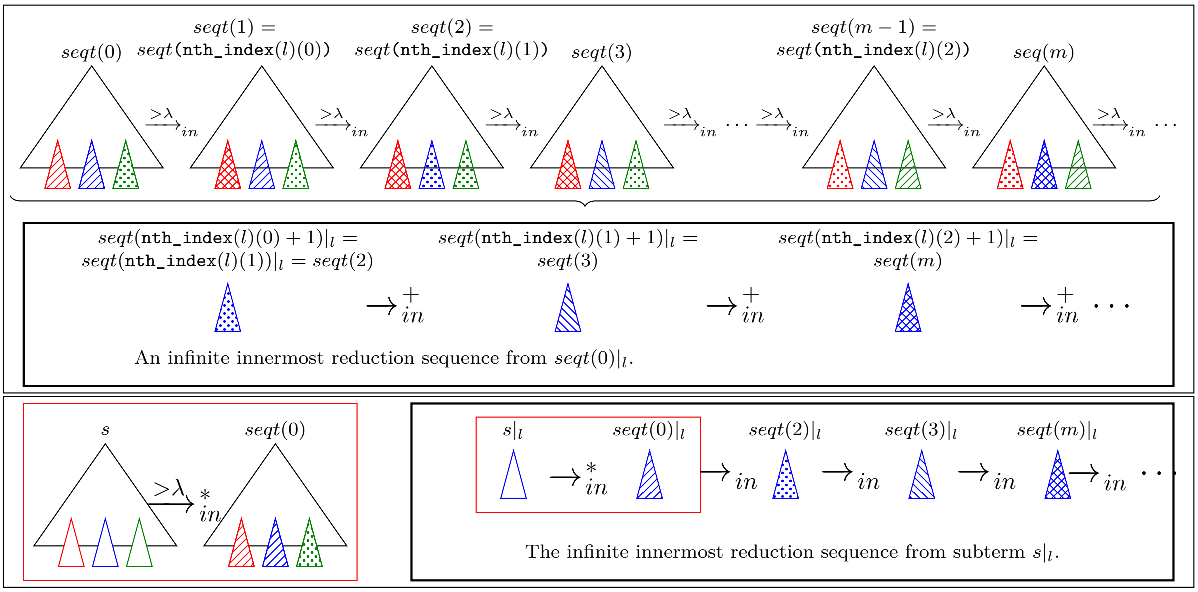


Figure 5.3: Proof intuition: building an infinite innermost derivation of an argument l as concatenation of a finite and an infinite non-root innermost derivation of terms (c.f. [AAAR20]).

Construction of Non-root Innermost Normal Forms for *mint* terms

Since a *mint* term s is Noetherian regarding $\xrightarrow[s]{>\lambda}_{in}$, as previously shown, in an infinite derivation starting from s there exists an index where the first innermost reduction in the root position occurs. This result is formalized in lemma `inf_inn_deriv_of_mint_has_min_root_reduction_index`.

Specification 5.10: Obligation of a first root reduction on infinite innermost derivations.

$$
\begin{array}{l}
\texttt{inf\_inn\_deriv\_of\_mint\_has\_min\_root\_reduction\_index} : \texttt{LEMMA} \\
\quad \forall (seq : \texttt{sequence}[\texttt{term}]) : \\
\qquad (\Uparrow_{in} (seq(0)) \ \wedge \ \forall (i : \texttt{nat}) : \texttt{innermost\_reduction?}(E)(seq(i), seq(i+1))) \Rightarrow \\
\qquad \exists (j : \texttt{nat}) : seq(j) \xrightarrow[s]{\lambda}_{in} seq(j+1) \ \wedge \ \forall (k : \texttt{nat}) : seq(k) \xrightarrow[s]{\lambda}_{in} seq(k+1) \Rightarrow k \geq j
\end{array}
$$

This lemma is formalized by providing as the first index required the minimum index of the infinite derivation where the reduction takes place at the root position. The function minimum (`min`) of PVS, just as function `choose`, also requires proof of non-emptiness of the set used as the parameter. With the Noetherianity provided by lemma `mint_is_nr_inn_terminating`, this non-emptiness constraint is obtained through an auxiliary result over Noetherian relations restricted to an initial element that are subsets of some non-Noetherian relation, which is given by lemma `non_Noetherian_and_Noetherian_rest_subset` in the `restricted_reduction.pvs` theory. This lemma provides an index of this infinite derivation where the given relation, i.e., $\xrightarrow[s]{>\lambda}_{in}$ does not hold.

Notice that, until this point, some infinite reduction sequence is being considered in the proof. However, the DPs are not extracted from the whole terms in this derivation.

Instead, a *mint* term is innermostly reduced until reaching an innermost normal form, and then the rule applied to the root builds the DP. Thus, at this point, the extraction of the DP would be possible. But since the instance of this DPs is crucial for building an infinite chain, it is important to know that not only the term that initiated the infinite derivation will be at some point reduced at the root position, but which exact term was reached before such reduction.

To be able to extract the DP and substitution required to proceed with the proof, one obtains finally that every *mint* term non-root innermost derives into a term that has its arguments in normal form as lemma `mint_reduces_to_int_nrnf_term`.

Specification 5.11: Lemma for obtaining a non-root normal form term from a *mint* term.

$\texttt{mint\_reduces\_to\_int\_nrnf\_term} : \texttt{LEMMA}$
$\quad \forall (s| \uparrow_{in} (s)) :$
$\quad\quad \exists (t| \uparrow_{in} (t)) :$
$\quad\quad\quad s \xrightarrow{>\lambda}{}^{*}_{in} t \;\wedge\; nf(\xrightarrow{>\lambda})(t)$

The proof follows as an application of the previous lemma, choosing the term at the index where the first reduction at the root position takes place since this term is in innermost normal form. Indeed, this term will be a normal instance of the *lhs* of some rule.

5.2.3 Existence of DPs

The term obtained in the previous subsection is an innermost non-terminating term such that it is also non-root innermost normalized. Such non-root normalized terms should innermost reduce at the root position (see $\xrightarrow{\lambda}_{in}$-reductions in Figure 5.2). These reductions from vertically to diagonally striped triangles give rise to the desired DPs. An important observation is that such terms reduce at the root position with a rule and a normal substitution. The substitution should be normal since the terms are non-root innermost normal forms.

The following key auxiliary lemma `normal_inst_of_rule_with_mint_on_rhs_gives_dp_alt` provides the important result that such normal instances of *rhs*'s of rules applied as before and that have minimal innermost non-terminating subterms give rise to DPs. The innermost non-terminality of the terms will guarantee the existence of such subterms.

Specification 5.12: Obtaining the desired DP from a *mint* term with normal substitution.

$\texttt{normal\_inst\_of\_rule\_with\_mint\_on\_rhs\_gives\_dp\_alt} : \texttt{LEMMA}$
$\quad \forall (e \in E, \sigma : (\texttt{normal\_sub?}(E)),\ \pi \in Pos(rhs(e)\sigma)) :$
$\quad\quad \uparrow_{in} (rhs(e)\sigma|_{\pi}) \Rightarrow \texttt{dep\_pair\_alt?}(E)(e, \pi)$

The proof only requires showing that $rhs(e)|_{\pi}$ is defined. For this, initially it must be ensured that π is indeed a non-variable position of $rhs(e)$. But σ is normal, thus, since the premise $\uparrow_{in} (rhs(e)\sigma|_{\pi})$ implies innermost reducibility of $rhs(e)\sigma|_{\pi}$, if π were a variable position or a position introduced by this substitution, there would be a contradiction to its normality. This is proved separately in lemma `reducible_position_of_normal_inst_is_app_pos_of_term` that states that reducible subterms of normal instances of terms appear only at non-variable positions of the original term. Then, by the main result of the last subsection, that is lemma `mint_reduces_to_int_nrnf_term`, one has that $rhs(e)\sigma|_\pi \xrightarrow{>\lambda}{}^*_{in} t$ for some term t such that $\Uparrow_{in}(t)$ and $nf(\xrightarrow{>\lambda})(t)$. Then, the term t has a defined symbol on its root. Thus, it only remains to prove that the root symbol of $rhs(e)\sigma|_\pi$ and t is the same, which is an auxiliary result formalized by induction on the length of the non-root (innermost) derivation in corollary `non_root_ir_preserves_root_symbol` for non-root innermost derivations.

5.2.4 Construction of Chained DPs

So far the existence of the elements needed for the proof was formalized. Now, one builds in fact the elements, as in Figure 5.2. Initially, a *mint* term is non-root innermost normalized through the function `mint_to_nit_nrnf` in Specification 5.13. The existential result given by the lemma in Specification 5.11 on subsection 5.2.2 allows the use of the PVS `choose` operator.

Specification 5.13: Innermost non-terminating non-root normal forms from *mint* terms.

$$\begin{array}{l}\texttt{mint\_to\_nit\_nrnf}(E)(s|\Uparrow_{in}(s)) : \texttt{term} = \\ \quad \texttt{choose}(\{t|s \xrightarrow{>\lambda}{}^*_{in} t) \wedge nf(\xrightarrow{>\lambda})(t) \wedge \Uparrow_{in}(t)\})\end{array}$$

Since this new non-root innermost normalized term is also innermost non-terminating, there exists some rule and normal substitution for allowing innermost reduction of this term at its root. Furthermore, the term obtained from this reduction will be also innermost non-terminating, i.e., it will have a *mint* subterm at some position of the *rhs* of the used rule. This property is formalized in lemma `reduced_nit_nrnf_has_mint` specified as in Specification 5.14.

Specification 5.14: Ensuring existence of *mint* terms on reductions of innermost non-terminating non-root normal form.

$$\begin{array}{l}\texttt{reduced\_nit\_nrnf\_has\_mint} : \texttt{LEMMA} \\ \forall (s|\Uparrow_{in}(E)(s)) : \\ \quad \exists (\sigma : \texttt{Sub}, e : \texttt{rewrite\_rule} \mid e \in E, \pi \in Pos(rhs(e))) : \\ \quad\quad lhs(e)\sigma = \texttt{mint\_to\_nit\_nrnf}(E)(s) \wedge \Uparrow_{in}(rhs(e))\sigma|_\pi)\end{array}$$

This lemma is formalized applying the existential results of Subsection 5.2.3 for obtaining the normal substitution σ and the rule e and, the results of the Subsection 5.2.1 to obtain a position π such that $\Uparrow_{in}(rhs(e))\sigma|_\pi)$.

Lemma `reduced_nit_nrnf_has_mint` allows one to use `choose` to pick the rule and position leading to the DP and the substitution that will allow chaining the DP with the next DP originated from the *mint* term $rhs(e)\sigma|_\pi$ as specified in the function `dp_and_sub_` `from_int_nrnf` given in Specification 5.15. Here it is possible to see why this construction is facilitated by the use of the alternative definition of DPs that includes both the rule and the position. Since it is known exactly the rule and position used, it is possible to use such elements in a direct way.

Specification 5.15: Function to obtain the desired DP and substitution.

$$
\begin{array}{l}
\texttt{dp\_and\_sub\_from\_int\_nrnf}(E)(s|\uparrow_{in}(s)) : [\texttt{dep\_pair\_alt}(E), \texttt{Sub}] = \\
\quad \texttt{LET}\ sub\_e\_p = \texttt{choose}(\{(\sigma : \texttt{Sub}, e \in E, \pi \in Pos(rhs(e))) \mid lhs(e)\sigma = \texttt{mint\_to\_nit\_nrnf}(E)(s) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \uparrow_{in}(rhs(e))\sigma|_{\pi})\})\ \texttt{IN} \\
\quad ((sub\_e\_p`2, sub\_e\_p`3), sub\_e\_p`1)
\end{array}
$$

Whenever this function has as input a term that is an instance of the *rhs* of a DP that is in non-root innermost normal form, the resulting DP and substitution will be chained with the DP and substitution used to build the input term. This result is specified in lemma `next_inst_dp_is_inn_chained_and_mnt` given in Specification5.16, where the desired alternative DPs are transformed into standard DPs in order to allow the analysis through the predicate `inn_chained_dp?`:

Specification 5.16: Lemma ensuring that the obtained DPs and substitutions are chained.

$$
\begin{array}{l}
\texttt{next\_inst\_dp\_is\_inn\_chained\_and\_mnt} : \texttt{LEMMA} \\
\forall (E)(\ dp : \texttt{dep\_pair\_alt}(E), \sigma : Sub \mid\ \uparrow_{in}(rhs(dp`1)\sigma|_{dp`2}))\ \wedge\ nf(\xrightarrow{>\lambda})(lhs(dp`1)\sigma)\ )\ : \\
\texttt{LET}\ std\_dp = (lhs(dp`1), rhs(dp`1)|_{dp`2}), \\
\qquad next\_dp\_sub = \texttt{dp\_and\_sub\_from\_int\_nrnf}(E)(rhs(dp`1)\sigma|_{dp`2}), \\
\qquad next\_std\_dp = (lhs(next\_dp\_sub`1`1), rhs(next\_dp\_sub`1`1)|_{next\_dp\_sub`1`2}), \\
\qquad \sigma` = next\_dp\_sub`2\ \texttt{IN} \\
\quad \texttt{inn\_chained\_dp?}(E)(std\_dp, next\_std\_dp)(\sigma, \sigma`)\ \wedge \quad \uparrow_{in}((next\_std\_dp`2)\sigma`)
\end{array}
$$

The formalization of this lemma is quite simple in its core. However, since transformations between the standard and alternative notions of DPs are used, the proof of some typing conditions are required to ensure type correctness. Once circumvented the typing issues, one must only guarantee the innermost chained property for the input DP and substitution and the resulting DP and substitution created and that the instantiated subterm of the *rhs* of the new DP is a *mint* term. Notice that the latter property is a direct result of the type of the PVS `choose` operator used in function `dp_and_sub_from_int_nrnf`; indeed, this property was included (and formalized) as part of this lemma just to avoid needing to repeatedly ensure non-emptiness of the used set, since this result is used several times throughout the rest of the formalization. To guarantee that the DPs are chained is also straightforward, since `dp_and_sub_from_int_nrnf` is defined over `mint_to_nit_nrnf`, which gives a term with type as a non-root innermost normal form of the *mint* input, i.e., exactly the definition given by predicate `inn_chained_dp?`; using notation of the lemma: $rhs(dp`1)|_{dp`2}\sigma \xrightarrow{>\lambda}{}^{*}_{in}\ lhs(next\_dp\_sub`1`1)\sigma`$.

This result allows the specification of a function using predicate subtyping, a very interesting feature available in PVS. Using this feature, elaborate predicate types can be assigned to the outputs of functions, and type checking will automatically generate the TCCs to ensure well-definedness of the function. Although used in other functions through the formalization, the most interesting application of this feature happens in the next function that outputs a pair for an input pair of DP and substitution, and where the type of the output uses the predicates inn_chained_dp? and $\Uparrow_{in}$. The generated TCCs are not proved automatically; however, to ensure that the type predicates hold, typing provided in the lemma next_inst_dp_is_inn_chained_and_mnt given in Specification 5.16 are applied.

Specification 5.17: Function to obtain adequate next DP and substitution.

$$\begin{array}{l}
\texttt{next_dp_and_sub}(E)(\ dp:\ \texttt{dep_pair_alt}(E), \sigma:\ \texttt{Sub}\ |\ \Uparrow_{in}(rhs(dp`1)\sigma|_{dp`2}) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad nf(\xrightarrow{\geq\lambda})(lhs(dp`1)\sigma)\ )\ : \\
\{\ (next_dp : \texttt{dep_pair_alt}(E),\ next_\sigma:\ \texttt{Sub}) | \texttt{inn_chained_dp?}(E)(dp, next_dp)(\sigma, next_\sigma) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \Uparrow_{in}(rhs(next_dp`1)next\_\sigma|_{next\_dp`2}))\ \} = \\
\quad \texttt{dp_and_sub_from_int_nrnf}(E)(rhs(dp`1)\sigma)|_{dp`2})
\end{array}$$

Applying dp_and_sub_from_int_nrnf (Specification 5.15) to a *mint* term built from a pair of DP and substitution (in the way done in the body of the function next_dp_and_sub), one provides as output a pair of DP and substitution with the specified subtyping predicates, guaranteeing that the input and output are chained.

5.2.5 Construction of the Infinite Innermost Dependency Chain

With the possibility of creating new DPs and substitutions from *mint* terms, it is possible to build, inductively, an infinite DP chain from any innermost non-terminating term. However, PVS syntax makes this construction a bit tricky, since its functional language only allows direct construction of lambda-style or recursive functions. A lambda-style function to create such an infinite chain is not possible, since the construction of every pair of DP and substitution depends on the previous one in the chain. But a direct construction of a recursive function is also problematic since the use of the choose operator in several steps of this construction makes it difficult to guarantee its determinism and then its functionality.

A simple solution for this problem is to use the recursion theorem to provide the existence of a function from naturals to pairs of a DP and a substitution such that each pair generates the next pair in the chain according to the function next_dp_and_sub, implying that contiguous images are chained.

The recursion theorem is given in Specification 5.18. It states that for all predicates X over a set T, initial element a in X and function f over elements of X, there exists a function u from naturals to X such that the images of u are given by the sequence $a, f(a), \ldots, f^n(a), \ldots$.

Specification 5.18: The recursion Theorem.

$\texttt{recursion\_theorem} \ : \texttt{THEOREM}$
$\quad \forall(X : set[T], a \in X, f : [(X) \to (X)]) :$
$\quad\quad \exists(u : [\texttt{nat} \to (X)]]) :$
$\quad\quad\quad u(0) = a \wedge \forall(n : \texttt{nat}) : u(n+1) = f(u(n))$

To use this theorem, the predicate is instantiated with pairs of DP and substitution of the type of the parameters of the function `next_dp_and_sub`, i.e., $(dp : \texttt{dep\_pair\_alt}(E), \sigma : \texttt{Sub} \mid \uparrow_{in} (rhs(dp`1)\sigma|_{dp`2}) \wedge nf(\xrightarrow{>\lambda})(lhs(dp`1)\sigma))$.

The first element of the sequence a is instantiated as the pair of DP and substitution, obtained from the initial term starting any infinite innermost derivation, according to the techniques given in subsections 5.2.1, 5.2.2 and 5.2.3. As expected, the function from pairs to pairs is chosen as `next_dp_and_sub`. The recursion theorem guarantees just the existence of a total function from naturals to the sequence inductively built using function `next_dp_and_sub` starting from the initial pair. But the choice of this function assures by its predicate subtyping that each pair of consecutive pairs are indeed chained.

As a consequence of all that, the sufficiency lemma given in Specification 5.19 is obtained.

Specification 5.19: The sufficiency lemma for DP termination.

$\texttt{inn\_dp\_termination\_implies\_Noetherian} : \texttt{LEMMA}$
$\quad \forall(E) :$
$\quad\quad \texttt{inn\_dp\_termination?}(E) \Rightarrow \texttt{Noetherian?}(\to_{in})$

5.3 Formalization of DP termination for other rewriting relations

The formalizations of the equivalence between the DP Criterion and Noetherianity for the ordinary rewriting relation, given in Definition 3.3.4 and the Q-restricted rewriting relation, given in Definition 3.3.5 are also part of the theories added to the `TRS` library. These results are formalized, respectively, as

$$\texttt{dp\_termination}(E) \Leftrightarrow \texttt{Noetherian?}(\to) \text{ and}$$

$$\texttt{dp\_termination\_criterion?}(E, Q) \Leftrightarrow \texttt{noetherian?}(\xrightarrow{Q}_E).$$

In PVS, the ordinary and Q-restricted relations were specified in a similar way to the one for innermost reduction (given in Specification 4.3). The differences in the specification mainly concern the conditions required on the chains for either reduction relation.

The formalizations of the DP criteria for the ordinary and Q-restricted relations are very similar to the one done for the innermost relation and follow the same steps described in Sections 5.1 and 5.2. The biggest difference is regarding the second step for the sufficiency proof (Section 5.2.2). In the innermost case, the proof innermost normalizes a *mint* term at non-root positions. This non-root innermost normalized term is then reduced at the root position with a rule that is used to build the desired DP. For the ordinary and Q-restricted cases, it is only required to show that derivations starting from a *mnt* or a *minimal Q-restricted non terminating* term are eventually reduced at root position. The desired DP is extracted from the rewriting rule applied for the reduction at the root position.

5.4 Library - `TRS` Theory Summary

The TRS Nasalib Library is very extensive. For better understanding on the extension of this library, this Section provides a visual guide on its organization. For this, the library will be visually split into three sublibraries: `ARS`, `REDUCTION`, `TRS Properties` and `DEPENDENCY PAIRS`, where this last one concentrates the majority of the efforts presented in this work as specifications in Section 4.1 and formalized in Chapter 5. Also, a color scheme is used for distinguish the elements in the library, where predicates are colored red, functions are blue, lemmas are green and types are pink. The same color notation is used in Sections 6.4 and 6.5.

A broad vision of parts of the library that were not discussed in this work is given providing only the name of the theories, such as in the Figure 5.4 for the `ARS` and Figure 5.6 for `TRS Properties` sublibraries. The former provides the basic elements of abstract reduction systems, such as reducibility, confluence and Noetherianity regarding a given relation. The latter embraces several elaborate formalizations regarding such systems, such as confluence of abstract reduction systems (see [GAR08]), the Critical Pair Theorem (see [GAR10]) and orthogonal TRSs and their confluence (see [ROGAR17]).

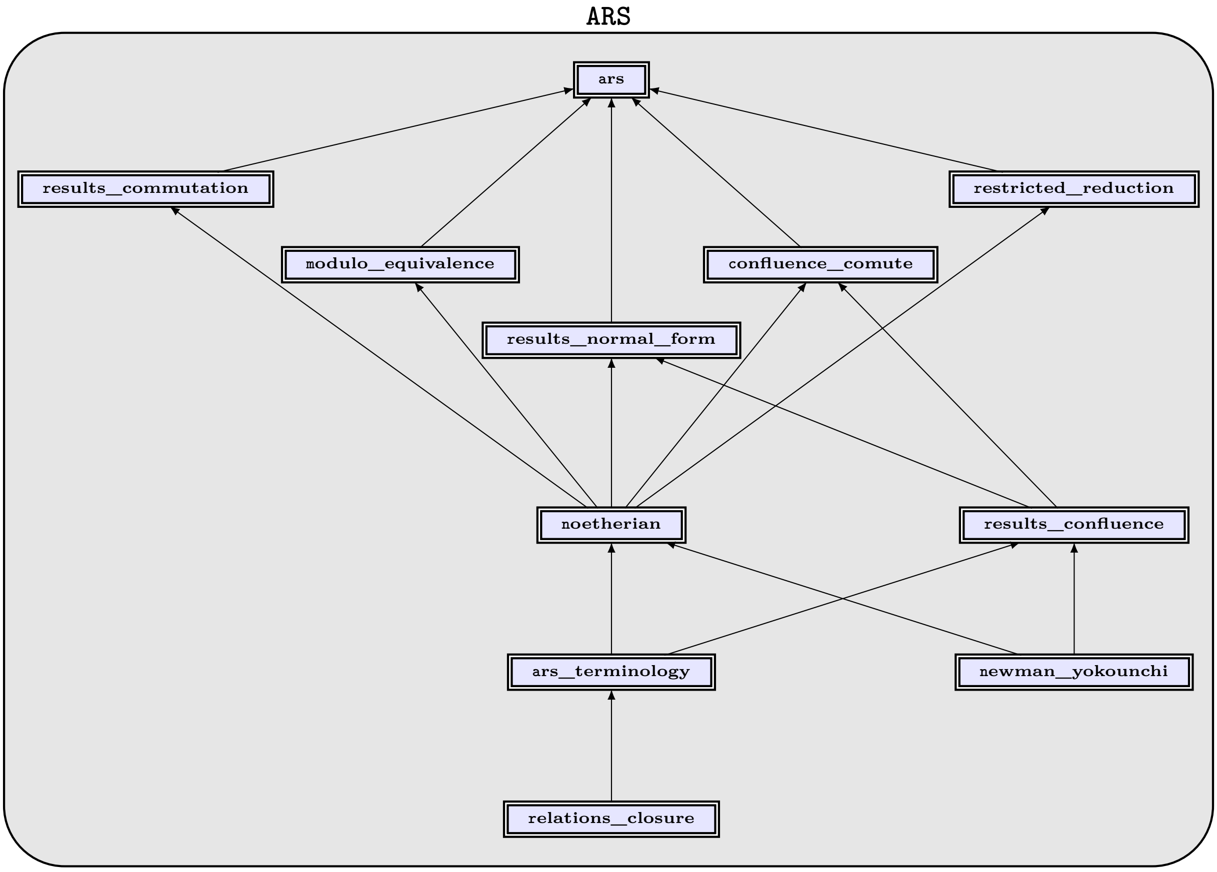


Figure 5.4: The ARS sublibrary

Subtheory TRS encapsulates the majority of the basic formalizations regarding TRSs. The theory term gives the datatype for terms and theory variables_term provides a set of variable terms. Theory positions has the function positionsOF, that specifies the positions of terms according Definition 2.2.2 and several properties over it, such as the notion of parallel positions used in formalizations of Orthogonality. Theory subterm gives the function subtermOF that specifies Definition 2.2.3 and means to manipulate such subterms, such as properties over the positions of subterms. The replacement of terms is given by function replaceTerm in theory replacement, that also contains results such as associativity and distributivity of replacement and properties over replacement of subterms.

The compatibility with contexts and closures over relations provided by the ARS sublibrary are given in theory compatibility. The substitutions needed in the reduction operation are given as type Sub in theory substitution along with the function ext that applies such substitution to a term and also several properties such as preservation of positions after application of substitutions and distributivity of substitutions over replacement.

The rewrite rules in Definition 2.2.4 are specified as the type `rewrite_rule` in theory `rewrite_rules`, where the notion of defined symbols in Specification 4.1 given as predicate `defined?` is also present.

The reduction and non-root reduction relations given in Definitions 2.2.6 and 2.2.8 and in Specification 4.2 are given in theory `reduction` as predicates `reduction_fix?`, `reduction?` and `non_root_reduction_fix?`. In this theory it is also present the rewriting reduction restricted to descendants relation given in Definition 2.2.10 (and Specification 4.4) as predicate `arg_rest_std?`. Furthermore, the theory also brings necessary results to several formalizations, including the one for DPs for ordinary reduction. Such results include:

- the non-root reduction being a subset of the ordinary reduction in lemma `non_root_subset_reduction`,
- the compatibility with contexts and substitutions of the reduction relation and its closures (as lemmas `reduction_is_subs_op` and `closure_close_subs`)
- the property of termination for the subterms of a terminating term in lemma `terminating_all_subterms`,
- the notion of a terminating substitution as predicate `terminating_sub?`
- the preservation of root symbols and arguments positions of terms when non-root derivating terms (lemmas `non_root_rtc_preserves_root_symbo` and `non_root_rtc_preserves_pos_args`)
- the preservation of other arguments when a specific argument is derivated in lemma `arg_preservation_in_finite_rtc` and
- the propagation of the derivations to arguments when the whore term is derivated in lemma `non_root_rtc_rtc_of_argument`.

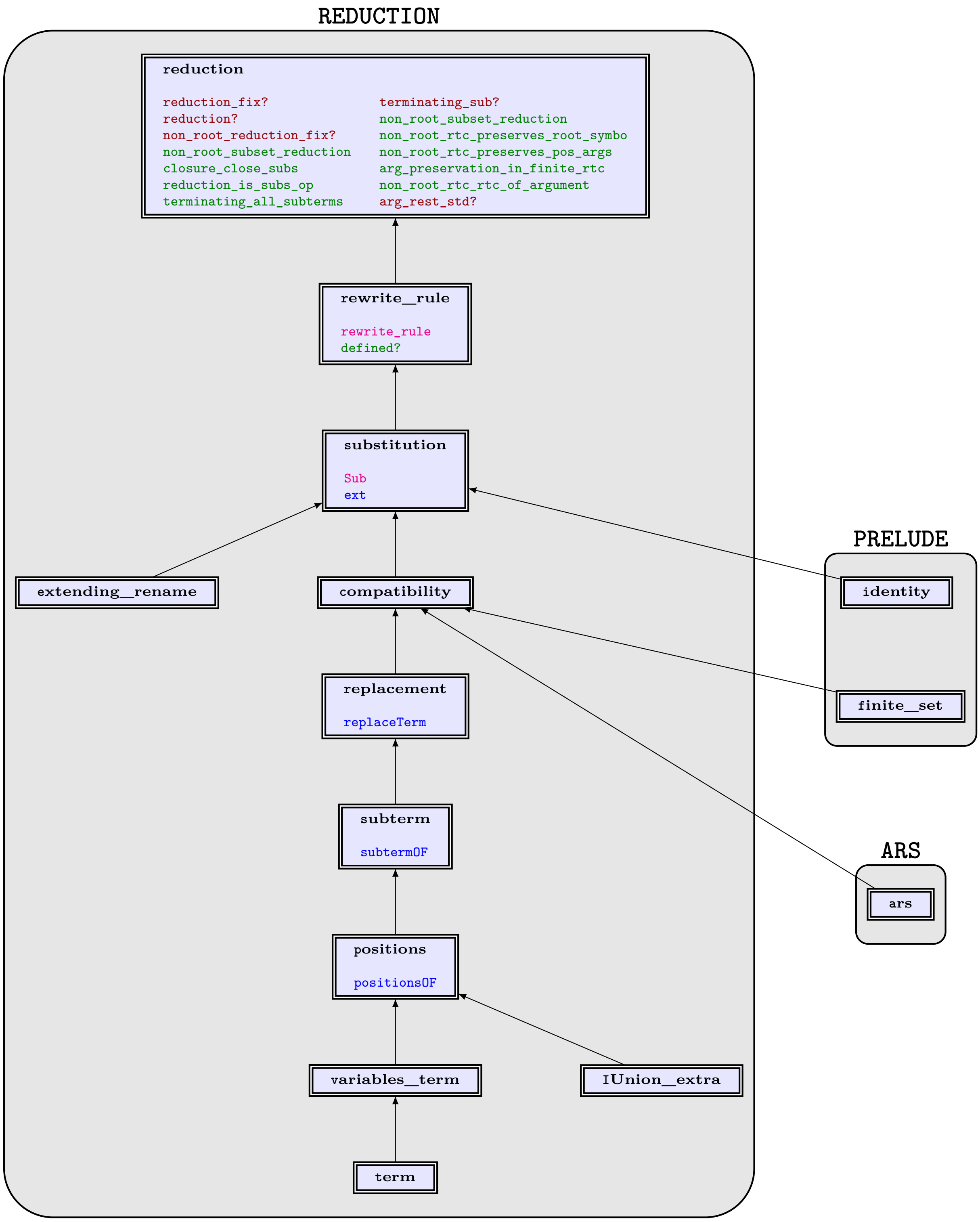


Figure 5.5: The REDUCTION sublibrary

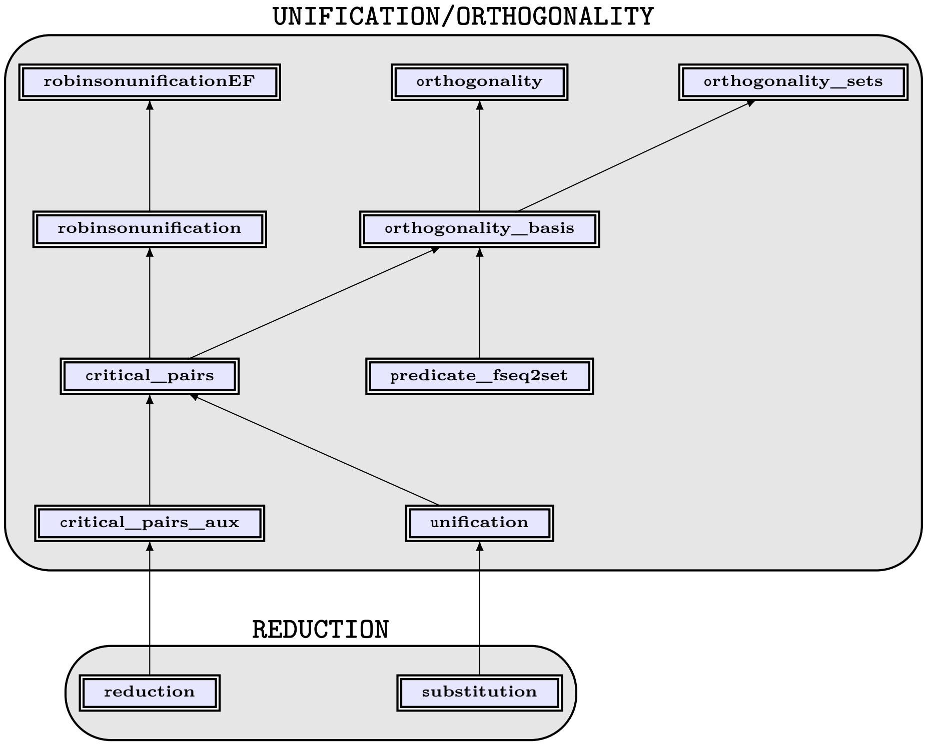


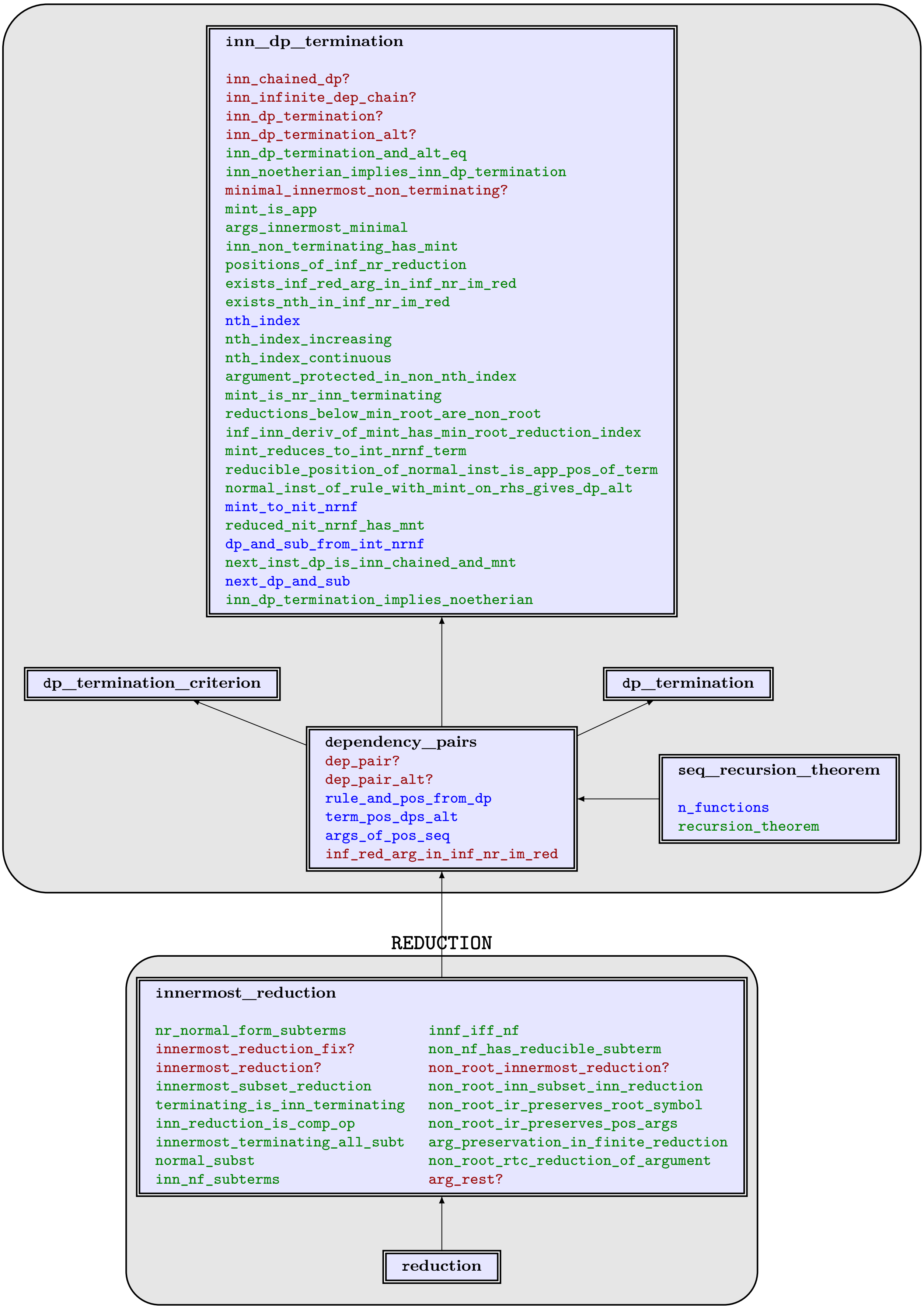
Figure 5.6: The `TRS Properties` sublibrary

The sublibrary `DEPENDENCY PAIRS` brings theory `innermost_reduction` which have the specification of relations `innermost_reduction_fix?`, `innermost_reduction?` and `non_root_innermost_reduction?` given in Definition 2.2.9 and in Specification 4.3. predicate `arg_rest?` and several results, such as:

- the one regarding the normal form for reduction of terms that are normal form for the non-root reduction relation (used to specify the innermost reduction relation itself) in lemma `nr_normal_form_subterms`,
- the relation between normal form and innermost normal form of terms, given in lemma `innf_iff_nf`,
- the reducibility of subterms in terms that are not in normal form in lemma `non_nf_has_reducible_subterm`,
- the fact that the innermost and non-root innermost relations are subsets of the ordinary reduction relation (lemmas `innermost_subset_reduction` and `non_root_inn_subset_inn_reduction`),
- the relation between terminating and innermost terminating terms, given in lemma `terminating_is_inn_terminating`,

- the preservation of the root symbol and the argument positions when non-root innermost reductions are performed (lemmas `non_root_ir_preserves_root_symbol` and `non_root_ir_preserves_pos_args`),
- the compatibility with contexts (lemma `inn_reduction_is_comp_op`),
- the innermost termination of subterms of innermost terminating terms (lemma `innermost_terminating_all_subterms`),
- the preservation of arguments when a derivation takes place in a specific argument (lemma `arg_preservation_in_finite_reduction`),
- the fact that terms in non-root normal form provide normal substitutions (lemma `normal_subst`),
- the propagation of the derivations to arguments when the whore term is derivated in lemma `non_root_rtc_reduction_of_argument` and
- the non-innermost reducibility of subterms of terms that are in innermost normal form given in lemma `inn_nf_subterms`.

The theories `dependency_pairs` and `inn_dp_termination` are also in the sublibrary `DEPENDENCY PAIRS`. The former includes the basic specifications of dependency pais and functions to extract the rule and *rhs* position that originated a given DP (`dep_pair?`, `dep_pair_alt?` and `term_pos_dps_alt` in Specifications 4.5, 4.6 and 5.2). The latter contains the majority of the elements discussed in Sections 5.1 and 5.2. Among them, the predicates that specify innermost chained dependency pais (`inn_chained_dp?`, from Definition 3.3.3 and Specification 4.8), innermost dependency chains and innermost termination (`inn_infinite_dep_chain?`, `inn_dp_termination?` and `inn_dp_termination_alt?` given in Definitions 3.3.3 and 3.3.4 and Specifications 4.9 and 4.8) and the main necessity lemma (`inn_Noetherian_implies_inn_dp_termination` in Specification 5.1). This theory also includes the functions to extract the elements required to construct the DPs for the sufficiency proof, such as `mint_to_nit_nrnf`, `dp_and_sub_from_int_nrnf` and `next_dp_and_sub` (Specifications 5.13, 5.15 and 5.17), the lemmas that allow the correction of such functions and the final sufficiency result `inn_dp_termination_implies_noetherian` (including the lemmas `inn_non_terminating_has_mint` in Specification 5.4, `mint_is_nr_inn_terminating` in Specification 5.9, `mint_reduces_to_int_nrnf_term` in Specification 5.11, `normal_inst_of_rule_with_mint_on_rhs_gives_dp_alt` in Specification 5.12, `reduced_nit_nrnf_has_mnt` in Specification 5.14 and `next_inst_dp_is_inn_chained_and_mnt` in Specification 5.16; and also the auxiliary lemmas that allow such proofs). This sublibrary also include the theory `dp_termination`, that formalizes the DP Criterion for ordinary reductions.



inn_dp_termination
inn_chained_dp?
inn_infinite_dep_chain?
inn_dp_termination?
inn_dp_termination_alt?
inn_dp_termination_and_alt_eq
inn_noetherian_implies_inn_dp_termination
minimal_innermost_non_terminating?
mint_is_app
args_innermost_minimal
inn_non_terminating_has_mint
positions_of_inf_nr_reduction
exists_inf_red_arg_in_inf_nr_im_red
exists_nth_in_inf_nr_im_red
nth_index
nth_index_increasing
nth_index_continuous
argument_protected_in_non_nth_index
mint_is_nr_inn_terminating
reductions_below_min_root_are_non_root
inf_inn_deriv_of_mint_has_min_root_reduction_index
mint_reduces_to_int_nrnf_term
reducible_position_of_normal_inst_is_app_pos_of_term
normal_inst_of_rule_with_mint_on_rhs_gives_dp_alt
mint_to_nit_nrnf
reduced_nit_nrnf_has_mnt
dp_and_sub_from_int_nrnf
next_inst_dp_is_inn_chained_and_mnt
next_dp_and_sub
inn_dp_termination_implies_noetherian
dp_termination_criterion
dp_termination
dependency_pairs
dep_pair?
dep_pair_alt?
rule_and_pos_from_dp
term_pos_dps_alt
args_of_pos_seq
inf_red_arg_in_inf_nr_im_red
seq_recursion_theorem
n_functions
recursion_theorem
REDUCTION
innermost_reduction
nr_normal_form_subterms
innermost_reduction_fix?
innermost_reduction?
innermost_subset_reduction
terminating_is_inn_terminating
inn_reduction_is_comp_op
innermost_terminating_all_subt
normal_subst
inn_nf_subterms
innf_iff_nf
non_nf_has_reducible_subterm
non_root_innermost_reduction?
non_root_inn_subset_inn_reduction
non_root_ir_preserves_root_symbol
non_root_ir_preserves_pos_args
arg_preservation_in_finite_reduction
non_root_rtc_reduction_of_argument
arg_rest?
reduction

5.5 Related work: other formalizations of DPs

Formalizations of the theorem of soundness and completeness of DPs (DP theorem, for short) are available in several proof assistants. In [BK11], Blanqui and Koprowski described a formalization of the DP theorem for the ordinary reduction relation that is part of the CoLoR library developed in Coq for certifying proofs of termination. The formalized result is the DP theorem for the ordinary reduction relation, and not for the innermost termination. The proof in [BK11], as the current formalization, uses the non-root reduction relation (internal reduction) and the reduction at root position relation (head reduction). Instead of building infinite chains from infinite derivations, it assumes a well-founded relation over the set of chained DPs to conclude Noetherianity of the ordinary reduction relation.

The library library Coccinelle [CCF+07] contains a Coq formalization for the DP Criterion. This formalization includes a relation between instances of *lhs* of DPs and proves the equivalence between well-foundedness of this relation and well-foundedness of the reduction relation of a given TRS. Their work also avoids using tuple symbols to avoid root reduction between chained DPs, instead, uses instances of the lists of arguments for the *lhs*'s and *rhs*'s of DPs related by the reflexive-transitive closure of the rewriting relation. The formalization also considers a refinement of the notion of DPs, which avoids DPs generated by a rules where the *rhs* of the DP appears also as a subterm of the *lhs* of the rule.

The $\mathsf{T_TT_2}$ tool implements techniques such as KBO and polynomial interpretations in a modular way following the Dependency Pair Framework [KSZM09] and allows the user to control the termination methods applied by configurable strategies.

The DP Criterion is automated in the tool AProVe [GSKT06], and also adapted to deal with termination of term rewriting systems modulo AC operators [YSTK16]. AProVe is a powerfull system for automated termination and complexity proofs. It also provides simpler termination analysis techniques, such as reduction orderings based on multiset orderings and integer transition systems among others ([DM79b], [TAN12], [SGB+17]). AProVe also can deal with termination and complexity analysis over Java, C, Haskell and Prolog programs [GBE+14, GAB+17].

A formalization of the DP theorem for the ordinary reduction relation is also present in the proof assistant Isabelle, as part of the library for rewriting IsaFoR briefly described in [ST10]. In this formalization the original signature of the TRS is extended with new tuple symbols for substituting the defined symbols (see comments after Definition 3.3.1 of DPs), which implies the analysis of additional properties of the new term rewriting system induced over the extended signature and also properties relating this new rewriting system with the original one. The proof, as in the current formalization, builds an infinite chain from an infinite derivation and vice-versa. This work brings interesting features, such as the use of the same refinement of DPs as the formalization in Coccinelle and also formalization for the Q-restricted rewriting relation, providing a general result that has as corollaries the results explicitly proved before for both the DP theorem for the ordinary and the innermost reduction relations.

The formalization for the termination of Q-restricted relation is used to provide a sound environment to certify concrete termination proofs in an automatic way by the tool CeTA [TS09]. Formalization of the DP Criterion for the ordinary rewriting relation is also included in the PVS theory `TRS` (as mentioned in Section 5.3), but as mentioned in the introduction, the emphasis in this work is on the innermost case since it is the one related to the operational semantics of first-order functional the `PVS0` language (eager evaluation) which models first-order PVS specifications.

Chapter 6

Formalization of Termination Criteria in PVS0

The theory PVS0 is also very extensive and have been gathering efforts from several researches in order to provide an adequate language to reason over the PVS specification language in a simplified way [1] [MARM$^+$21]. A lot of the efforts are in providing means to automate termination proofs, and the equivalence between the termination criteria specified in Section 4.2 is a fundamental part of this development. The formalizations on such equivalences are summarized in this Chapter, following from the semantic notions of termination to more constructive means to analyze such property.

6.1 Equivalence between semantic criteria

Several auxiliary properties have been formalized to prove the equivalence between the termination criteria T_ε and T_χ given is Section 4.2.1. For instance, the type of the function eval_expr abbreviated as χ, is shown to satisfy the two recursive judgments below. The first one, eval_expr_ge_n_j, means that whenever the type of χ is some value different from $\diamondsuit$, obtained allowing a number $n > 0$ of nested recursive calls, this value remains the same when one allows a number greater than or equal to the number of nested recursive calls provided. The second one, eval_expr_semantic_rel_j, means that when the type of χ is some value different from $\diamondsuit$, this value is exactly the one that satisfies the predicate ε. The proofs are by induction on PVS0 expressions expanding the definitions of χ and ε.

Specification 6.1: Typing results over evaluation of PVS0 expressions

eval_expr_ge_n_j $=$ $\chi(\mathtt{expr}, \mathtt{v}_{in}, \mathtt{n})$HAS_TYPE $\{myv : \mathtt{T} \cup \{\diamondsuit\} \mid some?(myv) \Rightarrow$ $\mathtt{n} > 0 \wedge \forall (\mathtt{m} \geq \mathtt{n}) : myv = \chi(\mathtt{expr}, \mathtt{v}_{in}, \mathtt{m})\}$	eval_expr_semantic_rel_j $=$ $\chi(\mathtt{expr}, \mathtt{v}_{in}, \mathtt{n})$HAS_TYPE $\{myv : \mathtt{T} \cup \{\diamondsuit\}; \mid some?(myv) \Rightarrow$ $\varepsilon(\mathtt{expr}, \mathtt{v}_{in}, \mathtt{get\_val(myv)})\}$

[1]Indeed, this is a joint work with the Formal Methods group at NASA Langley and the Formal Methods group at Universidade de Brasília

From these judgments, it is possible to prove the relation between the two notions of evaluation in the two lemmas specified below. The first portion of the currying of ε and χ are properly instantiated with the elements of program `def`.

Specification 6.2: Relations between Semantic Evaluation and Evaluation by number of nested calls of PVS0 expressions

$\texttt{semantic\_rel\_eval\_expr} : \text{LEMMA}$ $\forall(\texttt{def}, \texttt{expr}, \texttt{v}_{in}, \texttt{v}_{out}):$ $\varepsilon(\texttt{expr}, \texttt{v}_{in}, \texttt{v}_{out}) \Rightarrow$ $\exists(\texttt{n}) : \mathit{some?}(\chi(\texttt{expr}, \texttt{v}_{in}, \texttt{n})) \wedge$ $\texttt{v}_{out} = \mathit{get\_val}(\chi(\texttt{expr}, \texttt{v}_{in}, \texttt{n}))$	$\texttt{eval\_expr \_semantic\_rel} : \text{LEMMA}$ $\forall(\texttt{def})(\texttt{v}_{in}, \texttt{expr}, \texttt{v}_{out})(\texttt{n}):$ $\mathit{some?}(\chi(\texttt{expr}, \texttt{v}_{in}, \texttt{n})) \wedge$ $\texttt{v}_{out} = \mathit{get\_val}(\chi(\texttt{expr}, \texttt{v}_{in}, \texttt{n})) \Rightarrow$ $\varepsilon(\texttt{expr}, \texttt{v}_{in}, \texttt{v}_{out})$

The lemma `eval_expr_semantic_rel` has a trivial proof, being enough just to use the type judgement `eval_expr_semantic_rel_j`, while lemma `semantic_rel_eval_expr` is proved inductively on the structure of the inductive predicate ε. Thus, the induction requires a predicate that states invariance properties over the arguments that undergo alterations in the definition (i.e., `expr`, $\texttt{v}_{in}$ and $\texttt{v}_{out}$). The required predicate is given as the predicate in Specification 6.3.

Specification 6.3: Invariance predicate for evaluating PVS0 expressions

$$
\begin{aligned}
&P(\texttt{expr}, \texttt{v}_{in}, \texttt{v}_{out}) = \\
&\quad \exists(\texttt{n}) : \mathit{some?}(\chi(\texttt{expr}, \texttt{v}_{in}, \texttt{n})) \wedge \texttt{v}_{out} = \mathit{get\_val}(\chi(\texttt{expr}, \texttt{v}_{in}, \texttt{n}))
\end{aligned}
$$

Thus, the inductive scheme generated by PVS is given below.

Specification 6.4: Inductive Scheme over Semantic Evaluation of PVS0 expressions

$$
\begin{array}{lll}
(((\texttt{cnst?}(\texttt{expr}) & \wedge\ \texttt{v}_{out} = \mathit{get\_val}(\texttt{expr})) & \vee \\
\ (\texttt{vr?}(\texttt{expr}) & \wedge\ \texttt{v}_{out} = \texttt{v}_{in}) & \vee \\
\ (\texttt{op1?}(\texttt{expr}) & \wedge\ \exists(\texttt{v}_1) : \varepsilon(\mathit{get\_arg}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_1) \wedge & \\
 & \qquad P(\mathit{get\_arg}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_1) \wedge & \\
 & \qquad\quad \texttt{v}_{out} = O_1(\mathit{get\_op}(\texttt{expr}))(\texttt{v}_1)) & \\
\ (\texttt{op2?}(\texttt{expr}) & \wedge\ \exists(\texttt{v}_1, \texttt{v}_2) : \varepsilon(\mathit{get\_arg1}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_1) \wedge & \\
 & \qquad P(\mathit{get\_arg1}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_1) \wedge & \\
 & \qquad \varepsilon(\mathit{get\_arg2}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_2) \wedge & \\
 & \qquad P(\mathit{get\_arg2}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_2) \wedge & \\
 & \qquad\quad \texttt{v}_{out} = O_2(\mathit{get\_op}(\texttt{expr}))(\texttt{v}_1, \texttt{v}_2)) & \\
\ (\mathit{ite?}(\texttt{expr}) & \wedge\ \exists(\texttt{v}_1) : \varepsilon(\mathit{get\_cond}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_1) \wedge & \\
 & \qquad P(\mathit{get\_cond}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_1) \wedge & \\
 & \qquad ((\texttt{v}_1 \neq \Diamond \wedge\ \varepsilon(\mathit{get\_if}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_{out}) \wedge & \\
 & \qquad\ P(\mathit{get\_if}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_{out})) \vee & \\
 & \qquad (\texttt{v}_1 = \Diamond \wedge\ \varepsilon(\mathit{get\_else}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_{out}) \wedge & \\
 & \qquad\ P(\mathit{get\_else}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_{out})))) \vee & \\
\ (\mathit{rec?}(\texttt{expr}) & \wedge\ \exists(\texttt{v}_1) : \varepsilon(\mathit{get\_arg}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_1) \wedge & \\
 & \qquad P(\mathit{get\_arg}(\texttt{expr}), \texttt{v}_{in}, \texttt{v}_1) \wedge & \\
 & \qquad \varepsilon(\texttt{e}_f, \texttt{v}_1, \texttt{v}_{out}) \wedge & \\
 & \qquad P(\texttt{e}_f, \texttt{v}_1, \texttt{out}))) \Rightarrow\ P(\texttt{expr}, \texttt{v}_{in}, \texttt{v}_{out})) & \Rightarrow \\
(\forall(\texttt{expr}, \texttt{v}_{in}, \texttt{v}_{out}) : & \varepsilon(\texttt{expr}, \texttt{v}_{in}, \texttt{v}_{out}) \Rightarrow P(\texttt{expr}, \texttt{v}_{in}, \texttt{v}_{out})) &
\end{array}
$$

The predicate P makes straightforward the inductive step, since one has ε already and wants to prove χ, that is exactly the succedent of the induction hypothesis. The other part of the proof proceeds by case analysis accordingly to the ε definition, where one must provide the adequate n for the size of nested recursions to be allowed in each case. For variables and constants, $\mathtt{n} = 1$; for unary and binary operators and for branching instructions, n is the maximum number of nested recursions necessary to evaluate its arguments; finally, for recursive expressions, the required number of nested recursions is one more than necessary to evaluate its argument. Since the maximum number of recursions to evaluate arguments is used when the expression has more than one argument, the type judgement eval_expr_ge_n_j is needed to ensure that the result will be the same.

The formalization of equivalence between the two semantic notions of termination, specified as below, is then established by the application of the two lemmas above.

Specification 6.5: Semantic Evaluation Equivalence of PVS0 expressions

$$\begin{array}{l}\mathtt{eval\_expr\_terminates} : \mathtt{LEMMA} \\ \quad \forall(\mathtt{expr}) : T_{\chi}(\mathtt{expr}) \Leftrightarrow T_{\varepsilon}(\mathtt{expr})\end{array}$$

6.2 Equivalence between TCC termination and semantic termination

First, to prove that semantic termination (i.e., either T_{χ} or T_{ε}) implies TCC termination (T_{ς}), a function is specified that provides the minimum number of nested recursive calls needed to evaluate an output value for a *determined* pair of PVS0 definition and input value. This is done through evaluation with the function χ accordingly to the function mu specified as below.

Specification 6.6: Minimum number of recursive calls for evaluation of PVS0 definitions

$$\mathtt{mu(def)}(\mathtt{v}_{in} \mid \mathtt{determined?}(\mathtt{def}, \mathtt{v}_{in})) = min(\{\mathtt{n} \mid \chi(\mathtt{def\grave{}4}, \mathtt{v}_{in}, \mathtt{n}) \neq \Diamond\})$$

When a minimum number n can be used to evaluate a given PVS0 program definition, then the minimum number of nested recursive calls necessary to evaluate the argument of some recursive subexpression of this definition is proved to be smaller than n in lemma rec_mu_decreasing specified below.

Specification 6.7: Decreasingness of mu

$$\begin{array}{l}\mathtt{rec\_mu\_decreasing} : \mathtt{LEMMA} \\ \quad \forall(\mathtt{v}_{in})(\mathtt{n})(\mathtt{def}|T_{\varepsilon}(\mathtt{def\grave{}4}))(\mathtt{path} : (P(\mathtt{def\grave{}4}))) : \\ \quad\quad (\mathtt{mu(def)}(\mathtt{v}_{in}) = \mathtt{n} \wedge \\ \quad\quad\; \mathcal{C}(\mathtt{path\_conditions}(\mathtt{def\grave{}4}, \mathtt{path}), \mathtt{v}_{in}) \wedge \\ \quad\quad\; rec?(\mathtt{subterm\_at}(\mathtt{def\grave{}4}, \mathtt{path}))) \Rightarrow \\ \quad\quad \forall(\mathtt{v}_{out}) : \varepsilon(get\_arg(\mathtt{subterm\_at}(\mathtt{def\grave{}4}, \mathtt{path})), \mathtt{def\grave{}4}, \mathtt{v}_{in}, \mathtt{v}_{out}) \Rightarrow \\ \quad\quad\quad\quad \mathtt{mu(def)}(\mathtt{v}_{out}) < \mathtt{n}\end{array}$$

The lemma above is formalized through an auxiliary lemma, that is proved by induction in the structure of PVS0 expressions occurring in a PVS0 program. The auxiliary lemma states that whenever the expression, say `expr`, in the program `pvs0`, can be evaluated allowing `n` nested recursive calls, and there is a subexpression of `expr`, say `sexpr`, which is a recursive call, whose conditions hold (i.e., evaluate to true), then the arguments of this recursive call can be evaluated allowing minimum `n-1` nested recursive calls.

Then, for semantic terminating `PVS0` program definitions, this minimum number of nested recursive calls is proved to decrease over parameters and arguments of each CC `cc` of the `PVS0` program for which the conditions hold. This decreasing condition of semantic termination definitions is formalized as the lemma `mu_soundness` below. This is a straightforward consequence of the previous lemma.

Specification 6.8: Soundness of `mu`

$$\begin{array}{l}
\mathtt{mu_soundness} : \mathtt{LEMMA} \\
\quad T_{\varepsilon}(\mathtt{def`4}) \Rightarrow \\
\quad \forall(\mathtt{cc}, \mathtt{v}_{in}, \mathtt{v}_{out}) : \\
\qquad \varepsilon(get_arg(\mathtt{cc`}rec\_expr), \mathtt{v}_{in}, \mathtt{v}_{out}) \wedge \mathcal{C}(\mathtt{def}, \mathtt{cc`}cnds, \mathtt{v}_{in}) \\
\qquad \Rightarrow \mathtt{mu(pvs0)}(\mathtt{v}_{out}) < \mathtt{mu(def)}(\mathtt{v}_{in})
\end{array}$$

With this result, `mu` can be used as the `wfm` (using the order $>$ over $\mathbb{N}$) required to guarantee termination according T_{ς}. This concludes the first side of the equivalence proof, formalized as lemma `terminates_implies_pvs0_tcc` = $T_{\varepsilon}(\mathtt{def`4}) \Rightarrow T_{\varsigma}(\mathtt{def})$.

Second, to formalize the converse, that is $T_{\varsigma} \Rightarrow T_{\varepsilon}$, the goal is to relate the values used as arguments of nested recursive calls when a `PVS0` program is evaluated. The predicate `lt_val` below relates arguments of direct recursive calls with the input argument between a recursive definition, according to a given `wfm` and associated ordering `lt` (used to guarantee TCC termination).

Specification 6.9: `lt_val`

$$\begin{array}{l}
\mathtt{lt_val(def)(wfm)}(\mathtt{v}_{out}, \mathtt{v}_{in}) = \\
\quad \exists(\mathtt{cc} | (\mathtt{pvs0_tcc_valid_cc(def`4)(cc)})) : \\
\qquad \varepsilon(get_arg(\mathtt{cc`}rec\_expr), \mathtt{v}_{in}, \mathtt{v}_{out}) \wedge \mathcal{C}(\mathtt{def}, \mathtt{cc`}cnds, \mathtt{v}_{in}) \wedge \mathtt{lt}(\mathtt{wfm}(\mathtt{v}_{out}), \mathtt{wfm}(\mathtt{v}_{in}))
\end{array}$$

Notice that `lt_val` only relates input arguments and arguments of recursive calls regarding the static analysis used by TCC termination. To extend this relation to arguments that appear in recursive calls when a `PVS0` definition is evaluated, that is to arguments in nested recursive calls, the transitive n-closure of the inverse of this relation is built using the predicate `gt_n`:

Specification 6.10: `gt_n`

$$\begin{array}{ll}
\mathtt{gt_n(lt_val)(n)(a,b)} = & (\mathtt{n} = 1 \wedge \mathtt{lt_val(b,a)}) \vee \\
& (\mathtt{n} > 1 \wedge \exists(\mathtt{c}) : \mathtt{lt_val(c,a)} \wedge \mathtt{gt_n(lt_val)(n}-1\mathtt{)(c,b)})
\end{array}$$

Indeed, the transitive closure is the relation $\exists(n)$: `gt_n(lt_val)(n)`. Then, the function `Omega` is used to specify the height of the tree of evaluation of a recursive definition for a given input value as below. Notice that this corresponds also to the maximum number of nested recursive calls generated in the semantic evaluation of the recursive definition with the given input.

Specification 6.11: `Omega`

$$\texttt{Omega}(\texttt{v}_{in}) = min(\texttt{n} : above(0) \mid \forall(\texttt{v}_{ar}) : \neg\texttt{gt\_n}(\texttt{lt\_val})(\texttt{n})(\texttt{v}_{in}, \texttt{v}_{ar}))$$

The lemma `omega_is_eval_ub` shows that the `Omega` function provides an upper bound for the length of nested recursive calls for TCC terminating `PVS0` definitions, which also guarantees the semantic evaluation of some value different from $\diamondsuit$.

Specification 6.12: `omega_is_eval_ub`

$$\begin{array}{ll}
\multicolumn{2}{l}{\texttt{omega_is_eval_ub} : \texttt{LEMMA}} \\
\quad \varsigma(\texttt{def}, \texttt{wfm}) \Rightarrow \forall(\texttt{expr}, \texttt{v}_{in}, \texttt{path}) : & \texttt{expr} = \texttt{subterm_at}(\texttt{def`4}, \texttt{path}) \wedge \\
 & \mathcal{C}(\texttt{def}, \texttt{path_conditions}(\texttt{def`4}, \texttt{path}), \texttt{v}_{in}) \Rightarrow \\
\qquad\qquad \exists(\texttt{n} \leq \texttt{Omega}(\texttt{v}_{in})) : & some?(\chi(\texttt{expr}, \texttt{v}_{in}, \texttt{n})) \wedge \\
 & \varepsilon(\texttt{expr}, \texttt{v}_{in}, val(\chi(\texttt{expr}, \texttt{v}_{in}, \texttt{n})))
\end{array}$$

The proof is by induction on a lexicographic order over the size of the `PVS0` expressions to be evaluated and the measure of the input value. For expressions that are not recursive, the proof is trivial, since the number of nested recursive calls will be the same as for its arguments, what is given by induction hypothesis. As for recursive expressions, since predicate ς holds, the semantic evaluation of this function is use in order to provide the measure necessary for the `Omega` function. Then the result provided for `Omega` applied to the input value used for the expression evaluation is used as the number of nested recursive calls. The induction hypothesis instantiated with the output value evaluated for the first input value provided provides a number of nested recursive calls for the recursive expression argument, that is indeed smaller than the initial result of `Omega`, thus concluding the proof.

The theorem `pvs0_tcc_implies_terminates`, specified below, is a direct result from the equivalence between T_ε and T_χ and lemma `omega_is_eval_ub`.

Specification 6.13: TCC vs Semantic Termination for PVS0

$$\begin{array}{l}
\texttt{pvs0_tcc_implies_terminates} : \texttt{LEMMA} \\
\quad T_\varsigma(\texttt{def}) \Rightarrow T_\varepsilon(\texttt{def`4})
\end{array}$$

6.3 Equivalence between TCC and SCP technologies

It will be described how the equivalences between TCC and SCP termination and between SCP and CCG termination were formalized.

TCC versus SCP

The relation between TCC termination T_ς and SCP termination for PVS0 program definitions is specified in lemmas below.

Specification 6.14: Equivalence between TCC and SCP for PVS0

$\texttt{pvs0\_tcc\_implies\_scp} : \texttt{LEMMA}$
$T_\varsigma(\texttt{def}) \Rightarrow \texttt{scp\_termination\_pvs0}(\texttt{def})$

$\texttt{scp\_implies\_pvs0\_tcc} : \texttt{LEMMA}$
$\texttt{scp\_termination\_pvs0}(\texttt{def}) \Rightarrow T_\varsigma(\texttt{def})$

For the proof of the first lemma, from TCC termination it is provided a well-founded measure, say `wfm`, that decreases over the parameters and arguments of every recursive call. This measure is used to build the required relation over the values regarding the evaluation mechanisms using the predicate `lt_val` described in Section 6.2. Then, to conclude the proof, the relation `lt_val` built using `wfm` is used to instantiate the existential quantifier required for the predicate `SCP` presented in Section 4.2.3.

As for the second, the proof requires providing a well-founded measure to be used in TCC termination. Since one has SCP termination, the possible sequences of CCs are finite. Therefore, for each context of the PVS0 program definition it is possible to relate its parameters and arguments accordingly to the input and output values through the evaluation mechanisms. This relation is given by predicate `R` below.

Specification 6.15: Relating parameters and arguments

$\texttt{R}(\texttt{def})(\texttt{v}_{out}, \texttt{v}_{in}) =$
$\exists(\texttt{cc} \mid \texttt{pvs0\_tcc\_valid\_cc}(\texttt{def})(\texttt{cc})) : \mathcal{C}(\texttt{def}, \texttt{cc}`cnds, \texttt{v}_{in}) \wedge \varepsilon(\texttt{cc}`actuals, \texttt{v}_{in}, \texttt{v}_{out})$

Then, similarly to the proof for $T_\varsigma \Rightarrow T_\varepsilon$ on Section 6.2, this predicate only relates parameters and arguments of CCs in a static analysis. Notice that, once again, the predicate can be extended to relate arguments of nested recursive calls. This is done by using the n-closure `gt_n` as specified in Section 6.2, but now with `R` as its argument.

Finally, it is possible to use the height of the tree of evaluation of recursive calls as an adequate measure (over naturals) required for TCC. This height is given by function the `Omega` defined over the n-closure of predicate `R`, that captures the notion of SCP, and is given as below:

Specification 6.16: Hight of evaluation tree

$\texttt{Omega}(\texttt{v}_{in}) = min(\texttt{n} : above(0) \mid \forall(\texttt{v}_{ar}) : \neg\texttt{gt\_n}(\texttt{R})(\texttt{n})(\texttt{v}_{in}, \texttt{v}_{ar}))$

Notice that the premises of the definition of TCC termination (see Section 4.2.2) coincide with the predicate `R`. Then, it only remains to prove that `Omega` is a well-founded measure. Since `Omega` is defined only over well-founded relations, it is enough to prove that the relation given by predicate `R` is indeed well-founded. This proof is given by the SCP termination hypothesis, through the non-existence of infinite decreasing sequences related by `R`.

This concludes the proof of equivalence between TCC and SCP termination.

SCP versus CCG

Consider two generic evaluation mechanisms `cond_eval` and `sem_eval`.

First, consider that the SCP termination predicate `scp_termination?` (see Section 4.2.3) holds for these evaluation mechanisms. Then, it is proved that using these evaluation mechanisms the predicate `ccg_termination` also holds. This can be achieved since every graph whose vertices are CCs will have measures allowing this termination criterion, as specified in lemma `scp_implies_ccg_termination` below.

Specification 6.17: SCP implies CCG

$$\begin{array}{l}\texttt{scp\_implies\_ccg\_termination} : \text{LEMMA} \\ \quad \forall(\texttt{dg}) : \exists(\texttt{measures}) : \texttt{ccg\_termination?}(\texttt{make\_ccg}(\texttt{dg}, \texttt{measures}))\end{array}$$

For the proof, it must be provided the decreasing combination of measures that guarantees CCG termination. In order to do this, it is necessary first to build the family of measures that can be chosen to make this combination. In this formalization, the family of measures is composed by a single measure given by the function `Omega` over a generic version of a predicate `R`, similarly to the one previously used in this section, given as below, where `cc` is a vertex of the graph:

Specification 6.18: `R`

$$\texttt{R}(\texttt{v}_{out}, \texttt{v}_{in}) = \exists(\texttt{cc}) : \texttt{cond\_eval}(\texttt{cc}`cnds, \texttt{v}_{in}) \wedge \texttt{sem\_eval}(\texttt{cc}`actuals, \texttt{v}_{in}, \texttt{v}_{out})$$

Since the family of measures is a singleton, the combination to be associated with every (walk and) circuit is a sequence whose length is the same as the length of the circuit and whose components are always the unique available index: zero. Since SCP holds for the evaluation mechanisms, the proof is completed by the well-foundedness of `Omega` (that as in the previous equivalence formalization is a consequence of the well-foundedness of `R`).

Secondly, the converse is specified in lemma `ccg_termination_implies_scp` below.

Specification 6.19: CCG implies SCP

$$\begin{array}{l}\texttt{ccg\_termination\_implies\_scp} : \text{LEMMA} \\ \quad \forall(\texttt{dg}) : (\exists(\texttt{measures}) : \\ \qquad \texttt{ccg\_termination?}(\texttt{make\_ccg}(\texttt{dg}, \texttt{measures}))) \Rightarrow \texttt{scp\_termination?}(\texttt{sem\_eval}, \texttt{cond\_eval})\end{array}$$

Specification 6.20: Extracting infinite descent

$$\begin{array}{l}
\texttt{extract_infinite_descent} = \\
\quad \forall (J : \texttt{IncSub}, F : [\texttt{nat} \rightarrow \texttt{MT}], K : \texttt{nat}) : \\
\qquad \neg((\forall (i) : \exists (j) : J(i) + j < J(i+1) \wedge \texttt{gt}(F(J(i)+j), F(J(i)+j+1))) \wedge \\
\qquad\quad (\forall (i) : i \geq K \Rightarrow \texttt{ge}(F(i), F(i+1))))
\end{array}$$

$$\begin{array}{l}
\texttt{build_infinite_descent} = \\
\quad \neg(\exists (J : \texttt{IncSub}, F : [\texttt{nat} \rightarrow [\texttt{nat} \rightarrow [\texttt{Val} \rightarrow \texttt{MT}]]], vals : [\texttt{nat} \rightarrow \texttt{Val}]) : \\
\qquad \forall (j) : (\forall (i) : i < J(j+1) - J(j) \Rightarrow \\
\qquad\qquad\qquad \texttt{ge}(F(j)(i)(vals(J(j)+i)), F(j)(i+1)(vals(J(j)+i+1)))) \wedge \\
\qquad\qquad (\exists (i) : i < J(j+1) - J(j) \wedge \\
\qquad\qquad\qquad \texttt{gt}(F(j)(i)(vals(J(j)+i)), F(j)(i+1)(vals(J(j)+i+1)))) \wedge \\
\qquad\qquad F(j)(J(j+1) - J(j)) = F(j+1)(0))
\end{array}$$

$$\begin{array}{l}
\texttt{ccg_pigeonhole} = \\
\quad \forall (dg, (ccs : \texttt{Seq_cc(dg)})) : \exists (J : [\texttt{nat} \rightarrow \texttt{nat}]) : \\
\qquad (\forall (i, j : \texttt{nat}) : i < j \Rightarrow J(i) < J(j) \wedge ccs(J(i)) = ccs(J(j)))
\end{array}$$

The proof is by contraposition and consists of building a circuit that decreases infinitely from the infinite sequence of calling contexts provided by SCP (that requires the lemmas in Specification 6.20). Since the number of possible calling contexts is finite, whenever there is an infinite sequence of CCs some of those appear infinitely many times in the sequence. Thus, the same vertex of the graph must be assigned to every occurrence of a given context in the infinite sequence.

Note that even though the equivalence of SCP and CCG termination is given generically, using this result for `PVS0` programs is straightforward; in fact, this is achieved by a simple parameterization and adaptation of lemmas `scp_implies_ccg_termination` and `ccg_termination_implies_scp`. The necessary adjustments are related to how the CCG from a `PVS0` program definition is built and how generic contexts and specific calling contexts are related. This can be seen in the formalization of lemmas `ccg_implies_scp_pvs0` and `scp_implies_ccg_pvs0`.

Thus, it is possible to state the relation between CCG and TCC termination for `PVS0` programs, such as in lemmas `ccg_implies_pvs0_tcc` and `pvs0_tcc_implies_ccg` . The former is stated below, and its proof is obtained through the application of two intermediate results, given by lemmas `scp_implies_pvs0_tcc` and `ccg_implies_scp_pvs0`. The proof of the later follows the same principle.

Specification 6.21: CCG implies TCC for `PVS0` Programs

$$\texttt{ccg\_implies\_pvs0\_tcc} = \texttt{ccg\_termination\_pvs0}(\texttt{def}) \Rightarrow T_{\varsigma}(\texttt{def})$$

6.4 NASA PVS Library - `PVS0` Theory Summary

The syntax of `PVS0` expressions, gived in Section 4.2, is specified in the file `PVS0Expr` (see Scheme in Figure 6.1). Definitions of the PVS0 language such as the two different notions of semantic termination (terminates_expr and eval_expr_termination), described in Section 4.2.1, and their equivalence formalized by the lemma eval_expr_terminates along with the auxiliary results presented in Section 6.1 (eval_expr_ge_n_n, eval_expr_semantic_rel_j, semantic_rel_eval_expr and eval_expr_semantic_rel), are specified in theory **pvs0_expr**. The former definition of semantic termination is given over the predicate semantic_rel_expr, and the latter over the recursive function eval_expr.

Theory **pvs0_lang** provides the type PVS0 used for PVS0 program definitions. The semantic elements over a given PVS0 program definition, say def, are also given in this theory. This is done by instantiating the curried version provided in theory **pvs0_expr** with the desired evaluation environment and PVS0 expression as the def tuple. This theory also provides the predicate determined? described in Section 4.2.1 and the key function mu specified in Section 6.2.

The elements path, conditions and subterm as described in the beginning of Section (e.g., valid_path, subterm_at and spath_condition) are specified in theory **pvs0_cc**. The elements related to the specification of calling contexts of PVS0 expressions, such as evaluation of conditions (eval_conds), the type of calling contexts for PVS0 (PVS0Expr_CC) and validity for a given PVS0 expression (pvs0_tcc_valid_cc), described in Section 4.2.2, are also specified in this theory.

Several properties related to auxiliary mechanisms used to characterize well-defined, semantically terminating PVS0 programs are provided in theory **pvs0_props**. In particular, the property of decreasement of function mu regarding the arguments that appear in the evaluation of a chain of recursive calls as specified by lemma rec_mu_decreasing, explained in Section 6.2, belongs to this theory.

Theory **pvs0_to_dg** brings several functions for manipulation of program paths and conditions that are used to build the CCs in paths of a given PVS0 program. Then, this theory specifies sound CCGs (sound_ccg_digraphs) over these conjunctions or chains of CCs that are used for a correct specification of the CCG termination criterion, as described in Section 4.2.3.

Theory **measure_termination** specifies the notion of TCC termination, T_τ, described in Section 4.2.2. This theory imports the sub theories measure_termination_defs omitted from the scheme). The criterion itself, given by predicate pvs0_tcc_termination, uses a well-founded measure of type WFM. This theory also contains formalizations related to the fact that semantic termination implies TCC termination, given in Section 6.2, such as lemmas terminates_implies_pvs0_tcc and mu_soundness used in this proof.

The converse proof to complete the formalization of equivalence between semantic and TCC termination, given by lemma pvs0_tcc_implies_terminates, explained in Section 6.2, requires the transitive n-closure gt_n and function Omega given in theory omega (part of the orders NASA PVS Libraries, omitted in the scheme). As described in Section 6.2, this proof also requires the relation to be used over the calling contexts (lt_val) and the upper bound result for Omega (omega_is_eval_ub); both them specified in theory **pvs0__termination** along with the main lemma pvs0_tcc_implies_terminates itself.

The notion of SCP termination for PVS0 programs, given by scp_termination_pvs0 in Section 4.2.3, is specified in theory **scp__iff__pvs0**. This theory also provides the equivalence lemmas between SCP and TCC termination criteria described in Section 6.3; namely, lemmas pvs0_tcc_implies_scp and scp_implies_pvs0_tcc).

The equivalence between CCG and SCP/TCC termination for PVS0 programs is given in two different theories through instantiation of the results relating CCG and SCP. Theory **ccg__to__pvs0** specifies the notion of CCG, described in Section 4.2.3, with the mechanisms of semantic evaluation and evaluation of conditions for PVS0 programs as the predicate ccg_termination_pvs0 and formalizes lemmas (mentioned in Section 6.3) such as ccg_implies_scp_pvs0 and ccg_implies_pvs0_tcc. Sufficiency of the related equivalences, that is lemmas scp_implies_ccg_pvs0 and pvs0_tcc_implies_ccg, are formalized in theory **pvs0__to__ccg**.

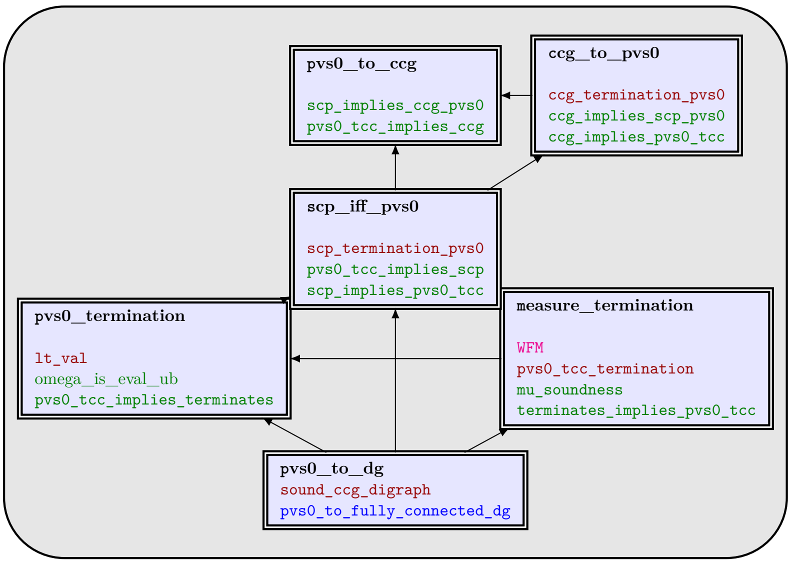


Figure 6.1: Hierarchy scheme for the PVS0 theory files

Table 6.1 summarizes the localization of the main lemmas in the PVS0 library.

Proof	Lemma name	Subtheory	Theory
$Sem_F \Rightarrow TCC$	terminates_implies_pvs0_tcc	**measure_termination**	PVS0
$TCC \Rightarrow Sem_F$	pvs0_tcc_implies_terminates	**pvs0_termination**	PVS0
$Sem_F \Leftrightarrow Sem_V$	eval_expr_terminates	**pvs0_expr**	PVS0
$SCP \Rightarrow TCC$	scp_implies_pvs0_tcc	**scp_iff_pvs0**	PVS0
$TCC \Rightarrow SCP$	pvs0_tcc_implies_tcc	**scp_iff_pvs0**	PVS0
$SCP \Rightarrow CCG$	scp_implies_ccg_pvs0	**pvs0_to_ccg**	PVS0
$TCC \Rightarrow CCG$	pvs0_tcc_implies_ccg	**pvs0_to_ccg**	PVS0
$SCP \Rightarrow CCG$	scp_implies_ccg_termination	**scp_to_ccg**	CCG
$CCG \Rightarrow SCP$	ccg_termination_implies_scp	**ccg**	CCG

Table 6.1: Termination equivalences on PVS0 and where to find them

6.5 NASA PVS Library - CCG Theory Summary

The generic CC type CallingContext, described in Section 4.2.3, is specified in theory **cc_def** (See scheme in Figure 6.2). The theory **scp** specifies infinite_seq_ccs, a predicate that checks if a sequence of CCs is infinite and is used to specify SCP termination through the predicates SCP and scp_termination?, as described in Section 4.2.3.

Theory **ccg_defs** specifies the type FunMeasures for the family of *measures* associated to a digraph. The type CCG for these enriched graphs and also the function make_ccg that given a digraph and some *measures* provides such graph (explained in Section 4.2.3), are also specified in this theory.

The type of the measure combination (measures_combination) associated to the walks of a graph of type CCG and the predicate that checks its decreasement over a walk of this graph (gt_mc?), described in Section 4.2.3, are specified in theory **ccg**. In this theory there are also included the notion of CCG termination given by ccg_termination? and lemma ccg_termination_implies_scp, described in Section 6.3, along with the auxiliary lemmas (extract_infinite_descent, build_infinite_descent and ccg_pigeonhole). The type IncSub used for the functions required in these lemmas is omitted from the scheme, but can be found in file **ramsey_graph** from theory **ints**, also part of the NASA PVS Library. The other direction of the proof, lemma scp_implies_ccg_termination, is specified in theory **scp_to_ccg**. Also in this theory is specified the generic predicate R, that checks the existence of a CC such that its conditions and the evaluation of a given input value are possible in this context. This predicate is used to build the single measure, used as the family of measures, through the function Omega (from theory **omega**), as described in Section 6.3.

The CCG theory summarized in Figure 6.2 also includes the theories **matrix_wdg** and **ccg_to_mwg** with their main elements regarding the MWGs of [Ave14].Although these theories ate not described in this paper, they are an important part of the automation process for termination of PVS0 programs, and its equivalence is also formalized in this library.

These theories, respectively, specify the notion of MWG termination (`mwg_termination?`) and formalize the theorem of equivalence between MWG and CCG termination criteria.

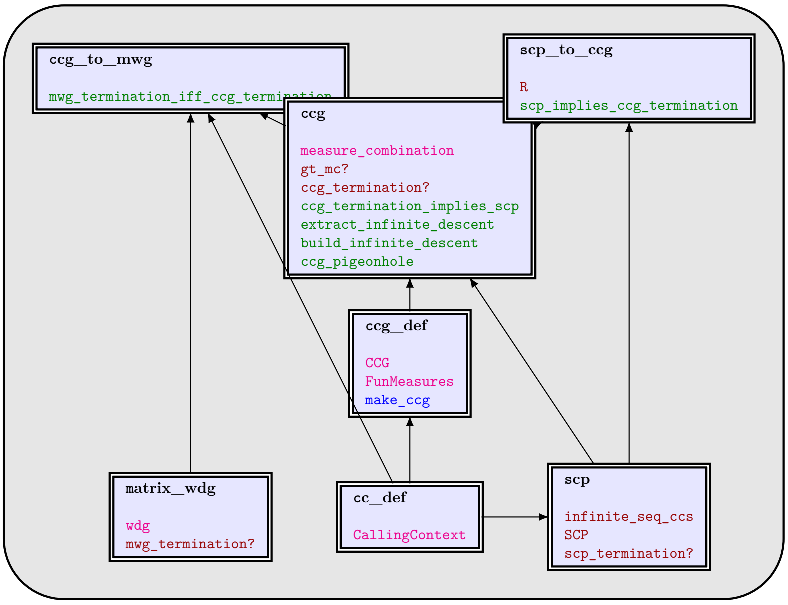


Figure 6.2: Hierarchy scheme for the CCG theory files

Chapter 7

Connecting FP and TRS Termination Criteria

Formalizing termination via DPs for TRSs allows investigating how to apply this termination criterion to provide automation of termination analysis of FPs. This Chapter provides some observations and speculates how it would be possible to build an adequate correspondence that produces DPs in the TRS similar to the CCs in the CCG from the original program and the derivations and derivations considered in DP and CCG criteria.

Challenges to obtaining such correspondence in a way that the TRS obtained from a givem FP is not an over-approximation (which can also be applied), include ensuring that this TRS is confluent since the FPs are deterministic. Also, since we are dealing with non-conditional TRSs and we want to avoid the increase of the signature, the guards of branching instructions of FPs cannot be expressed as rule conditions. The proposal is to translate guards into matching problems for the *lhs* of the rules to decide (one-step) reduction. Such matching decision problems may be reached by narrowing but without guaranteeing confluence. This chapter also briefly presents the translation proposed by Krauss et al [Kra09] that generates orthogonal TRSs, thus ensuring confluence.

7.1 CC versus DP

In FPs, the functions have formal parameters that are instantiated to produce a state when this function is called during an evaluation. TRS rules are applied to terms that match some instantiation of the *lhs* of these rules during a derivation. Therefore, since the specifications are written as FP, the conditions leading to some function call are related to the matching conditions to apply some rule in corresponding FPs and TRS.

When dealing with functional programs whose arithmetic guards are conditions over decidable theories, the matching conditions for the *lhs* of the rule associated with the branches of these conditions can be provided by narrowing with the TRS for such theories.

The solutions obtained by narrowing are applied to the expressions of the CC from which the solution was obtained. The term built by the application of the solution in the first expression of the CC is the *lhs* of the corresponding rewrite rule. Since the solution was obtained by narrowing only the conditions that must hold for the first expression in the CC, the *rhs* of the rule requires to innermost normalize the term obtained with the TRS used for the narrowing after applying the solution to the second expression in the CC. Then, the DPs can be extracted form the resulting TRS.

In particular, the Presburger Arithmetic (PA) expanded with usual algebraic relations (a known decidable theory [Coo72a, Coo72b]), and with the operations of multiplication by constants and subtraction defined over sum and successor allows to express the guards given over arithmetic conditions for many FPs specified in PVS. Since there is no canonical context-free TRS to axiomatize PA [Vor88], Example 3.3.4 provides a PA axiomatization to be used for the narrowing process in the remainder of the document.

To provide correspondence from FPs with guards defined over the PA to TRSs, a signature for this translation includes constructors $\{0, s\}$ and a mapping $n \mapsto s^n(0)$ for natural numbers. The Fibonacci and Ackermann functions, which have comparisons with ground expressions in their guards, give straightforward examples on how to use narrowing to obtain the matching conditions, and thus the rules of a TRS from a FP (Examples 7.1.1 and 7.1.2.

Example 7.1.1 (Fibonacci)**.** *Consider the TRS given in Example 3.3.4 and the following functional specification for the Fibonacci function:*

$$fib(n) := ite(\leq (n,1), 1, +(fib(-(n,1)), fib(-(n,2))))$$

Two conditions of this FP must be considered to reach a correspondent TRS. And to provide the matching condition for the rewriting rules, such conditions must evaluate to a $TRUE$ value:

$$\stackrel{?}{=} (\leq (n,1), \top)$$
$$\stackrel{?}{=} (> (n,1), \top)$$

By narrowing, the first condition is solved as the two possibilities below:

- $\stackrel{?}{=} (\leq (n, s(0)), \top) \rightsquigarrow_{[R_1,\{n/0\}]} \stackrel{?}{=} (\top,\top) \rightsquigarrow_{[R_5]} \top$ *that corresponds to the solution* $\{n/0\}$.
- $\stackrel{?}{=} (\leq (n, s(0)), \top) \rightsquigarrow_{[R_3,\{n/s(x)\}]} \stackrel{?}{=} (\leq (x,0), \top) \rightsquigarrow_{[R_1,\{x/0\}]} \stackrel{?}{=} (\top,\top) \rightsquigarrow_{[R_5]} \top$, *that corresponds to the solution* $\{n/s(0)\}$.

And the second is solved as:

- $\stackrel{?}{=} (> (n, s(0)), \top) \rightsquigarrow_{[R_2,\{n/s(x)\}]} \stackrel{?}{=} (> (x,0), \top) \rightsquigarrow_{[R_4,\{x/s(x')\}]} \stackrel{?}{=} (\top,\top) \rightsquigarrow_{[R_5]} \top$ *that corresponds to the solution* $\{n/s(s(x'))\}$.

Then, even with only two paths for the execution in the FP, the TRS must have three rules obtained through the application of the solutions above in the expressions of the CCs.

$$
\begin{array}{l}
fib(0) \to s(0) \\
fib(s(0)) \to s(0) \\
fib(s(s(y))) \to +(fib(-(s(s(y)), s(0))), fib(-(s(s(y)), s(s(0)))))
\end{array}
$$

Finally, the rhs of the rules are normalized to effectively obtain the rules:

$$
\begin{array}{l}
fib(0) \to s(0) \\
fib(s(0)) \to s(0) \\
fib(s(s(y))) \to +(fib(s(y)), fib(y))
\end{array}
$$

From where the following DPs are obtained:

$$
\begin{array}{l}
\langle fib(s(s(y))), fib(s(y)) \rangle \\
\langle fib(s(s(y))), fib(y) \rangle
\end{array}
$$

Example 7.1.2 (Building matching conditions for Ackermann)**.** *Consider the Ackermann function on Example 2.1.1. The TRS and related DPs will be built from the CCs on Example 3.1.1:*

$$
\begin{array}{l}
\langle ack(m,n),\ \neg(=(m,0)) \wedge =(n,0),\ ack(-(m,1),1) \rangle \\
\langle ack(m,n),\ \neg(=(m,0)) \wedge \neg(=(n,0)),\ ack(-(m,1), ack(m,-(n,1))) \rangle \\
\langle ack(m,n),\ \neg(=(m,0)) \wedge \neg(=(n,0)),\ ack(m,-(n,1)) \rangle
\end{array}
$$

Then, the conditions to be considered are:

$$
\begin{array}{l}
\stackrel{?}{=} (>(m,0) \wedge =(n,0)) \\
\stackrel{?}{=} (>(m,0) \wedge >(n,0))
\end{array}
$$

By narrowing, the first and second conditions are solved, respectively, as $\{m/s(x), n/0\}$ *and* $\{m/s(x), n/s(y)\}$*:*

- $\stackrel{?}{=} (>(m,0) \wedge =(n,0), \top) \rightsquigarrow_{[R_4,\{m/s(x)\}]} \stackrel{?}{=} (\top \wedge =(n,0), \top) \rightsquigarrow_{[R_5,\{n/0\}]} \stackrel{?}{=} (\top \wedge \top, \top) \rightsquigarrow^2_{[R_6],[R_5]} \top$.
- $\stackrel{?}{=} (>(m,0) \wedge >(n,0), \top) \rightsquigarrow_{[R_4,\{m/s(x)\}]} \stackrel{?}{=} (\top \wedge >(n,0), \top) \rightsquigarrow_{[R_4,\{n/s(y)\}]} \stackrel{?}{=} (\top \wedge \top, \top) \rightsquigarrow_{[R_6],[R_5]} \top$.

Thus, one would have as lhs of the first, and second and third DPs, respectively, $ack(s(x), 0)$ *and* $ack(s(x), s(y))$*.*

The rhs of the second DP is obtained by rewriting as below.

$$
ack(-(s(x), s(0)), ack(s(x), -(s(y), s(0)))) \to^2_{R_{14}} ack(-(x,0), ack(s(x), -(y,0))) \to^2_{R_{13}} ack(x, ack(s(x), y))
$$

Proceeding similarly for the first and third CCs, one obtains three associated DP's:

$$
\begin{array}{l}
\langle ack(s(x), 0), ack(x, s(0)) \rangle \\
\langle ack(s(x), s(y)), ack(x, ack(s(x), y)) \rangle \\
\langle ack(s(x), s(y)), ack(s(x), y) \rangle
\end{array}
$$

These are the DPs of Example 2.2.2 in Example 3.3.1.

For (Presburger) arithmetic conditions, when there are no ground expressions in the guards, the construction of DPs can also be achieved. Take for instance the FP for GCD:

Example 7.1.3 (Building conditions for *gcd*)**.** *Consider the case of the following functional specification of gcd over non-simultaneously null naturals (related to Example 3.3.3):*

$$\begin{aligned} gcd(m,n:\ nat\ |m>0 \vee n>0) := ite(&= (m,0),\\ &(n,\_),\\ &ite(\ \ = (n,0),\\ &\qquad (m,\_),\\ &\qquad ite(\leq (n,m),\\ &\qquad gcd(-(m,n),n),\\ &\qquad gcd(n,m)))) \end{aligned}$$

The two CCs are given by:

$$\begin{array}{l} \langle gcd(m,n), > (m,0) \wedge > (n,0) \wedge \leq (n,m), gcd(m-n,n)\rangle \\ \langle gcd(m,n), > (m,0) \wedge > (n,0) \wedge > (n,m), gcd(n,m)\rangle \end{array}$$

The conditions from the functional specification to be considered to the translation are given as:

$$\begin{array}{l} \stackrel{?}{=} (> (m,0) \wedge > (n,0) \wedge \leq (n,m), \top) \\ \stackrel{?}{=} (> (m,0) \wedge > (n,0) \wedge > (n,m), \top) \end{array}$$

Considering the rewriting system for PA given in Example 3.3.4, if one desires to obtain the rules for all the branches of the FP, it is possible to obtain the following matching conditions and rewrite rules for the paths with no recursive calls (see Example 3.3.3):

$$\begin{array}{l} gcd(0,s(y)) \to s(y) \\ gcd(s(x),0) \to s(x) \end{array}$$

The condition on the first CC leads to the third matching condition, that gives the following narrowing solution:

$$\{m/s(u), n/s(v)\} \wedge \stackrel{?}{=} (s(v) \leq s(u), \top)$$

Where the last equation is solved as:

- $\stackrel{?}{=} (s(v) \leq s(u), \top) \rightsquigarrow_{[R_3]} \stackrel{?}{=} (\leq (v,u), \top) \rightsquigarrow_{[R_{10},\{v/x,u/x+y\}]} \stackrel{?}{=} (\top,\top) \rightsquigarrow \top$, *which gives the final solution* $\{m/s(x+y), n/s(x)\}$.

Using this matching condition with proper substitution on the adequate branch expression, the third rule obtained would be:

$$gcd(s(x+y), s(x)) \to gcd(s(x+y) - s(x), s(x))$$

And after normalizing the rhs of the rule one has:

$$gcd(s(x+y), s(y)) \to gcd(x, s(y))$$

From which one can extract the DP $\langle gcd(s(x+y), s(y)), gcd(x, s(y))\rangle$
Finally, the fourth CC condition has as initial solution

$$\{m/s(u), n/s(v)\} \wedge \overset{?}{=} (> (s(v), s(u), \top)$$

From this point:

- $\overset{?}{=} (> (s(v), s(u)), \top) \rightsquigarrow_{[R_2]} \overset{?}{=} (> (v, u), \top) \rightsquigarrow_{R_{11}, \{v/x+s(u)\}} \overset{?}{=} (\top, \top) \rightsquigarrow \top$, *which gives the final solution* $\{m/s(u), n/s(x+s(u))\}$.

From which the fourth rule below is obtained.

$$gcd(s(u), s(x+s(u))) \to gcd(s(x+s(u)), s(u))$$

And the second DP $\langle gcd(s(u), s(x+s(u))), gcd(s(x+s(u)), s(u))\rangle$ *is extracted.*

7.2 Evaluation versus Derivation

The operational semantics of the two computational models, TRS and FP, also must be considered since the analysis of termination by CCG and DP criteria relies, respectively, on the evaluation of values connecting CCs and derivation of terms connecting DPs. For TRSs, a dependency chain consisting of just two DPs is built whenever there exists a substitution such that it allows a reduction (in non-root position) between the *rhs* and the *lhs* of the first and second DP, respectively (after renaming of variables). The *rhs* of the first DP have subterms corresponding to the actual parameters of its corresponding CC, whereas the *lhs* (of the second DP) has subterms with matching conditions obtained from the *lhs* expression of its associated CC. Thus its subterms are replaced by fresh variables representing the function generating a call and its formal parameters on a calling context.

Provided that different occurrences of contiguous DPs in a dependency chain have disjoint variables, for a dependency chain $\langle f_1, g_1\rangle\langle f_2, g_2\rangle$, the required substitution, such that $g_1\sigma \to^* f_2\sigma$, can be split as $\sigma = \sigma_1 \cup \sigma_2$, such that the disjoint domains of σ_1 and σ_2 are subsets of the variables occurring in g_1 and f_2. It is necessary to work with concrete assignments and their evaluation for analyzing the relation between DPs for functional programs to check termination.

Notice that for CCGs the connection between CCs requires to normalize the arguments of the second expression of the first CC. In contrast, the non-root derivation between DPs in a chain does not require such normalization.

Whenever a substitution σ_1 instantiates a DP $\langle f_1, g_1\rangle$, the variables in the subterms of both terms $f(s_1, ..., s_{n_f})$ and $g(t_1, ..., t_{n_g})$ are instantiated. Since in a corresponding CC $\langle f, CConds(\pi, e_f), g\rangle$ the first element of the tuple gives the formal parameters, the corresponding assignment β must be such that $f(s_1, ..., s_{n_f})\sigma_1 = f(x_1, ..., x_{n_f})\beta$ and when the *rhs* (sub)term is instantiated its substitution must correspond to the evaluation of the parameters of f leading to the formals in g. Thus, $g(t_1, ..., t_{n_g})\sigma_1 = g(e_1, ..., e_{n_g})\beta^*$, i.e., there exists a nested call $(f, \beta) \xrightarrow{\pi,k} (g, \beta^*)$.

The substitution σ_2 instantiating both the formal parameters of f_2 and the defined symbol for the tuple symbols g_1 and f_2 must be the same. Furthermore, the assignment σ_2 instantiates the actual parameters of g_2, that corresponds to a call rooted with function symbol g_2 and this will have an assignment built from the evaluation of its actual parameters with the assignment for the function that generated this call, i.e., f_2.

Consider, for instance, the TRS and DPs for the Ackermann function as given in Examples 2.2.2 and 3.3.1, whenever a pair of naturals (m, n) *matches* $(s(x), 0)$, exactly the condition of the first CC holds: $m > 0 \wedge n = 0$. In addition, the actual parameter of the first CC i.e., $(m - 1, 1)$, *matches* $(x, s(0))$. Similarly, this happens for the conditions and actual parameters of the second and third calling contexts.

The idea of relating the CCG and DP termination criteria can be intuitively seen through the similarity between the structure of CCs and DPs and between the behaviour of innermost derivations and eager evaluation. However, to state such relation in a concrete way is not that trivial. First, a formal definition of the desired correspondence must be provided. Then, the different signatures from the FP and the TRS must be provided in a way such that the structure of terms and expressions relate, as well as the structure of function calls and rewrite rules. Also, derivations linking DPs through non-root innermost normalization must, in a sense, "emulate" the semantics of eager evaluation of corresponding functional expressions.

7.3 Using the Dependency Pairs Termination Criterion for `PVS0` Programs

This section presents a preliminary sketch on how to state correspondence for the DP Criterion formalized in Chapter 5 and the criteria for FP in Chapter 6 considering the discussion in Section 7.1. The main discussion concerns the requirements for a translation from FPs to TRSs to allow the application of the DP Criterion to check the termination of FPs. The formalization of such translation and the correspondence between the CCG criteria for FPs and the DP criteria for TRSs are left as future work.

For this aim, a given `PVS0` program `def` must be "conservatively" translated by some mapping `Tr` into a TRS. The DPs of the TRS are then extracted and the DP Criterion is applied to prove its termination. Of course, "conservativeness" must be formaly defined, but intuitively one can consider as the translation providing a TRS such that all guard evaluations (in branching instructions of the program) are represented by matching conditions for the *lhs* of the rules and the *rhs* of rules can represent the evaluation resulting from such condition. Also, properties such as the confluence of the generated TRS must hold to ensure an exact modeling for the determinism of `def`. Then the DPs can be extracted and the DP Criterion used to state termination of `def`.

Take for instance the functional program for GCD and the corresponding TRS obtained in Example 7.1.3.

FP	**TRS**
$gcd(m, n: \text{ nat } \vert m > 0 \vee n > 0) :=$	
$ite(\ = (m, 0),$	$gcd(0, s(y)) \to s(y)$
$\quad (n, \_),$	$gcd(s(x), 0) \to s(x)$
$\quad ite(\ = (n, 0),$	$gcd(s(x+y), s(y)) \to gcd(x, s(y))$
$\qquad (m, \_),$	$gcd(s(u), s(x+s(u))) \to gcd(s(x+s(u)), s(u))$
$\qquad ite(\leq (n, m),$	
$\qquad\quad gcd(-(m, n), n),$	
$\qquad\quad gcd(n, m))))$	

Notice that the obtained TRS is not orthogonal, given that the third and fourth rules are not left-linear. This property is the one that ensures determinism, which is required in a TRS that indeed represents a FP. However, the TRSs generated by the translation does not lose determinism, since the matching conditions for the rules are disjoint:

- For the first rule, the first and second arguments must be m, n such that $m = 0$ and $n > 0$;
- For the second rule, the first and second arguments must be m, n such that $m > 0$ and $n = 0$;
- For the third rule, the first and second arguments must be m, n such that $m \neq 0$, $n \neq 0$ and $m \geq n$;
- For the fourth rule the first and second arguments must be m, n such that $m = 0$ and $n < 0$.

This disjunction of matching conditions ensures that the *gcd* TRS has no rules $R_1 = l_1 \to r_1$ and $R_2 = l_2 \to r_2$ such that, given a position π of l_1 and the most general unifier σ for $l_1|_\pi$ and l_2, two different terms $s = r_1\sigma$ and $s_2 l_1 \sigma[\pi \leftarrow r_2\sigma$ are reached (i.e. R_1 and R_2 are not *overlapping rules* and the ordered pair of terms (s_1, s_2) is not a *critical pair* or *CP* for short). Also, the TRS for PA does have critical pairs, but they are either trivial (such as the CP $(0, 0)$, obtained from rules R_{12} and R_{13}) or joinable (such as the CP

$(\leq (0,x), T)$, obtained from rules R_{10} and R_7), i.e., both systems are locally confluent. Since PA is decidable, it is also terminating, i.e., the system is convergent.

However, the termination of *gcd* is exactly the goal property to be stated. Thus, for *gcd* the Newman's Lemma can not be used yet. Other than that, the union of these two sets of rules result in a TRS that is not even locally confluent.

Example 7.3.1 (Non-Convergent Critical Pairs of GCD)**.** *Consider R as the union of the PA TRS given in Example 3.3.4 and the gcd TRS obtained in Example 7.1.3. The rules from PA keep their labels and the ones from gcd are labeled in R:*

$$\begin{array}{ll} R_{15} & gcd(0, s(y)) \to s(y) \\ R_{16} & gcd(s(x), 0) \to s(x) \\ R_{17} & gcd(s(x+y), s(y)) \to gcd(x, s(y)) \\ R_{18} & gcd(s(y), s(x+s(y))) \to gcd(s(x+s(y)), s(y)) \end{array}$$

From R the following non-convergent critical pairs are obtained:

From rules		*Critical Pairs*
R_{17} *and* R_7	CP_1	$\langle gcd(s(y), s(y)), gcd(0, s(y))\rangle$
R_{17} *and* R_8	CP_2	$\langle gcd(s(s((x+y))), s(y)), gcd(s(x), s(y))\rangle$
R_{18} *and* R_7	CP_3	$\langle gcd(s(y), s(s(y))), gcd(s(0+s(y)), s(y))\rangle$
R_{18} *and* R_8	CP_4	$\langle gcd(s(y), s(s(x+s(y))))), gcd(s(s(x)+s(y))), s(y))\rangle$

One possibility to solve this problem and obtain a locally confluent system would be to use Knuth-Bendix completion procedure on R. This however can provide an infinite TRS, as is the case for *gcd*, that after solving some CP adds these three rules:

$$\begin{array}{ll} R_{19} & gcd(s(y), s(y)) \to s(y) \\ R_{20} & gcd(s(s(x+y)), s(y)) \to gcd(s(x), s(y)) \\ R_{21} & gcd(s(y), s(s(y))) \to gcd(s(s(y)), s(y)) \end{array}$$

Notice that R_{21} is the commutativity property of *gcd* regarding arguments that are a positive term and its successor, that adds the non-covergent critical pair regarding the commutativity of *gcd* over a positive term and the second successor of this term added to another term, i.e.

$$\langle gcd(s(y), s(s(+(x, s(y))))), gcd(s(+(s(x), s(y))), s(y))\rangle$$

By solving it, the rule

$$R_{22} \quad gcd(s(y), s(s(+(x, s(y))))) \to gcd(s(s(+(x, s(y)))), s(y))$$

is obtained. Notice however that this rule only concerns adding successor to the sum in the terms and not to the positive term. This will, along with the rule R_{21}, lead to

CPs always adding another successor to the sum and providing commutativity with this successor. This will always lead to similar critical pairs and consequently, to an infinite system:

$$\begin{array}{c} gcd(s(y), s(s(s(+(x, s(y)))))) \to gcd(s(s(s(+(x, s(y))))), s(y)) \\ gcd(s(y), s(s(s(s(+(x, s(y))))))) \to gcd(s(s(s(s(+(x, s(y)))))), s(y)) \\ gcd(s(y), s(s(s(s(s(+(x, s(y)))))))) \to gcd(s(s(s(s(s(+(x, s(y))))))), s(y)) \\ \vdots \end{array}$$

This happens because the rules are all oriented, so the completion is not able to deal with some results over arithmetic symbols that are obvious, but require inductive proofs that are not included in the process. However, if the TRS for PA is given separately as a set E of equations, this set can be used to join the previously non-convergent critical pairs (as given in this example) by completion modulo E (e.g., Chapter 11 in [BN98]), since its decidabity makes it suitable its effective use in the process of rewriting modulo theories [AR93]. Thus, consider $\approx_{PA}$ as the equivalence relation over a set of equations from PA. If the equation E_i leading to the equivalence is worth mention in some step, the notation used is $\approx_{PA}^{E_i}$. Also, let $\approx_{R \cup PA}$ denote the relation rewriting modulo PA.

Then, for the running example of gcd, every rule representing PA expressions R_i, $1 \leq i \leq 14$ are replaced by a corresponding equational rule E_i, $1 \leq i \leq 14$. And by rewriting modulo PA one has:

- For CP_1: $gcd(s(y), s(y)) \approx_{R \cup PA} gcd(0, s(y))$ is provided by

$$gcd(s(x), s(x)) \approx_{PA}^{E_7} gcd(s(0 + x), s(x)) \to_R^{R_{17}} gcd(0, s(x))$$

- For CP_2: $gcd(s(s(x + y)), s(y)) \approx_{R \cup PA} gcd(s(x), s(y))$ is provided by

$$gcd(s(s(x + y)), s(y)) \approx_{PA}^{E_8} gcd(s(s(x) + y), s(y)) \to_R^{R_{17}} gcd(s(x), s(y))$$

- For CP_3: $gcd(s(y), s(s(y))) \approx_{R \cup PA} gcd(s(0 + s(y)), s(y))$ is provided by

$$gcd(s(y), s(s(y))) \approx_{PA}^{R_7} gcd(s(y), s(0 + s(y))) \to_R^{R_{18}} gcd(s(0 + s(y)), s(y))$$

- And for CP_4: $gcd(s(y), s(s(x + s(y)))) \approx_{R \cup PA} gcd(s(s(x) + s(y)), s(y))$ is provided by

$$gcd(s(y), s(s(x+s(y)))) \approx_{PA}^{E_8} gcd(s(y), s(s(x)+s(y))) \to_R^{R_{18}} gcd(s(s(x)+s(y)), s(y))$$

Notice that this arithmetic is the one proposed to deal with the creation of matching conditions to obtain the rules of the TRS used to the analysis by DP in this examples.

Thus, the theory of PA can be provided as a separate set of equations PA, after obtaining the rules but before checking the local confluence of the TRS. This allows the use of rewriting modulo theory as in [Vir95], avoiding the necessity of performing completion modulo the equational theory PA on the critical pairs obtained. Furthermore, this whole rewriting system will have only the four rewriting rules that models the four possible branches of the functional program such that each matching condition is representing an equivalent result of evaluating the guard of the program.

This discussion leads to a possible way to conservatively translate the FP's into TRS's without adding new symbols or rules (except the rules to deal with disjunctive guards) to the original signature, but keeping the essential property of functional programs: their determinism. From here, it is necessary to show that the translation used in the process indeed produces a TRS syntacticly and semanticlly equivalent to the original FP. This will allow to ensure that the DP Criterion can state termination of the given FP.

Once such a `Tr` is obtained and proven conservative, the correspondence between DP termination of `def` and CCG termination of the `PVS0` program `Tr(def)` must be proven to allow formalizing the correct application of the DP Criterion for `PVS0` programs.

$$
\begin{array}{l}
\texttt{dp\_termination\_implies\_ccg} : \mathrm{LEMMA} \\
\quad \forall(\texttt{Tr} | \text{``}\mathit{conservative}\text{''}) : \\
\qquad \forall(\texttt{def}) : \\
\qquad\quad \mathcal{T}_{DP}(\texttt{Tr(def)}) \Leftrightarrow \mathcal{T}_{\varrho}(\texttt{def})
\end{array}
$$

To achieve the main goal of this proposal, i.e., to use the DP Criterion to prove termination of FPs, the sufficiency of this lemma would be enough. By contraposition, if `def` is not CCG terminating, there exists an infinite circuit of CCs in the CCG of `def` representing an infinite evaluation of some input value $\texttt{v}_1$. By the conservativeness of the mapping `Tr`, one has the translation of $\texttt{v}_1$ into a ground term t, a corresponding set of DPs corresponding to the set of CCs and the "emulation" of the evaluation of values in `def` as derivations of corresponding terms in $\texttt{Tr}(def)$. From this, the pairs of connected CCs in the circuit provide the chained DPs, by using the terms obtained from the evaluated values to state the links. The DP chain created will be infinite, completing the proof.

The necessity however, requires some more observation. The operational semantics of TRSs is more expressive than the one of FPs. Even allowing only innermost derivations for the TRS, it is possible to derive terms that do not represent exactly values, as long as it has some subterm that matches with the *lhs* of some rule. Notice that values can only be translated into ground terms. Thus, the TRS analysed must be restricted to derivations over ground terms only.

7.4 Related work

The subjects of speculation in Sections 7.1, 7.2 and 7.3 are explored in the works of [KST$^+$11] and [GSSK$^+$06] in a different way than the one in this document. However, their approaches are interesting and could also be explored in the context of the `PVSO` language to expand the possibilities of reason over this functional language termination.

Giesl et al. [GSSK$^+$06] propose an approach to state termination of Haskell programs using DPs. The peculiarities of Haskell such as lazy evaluation and higher-order functions are considered. These features are not present in the `PVSO` language, but could be used to eventually model another simplified language to reason over less restrictive PVS functions. Their approach does not require direct translation from the FPs in Haskell to TRSs. Instead, they initially develop an heuristic to build termination graph for terms t which whenever instantiated with ground substitution σ, the termination of t implies termination of $t\sigma$. The termination graph is built by expanding the term t with five expansion rules, defined to allow the evaluation of terms. The application of such rules gives rise to descending nodes of t, from where the process follows repeatedly until it leads leafs where the rules can not be applied anymore. In this graph it is possible to obtain edges between a new node and some other node already in the graph, which represents cycles which could lead to non termination. Once such graph is provided, it is transformed into a DP problem, from which finiteness (if reached) implies termination of all terms in the termination graph.

The work from Krauss et al. is closer to the one pursued by the discussion in this chapter. They provide a translation from FPs to TRSs and then prove that this translation result in a system that indeed simulate the original FP and thus, that providing termination proofs for one computational model through such translation is enough to state termination of the original one.

7.4.1 A translation to orthogonal TRSs

The translation presented by Krauss et al. in [KST$^+$11] is done for Isabelle specifications. Their work also presents results to ensure the adequacy of such translation by stating that the TRS obtained captures the evaluation of the original FP in their Simulation Theorem. The translation proposed enlarges the signature of the TRS with new function symbols to deal with the conditions in the guards of the FP. The new function symbols and rules are obtained similarly to the transformation from conditional to unconditional TRSs. This process provides rules which are left-linear and do not overlap. This ensures the creation of an orthogonal TRS, that has confluence as one of its properties.

Their work restricts the signatures of first-order rewriting systems and functional programs to have the same constructors, with no mutual nor nested recursion. Also, the branching instruction used in the programming language is `case-of`, that allows several branches depending on the structure type used in the guards, what can avoid some of nested conditional instructions. Similar to the translation sketched in Section 7.3, they also restricted the guards to equational conditions. In the guards, the patterns used are constructor terms that must be linear and non-overlapping. To illustrate this translation, consider the program below using the syntax of [KST^{+}11].

Example 7.4.1 (GCD in the syntax used by [KST^{+}11])**.**

```
gcd(m,n) = (case m of
              0 => n |
              s(x) => (case n of
                         0 => s(x) |
                         s(y) => (case lte(s(y),s(x)) of
                                    true => gcd(minus(s(x),s(y)),s(y)) |
                                    false => gcd(s(y),s(x)) )))
```

With auxiliary functions:

```
lte(m,n) = (case m of
              0 => true |
              s(x) => (case n of
                         0 => false |
                         s(y) => lte(x,y)))
```

and

```
minus(m,n) = (case m of
               0 => 0 |
               s(x) => (case n of
                           0 => s(x) |
                         s(y) => minus(x,y)))
```

In their approach, the method used to ensure that termination of a TRS implies the termination of the FP is done trough this conservative translation of a FP to a corresponding TRS and then proceed by ensuring that strong normalization for each rule representing the calls of the program states its termination. This is done by their simulation lemma to state the simulation of the computation of the program through the corresponding TRS by induction on the expressions from the program. Thus, if the TRSs allows an infinite derivation, the evaluation that it simulates must also be infinite.

The program calls are extracted automatically from recursive calls using an operation $\texttt{CALLS}_{\texttt{f}}$, which is very similar to the one specified for obtaining the CCs of `PVS0` functions in Definition 3.1.1. In their approach, the calls are given by the whole expressions of the conditions, whereas in this work only the path for the condition is used as reference to trace the function calls.

Then, each expression in the calls is encoded by a meta-level operation ENC, that maps variables and functions into term variables and application, respectively. The mapping of case expressions are replaced by a new function symbol $\texttt{case}^{\texttt{f}_1}$:

$$
\begin{array}{rcl}
\texttt{ENC}(x) & \equiv & \texttt{x} \\
\texttt{ENC}(f e_1 \ldots e_n) & \equiv & \texttt{f}(\texttt{ENC}(e_1), \ldots, \texttt{ENC}(en)) \\
\texttt{ENC}(\texttt{case}^{\texttt{f}_1}\ e \text{ of } p_1 \ => \ e_1 \mid \ \ldots | pk => e_k) & & \texttt{case}^{\texttt{f}_1}(\texttt{ENC}(e), \texttt{ENC}(y_1), \ldots, \texttt{ENC}(y_m))
\end{array}
$$

After mapping the expressions into terms, the rules are obtained from operation RULES for function symbols (with defining equations $l_1 = r_1, \ldots l_k = r_k$) and case expressions as below:

$$
\begin{array}{rcl}
\texttt{RULES (f)} & \equiv & \{\texttt{ENC}(l_1) \to \texttt{ENC}(r_1), \ldots, \texttt{ENC}(l_k) \to \texttt{ENC}(r_k)\} \\
\texttt{RULES}(\texttt{case}^{\texttt{f}}) & \equiv & \{\texttt{case}^{\texttt{f}_1}(\texttt{ENC}(p1), \texttt{ENC}(y_1), \ldots, \texttt{ENC}(y_m)) \to \texttt{ENC}(e_1), \\
 & & \ldots \\
 & & \texttt{case}^{f}_{k}(\texttt{ENC}(p_k)), \texttt{ENC}(y_1), \ldots, \texttt{ENC}(y_m)) \to \texttt{ENC}(e_k)\}
\end{array}
$$

Example 7.4.2 (Rules for the program in Example 7.4.1)**.** *For the auxiliary functions the rules are:*

$$
\begin{array}{l}
\texttt{lte(m,n)} \to \texttt{case}^{\texttt{lte}}_0\texttt{(m,n)} \\
\texttt{case}^{\texttt{lte}}_0\texttt{(0,n)} \to \texttt{true} \\
\texttt{case}^{\texttt{lte}}_0\texttt{(s(x),n)} \to \texttt{case}^{\texttt{lte}}_1\texttt{(s(x), n)} \\
\texttt{case}^{\texttt{lte}}_1\texttt{(s(x),0)} \to \texttt{false} \\
\texttt{case}^{\texttt{lte}}_1\texttt{(s(x),s(y))} \to \texttt{lte(x,y)} \\
\\
\texttt{minus(m,n)} \to \texttt{case}^{\texttt{minus}}_0\texttt{(m,n)} \\
\texttt{case}^{\texttt{minus}}_0\texttt{(0,n)} \to \texttt{0} \\
\texttt{case}^{\texttt{minus}}_0\texttt{(s(x),n)} \to \texttt{case}^{\texttt{minus}}_1\texttt{(s(x),n)} \\
\texttt{case}^{\texttt{minus}}_1\texttt{(s(x),0)} \to \texttt{s(x)} \\
\texttt{case}^{\texttt{minus}}_1\texttt{(s(x),s(y))} \to \texttt{minus(x,y)}
\end{array}
$$

And for the main gcd *function:*

$$
\begin{aligned}
&\texttt{gcd(m,n)} \rightarrow \texttt{case}_0^{\texttt{gcd}}\texttt{(m,n)}\\
&\texttt{case}_0^{\texttt{gcd}}\texttt{(0,n)} \rightarrow \texttt{n}\\
&\texttt{case}_0^{\texttt{gcd}}\texttt{(s(x),n)} \rightarrow \texttt{case}_1^{\texttt{gcd}}\texttt{(s(x),n)}\\
&\texttt{case}_1^{\texttt{gcd}}\texttt{(s(x),0)} \rightarrow \texttt{s(x)}\\
&\texttt{case}_1^{\texttt{gcd}}\texttt{(s(x),s(y))} \rightarrow \texttt{case}_2^{\texttt{gcd}}\texttt{(lte(s(y),s(x)),s(x),s(y))}\\
&\texttt{case}_2^{\texttt{gcd}}\texttt{(true,s(x),s(y))} \rightarrow \texttt{gcd(minus(s(x),s(y)), s(y))}\\
&\texttt{case}_2^{\texttt{gcd}}\texttt{(false,s(x),s(y))} \rightarrow \texttt{gcd(s(y), s(x))}
\end{aligned}
$$

From here, it is shown that the TRS termination goals from the FP are equivalent to find a well-founded measure for the rules in the TRS. Also, it is ensured that the TRS models the FP behaviour through the simulation lemma, allowing to provide certificates that are verifiable by automatic checking tools, giving the final certificate to the initial FP. This is also a path that the translation proposed in this Chapter has to follow to allow the effective use of the DP innermost termination criterion to automate termination of PVS first order functions modeled by the `PVS0` language.

Chapter 8

Conclusion and Future Work

This document presented a formalization in the PVS proof assistant of the Dependency Pairs Termination Criterion for innermost term rewriting. The primary motivation of such formalization was to enable the formulation of an additional criterion to automate verification of termination of `PVS0` functional programs. The proofs follow a constructive pen-and-paper design close to the proofs proposed by Arts and Giesl's in their seminal papers [AG97] and [AG00]. Analytical proofs avoid reducing subterms rooted by defined function symbols by extending the language with tuple symbols. In contrast, our formalization specifies specialized reduction relations to avoid reduction on root positions: non-root (innermost/Q-restricted) reductions.

The formalization of some lemmas relies on set properties between these specialized relations and the ordinary reduction relation. For instance, the compatibility with contexts of $\xrightarrow{>\lambda}{}^{*}_{in}$ is proved through monotony of $\rightarrow$, since $\rightarrow$ is compatible with contexts and $\xrightarrow{>\lambda}_{in} \subseteq \rightarrow_{in} \subseteq \rightarrow$, as discussed in Section 5.1.

We slightly adapted the definition of DPS to include the information of which rule generated it and proved that the two definitions are equivalent. Such adaptation made it easier to follow a constructive approach to build the infinite derivations from infinite innermost DP chains explicitly in the formalization of necessity (Noetherianity implies innermost termination by DPS). The construction of an infinite derivation from an infinite DP chain is easily given by recursion, accumulating the contexts from the terms in the chain, allowing a simple inductive proof. On the other hand, for sufficiency (termination by DPs implies Noetherianity), since DPs are pairs of *lhs*'s of rules and subterms of their *rhs*'s, the recursive construction of chains of DPs from derivations is not that easy since it requires the removal of contexts instead.

In a pen-and-paper proof of the sufficiency of the innermost DP Criterion, some crucial steps are given as mere simple observations but turned out not so simple to formalize. For instance, the formalization of non-root innermost normalization of minimal non-innermost terminating terms presented in Section 5.2.2 was quite extensive. The use of subtype predicates helped to manage such difficulties. Such predicates were used in recursive functions to directly state relevant properties for every output, allowing to provide inductive proofs more efficiently, with no need to add lemmas over these properties along with the formalization. For instance, such typing information ease results over the non-emptiness of sets of terms or with specific properties used in the sufficiency proof described in Section 5.2.

The formalization of the innermost DP Criterion added to the NASA PVS Library substantially enriched TRS, the term rewriting systems library. The formalization of the innermost DP Criterion added its specification and around 55 lemmas directly related to such formalization. Other than that, the formalization for the innermost case could not be used directly to formalize termination by DP for the ordinary and Q-restricted reduction relations. These formalizations were done separately and also added around 55 lemmas each. To allow the specification of these three criteria, specialized reduction strategies were also specified. These strategies include innermost reduction, reduction restricted to descendants of a term, and non-root reduction for the innermost and ordinary relations (see Subsection 5.4). Results over these strategies added about 42 lemmas to the theory and were formalized in a generic manner allowing further independent applications. Other auxiliary definitions were included in the previously existing theories `reduction`, `rewrite_rule`, `ars`, `subterm`, `positions`, `noetherian` and `relations_closure`. Such definitions include, for instance, the closure monotony and the relation between relations closures and sequences (that allows the construction of one through the other). Results over the auxiliary definitions added eight lemmas to the theory.

The document also compiles formalizations on the equivalence of several termination criteria for the `PVS0` functional language, a language designed to reason about the termination of first-order functions in the PVS specification language. The formalization of the equivalence includes six termination criteria, namely: the two semantic notions of termination based on the operational semantics of evaluation of PVS0, i.e., the existence of output for each possible input and the existence of a bound on the number of nested recursive calls required to provide a valid result; the termination implemented in PVS, the TCC termination, which requires a measure on the function parameters that should decrease after each recursive call; and finally, the formalizations of termination by the SCP [LJBA01], by Calling Context Graphs [MV06], and by Matrix Weighted Graphs MWG [Ave14], which are abstractions of CCGs by labelling the graph edges with square matrices whose entries express relationships between different measures applied to the formal and current parameters of a call. MWG also provides an operation over such matrices to check decreasingness of each circuit in the graph [Ave14].

Finally, the document discusses how the Dependency Pairs Termination Criterion for term rewriting systems might be applied to guarantee the termination for `PVS0` functional programs. The proposal consists of translating PVS0 programs into term rewriting systems such that the evaluation of programs "corresponds" to the derivation of rewriting systems. However, further investigation is required to obtain such a translation. The translation proposed is restricted to functional programs with arithmetic guards. It uses narrowing with rules for a decidable arithmetic theory to provide matching conditions that capture the conditions on guards of functional programs. The TRSs obtained do not satisfy confluence necessarily, losing in this manner the determinism of the functional programs. Ensuring orthogonality (as in Krauss et al. translation approach [KST$^+$11]) of the translation or seeing the rewriting system modulo arithmetic are alternatives to guarantee confluence. After establishing such a translation, and verifying termination of the term rewriting systems, applying the Dependency Pairs Termination Criterion would assure termination of the functional programs.

Several interesting subjects are worth exploring to extend this work and provide a robust environment to reason over the termination of FPs through termination of TRSs, such as higher-order rewriting (e.g. [Kra09, GTSK05a, KvR11, SWS01]) and the relation of termination with ordinal segments (e. g. [KSW60, Sch14]) for instance. However, to continue the primary goal of obtaining automation of termination analysis for PVS functions, there is still a long way to roam. The discussion on Chapter 7 and the scenario provided by relevant current work in termination lead to several exciting formalizations to be done, among them those mentioned below.

- The correspondence between FPs and TRSs through a complete and correct translation following the steps discussed in Section 7.3 by narrowing and rewriting modulo theory, to allow the use of TRS results to FPs.
- Alternatively, formalization of the translation from PVS0 programs to orthogonal TRSs and of the adequacy of such translation according to Krauss et al. approach [KST$^+$11] and/or from CCGs obtained from PVS0 specification to DP problems according to Giesl et al. approach[GSKT06].
- Extension of the formalization to include refinements for the DP Criterion allowing to improve the automation process of checking termination of TRSs and thus of PVS0 FPs.
- Extension of the formalization generalizing the DP Criterion as the DP Framework, which allows to combine TRSs termination techniques to provide flexibility for automation when checking termination [GTSK05b].
- Additionally, it is worth investigate how the work presented in [Blanqui2021] can be used to expand the possibilities of specifications in PVS and in PVS0, since

it is relevant in the researches regarding formalizations in general. They provide an encoding for PVS, called PVS-Cert, to reason about the features of predicate subtyping. Such encoding aims to provide an automatic translation from PVS to DEDUKTI, a checker that also provides encoding for several proof assistant specification languages. Although PVS-Cert was developed for a different goal than `PVS0`, there is room to pursue a combination of both encodings to allow future sharing and use of formalizations provided in one proof assistant to another oneanother one, easing the comparison between proof strategies applied and also allow the reuse of formalized results.

Bibliography

[AAAR20] Ariane Alves Almeida and Mauricio Ayala-Rincon. Formalizing the Dependency Pair Criterion for Innermost Termination. *Science of Computer Programming*, 195(102474), 2020. 6, 13, 33, 51, 53, 56, 61

[AG97] Thomas Arts and Jürgen Giesl. Automatically proving termination where simplification orderings fail. In *Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 261–272. Springer, 1997. 1, 6, 28, 30, 32, 102

[AG98] Thomas Arts and Jürgen Giesl. Modularity of termination using dependency pairs. In *Rewriting Techniques and Applications*, volume 1379 of *Lecture Notes in Computer Science*, pages 226–240. Springer, 1998. 1

[AG00] Thomas Arts and Jürgen Giesl. Termination of term rewriting using Dependency Pairs. *Theoretical Computer Science*, 236:133–178, 2000. 1, 29, 32, 55, 102

[AR93] Mauricio Ayala-Rincón. *Expressiveness of conditional equational systems with built-in predicates.* PhD thesis, Universität Kaiserslauten, 1993. 96

[Art96] Thomas Arts. Termination by absence of infinite chains of dependency pairs. In *Trees in Algebra and Programming CAAP*, volume 1059 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 1996. 1

[Ave14] Andréia Borges Avelar. *Formalização da Automação da Terminação Através de Grafos com Matrizes de Medida.* PhD thesis, Department of Mathematics, Universidade de Brasília, 2014. In Portuguese. 86, 103

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions.* Springer-Verlag Berlin Heidelberg, 2004. 2

[BK11] Frédéric Blanqui and Adam Koprowski. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Math. Struct. in Comp. Science*, 21:827–859, 2011. 74

[BKB+03] Marc Bezem, Jan Willem Klop, Erik Barendsen, Roel C. de Vrijer, and Terese. *Term Rewriting Systems.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003. 13

[BM79] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, 1979. 20

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. 13, 14, 17, 96

[CCF+07] Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Certification of automated termination proofs. In *International Symposium on Frontiers of Combining Systems FroCoS*, pages 148–162. Springer, 2007. 74

[Coo72a] David C. Cooper. Programs for mechanical program verification. *Machine intelligence*, 6:43–59, 1972. 89

[Coo72b] David C. Cooper. Theorem proving in arithmetic without multiplication. *Machine intelligence*, 7:91–99, 1972. 89

[Der79] Nachum Dershowitz. A note on simplification orderings. *Information Processing Letters*, 9(5):212–215, 1979. 1, 19

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982. 1, 19

[Der87] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, pages 69 – 115, 1987. 1, 19

[DM79a] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. In Hermann A. Maurer, editor, *Automata, Languages and Programming*, pages 188–202, Berlin, Heidelberg, 1979. Springer Berlin Heidelberg. 1, 19

[DM79b] Nachum Dershowitz and Zohar Manna. Proving Termination with Multiset Orderings. *Commun. ACM*, 22(8):465–476, 1979. 74

[DS04] Yuxin Deng and Davide Sangiorgi. Ensuring termination by typability. In Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics*, pages 619–632. Springer US, 2004. 19

[DX07] Kevin Donnelly and Hongwei Xi. A formalization of strong normalization for simply-typed lambda-calculus and system f. *Electronic Notes in Theoretical Computer Science*, 174(5):109 – 125, 2007. Proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2006). 19

[Flo67] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967. 20

[GAB+17] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Analyzing program termination and complexity automatically with AProVE. *J. Autom. Reasoning*, 58(1):3–31, 2017. 74

[GAR08] André Luiz Galdino and Mauricio Ayala-Rincón. A Formalization of Newman's and Yokouchi's Lemmas in a Higher-Order Language. *Journal of Formalized Reasoning*, 1(1), 2008. 33, 67

[GAR10] André Luiz Galdino and Mauricio Ayala-Rincón. A Formalization of the Knuth–Bendix(–Huet) Critical Pair Theorem. *Journal of Automated Reasoning*, 45(3):301–325, 2010. 33, 67

[GBE+14] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving termination of programs automatically with AProVE. In *Automated Reasoning - 7th International Joint Conference*, volume 8562 of *Lecture Notes in Computer Science*, pages 184–191, 2014. 74

[Gra96] Bernhard Gramlich. On proving termination by innermost termination. In Harald Ganzinger, editor, *Rewriting Techniques and Applications: 7th International Conference*, volume 1103 of *Lecture Notes in Computer Science*, pages 93–107. Springer Berlin Heidelberg, 1996. 17

[GSKT06] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In *International Joint Conference on Automated Reasoning*, pages 281–286. Springer, 2006. 74, 104

[GSSK+06] Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp, , and René Thiemann. Automated termination analysis for haskell: From term rewriting to programming languages. In *Term Rewriting and Applications*, pages 297–312, 08 2006. 98

[GTSK05a] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. Proving and Disproving Termination of Higher-order Functions. In *Proc. 5th International Conference on Frontiers of Combining Systems FROCOS*, volume 3717 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2005. 104

[GTSK05b] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The Dependency Pair Framework: Combining techniques for automated termination proofs. In *11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning LPAR 2004*, volume 3452 of *Lecture Notes in Computer Science*, pages 301–331. Springer, 2005. 31, 104

[Hin92] James Roger Hindley. Types with intersection: An introduction. *Formal Aspects of Computing*, 4(5):470–486, 1992. 19

[HL78] Gérard Huet and Dallas Lankford. On the uniform halting problem for term rewriting systems. Technical report, INRIA, 1978. 1

[HM03] Nao Hirokawa and Aart Middeldorp. Automating the Dependency Pair Method. In Franz Baader, editor, *Proceedings of 19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2003. 32

[HM04] Nao Hirokawa and Aart Middeldorp. Dependency pairs revisited. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications*, pages 249–268. Springer Berlin Heidelberg, 2004. 32

[Hof79] Douglas R. Hofstadter. *Gödel, Escher, Bach: an eternal golden braid; 1st ed.* Penguin books. Basic Books, New York, NY, 1979. 3

[KK96] Claude Kirchner and Hélene Kirchner. Rewriting solving proving. Technical report, LORIA, INRIA and CNRS, 1996. 15

[KMM00] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000. 2

[Kra09] Alexander Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic.* PhD thesis, Institut für Informatik der Technischen Universität München, 2009. 88, 104

[KST+11] Alexander Krauss, Christian Sternagel, René Thiemann, Carsten Fuhs, and Jürgen Giesl. Termination of Isabelle Functions via Termination of Rewriting. In *Proceedings Interactive Theorem Proving - Second International Conference, ITP 2011*, volume 6898 of *LNCS*, pages 152–167. Springer, 2011. 6, 98, 99, 104

[KSW60] Georg Kreisel, Joseph Shoenfield, and Hao Wang. Number theoretic concepts and recursive well-orderings. *Archiv für mathematische Logik und Grundlagenforschung*, 5(1):42–64, 1960. 104

[KSZM09] Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean termination tool 2. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304. Springer, 2009. 74

[KvR11] Cynthia Kop and Femke van Raamsdonk. Higher order dependency pairs for algebraic functional systems. In *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011*, volume 10 of *LIPIcs*, pages 203–218. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011. 104

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The Size-change Principle for Program Termination. In *ACM SIGPLAN Notices*, pages 81–92. ACM, 2001. 2, 6, 25, 103

[Mal91] Colin L. Mallows. Conway's challenge sequence. *The American Mathematical Monthly*, 98(1):5–20, 1991. 3

[MARM+21] Cesar Muñoz, Mauricio Ayala-Rincón, Mariano Moscato, Aaron Dutle, Anthony Narkawicz, Ariane Alves Almeida, Andréia Borges Avelar da Silva, and Thiago Mendonça Ferreira Ramos. Formal Verification of Termination Criteria for First-Order Recursive Functions. In *LIPIcs proceedings 12th Int. Conference on Interactive Theorem Proving*, 2021. Accepted. 5, 6, 76

[MM69] Zohar Manna and John McCarthy. Properties of programs and partial function logic. Technical report, Stanford Univ. Calif. Dept. of Computer Science, 1969. 3

[MP70] Zohar Manna and Amir Pnueli. Formalization of properties of functional programs. *J. ACM*, 17(3):555–569, 1970. 3

[MV06] Panagiotis Manolios and Daron Vroon. Termination Analysis with Calling Context Graphs. In *Proceedings of the 18th International Conference on Computer Aided Verification CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 401–414. Springer, 2006. 2, 6, 25, 103

[Ned94] Robert Peter Nederpelt. Strong normalization in a typed lambda calculus with lambda structured types. In Robert Pieter Nederpelt, J. Herman Geuvers, and Roel C. de Vrijer, editors, *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*, pages 389 – 468. Elsevier, 1994. 19

[ORS92] Sam Owre, John Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992. 2

[OSRSC99] Sam Owre, Natarajan Shankar, John M. Rushby, and David WJ Stringer-Calvert. PVS language reference. *Computer Science Laboratory, SRI International, Menlo Park, CA*, 1:21, 1999. 20

[ROGAR17] Ana Cristina Rocha-Oliveira, André Luiz Galdino, and Mauricio Ayala-Rincón. Confluence of Orthogonal Term Rewriting Systems in the Prototype Verification System. *Journal of Automated Reasoning*, 58(2):231–251, 2017. 33, 67

[San06] Davide Sangiorgi. Termination of processes. *Mathematical Structures in Computer Science*, 16(1):1–39, 2006. 19

[Sch14] Sylvain Schmitz. Complexity bounds for ordinal-based termination. In Joël Ouaknine, Igor Potapov, and James Worrell, editors, *Reachability Problems*, pages 1–19. Springer International Publishing, 2014. 1, 104

[SGB+17] Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann. Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic. *J. Autom. Reasoning*, 58(1):33–65, 2017. 74

[ST10] Christian Sternagel and René Thiemann. Signature extensions preserve termination - an alternative proof via dependency pairs. In *19th Computer Science Logic CSL*, volume 6247 of *Lecture Notes in Computer Science*, pages 514—528. Springer, 2010. 6, 31, 74

[Ste10] Christian Sternagel. *Automatic Certification of Termination Proofs*. PhD thesis, Universität Innsbruck, 2010. 37

[SWS01] Masahiko Sakai, Yoshitsugu Watanabe, and Toshiki Sakabe. An extension of the dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, 84(8):1025–1032, 2001. 104

[TAN12] René Thiemann, Guillaume Allais, and Julian Nagele. On the Formalization of Termination Techniques based on Multiset Orderings. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications*, volume 15 of *Leibniz International Proceedings in Informatics*, pages 339–354. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012. 74

[TG03] René Thiemann and Jürgen Giesl. Size-change Termination for Term Rewriting. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2003. 29

[TS09] René Thiemann and Christian Sternagel. Certification of Termination Proofs Using CeTA. In *Proc. 22nd International Conference on Theorem Proving in Higher Order Logics TPHOL*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer, 2009. 75

[TSSY20] René Thiemann, Jonas Schöpf, Christian Sternagel, and Akihisa Yamada. Certifying the Weighted Path Order (Invited Talk). In Zena Matilde Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:20. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. 1, 19

[Tur37] Alan Mathison Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceeding of the London Mathematical Society*, s2-42(1):230–265, 1937. 1

[Tur49] Alan M. Turing. Checking a large routine. In *Report of a Conference High Speed Automatic Calculating-Machines*, pages 67–69. University Mathematical Laboratory, 1949. 2, 6, 20

[Vaj89] Steven Vajda. *Fibonacci and Lucas Numbers, and the Golden Section: Theory and Applications.* Ellis Horwood series in mathematics and its applications. E. Horwood Limited, 1989. 3

[Vir95] Patrick Viry. Rewriting modulo a rewrite system. Technical report, University of Pisa, 1995. 97

[Vor88] Sergei G. Vorobyov. On the arithmetic inexpressiveness of term rewriting systems. In *[1988] Proceedings. Third Annual Symposium on Logic in Computer Science*, pages 212–217, 1988. 89

[YBH04] Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong normalisation in the π-calculus. *Information and Computation*, 191(2):145 – 202, 2004. 19

[YKS15] Akihisa Yamada, Keiichirou Kusakari, and Toshiki Sakabe. A unified ordering for termination proving. *Science of Computer Programming*, 111:110 – 134, 2015. Special Issue on Principles and Practice of Declarative Programming (PPDP 2013). 1, 19

[YSTK16] Akihisa Yamada, Christian Sternagel, René Thiemann, and Keiichirou Kusakari. AC Dependency Pairs Revisited. In Jean-Marc Talbot and Laurent Regnier, editors, *25th Annual Conference on Computer Science Logic*, volume 62 of *Leibniz International Proceedings in Informatics*, pages 8:1–8:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016. 74