






**On Semantical Methods for
Higher-Order Complexity Analysis**

Deivid Rodrigues do Vale



On Semantical Methods for Higher-Order Complexity Analysis

Deivid Vale

Radboud University



The research reported in this thesis has been carried out under the auspices of the Radboud University and the research school IPA (Institute for Programming research and Algorithmics). This work has been financially supported by the NWO TOP project “Implicit Complexity through Higher-Order Rewriting”, NWO 612.001.803/7571.

IPA Dissertation series, number 2024-01

Cover design by Yijun Guo

Printed by Gilderprint, The Netherlands

Copyright © **Deivid Rodrigues do Vale**, 2024

This work is licensed under Attribution-ShareAlike 4.0 International. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/>.

On Semantical Methods for Higher-Order Complexity Analysis

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.M. Sanders,
volgens besluit van het college college voor promoties
in het openbaar te verdedigen op

vrijdag 19 april 2024
om 10.30 uur precies

door

Deivid Rodrigues do Vale
geboren op 4 oktober 1992
te Unaí (Brazilië)

Promotor:

prof. dr. J.H. Geuvers

Copromotor:

dr. C.L.M. Kop

Manuscriptcommissie:

prof. dr. S.B. Scholz (voorzitter)

prof. dr. J.G. Simonsen (Københavns Universitet, Denemarken)

prof. dr. J. van de Pol (Aarhus Universitet, Denemarken)

dr. V. van Oostrom (University of Sussex, Verenigd Koninkrijk)

dr. C. Fuhs (Birkbeck, University of London, Verenigd Koninkrijk)

In the loving memory of Constance Kaak van Hoorn.
I shall forever miss your wisdom, always followed with stories of a life well lived.

Palavra puxa palavra,
uma ideia traz outra,
e assim se faz um livro,
um governo,
uma revolução,
ou até mesmo uma tese de doutorado,
um velho conhecido meu atribuiu a outrem
a ideia que seria assim que
a natureza compôs suas espécies.

Acknowledgment

It is believed amongst the common folk that the acknowledgment section is the most read segment of a Ph.D. thesis. Some may dare to say that it is the only section friends and family will ever read. Eagerly trying to find what is said about them. That may or may not be true, and I certainly would like to be able to list every single one of you here. If perchance you are not listed below, my dear reader. I apologize for that. Let me then start with a nameless *thank you* function: $\lambda n.$ Thank you, n , for everything.

Thank you, Cynthia, for taking me as your student and for guiding me throughout the whole Ph.D. process. It was for sure an arduous journey and now looking back at it, I feel grateful that you were always there for me. Thank you, for giving me the freedom to make mistakes but also for helping me get out of those rabbit holes I constantly dig myself into. On another note, thank you for introducing me to fencing. It is a shame my days as a medieval soldier didn't last longer. Also, thank you for bringing me food when I was slowly dying of COVID. Thank you, Herman, for the support, especially by the end of my Ph.D.

I would like to thank my collaborators from whom I learned a lot over the years. Thank you, Mauricio Ayala, Patrick Baillot, Ugo Dal Lago, Maribel Fernandez, Liye Guo, Daniele Nantes, and Niels van der Weide. It has been a pleasure to work with all of you.

Thank you, Celius Magalhães, for your mentorship over the years. You energetically encouraged me to do a Ph.D. abroad and even provided me with the financial means to do so. This changed my life. Thank you for supporting me through the dark times of my academic career. Thank you for believing it, even when I did not. Thank you, Daniele Nantes, for your long-term support and friendship. For teaching me how to appreciate mathematics and the little details on those “exemplos espírito de porcos”. Thank you for continually sharing with me the excitement of learning something new.

Moving to a new country can be a painful endeavor. I was lucky to meet Constance Kaak. Thank you, dear Constance. You became my Dutch grandmother, and I was filled with joy to be your Brazilian grandson. Thank you for taking me in and for being part of my life. Thank you, Casper Kaak, for all your help and words of encouragement.

To all my friends and colleagues from Software Science, thank you all for providing such a pleasant work environment. Bharat Garhewal, Dennis Gross, and Hans-Nikolai Viessmann, thank you for all those coffee breaks, jokes, and discussions we had over the

years. Having useless discussions during coffee breaks certainly kept me from being crazy. Even though Dennis — if asked — might disagree. Dennis, thank you for taking me on those bike trips to Germany. You always choose the best itinerary for such trips.

Bharat, thank you for your friendship and companionship. For teaching how India works, for our discussions on history and politics, and for buying an Xbox so we could play video games together. However, I *do not* thank you for that spoiler on Final Fantasy XV. And, for god's sake, stop complaining about Gears of War's lore.

Here's a second paragraph on your honor, as promised.

Liye, thank you for taking me to such amazing Chinese restaurants, and for beta testing those crazy recipes when I decided to mixture Brazilian and Chinese cuisine. Bilibili is indeed amazing, and I became a fan of "the uncle" thanks to you. Let us not forget the cat videos. Thank you for your help on those messy conference travels, and for walking 10km with me on that Tbilisi trip, and for everything else: "thank you, Sir". I could also not forget to say thank you to Kasper Hagens. You taught me how to perfectly fold a piece of paper into a very useful envelope which, indeed, can be used for all sorts of things. You showed me how Dutch people are concerned about their health, and that beers and cigarettes are okay, but god forgive me for adding a pinch of salt to our food. Thank you, Niels, for those discussions on formal methods and proof assistants. You certainly changed my perception of them. Also, thank you for answering my questions about Coq, academic life, and even the Dutch political system. Thank you, Mairielli Wessel, for being such a kind friend to me, and for all those Brazilian dinners we shared. Thank you, Dario Stein, for organizing the small details of our social events. Thank you, Marc Hermes, for those delicious dinners you cooked for me. You are the only one who could make a meat lover like myself satisfied without meat. Thank you, Marcos and Ilka, for caring for me and trying to make me work less.

I would like to register my appreciation for all the work put in by the reading committee on carefully reading this thesis and providing useful feedback on it.

Eu gostaria de agradecer o apoio da minha família e amigos no Brasil. Obrigada mamãe, por sempre acreditar em mim e por ser a única pessoa nesse mundo capaz de sacrificar tudo para que eu pudesse estudar nessa vida. Tudo que é de bom em mim, começou em ti. Agradeço aos meus irmãos, Thiago, Johnne, e Danúbia por tornarem minha vida mais feliz e por serem os melhores irmãos desse mundo. Agradeço à Edna Gomes por fazer parte da minha vida. Eu não poderia pedir por uma companheira melhor que ti. Obrigado Davi, pela amizade duradoura e pelo companheirismo ao longo dos anos. Bruna Nunes, obrigado pela amizade e carinho, apesar de eu nunca ter tempo de visitá-la. Aos amigos Tiago Araújo, Henrique Balbino, Bruno Caxito e Caio Sady. Obrigado por fazerem minha vida fora da academia interessante e por manterem constante contato, apesar de termos um oceano inteiro entre nós. Obrigado, Arthur Resende, pelas recentes jogatinas em Bauldur's Gate e pela amizade ao longo dos anos.

Table of contents

1	Introduction	1
1.1	Technical Overview	5
1.2	Related Work	6
1.3	Content Overview and Contributions	8
2	Higher-Order Rewriting	15
2.1	Curried Higher-Order Rewrite Systems	15
2.1.1	The Syntax of Types and Terms	15
2.1.2	Higher-Order Rewrite Rules	17
2.2	Ordered Sets and Monotonic Functions	20
3	First-Order Tuple Interpretations	23
3.1	Derivation Height and Complexity	23
3.2	From Termination Proofs to Complexity Bounds	24
3.3	Tuple Interpretations for Full Rewriting	28
3.3.1	Strongly Monotonic Tuple Algebras	28
3.3.2	Runtime Complexity Analysis	32
3.4	Cost-Size Products	36
3.5	Tuple Interpretations for Innermost Rewriting	37
3.5.1	Cost-Size Tuple Algebras	37
3.5.2	Innermost Compatibility Theorem	39
3.6	Upper Bounds for Innermost Runtime Complexity	41
3.7	Automation	42
3.7.1	Parametric Tuple Interpretations	43
3.7.2	Strategy-based Search for Tuple Interpretations.	46
3.7.3	Prototype Implementation	48
4	Higher-Order Tuple Interpretations	53
4.1	Strongly monotonic algebras	53
4.1.1	Higher-Order Compatibility	59
4.2	Interpreting abstractions	61

4.2.1	Strongly Monotonic Combinators	62
4.2.2	Making a MakeSM	67
4.2.3	Orienting Beta and Eta	68
4.3	Creating strongly monotonic interpretation functions	69
4.4	Finding Higher-Order Complexity Bounds	74
4.5	Conclusion	78
5	Higher-Order Tuple Interpretations for Call-by-Value	79
5.1	Call-by-Value Higher-order Rewriting	79
5.2	Cost-Size Overview	80
5.3	Cost-Size Semantics for Simple Types	83
5.4	Cost-Size Semantics for Terms	87
5.5	Complexity Analysis of Call-by-Value Rewriting	92
5.6	Conclusions and Future Work	96
6	A Rewriting Characterization of Higher-Order Feasibility	99
6.1	Higher-Order Feasibility	99
6.2	Basic Feasible Functionals	100
6.3	Oracle Turing Machines	104
6.3.1	Computing with Oracle Turing Machines	107
6.3.2	Complexity of Oracle Turing Machines	109
6.4	Kapron and Cook's Characterization of BFF	111
6.5	From Higher-Order Rewriting to BFF and Back Again	111
6.5.1	Type-2 Computability via Higher-Order Rewriting	111
6.6	Soundness	115
6.6.1	Interpreting the extended TRS	115
6.6.2	Term Graph Rewriting	119
6.7	Completeness	129
6.7.1	Constructors	129
6.7.2	Executing the machine	133
6.7.3	A bound on the number of steps	136
6.7.4	Finalizing execution	140
6.8	Conclusions and Future Work	144
7	Certification of Higher-Order Polynomial Interpretations	145
7.1	Certifying Termination Tools	145
7.2	The Basics of Higher-Order Rewriting in Coq	148
7.2.1	Terms and Rewrite Rules	148
7.2.2	Termination	152
7.3	Higher-Order Interpretation Method in Coq	153

7.3.1	Interpreting types and terms	153
7.3.2	Termination Models	157
7.4	The Higher-Order Polynomial Method in Coq	159
7.4.1	Polynomials	159
7.4.2	Polynomial Interpretation	161
7.4.3	Constraint Solving Tactic	162
7.5	Generating Proof Scripts	164
7.5.1	Proof traces for polynomial interpretation	164
7.5.2	Verifying Wanda’s Polynomial Interpretations	165
7.6	Conclusions and Future Work	166
8	Conclusions	167
	References	171
	Samenvatting	181
	Summary	183
	Research Data Management	185
	Titles in the IPA Dissertation Series since 2021	186
	Curriculum Vitae	189

Chapter 1

Introduction

Complexity Theory is a rich field of study that consists of several quite different techniques. This myriad of apparently distinct approaches, however, have two main characteristics in common: first, they are interested in *algorithms* and the problems solved by them; second, they focus on the computational *resources* required by those algorithms to solve problems.

The word algorithm here is taken in a broad sense. Different approaches to complexity theory establish distinct — sometimes even incompatible — notions of algorithm. The seemingly common denominator is that algorithms are usually understood as objects that can be computed via some notion of computation. For instance, in program analysis, an algorithm is a program written in some high-level programming language and is computed by the evaluation machine of said language. Meanwhile, in other approaches, algorithms are seen as boolean circuits, Turing machines, words written in some formal grammar, and so on. Each one of them has an associated notion of computation attached.

A somewhat easier notion to describe is that of resources. In general, by resources, we mean the amount of measurable entities that allow for computation to go on. In a concrete complexity analysis, for instance, the resources are the number of CPU instructions (or their execution time) and bits in memory needed to execute tasks. If one goes a level higher in abstraction, one forgoes CPU time and counts the number of loop iterations used by a program written in a high-level programming language.

The subject of this thesis is perhaps most broadly classified in the field of *structural complexity theory*. That is, our analysis is *structural* because we look for mathematical structures inside the problems and aim to consider a more abstract notion of computation in which algorithms are objects describing computations without reference to any concrete machine model. We are then interested in the classification of such algorithms according to their resource usage (or complexity). In this thesis, we mainly work with term rewriting systems as computational models.

Term Rewriting Systems are a simple but powerful model of computation that encompasses most declarative styles of computation. In such systems, we have *rewriting*

rules that describe how expressions (i.e., terms) can be *rewritten* to (i.e., replaced by) “simpler” ones. Let us consider a rewriting system that computes the `append` function; which given lists q and l produces the list `append q l` containing the elements of the first and second list in that order. We define `append` using the rules below.

$$(R_0) \text{append } [] \ q \rightarrow q \qquad (R_1) \text{append } (x :: q) \ l \rightarrow x :: (\text{append } q \ l)$$

Another important feature of this style of expression is that it is very close to functional programming syntax — at least syntactically; the operational semantics of rewriting can be quite different from that of languages like OCaml or Haskell — where functions are defined by *pattern matching* on the data structures.

The rules above suggest a “motion” in place. In essence, we compute in rewriting by replacing instances of a left-hand side of a rule by its right-hand side. If one views the rules R_0 and R_1 above as directed equations, then computing with rewriting is comparable to a “search and replace operation” on expressions, in the same fashion as in the “replacing equals by equals” principle we learn in primary school.

So if we start with the expression `append [1;2] [2;3]` — noting that `[1;2]` denotes the list `1 :: 2 :: []` — we then express computation using this system as follows:

$$\begin{aligned} \text{append } [1;2] \ [2;3] &\rightarrow 1 :: (\text{append } [2] \ [2;3]) \\ &\rightarrow 1 :: 2 :: (\text{append } [] \ [2;3]) \\ &\rightarrow 1 :: 2 :: [2;3] \\ &= [1;2;2;3] \end{aligned}$$

Hence, computations using term rewriting are done by step-wise transforming the sub-expressions that match one of the rules. The final expression is the list `[1;2;2;3]` and we say it is a *normal form* because it is not possible to compute with it anymore. This chain of reductions we call a *derivation* to suggest the process of deriving the normal form `[1;2;2;3]` from the initial expression `append [1;2] [2;3]`.

With a computation model at hand, the second important component in complexity theory is the notions of *time* and *space* complexities. In the step-by-step computational model induced by rewriting, time complexity is naturally understood as the number of rewriting steps needed to reach normal forms. The derivation of `append [1;2] [2;3]` above takes three units of time since we used three rewriting steps. This naturality comes from the tacit assumption that the cost of performing a computational step is either constant or negligible. Let us analyze the derivation of `append [1;2] [2;3]` once more. We use R_1 to reduce this expression to `1 :: (append [2] [2;3])`, and this matching of R_1 with the redex (reducible expression) `append [1;2] [2;3]` is not *atomic*: we need to first compute the substitution that sends $x \mapsto 1$, $q \mapsto 2 :: []$ and $l \mapsto [2;3]$.

Hence, in the unitary time cost model, the intricacies of a low-level rewriting realization (e.g., a concrete rewriting engine implementation) are ignored. This assumption does not pose a problem as long as the low-level concrete time complexity needed to apply a rule is kept reasonably low. While showing that this is indeed the case is non-trivial, it has been done if some additional mild conditions are imposed; see for instance [77] and Chapter 6.

In the rewriting setting, a *time complexity function* is a function ϕ from $\mathbb{N} \setminus \{0\}$ to \mathbb{N} such that, given n , $\phi(n)$ bounds the length of all rewrite sequences starting with terms of *size* $\leq n$. Two distinct complexity notions are commonly considered in rewriting complexity analysis. The first is *derivational complexity*. It measures the absolute worst-case complexity when deriving a normal form, so a derivational complexity function bounds the length of the longest derivation chain from all possible chains. The second complexity notion is that of *runtime complexity*. It comes from the intuition that if we want to majorize the actual “running time” of computing with a term rewriting system, we should restrict our analysis only to those systems where function symbols are applied only to basic data values, e.g., numbers or lists.

Therefore, by *complexity analysis* of term rewriting, we mean the set of formal tools used to provide bounds to those complexity functions. A common way to determine these bounds is by adapting the proof techniques used to show termination to deduce the complexity induced by the method. There is a myriad of works following this idea. To mention a few, see [13, 21, 26, 52, 53, 83] for interpretation methods, [22, 51, 107] for lexicographic and path orders, and [50, 88] for dependency pairs.

Up to now, we have discussed *first-order term rewriting*. A key feature of this thesis is that we consider *higher-order* systems. Those are the rewriting systems where functions themselves (so the higher-order aspect) can be abstracted away and captured by variables. A running example used throughout the thesis is that of the `map` function which is a higher-order function that takes a function F and a list l and applies F to each element of l . It can be defined in higher-order rewriting as follows:

$$(R_0) \text{ map } F [] \rightarrow [] \qquad (R_1) \text{ map } F (h :: tl) \rightarrow (F h) :: \text{ map } F tl$$

This allows us to define new functions in terms of those rules in a “functional programming style”. For instance, a function that increments a set of counters from a list can be written simply as `incrCounter l` \rightarrow `map` $(\lambda x. x + 1) l$. Computation in such higher-order systems is done similarly to its first-order counterpart. The key difference is that the matching mechanism to apply rules needs to take higher-order variables into account.

As an example, let us consider the following derivation chain:

$$\begin{aligned} \text{incrCounter } [3;5] &\rightarrow \text{map } (\lambda x. x + 1) [3;5] \\ &\xrightarrow{+} [(\lambda x. x + 1) 3; (\lambda x. x + 1) 5] \end{aligned}$$

In the formalism we consider rewriting is *union* β ; so we have infinitely many β -rules of the form $(\lambda x. e) e' \rightarrow e[x := e']$, one such rule for each pair of expressions e, e' . Here, $e[x := e']$ denotes the substitution of all *free* occurrences of x in e for e' . A formal treatment of these notions will be addressed in Chapter 2. By applying β -steps we continue the computation above as follows.

$$\begin{aligned} [(\lambda x. x + 1) 3; (\lambda x. x + 1) 5] &\xrightarrow{+} [3 + 1; 5 + 1] \\ &\xrightarrow{+} [4; 6] \end{aligned}$$

The techniques for structural complexity analysis for such higher-order systems do not always come from a direct generalization of first-order techniques. Indeed, the simple first-order notion of runtime complexity for instance cannot be directly extended as it is not obvious what “basic data” means in the higher-order setting. Additionally, to define a complexity function for higher-order terms we must also set notions of “size” for functional arguments: “what should the size of a function F given to map be?” Functional arguments are inherently infinite objects in the sense that we cannot possibly find a single number majorizing such objects. Furthermore, the behavior of such functional arguments arguably matters in some way. Another source of difficulty is that we have abstraction terms in our higher-order systems, so the cost of a β -step might be significantly higher than a redex step since the abstracted variable might occur deep inside the substituted term.

We set to answer those questions in this thesis by providing mathematical tooling for the complexity analysis of such higher-order systems. The termination method on which we base our complexity analysis framework is tuple interpretations, which is first introduced by the author in [69]. Tuple interpretations are an instance of the interpretation method where expressions are mapped to tuples, for example by mapping every base-type term to a cost–size pair of the following form:

$$\langle \text{cost of reducing terms to normal form, size of normal form} \rangle.$$

Thus, we seek to interpret terms in such a way that the rewrite relation can be embedded in a well-founded ordered set. So a rewriting step on terms implies a strict decrease in their interpretation. For instance, considering the computation of `incrCounter` above, we would like to give an interpretation $\llbracket \text{incrCounter } [3;5] \rrbracket$ of `incrCounter` $[3;5]$ such that

$\llbracket \text{incrCounter } [3; 5] \rrbracket > \llbracket \text{map } (\lambda x. x + 1) [3; 5] \rrbracket$, since the following reduction happens: $\text{incrCounter } [3; 5] \rightarrow \text{map } (\lambda x. x + 1) [3; 5]$.

Tuple interpretations do not provide a complete termination proof method: there are terminating systems for which interpretations cannot be found. Consequently, it does not induce a complete complexity analysis framework either. Notwithstanding, it has the potential to be very powerful if we choose the interpretation sets wisely. A second limitation is that the search for interpretations is undecidable in general, which is expected already in the polynomial case [81]. Undecidability never hindered computer scientists' efforts on mechanizing difficult problems, however. Indeed, several proof search methods have been developed over the years to find interpretations automatically [24, 28, 30, 52, 109].

1.1 Technical Overview

This thesis orbits the notion of semantic interpretations in term rewriting and their applicability to termination and complexity analysis of a variety of term rewriting systems. We consider both first- and higher-order rules in different rewriting strategies. More precisely, when dealing with first-order rewriting in Chapter 3, we consider full rewriting in Section 3.3 and innermost rewriting in Section 3.5. Then we move on to higher-order rules for the rest of the thesis. In the higher-order setting, full rewriting is considered in Chapters 4 and 7 while in Chapters 5 and 6 we use weak call-by-value.

The key idea of the *interpretation method* is that in order to prove termination of a system we find a function $\llbracket \cdot \rrbracket$ from the set of expressions to a well-founded set $(A, >)$. For instance $(\mathbb{N}, >)$. This interpretation function should embed the rewriting relation \rightarrow into the well-founded relation $>$ on A , that is, the following should hold

$$e_0 \rightarrow e_1 \cdots \quad \text{implies} \quad \llbracket e_0 \rrbracket > \llbracket e_1 \rrbracket > \cdots$$

For complexity analysis, by choosing the interpretation domain as \mathbb{N} , we have that the number $\llbracket e_0 \rrbracket$ bounds the number of steps needed to reduce e_0 to its normal form. The main technical difficulties of this approach are proving *compatibility theorems*: assuming some compatibility conditions on the interpretation function we prove that $\llbracket e \rrbracket > \llbracket e' \rrbracket$ whenever $e \rightarrow e'$ follows from $\llbracket \ell \rrbracket > \llbracket r \rrbracket$ being valid for all rules $\ell \rightarrow r$ in the system.

In this thesis, we follow the same approach and develop *tuple interpretations*. The key idea of tuple interpretations is that instead of mapping base-type expressions to a single set — for example \mathbb{N} — we map them to *tuples* — for example \mathbb{N}^k , $k \geq 1$. For instance:

$$\llbracket \text{append } [1; 2] [2; 3] \rrbracket = (3, 4, 3) > (2, 4, 3) = \llbracket 1 :: \text{append } [2] [2; 3] \rrbracket$$

Here, the first component, 3 bounds the *cost* (maximal length of the reduction); the second, 4 bounds the *length* of the resulting list; and, the third, 3 bounds the *maximum element size*. Higher-order expressions, however, are mapped to either functions (when we consider full rewriting) or to pairs of functions (when considering call-by-value rewriting). In this thesis, we are mainly concerned with applying tuple interpretations to study the complexity behavior of rewriting systems. Hence, we are interested in providing upper bounds to their complexity functions (i.e., derivational and runtime complexities), and finally, we apply this technique to provide a higher-order rewriting characterization of higher-order feasibility.

1.2 Related Work

First-Order Rewriting. There are several first-order complexity techniques based on interpretations. For example, in [21], the consequences of using additive, linear, and polynomial interpretations to natural numbers are investigated; and in [52], context-dependent interpretations are introduced, which map terms to real numbers to obtain tight bounds. Most closely related to our approach are *matrix interpretations* [37, 84], and a technique by Cynthia Kop, Aart Middeldorp, and Thomas Sternagel for complexity analysis of conditional term rewriting [67]. In both cases, terms are mapped to tuples as they are in our approach, although neither considers typing information, and matrix interpretations use linear interpretation functions and matrix-based polynomials [32]. The class of tuple interpretations we develop in this thesis is a generalization of both.

Tuple interpretations were originally developed for complexity analysis of higher-order rewriting in [69]. They were also independently developed by Yamada [108] for purposes of proving termination of first-order systems where the usage is focused on using weakly monotonic interpretations as processors for the dependency pair framework. Yamada [109] also shows that tuple interpretations subsume polynomial interpretations (with negative constants), (improved) matrix interpretations, and arctic interpretations, as well as the syntactic method *argument filtering*. Yamada’s work is focused on applying these interpretations to termination analysis of first-order systems while our main focus is complexity analysis of higher-order systems.

Higher-order Rewriting. In *higher-order* term rewriting (but a formalism without λ -abstraction), Baillot and Dal Lago [13] develop a version of higher-order polynomial interpretations which, like the present work, is based on van de Pol’s higher-order interpretations [92]. In similar ways to what we do here in Chapter 4, the authors enforce polynomial bounds on derivational complexity by imposing restrictions on the shape of interpretations. Their method differs from ours in various ways, most importantly by mapping terms to \mathbb{N} rather than tuples. In addition, the interpretations are limited

to higher-order polynomials. This yields an ordering with the subterm property, that is, the inequality $f \dots s \dots > s$ holds for all terms s . Consequently, TRSs like the ones in Example 3.3.7 cannot be handled. Moreover, it is not possible to find a general interpretation for functions like `foldl` or `rec`; the method can only handle instances of `foldl` if the argument function is linear. In this work, we also take inspiration from [38] (which is also based on van de Pol’s techniques) and the authors extend polynomial interpretations to the higher-order setting. Beyond this, it unfortunately seems that relatively little work has thus far been done on complexity analysis of higher-order term rewriting. However, complexity analysis of *functional programs* is an active field of research with a close relation to higher-order term rewriting.

Functional Programming. There are various techniques to statically analyze resource usage of functional programs. These may be fully automated [6, 17, 96], semi-automated designed to reason about programmer specified-bounds [25, 48, 105], or even manual techniques, integrated with type system or program logic semantics [23, 76]. We discuss the most pertinent ones. An approach using rewriting for full-program analysis is to translate functional programs to rewriting systems [7], which can be analyzed using first-order complexity techniques. This takes advantage of the large body of work on first-order complexity but loses information; the transformation often yields a system that is harder to analyze than the original. In this thesis, we work only on the rewriting side of the analysis. A translation from programs to rewriting is outside our scope.

The research methodology in most studies in functional programming differs significantly from rewriting techniques. Nevertheless, there are some studies with clear connections to our approach; in particular our separation of cost and size (and other structural properties). Most relevant, in [34] the authors use a similar approach by giving semantics to a complexity-aware intermediate language allowing arbitrary user-defined notions for size—such as list length or maximum element size; recurrence relations are then extracted to represent the complexity.

Additionally, most modern complexity analysis is done via enhancements at the type system level [4, 6, 36, 48, 54, 93]. For example, types may be annotated with a counter, the heap size or a data type’s size measure. Notably, a line of work on Resource-Aware ML [54, 59, 87] studies resource use of OCaml programs with methods based on Tarjan’s amortized analysis [100]. Types are annotated with *potentials* (a cost measure), and type inference generates a set of linear constraints which is sent over to an external solver. For Haskell, Liquid Haskell [94, 103] provides a language to annotate types, which can be used to prove properties of the program; this was recently extended to include complexity [48]. Unlike RAML, this approach is not fully automatic: type annotations are checked, not derived.

These works in functional programming have a slightly different purpose from ours: they study the resource use in a specific language, typically with a fixed evaluation strategy. Our method, in contrast, considers computation abstractly. First without any restriction on evaluation strategy in Chapter 4 and further considering a call-by-value evaluation strategy in Chapter 5. Our approach has the advantage of being abstract, so it potentially applies to more cases. But also the disadvantage that such an abstract treatment inherently abstracts away from some specific details of programming languages that affect how the computation is performed. Moreover, most of these works limit interest to full-program analysis. We do this for runtime complexity, but our method offers more, by providing interpretations for individual functions like `map` or `foldl`. On the other hand, many do consider (shallow) polymorphism, which we do not.

While in functional programming one considers resource usage [54, 93], rewriting is concerned with the number of steps, which can be translated to a form of resource measure if the true cost of each step is kept low, as we previously discussed. This can be achieved by imposing restrictions on the reduction strategy and the representation of terms [2, 77]. Our approach carries the blessing of being general and machine-independent and the curse of not necessarily being a reasonable cost model. Not all is lost, however, as in Chapter 6 we provide the necessary conditions needed for an efficient realization of our higher-order rewriting formalism by giving a characterization of higher-order feasibility via the rewriting formalism.

1.3 Content Overview and Contributions

This thesis contains a chapter with preliminaries (Chapter 2) and five more chapters (Chapters 3–7) that present its main research contributions. The research content is roughly divided into the following themes: first-order complexity analysis (Chapter 3), higher-order complexity analysis (Chapters 4 and 5), a characterization of higher-order feasibility (Chapter 6), and higher-order termination (Chapter 7). These are thematically connected by the fact that we use “semantic methods” to tackle those problems.

The main contributions of this thesis are twofold. The first is the development of *tuple interpretations*. As we have seen in Section 1.1, this is part of an algebraic interpretation method, and it can be used to reason about both the termination and complexity of rewriting systems. Secondly, we provide sufficient conditions for which tuple interpretations can be used to capture feasibility in the higher-order setting. I list below the main contributions of each chapter.

Chapter 2: Higher-Order Rewriting. This is a chapter on preliminaries. It contains the basic formal definitions of rewriting theory in both first- and higher-order setting. It

is here where most notations and naming conventions used throughout the thesis are set. The content of this chapter does not contain any new results.

Chapter 3: First-Order Tuple Interpretations. In this chapter we develop the notion of tuple interpretations in the context of first-order rewriting. We study two main complexity notions, introduced formally in the chapter, of *derivational* and *runtime* complexity. The main contribution present in this chapter is that we show how the newly introduced tuple interpretation is used to provide upper bounds to those complexity measures. This chapter is mainly based on two publications:

1. Cynthia Kop and Deivid Vale. “Tuple Interpretations for Higher-Order Complexity”. In: *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*. Ed. by Naoki Kobayashi. Vol. 195. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 31:1–31:22. DOI: [10.4230/LIPIcs.FSCD.2021.31](https://doi.org/10.4230/LIPIcs.FSCD.2021.31)
 - Here, we take the first-order content in this publication which corresponds mostly to its Section 3.
2. Liye Guo and Deivid Vale. “Analyzing Innermost Runtime Complexity Through Tuple Interpretations”. In: *Proceedings 17th International Workshop on Logical and Semantic Frameworks with Applications, LSFA 2022, Belo Horizonte, Brazil (hybrid), 23-24 September 2022*. Ed. by Daniele Nantes-Sobrinho and Pascal Fontaine. Vol. 376. EPTCS. 2022, pp. 34–48. DOI: [10.4204/EPTCS.376.5](https://doi.org/10.4204/EPTCS.376.5)

An extended (invited) journal version of the second publication is currently under review (submitted to *Mathematical Structures in Computer Science*). This chapter is mostly based on the extended journal version and includes the first-order contributions of the first paper. The main contributions of this project are:

- We introduce the notion of algebraic interpretations, which was introduced in the first paper, tailored to deal with first-order rewriting.
- We provide sufficient conditions for which tuple interpretations guarantee polynomial bounds to the derivational/runtime complexity of compatible systems.
- We show that tuple interpretations are amenable to automation by providing a tool, which we call Hermes, to automatically find tuple interpretations for first-order systems using the innermost evaluation strategies.

Chapter 4: Higher-Order Tuple Interpretations. In this chapter, we move to higher-order rewriting. We start by introducing strongly monotonic functionals as our interpretation domain and consider the full (unrestricted) evaluation strategy. In this formalism,

the interpretation mechanism needs to take abstractions into account. We show how to interpret abstractions as strongly monotonic functions, which allows us to fully interpret the β and η rules. This chapter is an extended version of

- Cynthia Kop and Deivid Vale. “Tuple Interpretations for Higher-Order Complexity”. In: *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*. Ed. by Naoki Kobayashi. Vol. 195. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 31:1–31:22. DOI: [10.4230/LIPIcs.FSCD.2021.31](https://doi.org/10.4230/LIPIcs.FSCD.2021.31)
 - Here, we take the higher-order content of this publication. Which mainly corresponds to Sections 4–5.

In this thesis, I expand on this paper by providing extended proofs of its results, which were omitted or shortened in the conference version due to space limitations. Additionally, some textual corrections and improvements are only present in this chapter.

The main contributions of this chapter are:

- We develop tuple interpretations for higher-order full rewriting, so no restrictions on the contraction of redexes.
- We show how the β and η rules are interpreted in this setting.
- We provide a conservative notion of *runtime complexity* for this higher-order setting, and we regain basic results from first-order complexity analysis in the higher-order world.

Chapter 5: Higher-Order Tuple Interpretations for Call-by-Value. In this chapter we consider higher-order rewriting following the call-by-value evaluation strategy. The goal here is to get closer to “real functional programs” as this strategy is commonly used for instance in the ML family of languages. We get closer but are not fully there yet, since a full treatment of real-world programs would require for instance a richer type system. Notwithstanding, this chapter provides the theoretical foundation and mathematical correctness for future iterations of tuple interpretations that can deal with actual programs. Hence, we focus on the theory. This chapter is fully based on the following publication

- Cynthia Kop and Deivid Vale. “Cost-Size Semantics for Call-By-Value Higher-Order Rewriting”. In: *8th International Conference on Formal Structures for Computation and Deduction, FSCD 2023, July 3-6, 2023, Rome, Italy*. Ed. by Marco Gaboardi and Femke van Raamsdonk. Vol. 260. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 15:1–15:19. DOI: [10.4230/LIPIcs.FSCD.2023.15](https://doi.org/10.4230/LIPIcs.FSCD.2023.15)

The main contributions of this chapter are:

- We provide a rewriting-based weak call-by-value strategy for higher-order term rewriting.
- We develop the tuple interpretation mechanism for this weak call-by-value rewriting strategy that is general enough to deal with rules of non-base types and partial applications.
- In the context of higher-order term rewriting, we show that the cost and size functions can be completely split.
- We show that we can use those cost–size tuples to define both a termination method under this strategy and a new definition of weak call-by-value runtime complexity along with a methodology to derive bounds for it.

This paper builds on the ideas of [69], which introduces tuple interpretations and a notion of runtime complexity for full higher-order rewriting (without evaluation strategy). The key difference here is our focus on a weak call-by-value evaluation strategy. This allows for tighter bounds but also requires significant technical changes since the “cost” for a term of higher type can no longer be captured by just a function. An additional change compared to [69] is that we have separated the cost and size components into distinct functions. In [69], it is in theory allowed for the size component to depend on the cost component, even though in practice this never happened. By fully separating the components, it is easier to prove the correctness of a given tuple interpretation.

Chapter 6: A Rewriting Characterization of Higher-Order Feasibility. The class of *Basic Feasible Functionals* (BFF) of second-order functionals that are computable in polynomial time was introduced by Kapron and Cook [60]. This class is a low-level machine characterization of the feasible functionals introduced by Mehlhorn [79]. In this chapter we provide a higher-order rewriting characterization of BFF via call-by-value tuple interpretations, so this presents an application for the theory we developed in Chapter 5. In this characterization, we build upon the notion of BFF given by Kapron and Cook [60]. This project is joint work with Patrick Baillot, Cynthia Kop, and Ugo dal Lago. This work has been accepted at the *27th International Conference on Foundations of Software Science and Computation Structures* (FoSSaCs 2024). The main contributions of this work are the following:

- We provide a rewriting-based sound and complete characterization of BFF.
- We introduce graph rewriting for an *applicative* format of our higher-order formalism (we consider the subset of terms that do not contain abstractions) to deal with sharing and duplication of variables.

- We prove that applicative higher-order rewriting with the call-by-value rewriting strategy can simulate (and be simulated) by Oracle Turing Machines with polynomial overhead on time and space whenever the underlying evaluation of terms uses graphs with sharing of duplicated variables. Here there is one additional condition: we only consider orthogonal systems. This restriction is in line with the conditions imposed to get reasonable cost models for first-order rewriting in [77].

Chapter 7: Certification of Higher-Order Polynomial Interpretations. In this chapter, we move away from tuple interpretations and concentrate on higher-order polynomial interpretations from a certification point of view. Our goal here is to provide mechanized verification of the output of termination tools for instance the termination tool Wanda [65]. To achieve this, we introduce the pair Nijn/ONijn consisting of a formalization library, which provides a formalization of basic results in higher-order rewriting theory, and a tool to compile the informal output of termination proofs in a formal format. This chapter is based on the following publication:

- Niels van der Weide, Deivid Vale, and Cynthia Kop. “Certifying Higher-Order Polynomial Interpretations”. In: *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*. Ed. by Adam Naumowicz and René Thiemann. Vol. 268. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 30:1–30:20. DOI: [10.4230/LIPIcs.ITP.2023.30](https://doi.org/10.4230/LIPIcs.ITP.2023.30)

The contributions of this work are as follows:

- We provide a formalization of higher-order algebraic functional systems.
- We provide a mechanized proof of the interpretation method using weakly monotonic algebras.
- We formalize the higher-order polynomial interpretation method.
- We develop a tactic that automatically solves the constraints that arise when using the higher-order polynomial interpretation method.
- We develop an OCaml program that transforms the output of a termination prover into a Coq script that represents the termination proof, i.e., a certification of the validity of the termination proof, the certification is then validated by the Coq’s compiler.

Dependency of Chapters. The chapters in this thesis are mostly independent and self-contained. The exception is the preliminary chapter (Chapter 2) which is a dependency for most chapters. The only completely “isolated” chapter is Chapter 7. Some chapters

contain repetitions and explain the same previously introduced concept but in a slightly different format or context. The notions of cost and size *tuples*, for example, are introduced in all chapters where we introduce a new variant of the tuple interpretation method. This is intentional as those concepts play an important role in each chapter. The figure below depicts the relationship between each chapter.

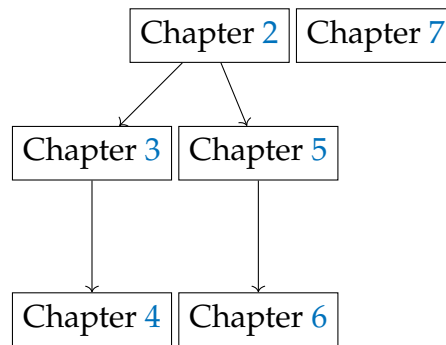


Fig. 1.1 Dependency of Chapters

The dependency of Chapter 4 with Chapter 3 comes from Sections 3.1–3.3.

Statement of Contributions. The publications leading to the production of this thesis had been carried out with collaborators. Therefore, in the paragraphs below I clarify my main involvement in each one of them.

The two publications leading to Chapter 3 are split as follows. In the first paper [69], I carried out the main research, under the supervision of Cynthia Kop. This is the first paper of the Ph.D. project, so major revisions and feedback on writing and proofreading were provided by Cynthia Kop. In the second paper [42], I took the lead on the research and writing the paper. Liye Guo collaborated with parts of the code in Hermes, which mainly comprises our internal implementation of tuple interpretations and the implementation of a wrapper API around the Z3 SMT solver. I was responsible for implementing the front end of Hermes and the algorithm for finding tuple interpretations.

The organization for the publication [69] leading to Chapter 4 follows the same one as previously mentioned. In the publication [70] leading to Chapter 5, which was written together with Cynthia Kop, I carried out the main research and Cynthia Kop took the advisory role. In Chapter 6, the co-authors took the advisory role as well. I owe Ugo dal Lago and Patrick Baillot important discussions for the proof of soundness and completeness of the characterization presented in this chapter. Ugo dal Lago provided us with the main “recipe” to prove the soundness and completeness of the characterization. The proof of soundness of the characterization was developed with advisory work of Ugo dal Lago. Furthermore, Cynthia Kop provided most of the low-level details of the rewriting encoding of Oracle Turing Machines, which drastically

improved my high-level version of the encoding. These contributions are mostly used for the proof of completeness in Chapter 6.

Finally, the paper [106] pertaining to Chapter 7, was a collaboration with my colleague Niels van der Weide who was responsible for the implementation of the formalization library, Nijn. My main contribution to Nijn was that of revisiting the proofs in [38] and help with the development of the correct proof idea that would be formalized, so the formalization can correctly reflect the aspects of rewriting theory. I am the designer and implementor of ONijn. Cynthia Kop provided most of the code changes necessary in Wanda. Niels van der Weide and I shared writing duties equally, with Cynthia Kop providing feedback and textual revisions.

External Publication. I co-authored the following publication during my Ph.D.

- Mauricio Ayala-Rincón, Maribel Fernández, Daniele Nantes-Sobrinho, and Deivid Vale. “Nominal Equational Problems”. In: *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*. Ed. by Stefan Kiefer and Christine Tasson. Vol. 12650. Lecture Notes in Computer Science. Springer, 2021, pp. 22–41. doi: [10.1007/978-3-030-71995-1_2](https://doi.org/10.1007/978-3-030-71995-1_2)

However, I decided not to include it as a chapter in the thesis since it does not fit into the “complexity/termination” narrative umbrella.

Chapter 2

Higher-Order Rewriting

In this chapter, we introduce the higher-order rewriting formalism of *Curried Functional Systems* (CFS). The nomenclature “formalism” has a specific meaning in this thesis. A rewriting formalism defines the conditions for the formation of terms and rules. Furthermore, it also defines equality over terms, the action of substitutions, and establishes the matching mechanism for applying rewriting rules.

2.1 Curried Higher-Order Rewrite Systems

Unlike first-order rewriting which has a uniform formal presentation, the nomenclature ‘higher-order rewriting’ might have a different signification varying according to the intended application domain. In this thesis, we work with *curried functional systems*. This formalism is a variation of *algebraic functional systems* [58] using curried notation and typed function symbols (but no arity) and a monomorphic simple type system. The matching mechanism we consider is a plain modulo alpha matching on syntax and β -reduction is added separately as reduction steps.

2.1.1 The Syntax of Types and Terms

Definition 2.1.1. Let \mathbb{B} be a nonempty set whose elements are called **base types** and range over ι, κ, ν . The set $\mathbb{T}(\mathbb{B})$ of **simple types** over \mathbb{B} is generated by the grammar:

$$\mathbb{T}(\mathbb{B}) ::= \mathbb{B} \mid \mathbb{T}(\mathbb{B}) \Rightarrow \mathbb{T}(\mathbb{B})$$

The letters σ, τ, ρ range over the set of types $\mathbb{T}(\mathbb{B})$. As usual, we will the arrow type constructor \Rightarrow to be right-associative, so we write $\sigma \Rightarrow \tau \Rightarrow \rho$ for $(\sigma \Rightarrow (\tau \Rightarrow \rho))$. Notice that every simple type σ — which is not a base type — can be uniquely written as $\tau_1 \Rightarrow \dots \Rightarrow \tau_n \Rightarrow \iota$ with $m \geq 1$. We informally say that the τ_i ’s are the *input types* and the base type ι is the *output type*. We abbreviate such types by $\tau \Rightarrow \iota$.

Definition 2.1.2. The **type order** (sometimes called **type level**) of a type is inductively defined as the natural number:

- (i) $\text{ord}(\iota) = 0$
- (ii) $\text{ord}(\sigma \Rightarrow \tau) = \max(1 + \text{ord}(\sigma), \text{ord}(\tau))$.

Definition 2.1.3. A **signature** \mathbb{F} is a triple $(\mathbb{B}, \Sigma, \text{typeOf})$ where \mathbb{B} is a set of base types, Σ is a nonempty finite set of function symbols, and typeOf is a function $\text{typeOf} : \Sigma \rightarrow \mathbb{T}(\mathbb{B})$.

For each type σ , we postulate the existence of a nonempty set \mathbb{X}_σ of countably many variables. Furthermore, we impose that $\mathbb{X}_\sigma \cap \mathbb{X}_\tau = \emptyset$ whenever $\sigma \neq \tau$. Let \mathbb{X} denote the family of sets $(\mathbb{X}_\sigma)_{\sigma \in \mathbb{T}(\mathbb{B})}$ indexed by $\mathbb{T}(\mathbb{B})$ and assume that $\Sigma \cap \mathbb{X} = \emptyset$. Mathematically, this means that for each $\sigma \in \mathbb{T}(\mathbb{B})$, the intersection $\Sigma \cap \mathbb{X}_\sigma$ is empty.

Now we have the ingredients to define the set of (uncurried) terms, namely, a signature $\mathbb{F} = (\mathbb{B}, \Sigma, \text{typeOf})$ and a family of variables \mathbb{X} . Intuitively, our notion of terms corresponds to a Simply Typed Lambda Calculus extended with a set of function symbols from Σ .

Definition 2.1.4. The set $\mathbb{T}(\mathbb{F}, \mathbb{X})$ — of **terms** built from \mathbb{F} and \mathbb{X} — collects those expressions s for which the judgment $s : \sigma$ can be deduced using the following rules:

$$\frac{x \in \mathbb{X}_\sigma}{x : \sigma} \text{ (var)} \qquad \frac{f \in \Sigma \quad \text{typeOf}(f) = \sigma}{f : \sigma} \text{ (symb)}$$

$$\frac{s : \sigma \Rightarrow \tau \quad t : \sigma}{(s t) : \tau} \text{ (app)} \qquad \frac{x \in \mathbb{X}_\sigma \quad s : \tau}{(\lambda x. s) : \sigma \Rightarrow \tau} \text{ (lam)}$$

We follow standard conventions in writing terms down. Application of terms is left-associative, so we write $s t u$ for $((s t) u)$. Abstraction is right-associative, so we write $\lambda x y z. s$ for $\lambda x. (\lambda y. (\lambda z. s))$. Application takes precedence over abstraction, which allows us to write $\lambda x. s t$ for $\lambda x. (s t)$. Unnecessary parentheses are removed, and we write terms following these rules. A symbol $f \in \Sigma$ is called the *head symbol* of s if $s = f s_1 \dots s_k$. We identify terms modulo α -equality, so $s = t$ denotes $s =_\alpha t$. We also write \equiv for the syntactical identity of terms.

Definition 2.1.5. A **subterm** of s is a term t (we write $s \triangleright t$) such that one of the following holds:

1. $s \equiv t$;
2. t is a subterm of s' or s'' , if $s \equiv s' s''$;
3. t is a subterm of s' , if $s \equiv \lambda x. s'$.

Let $f : \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \iota$ be a function symbol. Consider a term s of the form $s = f s_1 \dots s_m$. Then, we call s_1, \dots, s_m the **immediate subterms** of s .

Definition 2.1.6. The set $\text{fv}(s)$ collects the **free variables** in s , i.e., those variables not bound by a lambda abstractor. Formally, we define it as follows:

$$\begin{aligned} \text{fv}(f) &= \emptyset & \text{fv}(s t) &= \text{fv}(s) \cup \text{fv}(t) \\ \text{fv}(x) &= \{x\} & \text{fv}(\lambda x. s) &= \text{fv}(s) \setminus \{x\} \end{aligned}$$

A term s is **closed** if $\text{fv}(s) = \emptyset$. It is **ground** if no variable occurs in it.

Definition 2.1.7. A **substitution** γ is a type-preserving map from variables to terms such that the set $\text{supp}(\gamma) = \{x \in \mathbb{X} \mid \gamma(x) \neq x\}$ is finite. We may explicitly represent γ as a list of mappings $[x_1 := s_1, \dots, x_k := s_k]$.

Every substitution γ extends uniquely up to α -equivalence to a type-preserving endomorphism on the set of terms, whose image on s we denote by $s\gamma$. This is expressed in the definition below.

Definition 2.1.8. We define the *capture avoiding* application of γ to s by induction on the structure of s as follows:

$$\begin{aligned} x\gamma &= \gamma(x) & (s t)\gamma &= (s\gamma)(t\gamma) \\ f\gamma &= f & (\lambda x. s)\gamma &= \lambda y. (s^{\{x \mapsto y\}}\gamma), \text{ for } y \text{ fresh} \end{aligned}$$

Here, $s^{\{x \mapsto y\}}$ denotes the term obtained by replacing all free occurrences of x by y in s . By saying y is *fresh* we mean it is a completely new variable name that does not occur anywhere on s or in the range of γ .

It is worth to notice that the result of $s\gamma$ is unique modulo α -renaming.

2.1.2 Higher-Order Rewrite Rules

Notice that in order to express rules we need to fix an arbitrary but fixed signature \mathbb{F} and a set of variables \mathbb{X} . In this way, we keep this choice implicit whenever we talk about term rewriting systems generically. The reader should keep in mind that all definitions below depend on this choice. Let us start by defining rewrite rules.

Definition 2.1.9. A **rewrite rule** (of type σ) is a pair of terms (ℓ, r) of the same type σ , which we denote by $\ell \rightarrow r : \sigma$, such that:

- (i) $\ell = f \ell_1 \dots \ell_k$, and
- (ii) $\text{fv}(r) \subseteq \text{fv}(\ell)$.

The *order* of a rule $\ell \rightarrow r : \sigma$ is the number defined by $\max\{\text{ord}(\sigma) \mid s : \sigma \text{ and } \ell \triangleright s \text{ or } r \triangleright s\}$. For simplicity of notation, we write $\ell \rightarrow r$ instead of $\ell \rightarrow r : \sigma$ whenever the type of the rule can be deduced from context or is irrelevant.

Definition 2.1.10. A **term rewriting system** (TRS) \mathbb{R} over \mathbb{F} is a set of rules. The **order** of a TRS \mathbb{R} is the least n such that the order of all rules in \mathbb{R} is less than (or equal) to n . If no such n exists, we say \mathbb{R} is of infinite order.

Example 2.1.11. We will follow ubiquitous examples in higher-order rewriting and consider the rules implementing `map` and `foldl` with the usual constructor for lists: `[]` : list and `cons` : nat \Rightarrow list \Rightarrow list. The `map` function is of type (nat \Rightarrow nat) \Rightarrow list \Rightarrow list. It applies the functional argument of type nat \Rightarrow nat to each element of the argument list. Another common example is `foldl` of type (nat \Rightarrow nat \Rightarrow nat) \Rightarrow nat \Rightarrow list \Rightarrow list.

$$\begin{array}{ll} \text{map } F [] \rightarrow [] & \text{foldl } F z [] \rightarrow z \\ \text{map } F (x :: q) \rightarrow (F x) :: \text{map } F q & \text{foldl } F z (x :: q) \rightarrow \text{foldl } F (F z x) q \end{array}$$

The symbol `cons` in the rules above is written in infix notation. We use “`::`” for it.

Definition 2.1.12. Every rewrite rule $\ell \rightarrow r$ defines a symbol f , namely, the head symbol of ℓ . For each $f \in \Sigma$, let \mathbb{R}_f denote the set of rewrite rules that define f in \mathbb{R} . A symbol $f \in \Sigma$ is a **defined symbol** if $\mathbb{R}_f \neq \emptyset$. A **constructor symbol** $c \in \Sigma$ is such that $\mathbb{R}_c = \emptyset$.

Hence, whenever we want to refer to such rules we write $\mathbb{R}_{\text{foldl}}$, for the rules defining `foldl`, and \mathbb{R}_{map} , for the rules defining `map`. We let Σ^{def} be the set of defined symbols and Σ^{con} the set of constructor symbols. Hence, $\Sigma = \Sigma^{\text{def}} \uplus \Sigma^{\text{con}}$. For the rest of the thesis, we will assume that the subset $\Sigma^{\text{con}} \subset \Sigma$ is finite. This means that we only have a finite number of data constructors.

Definition 2.1.13. Let \mathbb{R} be a TRS. A **data term** is a term of the form $c d_1 \dots d_k$ where c is a constructor symbol and each d_i is a data term. A **basic term** is a term of base type and of form $f d_1 \dots d_m$ where f is a defined symbol and all d_1, \dots, d_m are data terms.

In most of the examples in this thesis data types appear quite often: `nat` to represent natural numbers (in unary representation) and `list` to represent lists of numbers.

The constructors for natural numbers are `0` : nat and `s` : nat \Rightarrow nat. All ground terms built using those are of the form `s(s(s...s0...))`, that is, such terms are composed of several successive applications of `s` to `0`. There is a bijective correspondence of such terms to natural numbers in \mathbb{N} . Indeed, this representation of numbers is called *unary representation* of the natural numbers. We define the following notational scheme, so we can write these terms in a more compact form.

Notation Scheme 2.1.14. We let $\ulcorner n \urcorner$ denote the ground term `s(s(s...s0...))`, of type nat that is composed of n successive applications of `s` to `0`.

The constructors for lists are `[]` : list (which represents the empty list) and `cons` : nat \Rightarrow list \Rightarrow list. We often write `cons` in infix notation, as in Example 2.1.11. We also may write

lists in the common format $[s_1; \dots, s_k]$. Sometimes it is more convenient to write lists in postfix notation, so we use the most convenient format at will.

Let \mathbb{R} be a TRS over $\mathsf{T}(\mathbb{F}, \mathbb{X})$. We define the notion of reducibility by defining whenever a term s *reduces* to a term t . Additionally to the rules in \mathbb{R} , we want to rewrite terms of the form $(\lambda x. s) t$, which we call β -redex. In our setting, we implement β reduction steps by considering a β rule-scheme.

Definition 2.1.15. The **higher-order rewrite relation** (union β) $\rightarrow_{\mathbb{R}}$, induced by a set of rules \mathbb{R} , is defined as follows:

1. $\ell \gamma \rightarrow_{\mathbb{R}} r \gamma$, for any substitution γ and rule $\ell \rightarrow r$ in \mathbb{R} ;
2. $(\lambda x. s) t \rightarrow_{\mathbb{R}} s[x := t]$;
3. $s t \rightarrow_{\mathbb{R}} s' t$, whenever $s \rightarrow_{\mathbb{R}} s'$, and $s t \rightarrow_{\mathbb{R}} s t'$, whenever $t \rightarrow_{\mathbb{R}} t'$;
4. $\lambda x. s \rightarrow_{\mathbb{R}} \lambda x. s'$, whenever $s \rightarrow_{\mathbb{R}} s'$.

The cases (1)-(2) say that (i) every instance of a rule in \mathbb{R} induces a rewriting step, and (ii) a beta redex also induces a step for any pair of terms s, t . The cases (3)-(4) are the structural congruence closure of (1)-(2). This allows us to apply a rule (or β) step at any subterm of a term s . We say an \mathbb{R} -*reducible expression* (redex) in a term s is a subterm of s of the form $\ell \gamma$, for some rule $\ell \rightarrow r$ and substitution γ . A β -redex is a subterm of s that is of form $(\lambda x. s') t$. We write $\xrightarrow{+}_{\mathbb{R}}$ for the transitive closure of $\rightarrow_{\mathbb{R}}$.

Remark 2.1.16. Note that we do not, by default, include the common η -reduction rule-scheme defined as: $\lambda x. s x \rightarrow s$, if x does not occur free in s .

We avoid this because not all sources consider it, and we want to maintain compatibility with the standard literature. Nonetheless, it is easy to add support for η -reduction in our formalism by including, for all types σ, τ , the rule schemata

$$\lambda x. F x \rightarrow F$$

with $F \in \mathbb{X}_{\sigma \Rightarrow \tau}$ in Definition 2.1.15 above. Notice that the variable side condition is guaranteed to hold by the substitution mechanism. Indeed, any instance of F in $\lambda x. F x$ cannot have free occurrences of the abstracted variable x as if that would be the case then x would get renamed to a new fresh variable y .

First-Order Rules. Intuitively, first-order functions only take primitive values as objects, e.g., numbers, lists, and so on. Contrary to the previously introduced terms where function symbols are allowed to take arguments of functional type. In the definition below, we specify the class of first-order terms that we will work with, mainly on Chapter 3.

Definition 2.1.17. In this thesis, by **first-order terms**, we mean the restriction of the possible expressions from $\mathbb{T}(\mathbb{F}, \mathbb{X})$ such that (i) the only way to form terms is by using the (var), (symb), and (app) rules, and (ii) the order of such terms cannot be greater than 1. We collect terms satisfying (i) and (ii) in the set $\mathbb{T}_{fo}(\mathbb{F}, \mathbb{X})$.

A first-order term rewriting system is a term rewriting system such that its order is at most 1.

Example 2.1.18. We can then write first-order rules. Let us illustrate this by defining a first-order system computing the Fibonacci sequence. We start by giving the types involved. So the base type set is $\mathbb{B} = \{\text{nat}\}$. For Σ we set $\Sigma = \{0, s, \text{add}, \text{fib}\}$ with types as follows: $0 : \text{nat}$, $s : \text{nat} \Rightarrow \text{nat}$, $\text{add} : \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$, and finally $\text{fib} : \text{nat} \Rightarrow \text{nat}$.

$$\begin{array}{ll} \text{add } x \ 0 \rightarrow x & \text{add } x \ (s \ y) \rightarrow s \ (\text{add } x \ y) \\ \text{fib } 0 \rightarrow 0 & \text{fib } (s \ 0) \rightarrow s \ 0 \\ \text{fib } (s \ (s \ x)) \rightarrow \text{add } (\text{fib } (s \ x)) \ (\text{fib } x) & \end{array}$$

Example 2.1.19. In the example below we provide two more examples of first-order systems. Here, $\text{dbl} : \text{nat} \Rightarrow \text{nat}$ and $\text{mult} : \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$.

$$\begin{array}{ll} \text{dbl } 0 \rightarrow 0 & \text{mult } x \ 0 \rightarrow 0 \\ \text{dbl } (s \ x) \rightarrow s \ (s \ (\text{dbl } x)) & \text{mult } x \ (s \ y) \rightarrow \text{add } x \ (\text{mult } x \ y) \end{array}$$

These rules will be used throughout the thesis as toy examples.

2.2 Ordered Sets and Monotonic Functions

In this section, we set basic definitions and notations on ordered sets and monotonic functions that we shall use throughout the thesis.

Definition 2.2.1. A **quasi-ordered set** (A, \sqsupseteq) consists of a set A and a quasi-order (reflexive and transitive) binary relation \sqsupseteq on A . An **extended well-founded set** $(A, >, \gtrsim)$ is a nonempty set A together with a well-founded order $>$ (i.e., an irreflexive and transitive well-founded relation) and a quasi-order \gtrsim on A such that \gtrsim is compatible with $>$ that is:

1. $x > y$ implies $x \gtrsim y$, and
2. $x > y \gtrsim z$ implies $x > z$.

Notice that it is permitted, but not required, that \gtrsim is the reflexive closure of $>$. In this thesis, we refer to an extended well-founded set simply as *well-founded set*.

Definition 2.2.2. The unit set is the quasi-ordered set $(\{u\}, \sqsupseteq)$, with $u \sqsupseteq u$.

Given quasi-ordered sets (A, \sqsupseteq) and (B, \succeq) , a function $f : A \rightarrow B$ is *weakly monotonic* if $x \sqsupseteq y$ implies $f(x) \succeq f(y)$. Let $A \Longrightarrow B$ denote the set of weakly monotonic functions from A to B . The comparison operator \succeq on B induces a point-wise comparison on $A \Longrightarrow B$ as follows: $f \succeq g$ if $f(x) \succeq g(x)$ for all $x \in A$. This way $(A \Longrightarrow B, \succeq)$ is also quasi-ordered.

Given well-founded sets $(A, >, \succeq)$ and $(B, >, \succeq)$, a function $f : A \rightarrow B$ is said to be *strongly monotonic* if $x > y$ implies $f(x) > f(y)$ and $x \succeq y$ implies $f(x) \succeq f(y)$. We say that a weakly monotonic function $f \in A_1 \Longrightarrow \dots \Longrightarrow A_k \Longrightarrow B$ is *strict* in argument i if A_i is a well-founded set and for all $x_1 \in A_1, \dots, x_i \in A_i, \dots, x_k \in A_k$ and $x'_1 \in A_1, \dots, x'_i \in A_i, \dots, x'_k \in A_k$ such that $x_i > x'_i$, we have $f(x_1, \dots, x_i, \dots, x_k) > f(x'_1, \dots, x'_i, \dots, x'_k)$. Note that if f is strict in argument i , then $f(x_1, \dots, x_{i-1})$ is a strongly monotonic function from A_i to $A_{i+1} \Longrightarrow \dots \Longrightarrow A_k \Longrightarrow B$. Hence, a weakly monotonic function that is strict in argument i is also strongly monotonic in the same argument.

In this thesis, we admit the functional extensionality axiom. Both in *pen-and-paper* reasoning up to Chapter 6 and in the formalization described in Chapter 7. So two function f, g are equal iff for all x , $f(x) = g(x)$.

Chapter 3

First-Order Tuple Interpretations

In this chapter, we consider a style of first-order many-sorted rewriting. This means that we restrict ourselves to base-type terms in $\mathsf{T}_{fo}(\mathbb{F}, \mathbb{X})$ as in Definition 2.1.17. Here we set out to develop the notions of tuple interpretation in this first-order setting for both full and innermost evaluation strategies.

3.1 Derivation Height and Complexity

Given a well-founded and finitely branching relation \rightsquigarrow on terms, we write $s \rightsquigarrow^n t$ if there is a sequence of steps $s = s_0 \rightsquigarrow \cdots \rightsquigarrow s_n = t$ of length n . The *derivation height* $\text{dh}_{\mathbb{R}}(s, \rightsquigarrow)$ of a term s with respect to \rightsquigarrow is the length of the longest \rightsquigarrow -sequence starting with s , i.e., $\text{dh}_{\mathbb{R}}(s, \rightsquigarrow) = \max\{n \mid \exists t \in \mathsf{T}(\mathbb{F}, \mathbb{X}) : s \rightsquigarrow^n t\}$. The *absolute size* of a term s , denoted by $|s|$, is 1 if s is a symbol in Σ or a variable, and $|s_1| + |s_2|$ if $s = s_1 s_2$. Let $\mathsf{Tm} \subseteq \mathsf{T}(\mathbb{F}, \mathbb{X})$. In order to express various complexity notions in the rewriting setting, we define the *complexity function* as follows: $\text{comp}(n, \rightsquigarrow, \mathsf{Tm}) = \max\{\text{dh}_{\mathbb{R}}(s, \rightsquigarrow) \mid s \in \mathsf{Tm} \text{ and } |s| \leq n\}$. Intuitively, $\text{comp}(n, \rightsquigarrow, \mathsf{Tm})$ is the length of the longest \rightsquigarrow -sequence starting with a term (that belongs to Tm) whose absolute size is at most n . We summarize four particular instances of complexity functions below:

	derivational	runtime
full	$\text{dc}_{\mathbb{R}}(n) = \text{comp}(n, \rightarrow_{\mathbb{R}}, \mathsf{T}(\mathbb{F}, \mathbb{X}))$	$\text{rc}_{\mathbb{R}}(n) = \text{comp}(n, \rightarrow_{\mathbb{R}}, \mathsf{T}_b(\mathbb{F}))$
innermost	$\text{idc}_{\mathbb{R}}(n) = \text{comp}(n, \rightarrow_{\mathbb{R}}^i, \mathsf{T}(\mathbb{F}, \mathbb{X}))$	$\text{irc}_{\mathbb{R}}(n) = \text{comp}(n, \rightarrow_{\mathbb{R}}^i, \mathsf{T}_b(\mathbb{F}))$

Here, $\mathsf{T}_b(\mathbb{F})$ denotes the set of basic terms built over \mathbb{F} . The notions of derivational and runtime complexity differ by the set of terms we allow reductions to start from. In the former, there is no restriction on the starting term of reductions, so $\text{dc}_{\mathbb{R}}(n)$ needs to bound the derivation height of all reductions starting with terms of absolute size $\leq n$.

Meanwhile, in the latter, we consider reductions only starting from basic terms, so we consider terms such that a single function symbol is applied to data terms.

The example below collects some first-order functions we will use throughout the chapter.

Example 3.1.1. We fix nat and list for the sorts of natural numbers and lists of natural numbers, respectively. In the TRS below, $0:\text{nat}$, $s:\text{nat} \Rightarrow \text{nat}$, $[]):\text{list}$ and $\text{cons}:\text{nat} \Rightarrow \text{list} \Rightarrow \text{list}$ are constructors while $\text{add}:\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$, $\text{append}:\text{list} \Rightarrow \text{list} \Rightarrow \text{list}$, $\text{sum}:\text{list} \Rightarrow \text{nat}$, and $\text{rev}:\text{list} \Rightarrow \text{list}$ are defined symbols.

$$\begin{array}{ll}
 \text{add } x \ 0 \rightarrow x & \text{sum } [] \rightarrow 0 \\
 \text{add } x \ (s \ y) \rightarrow s \ (\text{add } x \ y) & \text{sum } (\text{cons } x \ q) \rightarrow \text{add } (\text{sum } q) \ x \\
 \text{append } [] \ q' \rightarrow q' & \text{rev } [] \rightarrow [] \\
 \text{append } (\text{cons } x \ q) \ q' \rightarrow \text{cons } x \ (\text{append } q \ q') & \\
 \text{rev } (\text{cons } x \ q) \rightarrow \text{append } (\text{rev } q) \ (\text{cons } x \ []) &
 \end{array}$$

Remark 3.1.2. We could have used a simpler version of rev in the example above by using an accumulator. For instance, consider the rule $\text{rev } q \rightarrow \text{aux } q \ []$ and the two auxiliary rules $\text{aux } [] \ q' \rightarrow q'$ and $\text{aux } (\text{cons } x \ q) \ q' \rightarrow \text{aux } q \ (\text{cons } x \ q')$. The version that uses append is useful for showing how to combine different functions and how to interpret the result.

3.2 From Termination Proofs to Complexity Bounds

It is common in the rewriting literature to use termination proofs to assess the *difficulty* of rewriting a term to normal form [8, 53]. The intuition comes from the realization that termination proofs not only prove the absence of infinite reduction chains but also can be used to provide upper bounds on the derivational (or runtime) complexity function. Then one assesses the “power” of termination-proof techniques by looking at their induced bounds on the derivational (runtime) complexity functions. This program has been suggested in [53]. This principle applies not only to syntactic termination proofs [50, 51, 82]; but also to semantical methods [52, 53, 83].

This observation is natural for interpretation-based termination proofs. Indeed, let us consider interpretations into \mathbb{N} as an example, for instance, polynomial interpretations. In this setting, each term s is mapped to a natural number $\llbracket s \rrbracket$. The interpretation function $\llbracket \cdot \rrbracket$ is such that whenever a reduction is fired on terms there is a strict decrease on \mathbb{N} , i.e., if s reduces to a term t , then their respective interpretation decreases so $\llbracket s \rrbracket > \llbracket t \rrbracket$ in \mathbb{N} . As such, the number $\llbracket s \rrbracket$ gives an upper bound on the length of

reductions starting with s since this inequality holds for any reduction sequence starting with s . We shall make this discourse more formal below.

Since the term formalism we consider in this thesis is simply typed and uncurried, we update the classical treatment of *monotonic algebras* from [11] to it. Let $\mathbb{F} = (\mathbb{B}, \Sigma, \text{typeOf})$ be a signature, we call it *first-order* whenever for all $f \in \Sigma$, $\text{typeOf}(f)$ is of order at most 1.

Definition 3.2.1. Let $\mathbb{F} = (\mathbb{B}, \Sigma, \text{typeOf})$ be a first-order signature. A **strongly monotonic** \mathbb{F} -algebra $\mathcal{F} = (A, \mathcal{J})$ consists of a family of extended well-founded sets $A = \{(A_\iota, >_\iota, \geq_\iota)\}_{\iota \in \mathbb{B}}$ together with an interpretation \mathcal{J} that maps each $f : \iota_1 \Rightarrow \dots \Rightarrow \iota_m \Rightarrow \kappa$ in Σ to a strongly monotonic function (in all arguments) $\mathcal{J}_f : A_{\iota_1} \Rightarrow \dots \Rightarrow A_{\iota_m} \Rightarrow A_\kappa$.

In order to interpret terms from $\mathbb{T}(\mathbb{F}, \mathbb{X})$ we extend \mathbb{F} -algebras to the set of terms using a valuation function α mapping variables to elements in A , as usual. The difference is that now valuations have to reflect the typing information given by \mathbb{F} . Hence, we say a valuation is a function α that maps variables of base type ι to elements of A_ι .

Definition 3.2.2. We then extend \mathcal{J} to a function $\llbracket \cdot \rrbracket_\alpha^{\mathcal{J}}$ over the set of terms by letting:

1. $\llbracket x \rrbracket_\alpha^{\mathcal{J}} = \alpha(x)$ if x is a variable of type ι ,
2. $\llbracket f \rrbracket_\alpha^{\mathcal{J}} = \mathcal{J}_f$, and
3. $\llbracket s t \rrbracket_\alpha^{\mathcal{J}} = \llbracket s \rrbracket_\alpha^{\mathcal{J}}(\llbracket t \rrbracket_\alpha^{\mathcal{J}})$.

We will generally omit the annotations \mathcal{J} and α from $\llbracket \cdot \rrbracket_\alpha^{\mathcal{J}}$ when those are clear from the context. So we may write $\llbracket s \rrbracket$ instead of $\llbracket s \rrbracket_\alpha^{\mathcal{J}}$. Additionally, for the interpretations we consider in this section, we fix the extended well-founded set $(\mathbb{N}, >, \geq)$ for all base types $\iota \in \mathbb{B}$ and the interpretation functions \mathcal{J}_f are polynomial functions over \mathbb{N} . This is essentially the classical polynomial interpretations instantiated to our typed uncurried setting. We may write something like $\llbracket s \rrbracket = x + y$ to mean $\llbracket s \rrbracket_\alpha^{\mathcal{J}} = \alpha(x) + \alpha(y)$.

Definition 3.2.3. We say that a TRS \mathbb{R} is **compatible** with an \mathbb{F} -algebra \mathcal{A} whenever $\llbracket \ell \rrbracket_\alpha^{\mathcal{J}} > \llbracket r \rrbracket_\alpha^{\mathcal{J}}$ for all rules $\ell \rightarrow r \in \mathbb{R}$ and valuations α .

A classical result from strongly monotonic algebras is the following *compatibility theorem*, which formalizes the ideas we just mentioned.

Theorem 3.2.4 (Compatibility). If a TRS \mathbb{R} is compatible with an \mathbb{F} -algebra \mathcal{F} , then for all valuations α , we have $\llbracket s \rrbracket_\alpha^{\mathcal{J}} > \llbracket t \rrbracket_\alpha^{\mathcal{J}}$, whenever $s \rightarrow t$.

Corollary 3.2.5. If a TRS \mathbb{R} is compatible with an \mathbb{F} -algebra \mathcal{F} then it is terminating.

Recall that, intuitively, the $\text{dh}_{\mathbb{R}}(s)$ function describes the worst-case number of steps for all possible reductions starting with s . Then, as a consequence of the Compatibility Theorem, we see that $\text{dh}_{\mathbb{R}}(s) \leq \llbracket s \rrbracket$, for any term $s : \iota$. Hence, the interpretation function $\llbracket \cdot \rrbracket$ can be used to bound the derivation height function.

Example 3.2.6. Let us consider the TRS \mathbb{R}_{add} , which contains the two rules defining the symbol `add` from Example 2.1.18 $\text{add } x \ 0 \rightarrow x$ and $\text{add } x \ (\text{s } y) \rightarrow \text{s}(\text{add } x \ y)$. These rules are simple enough to suggest the following interpretation¹:

$$\mathcal{J}_0 = 0 \qquad \mathcal{J}_s = \lambda x.x + 1 \qquad \mathcal{J}_{\text{add}} = \lambda xy.x + 2y + 1$$

This interpretation is compatible with the rules in \mathbb{R}_{add} . Indeed, $\llbracket \text{add } x \ 0 \rrbracket = x + 1 > x = \llbracket x \rrbracket$ and $\llbracket \text{add } x \ (\text{s } y) \rrbracket = x + 2y + 3 > x + 2y + 2 = \llbracket \text{s}(\text{add } x \ y) \rrbracket$. Notice that every ground term s over this system normalizes to a term of the form $\text{s}(\text{s} \dots (\text{s } 0))$ which is the application of n successive `s`'s to 0. Recall that we use the notation $\ulcorner n \urcorner$ for such terms.

If we start with a term of the shape $s = \text{add } \ulcorner n \urcorner \ulcorner m \urcorner$, we get reductions like

$$\text{add } \ulcorner n \urcorner \ulcorner m \urcorner \rightarrow \text{s}(\text{add } \ulcorner n \urcorner \ulcorner m - 1 \urcorner) \xrightarrow{+} \text{s}^m(\text{add } \ulcorner n \urcorner \ 0) \rightarrow \ulcorner n + m \urcorner$$

The length of this reduction chain is exactly $m + 1$ and one can easily deduce that $\text{dh}_{\mathbb{R}}(s) = m + 1$. The interpretation $\llbracket \text{add } \ulcorner n \urcorner \ulcorner m \urcorner \rrbracket = n + 2m + 1$.

However, since in a reduction chain $s \xrightarrow{+} \ulcorner n \urcorner$ the interpretation of s bounds the length of any possible reduction starting with s , it may give a severe overestimation to the derivation height of s . This can be illustrated by a term s of the shape $\text{add } \ulcorner n \urcorner (\text{add } \ulcorner n \urcorner (\text{add } \ulcorner n \urcorner \ulcorner m \urcorner))$. Its interpretation is $n + 2n + 2^2n + 2^3m + 3 \leq 2^{c|s|}$, for some sufficiently large constant c . Meanwhile, $\text{dh}_{\mathbb{R}}(s)$ admits the linear upper bound $3(m + n + 1)$.

The reason for such a huge overestimation, already observed in [53], is that polynomial interpretations with nested function calls like ones in s above provide exponential bounds to the derivation height of terms. The further we move deeper in the term tree, the higher our overestimation. This motivates the definition of the runtime complexity notion: we remove those nested functional calls and consider reductions that only start with basic terms. In the lines of Example 3.2.6, we consider only terms of the shape $\text{add } \ulcorner n \urcorner \ulcorner m \urcorner$.

Runtime complexity is good enough when one is concerned with single functions applied to data. However, we are still not satisfied. A natural but subtle observation is that the interpretation $\mathcal{J}_{\text{add}} = \lambda xy.x + 2y + 1$ of `add` is not tight enough to provide good upper bounds. Indeed, successive nested applications of `add` in a term would

¹A note on notation: we use the notation $\mathcal{J}_s = \lambda x.x + 1$ to mean that the image of $\mathcal{J}(\cdot)$ under the symbol `s` is the function that maps x to $x + 1$, mathematically this is the nameless function $x \mapsto x + 1$. This λ style is particularly useful when dealing with successive maps, like with \mathcal{J}_{add} . We are not interpreting first-order terms as λ -terms in the object level but rather using this suggestive notation in the meta-level to refer to functions.

naturally give rise to exponential overheads in the interpretation. A follow-up question is naturally raised.

“Can we find a natural, tight, polynomial interpretation over \mathbb{N} that is compatible with \mathbb{R}_{add} and correctly captures its worst-case derivational complexity?”

It turns out the answer is negative. To see this let us consider a parametric (on the polynomial coefficients) interpretation of \mathbb{R}_{add} satisfying compatibility:

$$\mathcal{J}_0 = a \qquad \mathcal{J}_s = \lambda x.P(x) \qquad \mathcal{J}_{\text{add}} = \lambda x.x + y + b$$

The compatibility assumption imposes the following constraints

$$\llbracket \text{add } x \ 0 \rrbracket = x + a + b > x = \llbracket x \rrbracket \qquad \llbracket \text{add } x \ (\text{s } y) \rrbracket = x + P(y) + b > P(x + y + b)$$

The first constraint requires that either $a \geq 1$ or $b \geq 1$, while the second requires that $x + P(y) + b > P(x + y + b)$, which is impossible for we assume P is strongly monotonic over \mathbb{N} . Let us analyze a much simpler example, but with similar behavior.

Example 3.2.7. Let \mathbb{R}_a be the TRS with only a rule $a(b\ x) \rightarrow b(a\ x)$ and signature $a, b : \text{bnat} \Rightarrow \text{bnat}$ and $\varepsilon : \text{bnat}$. We can prove termination by the following interpretation:

$$\llbracket \varepsilon \rrbracket = 0 \qquad \mathcal{J}_b = \lambda x.x + 1 \qquad \mathcal{J}_a = \lambda x.2x$$

Indeed, we have $\llbracket \ell \rrbracket > \llbracket r \rrbracket$ for the only rule as $\llbracket a(b\ x) \rrbracket = 2x + 2 > 2x + 1 = \llbracket b(a\ x) \rrbracket$. Now consider a term $s = a^n(b^m\ \varepsilon)$. Then $\text{dh}_{\mathbb{R}}(s) = nm$ whereas $\llbracket t \rrbracket = 2^n m$.

We can find a tight upper bound for \mathbb{R}_a by a reasoning like the following: for every term s , let $\#bs(s)$ be the number of b occurrences in s . For a term t , let $\text{cost}(t)$ denote $\sum \{\#bs(s) \mid a(s) \text{ is a subterm of } t\}$. Then, the cost of a term decreases exactly by 1 in each step. As the normal form has cost 0, we find the tight bound $\text{cost}(a^n(b^m\ \varepsilon)) = nm$. This reasoning relies on tracking more than one value simultaneously, and it cannot be expressed directly as polynomials over \mathbb{N} .

Examples 3.2.6 and 3.2.7 albeit computing very different functions share a common pattern regarding the interplay between their complexity and termination proofs: it is easy to compute with those systems even in the presence of nested function calls, but their polynomial termination proofs encode all cost information at once. This is not a problem if we are only interested in proving termination, but such overestimation is problematic if we want to use interpretations to establish upper bounds to the complexity of such systems.

It turns out we still can formalize this reasoning using an algebraic interpretation: we consider algebras that split interpretations into two components; one for *cost* and another for *size*.

3.3 Tuple Interpretations for Full Rewriting

Let us develop a class of strongly monotonic \mathbb{F} -algebras that are capable of dealing with the problems we studied above. We start with a definition.

3.3.1 Strongly Monotonic Tuple Algebras

Definition 3.3.1. A **strongly monotonic tuple algebra** is an \mathbb{F} -algebra $\mathcal{F} = (A, \mathcal{J})$ with domain $A = \{(A_\iota, >_\iota, \geq_\iota)\}_{\iota \in \mathbb{B}}$ such that each A_ι has the form $\mathbb{N}^{K(\iota)}$ (for an integer $K(\iota) \geq 1$). We then define the relations $\geq_\iota, >_\iota$ as follows:

1. $\langle x_1, \dots, x_{K(\iota)} \rangle \geq_\iota \langle y_1, \dots, y_{K(\iota)} \rangle$, if each $x_i \geq y_i$,
2. $\langle x_1, \dots, x_{K(\iota)} \rangle >_\iota \langle y_1, \dots, y_{K(\iota)} \rangle$, if the above holds and additionally $x_1 > y_1$.

Notice that such tuple algebras are parametrized by the interpretation we give to each base type, which defines the domain A . So the interpretation domain is parametrized by a function K that maps each base type ι to the well-founded set A_ι . In the definition above, the $K : \mathbb{B} \rightarrow \mathbb{N}$ function maps each base type ι to a number $K(\iota)$.

Intuitively, the first component always indicates “cost”: the number of steps needed to reduce a term to normal form. This is the component that needs to decrease in each rewrite step to have $\llbracket s \rrbracket > \llbracket t \rrbracket$ whenever $s \rightarrow t$. The remaining components represent some value of interest for the base type. This could for example be the size of the term (or its normal form), the length of a list, or following Example 3.2.7, the number of occurrences of a specific symbol. For these components, we only require that they do not increase in a reduction step. So $\llbracket s \rrbracket \geq \llbracket t \rrbracket$ whenever $s \rightarrow t$.

By the definition of $>_\iota$, and using Theorem 3.2.4, we can conclude:

Corollary 3.3.2. If a TRS \mathbb{R} is compatible with a tuple algebra then it is terminating and $\text{dh}_{\mathbb{R}}(t) \leq \llbracket t \rrbracket_1$, for all terms t . (Here, $\llbracket t \rrbracket_1$ indicates the first component of the tuple $\llbracket t \rrbracket$).

Using this, we obtain a tight bound on the derivation height of $a^n(b^m(\varepsilon))$ in Example 3.2.7. Here, again, subscripts indicate tuple projections, i.e., $\langle x, y \rangle_1 = x$ and $\langle x, y \rangle_2 = y$.

Example 3.3.3. The TRS \mathbb{R}_a is compatible with the tuple algebra with $A_{\text{bnat}} = \mathbb{N}^2$ and

$$\mathcal{J}_a = \lambda x. \langle x_1 + x_2, x_2 \rangle \quad \mathcal{J}_b = \lambda x. \langle x_1, x_2 + 1 \rangle \quad \mathcal{J}_\varepsilon = \langle 0, 0 \rangle$$

The functions \mathcal{J}_a and \mathcal{J}_b are strongly monotonic. For example, considering \mathcal{J}_a : if $x \geq_{\text{bnat}} y$ then $x_1 + x_2 \geq y_1 + y_2$ and $x_2 \geq y_2$; if $x >_{\text{bnat}} y$ then $x_1 + x_2 > y_1 + y_2$ (since $x_1 > y_1$ and $x_2 \geq y_2$).

Note that the first component exactly sums $\#bs(s)$ for every subterm s which has the form $a\ t$, and for every ground term s we have $\llbracket s \rrbracket_2 = \#bs(s)$. The components in this interpretation are exactly those two measures we keep track of in Example 3.2.7. This interpretation is compatible with the rules of \mathbb{R}_a . Indeed, we have $\llbracket a(b\ x) \rrbracket = \langle x_1 + x_2 + 1, x_2 + 1 \rangle >_{\text{bnat}} \langle x_1 + x_2, x_2 + 1 \rangle = \llbracket b(a(x)) \rrbracket$. Finally, we can see that $\llbracket a^n(b^m\ \varepsilon) \rrbracket = \langle nm, m \rangle$.

To build strongly monotonic functions we can use the following observation:

Lemma 3.3.4. A function in $f : \mathbb{N}^{K(\iota_1)} \Longrightarrow \dots \Longrightarrow \mathbb{N}^{K(\iota_k)} \Longrightarrow \mathbb{N}^{K(\kappa)}$ is strongly monotonic if we can write f as follows

$$\lambda x^1 \dots x^k = \langle x_1^1 + \dots + x_1^k + S_1(x^1, \dots, x^k), S_2(x^1, \dots, x^k), \dots, S_{K(\kappa)}(x^1, \dots, x^k) \rangle,$$

where each S_i is a weakly monotonic function in $\mathbb{N}^{K(\iota_1)} \times \dots \times \mathbb{N}^{K(\iota_k)} \longrightarrow \mathbb{N}$. Moreover, a function $S : \mathbb{N}^{K(\iota_1)} \times \dots \times \mathbb{N}^{K(\iota_k)} \longrightarrow \mathbb{N}$ is weakly monotonic if it is built from constants in \mathbb{N} , variable components x_j^n , and weakly monotonic functions in $\mathbb{N}^n \longrightarrow \mathbb{N}$.

For the “weakly monotonic functions in $\mathbb{N}^n \longrightarrow \mathbb{N}^m$ ” we could for instance use $+$, $*$ or \max . We take a semantic approach (cf. [69]) to determine the number $K(\iota)$ for each base type ι . For instance nat is the base type of natural numbers in unary format, so a number $n \in \mathbb{N}$ is represented as the data term $\ulcorner n \urcorner = s^n 0$. With that in mind the number of occurrences of s in such terms is a reasonable measure for their size, so we let $K(\text{nat}) = 2$. Hence, a tuple in A_{nat} has the meaning $\langle \text{cost}, \text{size} \rangle$. A second example is that of list. To characterize the size of a list we may need information about its elements in addition to the length of the list. So we keep track of the length as well as the maximum size of their elements. This way $K(\text{list}) = 3$. A tuple in A_{list} has the meaning $\langle \text{cost}, \text{length of normal form}, \text{max. elem. size} \rangle$. In the example below we interpret the constructors for nat and list following this semantics. In the remainder of this thesis, we will use x_c as syntactic sugar for x_1 (the cost component of x), x_s and x_l as x_2 and x_m as x_3 .

Example 3.3.5. Consider the TRS defined in Example 3.1.1. Let us start by giving an interpretation for the constructor symbols 0 , $[]$, s , and cons . We interpret these symbols below to be consistent with the semantics we just described above, we let:

$$\begin{aligned} \mathcal{J}_0 &= \langle 0, 0 \rangle & \mathcal{J}_s &= \lambda x. \langle x_c, x_s + 1 \rangle \\ \mathcal{J}_{[]} &= \langle 0, 0, 0 \rangle & \mathcal{J}_{\text{cons}} &= \lambda x q. \langle x_c + q_c, q_l + 1, \max(x_s, q_m) \rangle \end{aligned}$$

This expresses that 0 has no evaluation cost and size 0. It is the same for $[]$, as no reduction is fired from constructors; additionally the empty list has zero length and no elements. The cost of evaluating a term $s\ t$ depends entirely on the cost of its argument

t ; the size component counts the number of s 's. The cost component for cons similarly sums the costs of its arguments, while the length is increased by 1, and the maximum element is the maximum between its head and tail.

Note that when interpreting data terms with these interpretations, the cost component is always zero. For instance, for nat terms of the form $\ulcorner n \urcorner$: we get $\llbracket \ulcorner n \urcorner \rrbracket = \langle 0, n \rangle$. This also holds for lists: $\llbracket \ulcorner 2 \urcorner :: \ulcorner 3 \urcorner :: \ulcorner 5 \urcorner :: [] \rrbracket = \langle 0, 3, 5 \rangle$. Different initial interpretations for 0, [] can produce different tuples.

We can answer the question posed earlier positively by using tuple interpretations with the following interpretation for add:

$$\mathcal{J}_{\text{add}} = \lambda xy. \langle x_c + y_c + y_s + 1, x_s + y_s \rangle$$

This interpretation captures the observation that the number of steps needed to reduce a term $\text{add } s \ t$ is the cost of reducing both arguments s and t and depends directly on the size of the normal form of t . This is the case since the “recursive argument” of add is the second argument.

As an example let us interpret the term $s = \text{add } \ulcorner n \urcorner (\text{add } \ulcorner n \urcorner (\text{add } \ulcorner n \urcorner \ulcorner m \urcorner))$, which we considered earlier. We get the following result with tuple interpretations:

$$\llbracket \text{add } \ulcorner n \urcorner (\text{add } \ulcorner n \urcorner (\text{add } \ulcorner n \urcorner \ulcorner m \urcorner)) \rrbracket = \langle 3(m + n + 1), 3n + m \rangle,$$

which provides us with a tight interpretation.

It is worth mentioning however that we can still have larger overheads when providing upper bounds using tuple interpretations. The interesting observation here is that by separating the notions of cost and size we can simplify the cost component. This is not possible using only polynomials, as we have seen earlier.

Example 3.3.6. Let us interpret the rest of the symbols from Example 3.1.1. For the remaining symbols we choose the following interpretations:

$$\begin{aligned} \mathcal{J}_{\text{sum}} &= \lambda q. \langle q_c + 2q_l + q_l q_m + 1, q_l q_m \rangle \\ \mathcal{J}_{\text{rev}} &= \lambda q. \langle q_c + q_l + q_l * (q_l + 1)/2 + 1, q_l, q_m \rangle \\ \mathcal{J}_{\text{append}} &= \lambda qq'. \langle q_c + q'_c + q_l + 1, q_l + q'_l, \max(q_m, q'_m) \rangle \end{aligned}$$

The strong monotonicity of this interpretation follows by Lemma 3.3.4 by observing that the function $n \mapsto n * (n + 1)/2$ in $\mathbb{N} \rightarrow \mathbb{N}$ is weakly monotonic. Checking compatibility is easily done by computing the interpretations for each rule.

We see that the cost of evaluating sum and rev is quadratic on length and size combined while evaluating append is linear in the first list length and independent of the size of the list elements.

Tuple interpretations have some similarities with matrix interpretations [37], where each term is also associated with an n -tuple. However, the shape of the interpretation functions \mathcal{J}_f in matrix interpretations is limited to functions following Lemma 3.3.4 where each S is a linear multivariate polynomial. In fact, the authors in [32] generalize matrix interpretations to a form of linear polynomials with coefficients as quadratic matrices of some fixed dimension d . In this work, the authors allow for a selection set E over the columns of the matrices that can be used for strict orientation. In this work [32] conjecture that such a polynomial approach to matrix interpretations may open the way for non-linear matrix interpretations. We believe our tuple interpretation approach is a simple solution to the problem proposed in [32]. Indeed, tuple interpretations allow for different “dimensions” (one $K(\iota)$ for each sort) and there is no need for such an intricate ordering ($\geq_{\mathbb{N}^{d \times d}}, >_{\mathbb{N}^{d \times d}}^E$). See [32, Section 3.1] for a formal definition of this ordering. Hence, our interpretations are a strict generalization of the matrix interpretations presented in [32, 37].

Example 3.3.7. A TRS that implements division, given by Arts and Giesl in [5], shows a limitation of polynomial interpretations. Let us consider the rules below.

$$\begin{array}{ll}
 \text{minus } x \ 0 \rightarrow x & \text{quot } 0 \ (s \ y) \rightarrow 0 \\
 \text{minus } 0 \ y \rightarrow 0 & \text{quot } (s \ x) \ (s \ y) \rightarrow s \ (\text{quot } (\text{minus } x \ y) \ (s \ y)) \\
 \text{minus } (s \ x) \ (s \ y) \rightarrow \text{minus } x \ y &
 \end{array}$$

The rule $\text{quot}(s(x), s(y)) \rightarrow s(\text{quot}(\text{minus}(x, y), s(y)))$ cannot satisfy compatibility by any polynomial interpretation because $\llbracket \text{minus}(x, s(x)) \rrbracket > \llbracket s(x) \rrbracket$ for any strongly monotonic polynomial $\mathcal{J}_{\text{minus}}$. Due to the duplication of y , this rule also cannot be handled by a matrix interpretation. However, we do have a compatible tuple interpretation:

$$\begin{aligned}
 \mathcal{J}_{\text{minus}} &= \lambda x y. \langle x_c + y_c + y_s + 1, x_s \rangle \\
 \mathcal{J}_{\text{quot}} &= \lambda x y. \langle x_c + x_s + y_c + x_s y_c + x_s y_s + 1, x_s \rangle
 \end{aligned}$$

In practice, for termination analysis, one would not exclusively use interpretations but rather a combination of different techniques. In that context, tuple interpretations may be used as one part of a larger toolbox. Indeed, this is demonstrated by Yamada [109] where the author uses a style of tuple interpretations combined with the dependency pairs termination method. However, developing a new technique for first-order termination is not our goal. As such, we concentrate our efforts on developing tuple interpretations for providing a more fine-grained complexity analysis. For instance, interpretations that may consider information such as the length of a list. Further combination of tuple interpretations with other techniques designed for complexity analysis, however, is a clear path for future investigation.

3.3.2 Runtime Complexity Analysis

Recall from Section 3.1 that the *runtime complexity* of a TRS \mathbb{R} is the function $\text{rc}_{\mathbb{R}}$ that maps each non-zero natural number n to a number $\text{rc}_{\mathbb{R}}(n)$. This number satisfies the property that for every basic term $f d_1 \dots d_k$ of absolute size at most n , $\text{dh}_{\mathbb{R}}(s) \leq \text{rc}_{\mathbb{R}}(n)$. We expressed this function in terms of the complexity function $\text{rc}_{\mathbb{R}}(n) = \text{comp}(n, \rightarrow_{\mathbb{R}}, \text{T}_b(\mathbb{F}))$.

The comparable notion of *derivational complexity* considers the derivation height for arbitrary ground terms of size n , but we will not use that here, since it can often give very high bounds that are not necessarily representative for realistic use of the system. In practice, a computation with a TRS would typically start with a main function, which takes *data* (e.g., natural numbers, lists) as input. This is exactly a basic term. Hence, the notion of runtime complexity roughly captures the worst-case number of steps for a realistic computation.

To derive runtime complexity for a TRS \mathbb{R} , our approach is to consider bounds for the interpretation functions \mathcal{J}_f .

Definition 3.3.8. Consider a type $\iota_1 \Rightarrow \dots \Rightarrow \iota_m \Rightarrow \kappa$ and a strongly monotonic function (in its first argument) f in $A_{\iota_1} \Rightarrow \dots \Rightarrow A_{\iota_m} \Rightarrow A_{\kappa}$. Suppose that f can be written as $\lambda x^1 \dots x^m = \langle f_1(x^1, \dots, x^m), \dots, f_{K(\kappa)}(x^1, \dots, x^m) \rangle$. Then f is

1. **linearly bounded** if each component function f_l of f is upper-bounded by a positive linear polynomial; that is, there is a constant $a \in \mathbb{N}$ such that

$$f_l(x^1, \dots, x^m) \leq a \left(1 + \sum_{i=1}^m \sum_{j=1}^{K(\iota_i)} x_j^i \right)$$

2. **additively bounded** if there exists a constant $a \in \mathbb{N}$ such that

$$\sum_{l=1}^{K(\kappa)} f_l(x^1, \dots, x^m) \leq a + \sum_{i=1}^m \sum_{j=1}^{K(\iota_i)} x_j^i$$

3. **polynomially bounded** if there exists a positive polynomial P that bounds each component f_l of f , i.e., $f_l(x^1, \dots, x^m) \leq P(x^1, \dots, x^m)$.

By this definition, f_l is not required to be a linear function, only to be bounded by one. This means for instance that taking the minimum of two values, e.g., $\min(x_j^i, 2x_b^a)$, is allowed while multiplying two values, e.g., $x_j^i x_b^a$, is not. It is easily checked that all the data constructors in this chapter have an additively bounded interpretation. For example, the interpretation function for `cons` satisfies: $(x_c + q_c) + (x_1 + 1) + \max(x_s, q_m) \leq 1 + x_c + x_s + q_c + q_1 + q_m$.

Next, we prove that bounding the interpretation functions of constructor symbols additively results in interpretations that are proportional to the absolute size of data terms. But going up just by a linear factor induces an exponential overhead related to their absolute size.

We use the following inequality to prove the lemma below. Let $2 \leq x^1, \dots, x^m$, then

$$\sum_{i=1}^m x^i \leq \prod_{i=1}^m x^i \quad (3.1)$$

Lemma 3.3.9. Let \mathbb{R} be a TRS that is compatible with a strongly monotonic algebra with interpretation function \mathcal{J} . Then

1. if \mathcal{J}_c is additively bounded for all data constructors c , then there exists a constant $b > 0$ in \mathbb{N} so that for all data terms s : if $|s| \leq n$ then $\llbracket s \rrbracket_l \leq bn$, for each component $\llbracket s \rrbracket_l$ of $\llbracket s \rrbracket$;
2. if \mathcal{J}_c is linearly bounded for all data constructors c , then there exists a constant $b > 0$ in \mathbb{N} so that for all data terms s : if $|s| \leq n$ then $\llbracket s \rrbracket_l \leq 2^{bn}$, for each component $\llbracket s \rrbracket_l$ of $\llbracket s \rrbracket$.

Proof. 1. Since the interpretation $\mathcal{J}_c = \langle f_1, \dots, f_{K(\kappa)} \rangle$ for each constructor c is additively bounded, by Definition 3.3.8, for each $c \in \Sigma$, there exists a constant a_c such that for all (x^1, \dots, x^m) , $\sum_{l=1}^{K(\kappa)} f_l(x^1, \dots, x^m) \leq a_c + \sum_{i=1}^m \sum_{j=1}^{K(\kappa)} x_j^i$. Let us set a to be the maximum of such a_c , so for the sum of components f_l of \mathcal{J}_c we have:

$$\sum_{l=1}^{K(\kappa)} f_l(x^1, \dots, x^m) \leq a + \sum_{i=1}^m \sum_{j=1}^{K(\kappa)} x_j^i. \quad (3.2)$$

Now, we reason by induction on the size of $s : \kappa$ that $\sum_{l=1}^{K(\kappa)} \llbracket s \rrbracket_l \leq a|s|$. Then certainly $\llbracket s \rrbracket_l \leq a|s|$ holds for any component $\llbracket s \rrbracket_l$, and the first part of the lemma holds.

For the base case we have $|s| = 1$ so s is a constant symbol c and $\sum_{l=1}^{K(\iota)} \llbracket c \rrbracket_l \leq a_c \leq a$, by (Equation (3.2)).

For the inductive case, let $|s| > 1$. Then $s = \mathbf{c} d_1 \dots d_m$ and using Equation (3.2) above, we can expand the summation as follows:

$$\begin{aligned}
\sum_{l=1}^{K(\kappa)} \llbracket \mathbf{c} d_1 \dots d_m \rrbracket_l &= \sum_{l=1}^{K(\kappa)} f_l(\llbracket d_1 \rrbracket, \dots, \llbracket d_m \rrbracket) \\
&\stackrel{(3.2)}{\leq} a + \sum_{i=1}^m \sum_{j=1}^{K(\iota_i)} \llbracket d_i \rrbracket_j \\
&\stackrel{(IH)}{\leq} a + \sum_{i=1}^m a |d_i| \\
&= a \left(1 + \sum_{i=1}^m |d_i| \right) \\
&= a |s|.
\end{aligned}$$

Hence, we are done choosing $b := a$.

2. The proof follows the same structure as before: by Definition 3.3.8, each $\mathcal{J}_{\mathbf{c}} = \langle P_1, \dots, P_{K(\kappa)} \rangle$ is now linearly bounded; that is, for each $\mathbf{c} \in \Sigma$, there exists a constant $a_{\mathbf{c}}$ such that for all (x^1, \dots, x^m) we have $P_l(x^1, \dots, x^m) \leq a_{\mathbf{c}} (1 + \sum_{i=1}^m \sum_{j=1}^{K(\iota_i)} x_j^i)$. Let us set, as before, a to be the maximum of such $a_{\mathbf{c}}$ and define $k = \max(2, \max_i K(\iota_i))$. Notice that k is determined when we define the interpretation's domain, so it does not depend on the size of s .

We prove by induction on the size of s that $\llbracket s \rrbracket_l \leq 2^{(ak)|s|}$, for each component P_l of $\llbracket s \rrbracket$. In the base case, where s is a constant constructor, we have trivially that

$\llbracket \mathbf{c} \rrbracket_l \leq a_{\mathbf{c}} \leq ak < 2^{ak}$ follows. For the inductive step we have $s = \mathbf{c} d_1 \dots d_m$. Then:

$$\begin{aligned}
f_l(\llbracket d_1 \rrbracket, \dots, \llbracket d_m \rrbracket) &\leq a_{\mathbf{c}} \left(1 + \sum_{i=1}^m \sum_{j=1}^{K(\iota_i)} \llbracket d_i \rrbracket_j \right) \\
&\leq a \left(2 \max \left(\sum_{i=1}^m \sum_{j=1}^{K(\iota_i)} \llbracket d_i \rrbracket_j, 1 \right) \right) \\
&\stackrel{(IH)}{\leq} 2a \sum_{i=1}^m \left(\sum_{j=1}^{K(\iota_i)} 2^{ak|d_i|} \right) \\
&\leq 2ak \sum_{i=1}^m 2^{ak|d_i|} \\
&\leq (2ak) \prod_{i=1}^m 2^{ak|d_i|} \text{ by claim (3.1)} \\
&\leq 2^{ak} \prod_{i=1}^m \left(2^{ak|d_i|} \right) \\
&= 2^{(ak)} 2^{(ak) \sum_{i=1}^m |d_i|} \\
&= 2^{ak \left(1 + \sum_{i=1}^m |d_i| \right)} \\
&= 2^{(ak)|s|}.
\end{aligned}$$

Hence, we are done choosing $b := ak$. □

By using Lemma 3.3.9, we quickly obtain some ways to bound runtime complexity:

Lemma 3.3.10. Let \mathbb{R} be a TRS that is compatible with a strongly monotonic tuple algebra \mathcal{F} . Then:

1. if \mathcal{J}_f is additively bounded for all $f \in \Sigma$, then \mathbb{R} has linear runtime complexity;
2. if \mathcal{J}_c is additively bounded for all constructors c and for all defined symbols f we have that $\mathcal{J}_f(\vec{x}) = (f_1(\vec{x}), \dots, f_k(\vec{x}))$ where f_1 is bounded by a polynomial, then \mathbb{R} has polynomial runtime complexity;
3. if \mathcal{J}_f is linearly bounded for all $f \in \Sigma$, then \mathbb{R} has exponential runtime complexity.

Proof. Let us consider $f: \iota_1 \Rightarrow \dots \Rightarrow \iota_m \Rightarrow \kappa$ and a basic term $s = f d_1 \dots d_m$ of type κ . In the first case, since \mathcal{J}_f is additively bounded we get the following by Definition 3.3.8

$$\llbracket s \rrbracket_1 \leq \sum_{l=1}^{K(\kappa)} f_l(\llbracket d_1 \rrbracket, \dots, \llbracket d_m \rrbracket) \leq a + \sum_{i=1}^m \sum_{j=1}^{K(\iota_i)} \llbracket d_i \rrbracket_j \leq a|s|,$$

where the last inequality follows directly from Lemma 3.3.9. The other cases follow similar reasoning. \square

Notice that we can give a stronger statement for (1) above. Indeed, it remains valid even for derivational complexity analysis. Such a strong requirement like additivity for all function symbols in Σ is hard to obtain, however.

3.4 Cost–Size Products

Thus far, we have used interpretations of the form $\mathbb{N}^{K(\iota)}$. But we can view this in a more general light: a single number identifying *cost*, and an element of $\mathbb{N}^{K(\iota)-1}$ identifying the various notions of size for the term. Hence, we alternatively define:

Definition 3.4.1. Given a well-founded set $(C, >, \succeq)$, called the **cost set**, and a quasi-ordered set (S, \sqsupseteq) , called the **size set**, we call $C \times S$ the **cost–size product** of $(C, >, \succeq)$ and (S, \sqsupseteq) and its elements *cost–size tuples*.

By this definition, the very minimum we need to capture the notion of *cost* in our rewriting setting is a well-founded set. Similarly, for size sets in which we impose a quasi-ordering structure. Given a cost–size product $C \times S$, the well-foundedness of C and quasi-ordering on S naturally induce an ordering structure on the cartesian product $C \times S$ as follows.

Definition 3.4.2. Let $(C, >, \succeq) \times (S, \sqsupseteq)$ be a cost–size product. Then we define the relations $>, \succcurlyeq$ over $C \times S$ as follows: for all $\langle x, y \rangle$ and $\langle x', y' \rangle$ in $C \times S$,

- (i) $\langle x, y \rangle > \langle x', y' \rangle$ if $x > x'$ and $y \sqsupseteq y'$, and
- (ii) $\langle x, y \rangle \succcurlyeq \langle x', y' \rangle$ if $x \succeq x'$ and $y \sqsupseteq y'$.

Next, we show that cost–size products ordered as above form a well-founded set.

Lemma 3.4.3. The triple $(C \times S, >, \succcurlyeq)$ is a well-founded set.

Proof. It follows immediately from Definition 3.4.1 that $>, \succcurlyeq$ are transitive and \succcurlyeq is reflexive. To prove that $>$ is well-founded, note that the existence of $\langle x_1, y_1 \rangle > \langle x_2, y_2 \rangle > \dots$ would imply $x_1 > x_2 > \dots$ which cannot be the case since $>$ is well-founded. We still need to check that \succcurlyeq is compatible with $>$.

- Suppose $\langle x, y \rangle > \langle x', y' \rangle$. Since $x > x'$ implies $x \succeq x'$, we have $\langle x, y \rangle \succcurlyeq \langle x', y' \rangle$.
- Suppose $\langle x, y \rangle > \langle x', y' \rangle \succcurlyeq \langle x'', y'' \rangle$. Since $x > x' \succeq x''$ implies $x > x''$ and \sqsupseteq is transitive, we have $\langle x, y \rangle > \langle x'', y'' \rangle$.

□

In this chapter, we always use C as \mathbb{N} . When dealing with higher-order rewriting in Chapter 5, we take different cost sets. Notice that Definition 3.3.1 can be seen as a cost–size product for strongly monotonic algebras. Indeed, the interpretation function \mathcal{J}_f can be seen as a pair $\langle \mathcal{J}_f^c, \mathcal{J}_f^s \rangle$ so that

$$\llbracket f s_1 \dots s_m \rrbracket = \langle \mathcal{J}_f^c(\llbracket s_1 \rrbracket, \dots, \llbracket s_m \rrbracket), \mathcal{J}_f^s(\llbracket s_1 \rrbracket, \dots, \llbracket s_m \rrbracket) \rangle$$

In the next section, we define a variation of tuple interpretations for *innermost* rewriting where both \mathcal{J}_f^c and \mathcal{J}_f^s only depend on the *size* component of their arguments.

3.5 Tuple Interpretations for Innermost Rewriting

Our first task is to formally interpret base types as a particular kind of cost–size products.

Definition 3.5.1. Let \mathbb{B} be a set of base types. An **interpretation key** $\mathcal{J}_{\mathbb{B}}$ for \mathbb{B} maps each type ι to a quasi-ordered set $(\mathcal{J}_{\mathbb{B}}(\iota), \sqsupseteq_{\iota})$. We let $\langle \iota \rangle$ be the cost–size product $(\mathbb{N}, >, \succeq) \times (\mathcal{J}_{\mathbb{B}}(\iota), \sqsupseteq_{\iota})$.

The intention is that a term of base type ι will be mapped to a pair $\langle c, s \rangle$ with $s \in \mathcal{J}_{\mathbb{B}}(\iota)$. To do this, we need to assign an interpretation to all function symbols in the same fashion of Definition 3.3.1.

3.5.1 Cost–Size Tuple Algebras

An interpretation of a signature $(\mathbb{B}, \Sigma, \text{typeOf})$ interprets the types in $\mathbb{T}(\mathbb{B})$ and each $f : \sigma \in \Sigma$ to an element of $\langle \sigma \rangle$. This is formally stated in the definition below.

Definition 3.5.2. A **cost–size tuple algebra** \mathcal{F} for a signature $\mathbb{F} = (\mathbb{B}, \Sigma, \text{typeOf})$ consists of a pair of functions $(\mathcal{J}_{\mathbb{B}}, \mathcal{J}_{\Sigma})$ where

1. $\mathcal{J}_{\mathbb{B}}$ is a type interpretation key,
2. \mathcal{J}_{Σ} is an *interpretation of symbols* in Σ which maps each $f \in \Sigma$ with $\text{typeOf}(f) = \iota_1 \Rightarrow \dots \Rightarrow \iota_m \Rightarrow \kappa$ to a pair of weakly monotonic functions

$$\begin{aligned} \mathcal{J}_{\Sigma}(f)^c : \mathcal{J}_{\mathbb{B}}(\iota_1) &\Longrightarrow \dots \Longrightarrow \mathcal{J}_{\mathbb{B}}(\iota_m) \Longrightarrow \mathbb{N} \\ \mathcal{J}_{\Sigma}(f)^s : \mathcal{J}_{\mathbb{B}}(\iota_1) &\Longrightarrow \dots \Longrightarrow \mathcal{J}_{\mathbb{B}}(\iota_m) \Longrightarrow \mathcal{J}_{\mathbb{B}}(\kappa) \end{aligned}$$

A cost–size tuple algebra is nothing more than a weakly monotonic \mathbb{F} -algebra for \mathbb{F} such that the interpretation key $\mathcal{J}_{\mathbb{B}}$ completely defines its carrier. In what follows we slightly abuse notation by writing \mathcal{J}_f for $\mathcal{J}_{\Sigma}(f)$ and just \mathcal{J} for \mathcal{J}_{Σ} .

Example 3.5.3. Let $\mathcal{J}_{\mathbb{B}}(\text{nat}) = \mathbb{N}$ and $\mathcal{J}_{\mathbb{B}}(\text{list}) = \mathbb{N}^2$ be respectively. Recall from Example 3.3.5 that the size of a natural number is the number of occurrences of s , and the size of a list is a pair $q = (q_l, q_m)$ where q_l is the length and q_m is the maximum size of the elements. We interpret the constructors as follows:

$$\begin{aligned} \mathcal{J}_0 &= \langle 0, 0 \rangle & \mathcal{J}_s &= \langle \lambda x.0, \lambda x.x + 1 \rangle \\ \mathcal{J}_{[]} &= \langle 0, (0, 0) \rangle & \mathcal{J}_{\text{cons}} &= \langle \lambda xq.0, \lambda xq.(q_l + 1, \max(x, q_m)) \rangle \end{aligned}$$

In Example 3.3.5 we interpreted for instance $\mathcal{J}_s = \lambda x.\langle x_c, x_s + 1 \rangle$. There are two major differences between them. The first is that here the cost and size functions for \mathcal{J}_s are completely split. The second is that since innermost rewriting only allows contraction of redexes in normal form, the cost component here is $\lambda x.0$.

Albeit different in shape and form the interpretation of data terms in both full and innermost strategies encode the same information. Indeed, as in Example 3.3.5 by interpreting $\ulcorner n \urcorner : \text{nat}$ we get $\llbracket \ulcorner n \urcorner \rrbracket = \langle 0, n \rangle$, which we can write as $\langle 0, n \rangle$. Analogously, $\llbracket \ulcorner 2 \urcorner :: \ulcorner 3 \urcorner :: \ulcorner 5 \urcorner :: [] \rrbracket = \langle 0, (3, 5) \rangle$, which we can write as $\langle 0, 3, 5 \rangle$.

We extend the notion of interpretation to terms, where we use a valuation to map variables of type ι to elements of (ι) . With innermost rewriting we assume that variables have no cost.

Definition 3.5.4. Fix a cost–size tuple algebra \mathcal{F} . A **zero-cost valuation** $\alpha : \mathbb{X} \rightarrow \mathcal{T}$ is a function which maps each variable $x : \iota$ to a zero-cost tuple $\langle 0, x^s \rangle \in (\iota)$. The **interpretation** of a term s under the valuation α , denoted by $\llbracket s \rrbracket_{\alpha}^{\mathcal{J}}$, is defined as follows:

$$\begin{aligned} \llbracket x \rrbracket_{\alpha}^{\mathcal{J}} &= \alpha(x) \\ \llbracket f s_1 \cdots s_m \rrbracket_{\alpha}^{\mathcal{J}} &= \langle \mathcal{J}_f^c(k_1, \dots, k_m) + c_1 + \cdots + c_m, \mathcal{J}_f^s(k_1, \dots, k_m) \rangle \end{aligned}$$

where $\llbracket s_i \rrbracket_{\alpha}^{\mathcal{J}} = (c_i, k_i)$ for all $1 \leq i \leq m$.

We write $\llbracket s \rrbracket$ instead of $\llbracket s \rrbracket_{\alpha}^{\mathcal{J}}$ whenever α and \mathcal{J} are universally quantified or clear from the context. In both cases we may write $\llbracket x \rrbracket = x$ instead of $\llbracket x \rrbracket_{\alpha}^{\mathcal{J}} = \alpha(x)$.

We collect in the lemma below the simple observation that our interpretation notion conforms with typing.

Lemma 3.5.5. If $s : \sigma$ then $\llbracket s \rrbracket \in (\sigma)$.

Remark 3.5.6. In Definition 3.5.4 we require that valuations interpret variables as zero-cost tuples. This is an important but subtle requirement that only works when reductions are innermost. Indeed, if reduction is unrestricted we can instantiate variables on the left-hand side of rules to terms containing redexes for which the cost should be accounted. Hence, not accounting for the cost of variables in full rewriting would lead to unsound analysis. Additionally, zero-cost tuples allow us to prove the innermost termination of the TRS \mathbb{R} in Example 3.5.14, which is non-terminating in full rewriting.

3.5.2 Innermost Compatibility Theorem

Roughly, the compatibility theorem (Theorem 3.5.12) states that if \mathbb{R} is compatible with a tuple algebra \mathcal{F} , then the innermost rewrite relation $\rightarrow_{\mathbb{R}}^i$ is embedded in the well-founded order on cost–size products. We need two additional technical results in order to prove it for the innermost case. Lemma 3.5.8 states that interpretations are closed under substitution and Lemma 3.5.10 allows us to safely assume that normal forms have cost 0.

Definition 3.5.7. Fix a cost–size tuple algebra \mathcal{F} . A substitution γ is **zero-cost** under valuation α if $\llbracket \gamma(x) \rrbracket_{\alpha}^{\mathcal{F}}$ is a zero-cost tuple for each variable x .

Given a valuation α and a zero-cost substitution γ , the function $\alpha^{\gamma} = \llbracket \cdot \rrbracket_{\alpha}^{\mathcal{F}} \circ \gamma = \llbracket \gamma(\cdot) \rrbracket_{\alpha}^{\mathcal{F}}$ is thus a valuation.

Lemma 3.5.8 (Substitution Lemma). If γ is a zero-cost substitution under valuation α , $\llbracket s\gamma \rrbracket_{\alpha}^{\mathcal{F}} = \llbracket s \rrbracket_{\alpha^{\gamma}}^{\mathcal{F}}$ for any term s .

Unfortunately, a substitution that maps each variable to a normal form is not guaranteed to be a zero-cost substitution: this is the case when $\gamma(x)$ is mapped to a data term, but not necessarily true if it is mapped to an irreducible term $f s_1 \cdots s_m$ with f a defined symbol. Hence, we consider a modification of terms where symbols in normal forms are *marked*:

Definition 3.5.9. For every function symbol $f \in \Sigma$ we introduce a fresh function symbol f_{fn} with the same type. Then we define:

$$\text{mark}(f s_1 \cdots s_m) = \begin{cases} f_{\text{fn}} \text{mark}(s_1) \cdots \text{mark}(s_m) & \text{if } f s_1 \cdots s_m \text{ is irreducible} \\ f \text{mark}(s_1) \cdots \text{mark}(s_m) & \text{otherwise} \end{cases}$$

We extend the interpretation function to marked symbols so that $\mathcal{J}_{f_{\text{fn}}} = \langle \lambda \vec{x}.0, \mathcal{J}_f^s \rangle$. Then we have:

Lemma 3.5.10. For all terms s :

1. if $\llbracket s \rrbracket = (c, s)$ then $\llbracket \text{mark}(s) \rrbracket = (c', s)$ with $c \geq c'$; if s is irreducible then $c' = 0$.
2. if $s\gamma$ is in normal form, then $\text{mark}(s\gamma) = \text{mark}(s)\gamma^{\text{mark}}$, where γ^{mark} is the substitution that maps x to $\text{mark}(s\gamma)$.
3. $\llbracket s\gamma^{\text{mark}} \rrbracket \succcurlyeq \llbracket \text{mark}(s\gamma) \rrbracket$, for all substitutions γ .

With this in hand, we do not have exactly compatibility like in Theorem 3.2.4.

Definition 3.5.11. A TRS \mathbb{R} is said to be **innermost compatible** with a cost-size tuple algebra \mathcal{F} if $\llbracket \text{mark}(\ell) \rrbracket_{\alpha}^{\mathcal{F}} > \llbracket r \rrbracket_{\alpha}^{\mathcal{F}}$ for all rules $\ell \rightarrow r \in \mathbb{R}$ and zero-cost valuations α .

Notice that, for a rule $\ell = f \ell_1 \cdots \ell_k \rightarrow r$ typically each ℓ_i would be in normal form (as otherwise this rule can never be fired), and of course ℓ itself is not in normal form. Hence, $\llbracket \text{mark}(\ell) \rrbracket = \mathcal{J}_f^c(\pi_2(\llbracket \ell_1 \rrbracket), \dots, \pi_2(\llbracket \ell_k \rrbracket))$.

Theorem 3.5.12 (Compatibility). Let \mathbb{R} be a TRS be innermost compatible with a cost-size tuple algebra \mathcal{F} . Then, for any pair of terms s and t , whenever $s \rightarrow_{\mathbb{R}}^i t$ we have $\llbracket \text{mark}(s) \rrbracket_{\alpha}^{\mathcal{F}} > \llbracket \text{mark}(t) \rrbracket_{\alpha}^{\mathcal{F}}$.

Proof. We proceed by induction on $\rightarrow_{\mathbb{R}}^i$. For the base case, $s \rightarrow_{\mathbb{R}}^i t$ by $\ell\gamma \rightarrow r\gamma$ and all subterms of $\ell\gamma$ are in $\rightarrow_{\mathbb{R}}$ normal form. Then $\llbracket \text{mark}(s) \rrbracket = \llbracket \text{mark}(\ell\gamma) \rrbracket = \llbracket \text{mark}(\ell)\gamma^{\text{mark}} \rrbracket$ by Lemma 3.5.10, Item 2, using that the immediate subterms of $\ell\gamma$ are in normal form, and the root of ℓ and $\ell\gamma$ are themselves not. Since $x\gamma$ is in normal form for all $x \in \text{vars}(s)$, by Lemma 3.5.10 Item 1 we get that γ^{mark} is a zero-cost substitution. Therefore, by Lemma 3.5.8, $\llbracket \text{mark}(\ell)\gamma^{\text{mark}} \rrbracket = \llbracket \text{mark}(\ell) \rrbracket_{\alpha\gamma^{\text{mark}}} > \llbracket r \rrbracket_{\alpha\gamma^{\text{mark}}} = \llbracket r\gamma^{\text{mark}} \rrbracket \succcurlyeq \llbracket \text{mark}(r\gamma) \rrbracket = \llbracket t \rrbracket$, where the last inequality follows from Lemma 3.5.10, Item 3.

In the inductive step, we use the monotonicity of \mathcal{J}_{Σ} combined with the (IH) as follows. Suppose $s \rightarrow t$ by $s = f s_1 \cdots s_j \cdots s_m$ and $t = f s_1 \cdots s'_j \cdots s_m$ with $s_j \rightarrow s'_j$. As s is not in normal form, $\text{mark}(s) = f \text{mark}(s_1) \cdots \text{mark}(s_m)$. Let $\llbracket s_1 \rrbracket = (c_1, k_1), \dots, \llbracket s_m \rrbracket = (c_m, k_m)$. By the (IH), we get $c_j > c'_j$ and $k_j \succeq k'_j$. By weak monotonicity of \mathcal{J}_f^c and \mathcal{J}_f^s , we have both $\mathcal{J}_f^c(k_1, \dots, k_j, \dots, k_m) \succeq \mathcal{J}_f^c(k_1, \dots, k'_j, \dots, k_m)$ and $\mathcal{J}_f^s(k_1, \dots, k_j, \dots, k_m) \sqsupseteq \mathcal{J}_f^s(k_1, \dots, k'_j, \dots, k_m)$. Thus,

$$\begin{aligned} \llbracket \text{mark}(s) \rrbracket &= \langle \mathcal{J}_f^c(k_1, \dots, k_j, \dots, k_m) + c_1 \cdots + c_j + \cdots + c_m, \mathcal{J}_f^s(k_1, \dots, k_j, \dots, k_m) \rangle \\ &> \langle \mathcal{J}_f^c(k_1, \dots, k'_j, \dots, k_m) + c_1 \cdots + c'_j + \cdots + c_m, \mathcal{J}_f^s(k_1, \dots, k'_j, \dots, k_m) \rangle \end{aligned}$$

If t is not in normal form this is exactly $\llbracket \text{mark}(t) \rrbracket$. In the case t is not in normal form, note that $\mathcal{J}_f^s = \mathcal{J}_{f_{\text{fn}}}^s$, so the inequality above continues as follows:

$$\begin{aligned} \llbracket \text{mark}(s) \rrbracket &= \langle \mathcal{J}_f^c(k_1, \dots, k_j, \dots, k_m) + c_1 \cdots + c_j + \cdots + c_m, \mathcal{J}_f^s(k_1, \dots, k_j, \dots, k_m) \rangle \\ &> \langle \mathcal{J}_f^c(k_1, \dots, k'_j, \dots, k_m) + c_1 \cdots + c'_j + \cdots + c_m, \mathcal{J}_f^s(k_1, \dots, k'_j, \dots, k_m) \rangle \\ &\succcurlyeq \langle 0 + c_1 \cdots + c'_j + \cdots + c_m, \mathcal{J}_f^s(k_1, \dots, k'_j, \dots, k_m) \rangle \end{aligned}$$

□

Corollary 3.5.13. $\text{dh}_{\mathbb{R}}(s, \rightarrow_{\mathbb{R}}^i) \leq \pi_1(\llbracket s \rrbracket)$.

Example 3.5.14. Let $a, b : \iota$, $g : \iota \Rightarrow \iota \Rightarrow \iota$, and $f : \iota \Rightarrow \iota \Rightarrow \iota \Rightarrow \iota$. The rewrite system introduced in [102] and defined by $\mathbb{R} = \{g x y \rightarrow x, g x y \rightarrow y, f a b z \rightarrow f z z z\}$ was given to show that termination is not modular for disjoint unions of TRSs. Indeed, it admits the infinite rewriting sequence $f a b (g a b) \rightarrow_{\mathbb{R}} f (g a b) (g a b) (g a b) \rightarrow_{\mathbb{R}}^+ f a b (g a b)$. However, the innermost relation $\rightarrow_{\mathbb{R}}^i$ is terminating. In order to prove it, we introduce a non-numeric notion of size. Let $\mathcal{J}_{\mathbb{B}}(\iota) = \mathcal{P}(\mathcal{T}(\mathbb{F}, \mathbb{X}))$, i.e., the set of all subsets of $\mathcal{T}(\mathbb{F}, \mathbb{X})$. This set is partially ordered by set inclusion, so $x \sqsupseteq y$ iff $x \supseteq y$, which is a quasi-order. Consider the following interpretation:

$$\begin{aligned} \mathcal{J}_a &= \langle 0, \{a\} \rangle & \mathcal{J}_b &= \langle 0, \{b\} \rangle \\ \mathcal{J}_g &= \langle \lambda x y. 1, \lambda x y. (x \cup y) \rangle & \mathcal{J}_f &= \langle \lambda x y z. H(x, y), \lambda x y z. \emptyset \rangle \end{aligned}$$

where H is a helper function defined by $H(x, y) = \text{if } x \sqsupseteq \{a\} \wedge y \sqsupseteq \{b\} \text{ then } 1 \text{ else } 0$. Notice that H is weakly monotonic and all terms in normal form are interpreted as sets of size ≤ 1 . Checking compatibility is straightforward: $\llbracket g x y \rrbracket = \langle 1, x \cup y \rangle > \langle 0, x \rangle = \llbracket x \rrbracket$ and $\llbracket g x y \rrbracket = \langle 1, x \cup y \rangle > \langle 0, y \rangle = \llbracket y \rrbracket$; and $\llbracket f a b z \rrbracket = \langle 1, \emptyset \rangle > \langle 0, \emptyset \rangle = \llbracket f z z z \rrbracket$, because any instantiation of z is necessarily in normal form, so its interpretation cannot include both a and b .

This example, albeit artificial, is interesting from a termination point of view. It shows that tuple interpretations are weak enough to be used to deal with rewrite systems that terminate via the innermost strategy but not via the full strategy. We cannot interpret this system using the strongly monotonic tuple interpretations from the last section as this system is non-terminating for full rewriting.

3.6 Upper Bounds for Innermost Runtime Complexity

In this section, we study the applications of tuple interpretations to innermost complexity analysis of compatible TRSs, i.e., rewriting systems that admit an interpretation in an innermost tuple algebra $(\langle \cdot \rangle, \mathcal{J})$ where the interpretation key is chosen over \mathbb{N}^k , for some $k \geq 1$.

Definition 3.6.1. We say an interpretation \mathcal{J} is **additive** if for each $c \in \Sigma^{\text{con}}$, its size interpretation \mathcal{J}_c^s is additively bounded.

Definition 3.6.2. Let $\mathcal{J}_f = \langle \mathcal{J}_f^c, \mathcal{J}_f^s \rangle$ be the cost–size interpretation of f . We say the cost interpretation of f is linearly (additively) bounded whenever \mathcal{J}_f^c is linearly (additively) bounded. Also, \mathcal{J}_f is bounded by a functional f if both \mathcal{J}_f^c and \mathcal{J}_f^s are bounded by f .

In the next lemma, we collect the appropriate induced upper-bounds on innermost runtime complexity given that we can provide bounds to the cost–size components of interpretations. This is the innermost analogue of Lemma 3.3.10.

Lemma 3.6.3. Suppose \mathbb{R} is a TRS compatible with a cost–size tuple algebra \mathcal{F} , then:

- (i) if, for all $f \in \Sigma$, \mathcal{J}_f^s is logarithmically and \mathcal{J}_f^c is additively bounded, then $\text{irc}_{\mathbb{R}}(n) = O(\log n)$;
- (ii) if, for all $f \in \Sigma$, \mathcal{J}_f is additively bounded, then $\text{irc}_{\mathbb{R}}(n) = O(n)$; and
- (iii) if, for all defined symbols f and constructors c , \mathcal{J}_c is additively and \mathcal{J}_f is polynomially bounded, then $\text{irc}_{\mathbb{R}}(n) = O(n^k)$, for some $k \in \mathbb{N}$.

Example 3.6.4. Let us illustrate this behavior by interpreting functions from Example 3.1.1. The interpretation for the constructors was given in Example 3.5.3.

$$\begin{aligned}
\mathcal{J}_{\text{add}} &= \langle \lambda xy. y + 1, \lambda xy. x + y \rangle \\
\mathcal{J}_{\text{sum}} &= \langle \lambda q. 2q_l + q_l q_m, \lambda q. q_l q_m \rangle \\
\mathcal{J}_{\text{minus}} &= \langle \lambda xy. y + 1, \lambda xy. x \rangle \\
\mathcal{J}_{\text{rev}} &= \langle \lambda q. q_l + q_l(q_l + 1)/2 + 1, \lambda q. q \rangle \\
\mathcal{J}_{\text{quot}} &= \langle \lambda xy. x + xy + 1, \lambda xy. x \rangle \\
\mathcal{J}_{\text{append}} &= \langle \lambda q q'. q_l + 1, \lambda q q'. \langle q_l + q'_l, \max(q_m, q'_m) \rangle \rangle
\end{aligned}$$

Checking the compatibility of this interpretation is straightforward. Notice that in the interpretation of data constructors for Example 3.1.1 the size components are additively bounded. Hence, by case (ii) of Lemma 3.6.3, we have that $\text{irc}_{\mathbb{R}_{\text{add}}}$, $\text{irc}_{\mathbb{R}_{\text{append}}}$, and $\text{irc}_{\mathbb{R}_{\text{minus}}}$ are linear. Quadratic bounds can be derived to $\text{irc}_{\mathbb{R}_{\text{quot}}}$, $\text{irc}_{\mathbb{R}_{\text{sum}}}$, and $\text{irc}_{\mathbb{R}_{\text{rev}}}$.

Notice that the cost component of interpretations does not only bound the innermost runtime complexity of \mathbb{R}_f but also provides additional information on the role each size component plays in the rewriting cost. For instance: the cost of adding two numbers depends solely on the size of `add`'s second argument; the cost of summing every element of a list has a linear dependency on its length and non-linear dependency on its length and maximum element. This is particularly useful in program analysis since one can detect a possible costly operation by analyzing the shape of interpretations themselves.

3.7 Automation

In this section, we propose a procedure that implements strategies for finding cost–size tuple interpretations. Then, we discuss the concrete implementation of such procedure, Hermes. Our goal with the procedure is to find interpretations that guarantee polynomial upper bounds to the innermost runtime complexity of the rewriting system at hand.

We exploit the theoretical results just proved, especially Lemma 3.6.3. Consequently, we posit the following conditions:

1. the interpretation key chosen is over \mathbb{N}^k with $k \geq 1$,
2. the size interpretation of constructor symbols is additively bounded, and
3. the interpretation of defined symbols is polynomially bounded.

3.7.1 Parametric Tuple Interpretations

Notice that the class of functions from which we can choose interpretations (weakly monotonic functions) is exorbitantly large. Hence, any practical search procedure operating in this class needs to be restricted. So we narrow down our search space to a limited class of polynomially bounded functions: max-polynomials, i.e., functions that combine polynomial terms and the max function. Examples of that are the interpretations of cons in Example 3.5.3 and append in Example 3.6.4.

Even though the class of max-polynomials is considerably smaller than the class of all weakly monotonic functions, searching for polynomials is hardly a trivial endeavor. To tackle this problem we follow the approach proposed in [30] which considers *parametric polynomial shapes*. Those are polynomial expressions for which their coefficients are parameters to be determined. We then choose generic max-polynomials for the cost and size components which are parametrized by their coefficients. Recall that we wish to find interpretations that satisfy the compatibility condition given by Theorem 3.5.12: for all rules $\ell \rightarrow r$ in \mathbb{R} , $\llbracket \ell \rrbracket_\alpha > \llbracket r \rrbracket_\alpha$ for any α .

Example 3.7.1. Let us illustrate the ideas above with a simple system defining the function `dbl` over natural numbers. So we consider the TRS with rules `dbl 0` \rightarrow `0` and `dbl (s x)` \rightarrow `s (s (dbl x))`. Let us choose the following parametric interpretation

$$\mathcal{J}_0 = \langle 0, a_0 \rangle \quad \mathcal{J}_s = \langle \lambda x.0, \lambda x.x + b_0 \rangle \quad \mathcal{J}_{\text{dbl}} = \langle \lambda x.c_1x + c_0, \lambda x.d_1x + d_0 \rangle,$$

which satisfy conditions (i)-(iii) above. This interpretation is *parametric* in the sense that the coefficients $a_0, b_0, c_0, c_1, d_0, d_1$ are arbitrary non-negative integers that are yet to be determined.

The compatibility condition for the first rule requires that $\llbracket \text{dbl } 0 \rrbracket > \llbracket 0 \rrbracket$, so we interpret such rules utilizing the parametric interpretation chosen above. This gives us

$$\llbracket \text{dbl } 0 \rrbracket = \langle c_1a_0 + c_0, d_1a_0 + d_0 \rangle > \langle 0, a_0 \rangle = \llbracket 0 \rrbracket,$$

which in consequence requires the validity of $C_0 = (c_1a_0 + c_0 > 0) \wedge (d_1a_0 + d_0 \geq a_0)$. Analogously, the compatibility condition for the second rule gives us $\llbracket \text{dbl (s x)} \rrbracket >$

$\llbracket s(s(\text{dbl } x)) \rrbracket$. Then we get that

$$\langle c_1x + c_1b_0 + c_0, d_1x + d_1b_0 + d_0 \rangle > \langle c_1x + c_0, d_1x + d_0 + 2b_0 \rangle,$$

must hold as well.

Therefore, to validate this parametric interpretation shape is to find values for the undeterminate coefficients $a_0, b_0, c_0, c_1, d_0, d_1$ which witness the validity of the formula

$$C_1 = (c_1x + c_1b_0 + c_0 > c_1x + c_0) \wedge (d_1x + d_1b_0 + d_0 \geq d_1x + d_0 + 2b_0).$$

When solving this formula, we also require that each one of the coefficients is non-negative. Whenever those values are found, the interpretation shape is validated. This is true since we use the absolute positiveness of polynomials over \mathbb{N} .

The example above is simple in nature but uses the main ideas of our procedure. Essentially, we choose parametric interpretations for function symbols in Σ and solve the constraints over the undeterminate coefficients that arise from the compatibility condition. As we have seen in Example 3.6.4, cost-size interpretations may become complicated, so more interpretation shapes are needed for a practical search procedure.

The procedure we give below implements an automation procedure for finding interpretations. It receives two functional parameters. The first is a *selector algorithm* which is responsible for selecting interpretation shapes for function symbols. The second is a constraint solver that is capable of solving non-linear arithmetic formulas. In practice, the first is implemented as a set of heuristics selecting the best possible shape for certain symbols and the second is an external SMT solver.

Parametric Search Procedure

Parameter: A selector algorithm \mathcal{S} and a constraint solver over non-linear integer arithmetic.

Data Input: A signature $\mathbb{F} = (\mathbb{B}, \Sigma, \text{typeOf})$, a rewrite system (\mathbb{F}, \mathbb{R}) , and a natural number $K \geq 1$.

Output: YES, if a cost–size tuple interpretation satisfying compatibility can be found and MAYBE, if all steps below were executed and no interpretation could be found².

1. Split Σ into two disjoint sets of constructors and defined symbols, i.e., $\Sigma = \Sigma^{\text{con}} \uplus \Sigma^{\text{def}}$.
2. Split Σ^{def} into sets $\Sigma_1^{\text{def}}, \dots, \Sigma_n^{\text{def}}$ such that for each $f \in \Sigma_i^{\text{def}}$, with $1 \leq i \leq n$, all function symbols occurring in the rules defining f are either constructors or in $\Sigma_1^{\text{def}} \cup \dots \cup \Sigma_i^{\text{def}}$.
3. We start with $K(\iota) = 1$, for all base types ι , and whenever $K(\iota) > K$ we stop and return MAYBE.
4. Set $\mathcal{J}_{\mathbb{B}}(\iota) = \mathbb{N}^{K(\iota)}$.
5. For each constructor $c : \iota_1 \Rightarrow \dots \Rightarrow \iota_m \Rightarrow \kappa$, choose its cost interpretation as the zero-valued cost function; size interpretations are additively bounded, as usual. So we choose \mathcal{J}_c^s according to the additive shape interpretation, as in Definition 3.7.4.
6. For each $1 \leq i \leq n$, choose an *interpretation shape* for the symbols in Σ_i^{def} based on the selector strategy \mathcal{S} .
 - If no choice can be made by \mathcal{S} , stop and return MAYBE.
 - (a) If $f \ell_1 \dots \ell_k \rightarrow r$ is a rule of type ι with $f \in \mathcal{D}_1 \cup \dots \cup \mathcal{D}_i$. *Simplify* $\llbracket \text{mark}(f \ell_1 \dots \ell_k) \rrbracket > \llbracket r \rrbracket$ so that the result is a set of inequality constraints C that does not depend on any interpreted variable.
 - (b) Check if the set of constraints C is satisfiable.
 - i. If C is satisfiable, we go back to step 6 with $i := i + 1$.
 - ii. If C is unsatisfiable, we go back to step 3, by setting $K(\iota) := K(\iota) + 1$.
7. If each of the constraint sets C_i , with $1 \leq i \leq n$, are satisfiable and the current $K(\iota) \leq K$. Then we successfully found an interpretation of all defined and constructor symbols that satisfies compatibility. The result is an interpretation function \mathcal{J} .

²Notice that in our setting we cannot possibly return NO.

Lemma 3.7.2. The Parametric Search Procedure above is correct as long as the constraint-solving algorithm is correct.

Indeed, it only chooses interpretations, simplifies the interpretations, and collects a set of constraints over the undeterminate coefficients of the interpretation shapes. The procedure is terminating if we assume that the selector algorithm provides a finite number of interpretation shapes (which is always true in practice). Notice that the parameter K determines an exit condition for the dimension of the tuples. Two key parameters of the procedure above remain to be defined. The strategy \mathcal{S} for selecting interpretation shapes and the constraint solver used in Step 6.

3.7.2 Strategy-based Search for Tuple Interpretations.

The first parameter of the procedure is the selector strategy. Intuitively, a selector strategy \mathcal{S} is an algorithm for choosing parametric interpretations for defined symbols in Σ_i^{def} . For instance, we could randomly pick an interpretation shape from a list (the **blind** strategy); we could incrementally select interpretations from a list of possible attempts (the **progressive** strategy); or we could select interpretations based on their syntax patterns (the **pattern** strategy). Therefore, in a concrete implementation multiple strategies with different heuristics for selecting interpretation shapes can be considered. The definition below lists some interpretation shapes we consider. They are based on the classes studied in [30, 98].

In Definition 3.7.4 below, we consider interpretation *shapes* that are used by Hermes in the search procedure. We first consider max-polynomial shapes which are essentially polynomial expressions that possibly contain the max constructor.

Definition 3.7.3. Let $X = \{x_1, \dots, x_m\}$ be a set of unknowns. The set of **max-polynomial** expressions over \mathbb{N} with unknowns in X — written $\text{Pol}_{\mathbb{N}}[x_1, \dots, x_m]$ — is defined by the grammar:

$$P, Q := x \mid a \mid P + Q \mid P * Q \mid \max(P, Q)$$

Here, x ranges over $\{x_1, \dots, x_m\}$ and a ranges over \mathbb{N} .

Examples of polynomial expressions we consider in the implementation are for instance as in Example 3.7.1. In line with this example, it is worth noticing that in our search procedure the expressions' coefficients — varying over \mathbb{N} — are undetermined and need to be found by the procedure.

Definition 3.7.4. Let $\sigma = \iota_1 \Rightarrow \dots \Rightarrow \iota_m \Rightarrow \kappa$ and fix an interpretation key $\mathcal{J}_{\mathbb{B}}$ over \mathbb{N} . We define **shapes** below as max-polynomial expressions.

- The **additive** class contains linear polynomial expressions $P_1, \dots, P_{K(\kappa)}$ written as $\langle P_1(x^1, \dots, x^m), \dots, P_{K(\kappa)}(x^1, \dots, x^m) \rangle$ and, in order to satisfy Definition 3.3.8, the following constraint is imposed:

$$\sum_{l=1}^{K(\kappa)} P_l(x^1, \dots, x^m) \leq a + \sum_{i=1}^m \sum_{j=1}^{K(t_i)} x_j^i$$

The rest of the shapes below follow the same tuple format written as

$$\langle P_1(x^1, \dots, x^m), \dots, P_{K(\kappa)}(x^1, \dots, x^m) \rangle.$$

But since we do not require additional constraints like in the additive class, we describe a generic shape for one of the $P_i(x^1, \dots, x^m)$.

- The **linear** shape are expressions written as:

$$\sum_{i=1}^m \sum_{j=1}^{K(t_i)} a_{ij} x_{ij}$$

- The **affine** (or simple) shape are expressions written as:

$$a + \sum_{i=1}^m \sum_{j=1}^{K(t_i)} a_{ij} x_{ij}$$

- The **quadratic** shape are expressions where we allow unknowns with degree 2:

$$a + \sum_{i=1}^m \sum_{j=1}^{K(t_i)} a_{ij} x_{ij}^{k_{ij}}, \text{ with } k_{ij} \in \{1, 2\}$$

- The **simple quadratic** shape are expressions written as follows:

$$a + \sum_{i=1}^m \sum_{j=1}^{K(t_i)} a_{ij} x_{ij} + \sum_{i=1}^m \sum_{j=1}^{K(t_i)} a_{ij} x_{ij}^{k_{ij}}, \text{ with } k_{ij} \in \{1, 2\}.$$

Remark 3.7.5. The usage of the max expression is determined by the implementation. So some shapes concretely might use max on certain positions. For instance, a strategy may decide to use $P(x, y) = a_1x + a_2y + \max(x, y)$, which consists of a linear component plus a max component. Whenever the strategy decides to add a max component it must respect the shape format. For instance, a component $x * \max(x, y)$ would not lead to

a linear functional, so that is not allowed to be added to the linear shape. Thus we consider these shapes above as the basic building blocks to build more complicated shapes. The usage of additional components is left to implementation details.

With those shapes in hand, the blind strategy randomly selects one of the shapes above. The incremental strategy chooses shapes in order, from additive ones to quadratic ones. The pattern strategy is slightly more difficult to realize since we need heuristic analysis on the shape of rules. For instance, every rule of the form $f x_1 \dots x_m \rightarrow x_i$ has constant cost functions $\lambda x_1 \dots x_m . 1$ and additively bounded size components. Rules that duplicate variables, as in the pattern $C[x] \rightarrow D[x, x]$, may induce a quadratic bound on cost. Notice that this is the case for all quadratic complexities in this chapter. The concrete implementation of a selector algorithm determines the efficiency of the main procedure for finding interpretations.

The second important parameter is the constraint solver. In order to simplify constraints $\llbracket \ell \rrbracket > \llbracket r \rrbracket$ we have to simplify inequalities between polynomials (max-polynomials). To simplify polynomial (max-polynomial) shapes, we need to compare polynomials $P_\ell^c > R_r^c$ and $P_{\ell_1}^s \supseteq P_{r_1}^s \wedge \dots \wedge P_{\ell_{K(\tau)}}^s \supseteq P_{r_{K(\kappa)}}^s$. These conditions are then reduced to formulas in QFNIA (Quantifier-Free Non-Linear Integer Arithmetic) and sent to an SMT solver, see [40]. Max-polynomials are simplified using the rules $\max(x, y) + z \rightsquigarrow \max(x + z, y + z)$ and $\max(x, y)z \rightsquigarrow \max(xz, yz)$. The result has the form $\max_l P_l$ where each P_l is a polynomial without max occurrences [28]. The usage of the max function depends on the concrete realization of the procedure and the SMT solver the implementer decides to use. Nonetheless, recall that constraints involving the max operator can always be converted to pure inequality constraints over \mathbb{N} . Indeed, $\max(x, y) \leq x + y$ holds in \mathbb{N} for any x, y . Notice that the unknowns in constraints are eliminated using absolute positiveness, e.g., $a_0x + b_0 > a_1x + b_1$ becomes $a_0 > a_1 \wedge b_0 > b_1$.

3.7.3 Prototype Implementation

We provide a concrete implementation of the Parametric Search Procedure as the innermost complexity tool Hermes. It contains implementations for our rewriting formalism, an algorithm to generate interpretation shapes, and the progressive/pattern strategy for selecting interpretation shapes. Hermes can be found at

<https://github.com/deividrvale/hermes>

Installation and building instructions are provided in the repository.

Hermes receives as input a description of a TRS and outputs a tuple interpretation if found. For instance, the system \mathbb{R}_{add} is represented as follows.


```

Signature: [
  zero : nat;
  suc  : nat -> nat;
  add  : nat -> nat -> nat
]
Vars: [
  x : nat;
  y : nat
]
Rules: [
  add x zero    => x;
  add x (suc y) => suc (add x y)
]

```

It can find interpretations for all examples we presented in this chapter except for the system in Example 3.5.14. Indeed, this system requires a non-numeric notion of size, which is currently not implemented in Hermes. The files containing such examples can be found in the folder “benchmarks” of the repository linked above.

Experimental Setting. The Termination Problem Database (TPDB) collects termination and complexity problems that are used in the annual termination competition, see [41]. For our experimental evaluation, we are interested in the subset of problems dedicated to innermost runtime complexity, AG01. This benchmark set contains 36 innermost complexity problems. Additionally, we run Hermes in an external benchmark set, DV23, provided in the folder “experiments” in the Hermes repository. This benchmark set contains a total of 51 and adds more examples to test AG01 including the TRSs we used as examples in this article. The experiment was run on a machine with M1 Pro 2021 processor (10 individual cores at 600 – 3220 MHz) with 16GB of RAM. Memory usage of Hermes during execution ranges from 62MB to 2.75GB, which depends largely on how many rules the input TRS has. Each run of Hermes is set to a time limit of 60 seconds. The table below summarizes the results of running Hermes on our benchmark sets.

Benchmark	Size	# Poly bounds	Timeout	MAYBE
AG01	36	16	20	0
DV23	51	23	25	3

Table 3.1 Results of running Hermes on two benchmark sets

These results can be reproduced by invoking the command

```
./run_experiments.sh
```

from the root of Hermes' repository, assuming one followed the provided installation instructions provided at <https://github.com/deividrvale/hermes>.

To invoke Hermes on a specific file, one runs:

```
hermes path/to/file.onijn
```

Hermes is invoked with the following default options. The number K is set to 2. The default initial search strategy is **progressive**. The maximum degree for the polynomial shapes is 2. These runtime parameters guarantee the termination of the procedure and reasonable resource usage.

Experimental Evaluation. In its current version, v1.0.0, Hermes explores the full power of SMT solvers, which contain state-of-the-art algorithms to solve problems in QFNIA, even though SMT solvers still perform poorly (in comparison to other logics) in this logic. We still obtain satisfactory results on our experimental evaluation as we can solve about 45% of the problems proposed. Notice that our experimental evaluation shows the power of the tuple interpretation technique alone. For a technique solely based on interpretations, our results show an increase in power compared to other interpretation-based techniques like matrix/polynomial interpretations. Indeed, tuple interpretations subsume both matrix and polynomial interpretations and can prove termination of systems for which there is no polynomial nor matrix interpretations, like in Example 3.3.7.

Notice, however that it is rather difficult to directly compare Hermes, implementing only tuple interpretations, with other complexity tools which usually implement a variety of techniques combined. As such, our a priori expectation was that Hermes would solve many of the problems proposed (we got 45%), but would not outperform other complexity analysis tools. Indeed, this hypothesis is confirmed by our experiments. The termination competition in 2022 had two tools competing in the category of Innermost Runtime Complexity. TcT, implemented in [50], and AProVE, implemented in [40]. Let us collect those results in the table below, which can be viewed at

https://termcomp.github.io/Y2022/Runtime_Complexity__TRS_Innermost

We then compare those results to Hermes on AG01.

Benchmark	TcT	AProVE	Hermes
AG01	26	31	16

Table 3.2 Comparison of complexity tools on the AG01 benchmark set on the number of TRSs for which a polynomial upper-bound can be given

This discrepancy is to be expected as both TcT and AProVE implements transformation techniques like dependency pairs together with various dependency pairs processors

that use both interpretations (matrix and polynomial) and syntactic methods (path orders). These results directly guide us to combine the dependency pair framework and tuple interpretations. This is out of the scope of the current work and we leave it for future work.

Chapter 4

Higher-Order Tuple Interpretations

In this chapter, we extend the ideas from Section 3.3 to the higher-order setting. To do this, we will build on the notion of *higher-order strongly monotonic algebras* originating in [92]. The rewriting relation here is that of Definition 2.1.15, so we do not yet impose any restrictions on the applicability of rules. The somewhat higher-order version of Section 3.5 is the subject of Chapter 5.

4.1 Strongly monotonic algebras

In first-order term rewriting, the complexity of a TRS is often measured as *runtime* or *derivational* complexity. Both measures consider initial terms s of a certain shape, and supply a bound on $\text{dh}_{\mathbb{R}}(s)$ given the size of s . However, this is not a good approach for higher-order terms: the behavior of a term of higher type generally cannot be captured in an integer.

Example 4.1.1. Consider the TRS obtained by combining Examples 2.1.11 and 3.1.1. The evaluation cost of a term $\text{foldl } F \ n \ q$ depends almost completely on the “internal” behavior of the functional subterm F , and not only on its evaluation cost. To see this, let us consider two examples: $F_1 := \lambda x. \lambda y. y + x$ and $F_2 := \lambda x. \lambda y. x + x$. For natural numbers n and m , the evaluation cost of both $F_1 \ n \ m$ and $F_2 \ n \ m$ is equal to $n + 1$. However, the absolute size of their normal forms is different. Hence, the number of steps needed to compute $\text{foldl } F_1 \ n \ q$ for a number n and list q is quadratic in the size of n and q , while the number of steps needed for $\text{foldl } F_2 \ n \ q$ is exponential.

Remark 4.1.2. This phenomenon happens here due to the evaluation machine, i.e., the definition of reduction in Definition 2.1.15, for the computation of terms. It is well-known in the literature that duplication of variables may cause an exponential explosion in the absolute size of normal forms. Interestingly enough, examples of this kind bootstrap our intuition for the naturality of the separation between notions of *cost* and *size*.

As Example 4.1.1 shows, higher-order rewriting is a natural place to separate cost and size. But more than that, we need to know what a function does with its arguments: whether it is size-increasing, how long it takes to evaluate them, and more. This is naturally captured by the notion of (weakly or strongly) monotonic algebras for higher-order rewriting introduced by van de Pol [92] where terms of arrow type are interpreted as functions, which allows the interpretation to retain all relevant information.

Higher-order Monotonic Interpretations were originally defined for a different higher-order rewriting formalism, which does make some difference in the way abstraction and application are handled. Weakly monotonic algebras were transposed to the formalism of Algebraic Functional Systems in [38]; however, here we extend the more natural notion of *hereditarily* monotonic algebras which van de Pol only briefly considered [92].

Remark 4.1.3. In [92], van de Pol rejects hereditarily (or: strongly) monotonic algebras because they are not so well-suited for analyzing the HRS format [86] where reasoning is modulo \rightarrow_β : it is impossible to both interpret all terms of functional type as strongly monotonic functions and interpret terms modulo beta, that is, $\llbracket (\lambda x. s) t \rrbracket = \llbracket s[x := t] \rrbracket$. In our higher-order format (also in AFS [38]), we do not have the latter requirement. The AFS format considered in [38] interprets terms to \mathbb{N} rather than to tuples. Weakly monotonic algebras were used because they are a more natural choice in the context of dependency pairs.

Definition 4.1.4. Let \mathbb{B} be a set of sorts and \mathbb{F} a higher-order signature. We assume given for every sort ι an extended well-founded set $(A_\iota, >_\iota, \geq_\iota)$. From this, we define the set of *strongly monotonic functionals*, as follows:

For all sorts ι ,

$$- \langle \iota \rangle := A_\iota \text{ and } \sqsupset_\iota := >_\iota \text{ and } \sqsupseteq_\iota := \geq_\iota.$$

For any arrow type $\sigma \Rightarrow \tau$,

$$- \langle \sigma \Rightarrow \tau \rangle := \{f \in \langle \sigma \rangle \longrightarrow \langle \tau \rangle \mid f \text{ is strongly monotonic}\}.$$

This set is ordered by pointwise comparison, so $f \sqsupset_{\sigma \Rightarrow \tau} g$ iff $\langle \sigma \rangle$ is non-empty and for all $x \in \langle \sigma \rangle$ we have $f(x) \sqsupset_\tau g(x)$, and $f \sqsupseteq_{\sigma \Rightarrow \tau} g$ iff for all $x \in \langle \sigma \rangle$ we have $f(x) \sqsupseteq_\tau g(x)$.

By a straightforward induction on types we have:

Lemma 4.1.5. For any type σ , the triple $(\langle \sigma \rangle, \sqsupset_\sigma, \sqsupseteq_\sigma)$ is an extended well-founded set; that is:

1. \sqsupset_σ is well-founded and \sqsupseteq_σ is reflexive;
2. both \sqsupset_σ and \sqsupseteq_σ are transitive;

3. for all $x, y, z \in \langle \sigma \rangle$, $x \sqsupset_{\sigma} y$ implies $x \sqsupseteq_{\sigma} y$ and $x \sqsupset_{\sigma} y \sqsupseteq_{\sigma} z$ implies $x \sqsupset_{\sigma} z$.

Proof. We prove the result by induction on the structure of the type σ . If σ is a base type we have $\sigma = \iota$, so all items are satisfied by the conditions we impose on extended well-founded sets $(A_{\iota}, >_{\iota}, \geq_{\iota})$. For $\sigma = \tau \Rightarrow \rho$ we reason as follows.

1. $\sqsupset_{\tau \Rightarrow \rho}$ is well-founded and $\sqsupseteq_{\tau \Rightarrow \rho}$ is reflexive.

Suppose, by contradiction, that there is an infinite chain $F_1 \sqsupset_{\tau \Rightarrow \rho} F_2 \sqsupset_{\tau \Rightarrow \rho} \dots$ in $\langle \tau \Rightarrow \rho \rangle$. Then by definition of $\sqsupset_{\tau \Rightarrow \rho}$: $\langle \tau \rangle$ is non-empty, and for all $x \in \langle \tau \rangle$, $F_1(x) \sqsupset_{\rho} F_2(x) \sqsupset_{\rho} \dots$. This induces an infinite \sqsupset_{ρ} -chain in $\langle \rho \rangle$, contradicting the IH. For reflexivity, notice that $F \sqsupseteq_{\tau \Rightarrow \rho} F$ iff for all $x \in \langle \tau \rangle$, $F(x) \sqsupseteq_{\rho} F(x)$, which follows directly by reflexivity of \sqsupseteq_{ρ} (IH).

2. Both relations are transitive.

For $\sqsupset_{\tau \Rightarrow \rho}$. Suppose $F \sqsupset_{\tau \Rightarrow \rho} G \sqsupset_{\tau \Rightarrow \rho} H$, then for all $x \in \langle \tau \rangle$, $F(x) \sqsupset_{\rho} G(x) \sqsupset_{\rho} H(x)$ holds by definition of $\sqsupset_{\tau \Rightarrow \rho}$. The IH give us $F(x) \sqsupset_{\rho} H(x)$, for all $x \in \langle \tau \rangle$, which is exactly $F \sqsupset_{\tau \Rightarrow \rho} H$. Non-emptiness of $\langle \tau \rangle$ holds by assumption. The case for $\sqsupseteq_{\tau \Rightarrow \rho}$ is analogous.

3. For all F, G, H in $\langle \tau \Rightarrow \rho \rangle$, $F \sqsupset_{\tau \Rightarrow \rho} G$ implies $F \sqsupseteq_{\tau \Rightarrow \rho} G$, and $F \sqsupset_{\tau \Rightarrow \rho} G \sqsupseteq_{\tau \Rightarrow \rho} H$ implies $F \sqsupset_{\tau \Rightarrow \rho} H$.

Suppose $F \sqsupset_{\tau \Rightarrow \rho} G$. By definition, $F(x) \sqsupset_{\rho} G(x)$ for all $x \in \langle \tau \rangle$. By IH $F(x) \sqsupseteq_{\rho} G(x)$, for all $x \in \langle \rho \rangle$, which means $F \sqsupseteq_{\rho} G$. If, moreover, $G \sqsupseteq_{\tau \Rightarrow \rho} H$, the reasoning is similar: expand the definitions and apply the induction hypothesis.

□

We define in this chapter the notion of *higher-order strongly monotonic algebras* as an extension of Definition 3.2.1 that now considers the full set of types, not only its first-order fragment like in Chapter 3. So, functional types are interpreted as functional spaces. With that, a term of type σ should be mapped to an element of $\langle \sigma \rangle$. This strategy poses a theoretical problem: we now also have to deal with application and abstraction.

Application is straightforward: since terms of higher type are mapped to functions, we can interpret the application of terms as functional application, so $\llbracket s t \rrbracket := \llbracket s \rrbracket(\llbracket t \rrbracket)$. Abstraction, however, is more difficult. The natural choice would be to view abstraction terms as defining functions which is indeed their standard intensional view as nameless functions. As such, we would be tempted to interpret $\llbracket \lambda x. s \rrbracket_{\alpha}^{\mathcal{J}}$ as the function $d \mapsto \llbracket s \rrbracket_{\alpha[x:=d]}$. This is the function that maps the parameter $d \in \langle \sigma \rangle$ (since $x : \sigma$ for some type σ) to the interpretation of the body s of the abstraction $\lambda x. s$ where we send free occurrences of x to d via the valuation $\alpha[x := d]$.

Unfortunately, the standard way of viewing abstractions does not necessarily give rise to strongly monotonic functions. Indeed, the function $d \mapsto \llbracket s \rrbracket_{\alpha[x:=d]}$ is strongly

monotonic only if x occurs freely in s , since if that is not the case there is no d to be captured in the function body and we get a *constant function*, which is clearly not strongly monotonic. Let us consider a simple example. Take the term $\lambda x. 0$. In the standard view, it would be interpreted as

$$\llbracket \lambda x. 0 \rrbracket = d \mapsto \llbracket 0 \rrbracket_{\alpha[x:=d]} = d \mapsto \mathcal{J}_0$$

which is a constant function since \mathcal{J}_0 does not depend on d , so it is not in $(\text{nat} \Rightarrow \text{nat})$. Interpretation functions should not send terms to different types of sets, in the sense that strongly monotonic interpretations should interpret all terms of functional type as strongly monotonic functions and symbols of base type as elements of non-functional sets. Furthermore, this definition would give $\llbracket (\lambda x. s) t \rrbracket = \llbracket s[x := t] \rrbracket$, so β -steps would not be counted toward the evaluation cost, which is a problem for complexity analysis. We emphasize that this is an important but somewhat subtle difference from the work of van de Pol [92]. In our setting, such property would mean that we “forget” the number of β -steps needed to β -normalize terms. So interpretations modulo β are not even desirable in our complexity point of view.

These considerations on the interpretation of abstractions pose the question if it would even be possible to give strongly monotonic interpretations to all abstraction terms. It turns out that it is indeed possible and surprisingly simpler than we initially thought.

We handle this problem by first postulating the existence of a functional, let us call it by the suggestive name `MakeSM`, that given a function f it always returns a strongly monotonic function or it turns constant functions into strongly monotonic ones. Such a functional would certainly solve our problems with interpreting abstractions. Indeed, whenever we want to interpret $\lambda x. s$ we can always decide whether $x \in \text{fv}(s)$, so we can always decide if the standard interpretation of application terms would not be strongly monotonic. We start by defining the following set.

Definition 4.1.6. For any pair of types σ and τ , we let the set $C_{\sigma,\tau}$ be defined as the union $(\sigma \Rightarrow \tau) \cup \{f \in (\sigma) \longrightarrow (\tau) \mid f(x) = f(y) \text{ for all } x, y \in (\sigma)\}$.

Here, $C_{\sigma,\tau}$ is ordered by pointwise comparison. So this set collects all strongly monotonic and constant functions from (σ) to (τ) .

Definition 4.1.7. A (σ, τ) -monotonicity function $\text{MakeSM}_{\sigma,\tau}$ is a strongly monotonic function in $C_{\sigma,\tau} \longrightarrow (\sigma \Rightarrow \tau)$.

The reader may wonder if such a functional even exists. We show that Definition 4.1.7 is not vacuous by constructing a concrete `MakeSM` in Section 4.2. For now, we take it for granted and move on to define our notion of higher-order strongly monotonic algebras.

Definition 4.1.8. Let $\mathbb{F} = (\mathbb{B}, \Sigma, \text{typeOf})$ be a signature and MakeSM be a family of (σ, τ) -monotonicity functions $\text{MakeSM}_{\sigma, \tau}$ parametrized by all pairs of types σ, τ in $\mathbb{T}(\mathbb{B})$. A **strongly monotonic** \mathbb{F} -algebra $\mathcal{F} = (A, \mathcal{J})$ is a structure with a domain A given by the family of strongly monotonic functionals $(\langle \sigma \rangle, \sqsupset_{\sigma}, \sqsubseteq_{\sigma})_{\sigma \in \mathbb{T}(\mathbb{B})}$, and an interpretation function \mathcal{J} which maps each $f : \sigma \in \Sigma$ to an element of $\langle \sigma \rangle$.

Let α be a **valuation function** that maps variables of type σ to elements of $\langle \sigma \rangle$. We extend \mathcal{J} to a function $\llbracket \cdot \rrbracket_{\alpha}^{\mathcal{J}}$ to the set of all terms as follows:

$$\begin{aligned} \llbracket x \rrbracket_{\alpha}^{\mathcal{J}} &= \alpha(x) & \llbracket f \rrbracket_{\alpha}^{\mathcal{J}} &= \mathcal{J}_f & \llbracket s t \rrbracket_{\alpha}^{\mathcal{J}} &= \llbracket s \rrbracket_{\alpha}^{\mathcal{J}}(\llbracket t \rrbracket_{\alpha}^{\mathcal{J}}) \\ \llbracket \lambda x. s \rrbracket_{\alpha}^{\mathcal{J}} &= \text{MakeSM}_{\sigma, \tau} \left(d \mapsto \llbracket s \rrbracket_{\alpha[x:=d]}^{\mathcal{J}} \right), & \text{if } x : \sigma \text{ and } s : \tau \end{aligned}$$

Notation. As we did in Chapter 3, here we typically omit the subscript α and superscript \mathcal{J} and use notations like $\llbracket s \rrbracket = F(x + 3)$ to denote $\llbracket s \rrbracket_{\alpha}^{\mathcal{J}} = \alpha(F)(\alpha(x) + 3)$. This will be used whenever the valuation α or interpretation \mathcal{J} are universally quantified. When types are not relevant, we will denote \sqsupset instead of specifying \sqsupset_{σ} .

The next lemma is a technical result that establishes the correctness of Definition 4.1.8, that is, whenever $s : \sigma$ we have $\llbracket s \rrbracket \in \langle \sigma \rangle$.

Lemma 4.1.9. For any term $s : \sigma$ we have $\llbracket s \rrbracket \in \langle \sigma \rangle$.

Proof. By induction on the structure of s . Notice that we are done if we prove the following statement: $\llbracket s \rrbracket_{\alpha} \in \langle \sigma \rangle$ and for all variables x occurring in the domain of α : either $d \mapsto \llbracket s \rrbracket_{\alpha[x:=d]}$ is a strongly monotonic function, or it is a constant function.

- If $s = x$ then $\llbracket x \rrbracket_{\alpha} = \alpha(x) \in \langle \sigma \rangle$ by assumption. Moreover, $d \mapsto \llbracket s \rrbracket_{\alpha[x:=d]}$ is the identity function $d \mapsto d$, which is trivially strongly monotonic. For all other variables $y \neq x$, the function $d \mapsto \llbracket y \rrbracket_{\alpha[y:=d]}$ is the constant function $d \mapsto \alpha(x)$.
- If $s = f$ with $f : \sigma$, then $\llbracket f \rrbracket_{\alpha} \in \langle \sigma \rangle$ by Definition 4.1.8. Also, we have that $d \mapsto \llbracket f \rrbracket_{\alpha[x:=d]}$ is exactly the function $d \mapsto \mathcal{J}_f$, which is constant.
- If $s = t u$ then $t : \tau \Rightarrow \sigma$ and $u : \tau$. By the induction hypothesis, $\llbracket t \rrbracket_{\alpha} \in \langle \tau \Rightarrow \sigma \rangle$ which is a subset of $\langle \tau \rangle \rightarrow \langle \sigma \rangle$, and $\llbracket u \rrbracket_{\alpha}^{\mathcal{J}} \in \langle \tau \rangle$. Consequently, $\llbracket t \rrbracket_{\alpha}^{\mathcal{J}}(\llbracket u \rrbracket_{\alpha}^{\mathcal{J}})$ is in $\langle \sigma \rangle$.

We also get by (IH) that $d \mapsto \llbracket t \rrbracket_{\alpha[x:=d]}$ is either strongly monotonic or constant. The same holds for $d \mapsto \llbracket u \rrbracket_{\alpha[x:=d]}$. This gives us four cases.

1. If both functions are constant, then $d \mapsto \llbracket t \rrbracket_{\alpha[x:=d]}(\llbracket u \rrbracket_{\alpha[x:=d]})$ is constant.
2. If $d \mapsto \llbracket t \rrbracket_{\alpha[x:=d]}$ is constant and $d \mapsto \llbracket u \rrbracket_{\alpha[x:=d]}$ is strongly monotonic, then for $a \sqsupset b$ we have $\llbracket t \rrbracket_{\alpha[x:=a]} = \llbracket t \rrbracket_{\alpha[x:=b]} = \llbracket t \rrbracket_{\alpha}$, and additionally

$\llbracket u \rrbracket_{\alpha[x:=a]} \sqsupset_{\tau} \llbracket u \rrbracket_{\alpha[x:=b]}$. Hence, by monotonicity of $\llbracket t \rrbracket_{\alpha}$ we get:

$$\begin{aligned} \llbracket s \rrbracket_{\alpha[x:=a]} &= \llbracket t \rrbracket(\llbracket u \rrbracket_{\alpha[x:=a]}) \\ &\sqsupset_{\sigma} \llbracket t \rrbracket_{\alpha}^{\mathcal{J}}(\llbracket u \rrbracket_{\alpha[x:=b]}) \\ &= \llbracket s \rrbracket_{[x:=b]} \end{aligned}$$

3. If $d \mapsto \llbracket t \rrbracket_{\alpha[x:=d]}$ is strongly monotonic and $d \mapsto \llbracket u \rrbracket_{\alpha[x:=d]}$ is constant, then for $a \sqsupset b$ we have: $\llbracket t \rrbracket_{\alpha[x:=a]} \sqsupset_{\tau \Rightarrow \sigma} \llbracket t \rrbracket_{\alpha[x:=b]}$, and additionally we get $\llbracket u \rrbracket_{\alpha[x:=a]} = \llbracket u \rrbracket_{\alpha[x:=b]} = \llbracket u \rrbracket$. By definition of $\sqsupset_{\tau \Rightarrow \sigma}$, we thus have

$$\begin{aligned} \llbracket s \rrbracket_{[x:=a]} &= \llbracket t \rrbracket_{\alpha[x:=a]}(\llbracket u \rrbracket_{\alpha}^{\mathcal{J}}) \\ &\sqsupset_{\sigma} \llbracket t \rrbracket_{\alpha[x:=b]}(\llbracket u \rrbracket_{\alpha}^{\mathcal{J}}) \\ &= \llbracket s \rrbracket_{[x:=b]}. \end{aligned}$$

4. If both are strongly monotonic, then by monotonicity of $\llbracket t \rrbracket_{\alpha[x:=a]}$ we have

$$\begin{aligned} \llbracket s \rrbracket_{\alpha[x:=a]} &= \llbracket t \rrbracket_{\alpha[x:=a]}(\llbracket u \rrbracket_{\alpha[x:=a]}) \\ &\sqsupset_{\sigma} \llbracket t \rrbracket_{\alpha[x:=a]}(\llbracket u \rrbracket_{\alpha[x:=b]}) \\ &\sqsupset_{\sigma} \llbracket t \rrbracket_{\alpha[x:=b]}(\llbracket u \rrbracket_{\alpha[x:=b]}) \\ &= \llbracket s \rrbracket_{\alpha[x:=b]} \end{aligned}$$

since $\llbracket t \rrbracket_{\alpha[x:=a]} \sqsupset_{\tau \Rightarrow \sigma} \llbracket t \rrbracket_{\alpha[x:=b]}$.

- Finally, the abstraction case. If $s = \lambda x. t$ with $\sigma = \tau \Rightarrow \rho$, then its interpretation is $\llbracket s \rrbracket_{\alpha} = \text{MakeSM}_{\tau, \rho}(d \mapsto \llbracket t \rrbracket_{\alpha[x:=d]})$. Since by the induction hypothesis the function $d \mapsto \llbracket t \rrbracket_{\alpha[x:=d]}$ is either a constant or a strongly monotonic function from $\langle \tau \rangle$ to $\langle \rho \rangle$, it is an element of $C_{\tau, \rho}$. Therefore, $\llbracket s \rrbracket_{\alpha} = \text{MakeSM}_{\tau, \rho}(d \mapsto \llbracket t \rrbracket_{\alpha[x:=d]})$ is well-defined and $\text{MakeSM}_{\tau, \rho}(d \mapsto \llbracket t \rrbracket_{\alpha[x:=d]})$ yields an element of $\langle \tau \Rightarrow \rho \rangle$. Now, consider a variable y .

- If $y = x$, then $e \mapsto \llbracket s \rrbracket_{\alpha[y:=e]}$ is clearly a constant function, so we have

$$\begin{aligned} \llbracket s \rrbracket_{\alpha[y:=e]} &= \text{MakeSM}_{\tau, \rho}(d \mapsto \llbracket t \rrbracket_{\alpha[x:=e][x:=d]}) \\ &= \text{MakeSM}_{\tau, \rho}(d \mapsto \llbracket t \rrbracket_{\alpha[x:=d]}) \end{aligned}$$

- If $y \neq x$ note that by the induction hypothesis either $e \mapsto \llbracket t \rrbracket_{\alpha[y:=e][x:=d]}$ is constant or strongly monotonic.

- * If it is constant, then for all a and b we have that

$$d \mapsto \llbracket t \rrbracket_{\alpha[y:=a][x:=d]} = d \mapsto \llbracket t \rrbracket_{\alpha[y:=b][x:=d]}$$

and therefore the equality below holds

$$\text{MakeSM}_{\tau,\rho}(d \mapsto \llbracket t \rrbracket_{\alpha[y:=a][x:=d]}) = \text{MakeSM}_{\tau,\rho}(d \mapsto \llbracket t \rrbracket_{\alpha[y:=b][x:=d]})$$

So, the function $e \mapsto \llbracket s \rrbracket_{\alpha[y:=e]}$ is constant too.

- * Otherwise, if this function is strongly monotonic, then the function $d \mapsto \llbracket t \rrbracket_{\alpha[y:=a][x:=d]}$ is pointwise greater than $d \mapsto \llbracket t \rrbracket_{\alpha[y:=b][x:=d]}$. Hence, $\llbracket s \rrbracket_{\alpha[y:=a]} \supseteq_{\sigma} \llbracket s \rrbracket_{\alpha[y:=b]}$ as well.

□

4.1.1 Higher-Order Compatibility

As it is expected, we need yet another version of the compatibility result to show that our notion of higher-order strongly monotonic algebras is able to prove termination of compatible systems. Compatibility is slightly different in our higher-order formalism since now we have β rule-schemes to orient.

Definition 4.1.10. We say a TRS (\mathbb{F}, \mathbb{R}) is **compatible** with a higher-order strongly monotonic algebra \mathcal{F} whenever the following two conditions hold:

1. (β -compatibility) for any pair of terms s and t , $\llbracket (\lambda x. s) t \rrbracket \supseteq \llbracket s[x := t] \rrbracket$; and
2. (\mathbb{R} -compatibility) for all rules $\ell \rightarrow r$ in \mathbb{R} , $\llbracket \ell \rrbracket \supseteq \llbracket r \rrbracket$.

To prove Theorem 4.1.13, we need a higher-order version of the so-called *Substitution Lemma*. We begin by giving a systematic way of extending a substitution (seen as a morphism between terms) to a valuation, seen as morphism from terms to elements of the domain of an algebra \mathcal{F} .

Definition 4.1.11. Given a substitution $\gamma = [x_1 := s_1, \dots, x_n := s_n]$ and a valuation α , we define α^γ as the valuation such that $\alpha^\gamma(x) = \alpha(x)$, if $x \notin \text{supp}(\gamma)$; and $\alpha^\gamma(x) = \llbracket x\gamma \rrbracket_{\alpha}$ otherwise.

Lemma 4.1.12. For any substitution γ and valuation α , $\llbracket s\gamma \rrbracket_{\alpha} = \llbracket s \rrbracket_{\alpha^\gamma}$. Additionally, if $\llbracket s \rrbracket \supseteq_{\sigma} \llbracket t \rrbracket$ ($\llbracket s \rrbracket \supseteq_{\sigma} \llbracket t \rrbracket$), then $\llbracket s\gamma \rrbracket \supseteq_{\sigma} \llbracket t\gamma \rrbracket$ ($\llbracket s\gamma \rrbracket \supseteq_{\sigma} \llbracket t\gamma \rrbracket$).

Proof. By inspection of Definition 4.1.11 it can be easily shown by induction on s that the following diagram commutes:

$$\begin{array}{ccc}
 & & \mathbb{T}(\mathbb{F}, \mathbb{X}) \\
 & \nearrow \gamma & \downarrow \llbracket \cdot \rrbracket_{\alpha} \\
 \mathbb{T}(\mathbb{F}, \mathbb{X}) & \xrightarrow{\llbracket \cdot \rrbracket_{\alpha^\gamma}} & \mathcal{F}
 \end{array}$$

As a consequence, if $\llbracket s \rrbracket_\alpha \sqsupset_\sigma \llbracket t \rrbracket_\alpha$ for any valuation α , then $\llbracket s \rrbracket_{\alpha\gamma} \sqsupset_\sigma \llbracket t \rrbracket_{\alpha\gamma}$ in particular. So $\llbracket s\gamma \rrbracket_\alpha \sqsupset_\sigma \llbracket t\gamma \rrbracket_\alpha$. The case for \sqsupset_σ is analogous. \square

We can now show the compatibility theorem by induction on the rewrite relation.

Theorem 4.1.13. If a TRS \mathbb{R} is compatible with a higher-order tuple algebra \mathcal{F} , then for all valuations α , $\llbracket s \rrbracket_\alpha \sqsupset \llbracket t \rrbracket_\alpha$ whenever $s \rightarrow t$.

Proof. We reason by induction on $s \rightarrow t$. According to Definition 2.1.15, we have the following cases to consider.

- Suppose $s \rightarrow t$ by $\ell\gamma \rightarrow r\gamma$. Compatibility gives $\llbracket \ell \rrbracket \sqsupset \llbracket r \rrbracket$, and by Lemma 4.1.12 we have $\llbracket \ell\gamma \rrbracket \sqsupset \llbracket r\gamma \rrbracket$.
- The case $(\lambda x. s) t \rightarrow s[x := t]$ follows directly by compatibility.
- We consider the case $s t \rightarrow s t'$, with $t \rightarrow t'$. The other application case is analogous. We have $\llbracket s t \rrbracket = \llbracket s \rrbracket(\llbracket t \rrbracket)$ by Definition 4.1.8 and by Lemma 4.1.9 $\llbracket s \rrbracket$ is strongly monotonic. The induction hypothesis gives us $\llbracket t \rrbracket \sqsupset \llbracket t' \rrbracket$. Finally, combining these results we get $\llbracket s \rrbracket(\llbracket t \rrbracket) \sqsupset \llbracket s \rrbracket(\llbracket t' \rrbracket)$.
- Suppose $\lambda x. s \rightarrow \lambda x. t$, with $s \rightarrow t$.
 - If $x \notin \text{fv}(s)$, then by (IH) we get $\llbracket s \rrbracket_{\alpha[x:=d]} \sqsupset \llbracket t \rrbracket_{\alpha[x:=d]}$. As a consequence, $d \mapsto \llbracket s \rrbracket_{\alpha[x:=d]} \sqsupset d \mapsto \llbracket t \rrbracket_{\alpha[x:=d]}$ are constant functions not in the domain of \mathcal{F} . By Definition 4.1.7, $\text{MakeSM}_{\sigma,\tau}$ is strongly monotonic, so

$$\text{MakeSM}_{\sigma,\tau}(d \mapsto \llbracket s \rrbracket_{\alpha[x:=d]}) \sqsupset \text{MakeSM}_{\sigma,\tau}(d \mapsto \llbracket t \rrbracket_{\alpha[x:=d]})$$

- On the other hand, if $x \in \text{fv}(s)$, then $d \mapsto \llbracket s \rrbracket_{\alpha[x:=d]} \sqsupset_{\sigma \Rightarrow \tau} d \mapsto \llbracket t \rrbracket_{\alpha[x:=d]}$ are strongly monotonic functions and the result follows by Definition 4.1.7.

\square

For Definition 4.1.4 and Theorem 4.1.13, we can choose well-founded sets $(A_l, >_l, \geq_l)$ for each sort, and the functions $\text{MakeSM}_{\sigma,\tau}$ for each pair of types as we desire. Let us take a first example of interpretation by interpreting the map function in this setting.

Example 4.1.14. Let $A_{\text{nat}} = \mathbb{N}^2$ and $A_{\text{list}} = \mathbb{N}^3$ and each interpretation of the constructors cons and $[]$ be as in Example 3.3.5. Consider the rules for map in Example 2.1.11. We let:

$$\mathcal{J}_{\text{map}} = \lambda F q. \langle (q_l + 1) * (F(\langle q_c, q_m \rangle)_c + 1), q_l, F(q_c, q_m)_s \rangle$$

This expresses that map does not increase the list's length (as the length component is just q_l), the greatest element of the result is bounded by the value of F on the greatest

element of q , and the evaluation cost is mostly expressed by a number of F steps that is linear in the length of q . We will see in Lemma 4.3.1 that \mathcal{J}_{map} is in fact strongly monotonic.

To prove the \mathbb{R} -compatibility of this system, we must first show that $\llbracket \ell \rrbracket \sqsupseteq \llbracket r \rrbracket$ for all rules $\ell \rightarrow r$. For the first map rule this is easy:

$$\llbracket \text{map}(F, []) \rrbracket = \langle F(\langle 0, 0 \rangle)_c + 1, 0, F(\langle 0, 0 \rangle)_s \rangle \sqsupseteq_{\text{list}} \langle 0, 0, 0 \rangle = \llbracket [] \rrbracket$$

For the second map rule, we must check that

$$\langle \text{cost-}\ell, \text{len-}\ell, \text{max-}\ell \rangle \sqsupseteq_{\text{list}} \langle \text{cost-}r, \text{len-}r, \text{max-}r \rangle,$$

that is, $\text{cost-}\ell > \text{cost-}r$ and $\text{len-}\ell \geq \text{len-}r$ and $\text{max-}\ell \geq \text{max-}r$, where:

$$\begin{aligned} \text{cost-}\ell &= \llbracket \text{map}(F, x : q) \rrbracket_c &= (q_l + 2) * (F(\langle x_c + q_c, \max(x_s, q_m) \rangle)_c + 1) \\ \text{cost-}r &= \llbracket F(x) : \text{map}(F, q) \rrbracket_c &= F(\langle x_c, x_s \rangle)_c + (q_l + 1) * (F(\langle q_c, q_m \rangle)_c + 1) \\ \text{len-}\ell &= \llbracket \text{map}(F, x : q) \rrbracket_l &= q_l + 1 = \llbracket F(x) : \text{map}(F, q) \rrbracket_l = \text{len-}r \\ \text{max-}\ell &= \llbracket \text{map}(F, x : q) \rrbracket_m &= F(\langle x_c + q_c, \max(x_s, q_m) \rangle)_s \\ \text{max-}r &= \llbracket F(x) : \text{map}(F, q) \rrbracket_m &= \max(F(\langle x_c, x_s \rangle)_s, F(\langle q_c, q_m \rangle)_s) \end{aligned}$$

To see why $\text{cost-}\ell > \text{cost-}r$, for instance, we observe that for all x, q the tuple given by $\langle x_c + q_c, \max(x_s + q_m) \rangle$ is \sqsupseteq_{nat} to both $\langle x_c, x_s \rangle$ and $\langle q_c, q_m \rangle$. Since $F \in (\text{nat} \Rightarrow \text{nat})$, we get that $F(\langle x_c + q_c, \max(x_s + q_m) \rangle)$ is \sqsupseteq_{nat} to both $F(\langle x_c, x_s \rangle)$ and $F(\langle q_c, q_m \rangle)$. The other cases are treated by similar reasoning.

4.2 Interpreting abstractions

The compatibility proof of \mathbb{R}_{map} is not complete. Indeed, we showed only \mathbb{R} -compatibility in Example 4.1.14. β -compatibility requires more work as we have not yet explicitly defined the functions $\text{MakeSM}_{\sigma, \tau}$, which is needed in order to show $\llbracket (\lambda x. s) t \rrbracket \sqsupseteq \llbracket s[x := t] \rrbracket$ always holds. To achieve this, we will define some standard functions to build elements of (σ) . This allows us to easily construct strongly monotonic functionals, both to build $\text{MakeSM}_{\sigma, \tau}$ and to create interpretation functions \mathcal{J}_f .

4.2.1 Strongly Monotonic Combinators

Definition 4.2.1. For every type σ , we define the functions $\mathbf{0}_\sigma \in \langle \sigma \rangle$, $\text{costof}_\sigma \in \langle \sigma \rangle \rightarrow \mathbb{N}$, and $\text{addc}_\sigma \in \mathbb{N} \times \langle \sigma \rangle \rightarrow \langle \sigma \rangle$ by mutual recursion on the structure of types as follows:

$$\begin{aligned} \mathbf{0}_\iota &= \langle 0_1, \dots, 0_{K(\iota)} \rangle & \mathbf{0}_{\sigma \Rightarrow \tau} &= d \mapsto \text{addc}_\tau(\text{costof}_\sigma(d), \mathbf{0}_\tau) \\ \text{costof}_\iota(\langle n_1, \dots, n_{K(\iota)} \rangle) &= n_1 & \text{costof}_{\sigma \Rightarrow \tau}(F) &= \text{costof}_\tau(F(\mathbf{0}_\sigma)) \\ \text{addc}_\iota(c, \langle n_1, \dots, n_{K(\iota)} \rangle) &= \langle c + n_1, n_2, \dots, n_{K(\iota)} \rangle & \text{addc}_{\sigma \Rightarrow \tau}(c, F) &= d \mapsto \text{addc}_\tau(c, F(d)) \end{aligned}$$

Here, $\mathbf{0}_\sigma$ gives a *minimal element* for $\langle \sigma \rangle$ with respect to \sqsupset_σ . The function costof_σ maps every F to the cost component of $F(\mathbf{0}_{\sigma_1}, \dots, \mathbf{0}_{\sigma_m})$; hence, if $F \sqsupset_\sigma G$ we have $\text{costof}_\sigma(F) > \text{costof}_\sigma(G)$. The function addc_σ point-wise adds a natural number to the cost component of an element of $\langle \sigma \rangle$. Therefore, if $F(x_1, \dots, x_m) = \langle n_1, \dots, n_k \rangle$, then $\text{addc}(c, F)(x_1, \dots, x_m) = \langle c + n_1, n_2, \dots, n_k \rangle$. In this next lemma we show the correctness of this definition.

Lemma 4.2.2. For all types σ :

1. $\mathbf{0}_\sigma \in \langle \sigma \rangle$;
2. for all $n \in \mathbb{N}$ and $x \in \langle \sigma \rangle$, $\text{addc}_\sigma(n, x) \in \langle \sigma \rangle$;
3. costof_σ is weakly monotonic and strict in its first argument;
4. addc_σ is weakly monotonic and strict in both its arguments.

Proof. Each item of the lemma follows by a mutual induction on the structure of σ .

- (1) If $\sigma = \iota \in \mathbb{B}$, then $\mathbf{0}_\sigma = \langle 0, \dots, 0 \rangle$ is clearly in $\langle \iota \rangle$. If $\sigma = \tau \Rightarrow \rho$ then $\mathbf{0}_{\tau \Rightarrow \rho} = d \mapsto \text{addc}_\rho(\text{costof}_\tau(d), \mathbf{0}_\rho)$. Clearly $\text{costof}_\tau(d) \in \mathbb{N}$ and by induction hypothesis (1) $\mathbf{0}_\rho \in \langle \rho \rangle$, so by induction hypothesis (2) $\text{addc}_\rho(\text{costof}_\tau(d), \mathbf{0}_\rho) \in \langle \rho \rangle$. We still need to prove this function is strongly monotonic. So let us assume $x \sqsupset_\sigma y$ and prove $\mathbf{0}_x \sqsupset_\tau \mathbf{0}_y$. The proof for $x \sqsupset_\iota y$ is similar and we omit it. With that in mind, notice that we have $\text{costof}_\tau(x) > \text{costof}_\tau(y)$ by induction hypothesis (3). Hence, $\text{addc}_\rho(\text{costof}_\tau(x), \mathbf{0}_\rho) \sqsupset_\rho \text{addc}_\rho(\text{costof}_\tau(y), \mathbf{0}_\rho)$ by induction hypothesis (4); that is $\mathbf{0}_\sigma(x) \sqsupset_\rho \mathbf{0}_\sigma(y)$.
- (2) If $\sigma = \iota$, then $\text{addc}_\sigma(n, x) = \langle n + x_1, x_2, \dots, x_{K(\iota)} \rangle$, which is trivially an element of $\langle \iota \rangle$. Otherwise, we let $\sigma = \tau \Rightarrow \rho$ and take $n \in \mathbb{N}$ and $F \in \langle \tau \Rightarrow \rho \rangle$. Then $\text{addc}_{\tau \Rightarrow \rho}(n, F) = d \mapsto \text{addc}_\rho(n, F(d))$. By induction hypothesis (2), $\text{addc}_\rho(n, F(d))$ is in $\langle \rho \rangle$, so $\text{addc}_{\tau \Rightarrow \rho}(n, F) \in \langle \tau \rangle \rightarrow \langle \rho \rangle$; we only need to see that it is strongly monotonic. So let us assume $u \sqsupset_\tau w$, we will see that $\text{addc}_{\tau \Rightarrow \rho}(n, F, u) \sqsupset_\rho \text{addc}_{\tau \Rightarrow \rho}(n, F, w)$. Indeed, we have $\text{addc}_{\tau \Rightarrow \rho}(n, F, u) = \text{addc}_\rho(n, F(u))$. Since

F is strongly monotonic, we get $F(u) \sqsupset_{\rho} F(w)$. By induction hypothesis (4), $\text{addc}_{\rho}(n, F(u)) \sqsupset_{\rho} \text{addc}_{\rho}(n, F(w))$ which is equal to $\text{addc}_{\tau \Rightarrow \rho}(n, F, w)$. The proof of the case $u \sqsupset_{\tau} w$ follows similar reasoning.

(3) Here, we also work out the case for $x \sqsupset_{\sigma} y$; the case for $x \sqsupseteq_{\sigma} y$ is similar. For the base case, we assume σ is a base type $\iota \in \mathbb{B}$, then $\text{costof}_{\sigma}(x) = x_1 > y_1 = \text{costof}_{\sigma}(y)$ by definition of $x \sqsupset_{\iota} y$. If $\sigma = \tau \Rightarrow \rho$ then $\text{costof}_{\sigma}(x) = \text{costof}_{\rho}(x(\mathbf{0}_{\tau}))$. Since $x \sqsupset_{\tau \Rightarrow \rho} y$ we have $x(\mathbf{0}_{\tau}) \sqsupset_{\rho} y(\mathbf{0}_{\tau})$. By induction hypothesis (3), $\text{costof}_{\rho}(x(\mathbf{0}_{\tau})) > \text{costof}_{\rho}(y(\mathbf{0}_{\tau}))$ follows as required.

(4) Suppose $n \geq m$ and $x \sqsupseteq_{\sigma} y$. We show that

- (a) $\text{addc}_{\sigma}(n, x) \sqsupseteq_{\sigma} \text{addc}_{\sigma}(m, y)$, and
- (b) if $n > m$ or $x \sqsupset_{\sigma} y$ then $\text{addc}_{\sigma}(n, x) \sqsupset_{\sigma} \text{addc}_{\sigma}(m, y)$.

For the induction base case, σ is a base type $\iota \in \mathbb{B}$, then for the case (a)

$$\text{addc}_{\sigma}(n, x) = \langle n + x_1, x_2, \dots, x_{K(\iota)} \rangle \sqsupseteq_{\iota} \langle m + y_1, y_2, \dots, y_{K(\iota)} \rangle$$

because each $x_i \geq y_i$ and $n \geq m$. In case (b), we have $n > m$ or $x_1 > y_1$ so certainly $n + x_1 > m + y_1$.

For the inductive step we have $\sigma = \tau \Rightarrow \rho$. Then we get $\text{addc}_{\sigma}(n, x) = d \mapsto \text{addc}_{\rho}(n, x(d))$. Notice that since $x \sqsupseteq_{\sigma} y$, $x(d) \sqsupseteq_{\rho} y(d)$. Additionally, if $x \sqsupset_{\sigma} y$ then $x(d) \sqsupset_{\rho} y(d)$. Hence, by induction hypothesis (4), $\text{addc}_{\rho}(n, x(d)) \sqsupseteq_{\rho} \text{addc}_{\rho}(m, y(d))$ and if $n > m$ or $x \sqsupset_{\sigma} y$ even $\text{addc}_{\rho}(n, x(d)) \sqsupset_{\rho} \text{addc}_{\rho}(m, y(d))$. This suffices, since $\sqsupseteq_{\tau \Rightarrow \rho}$ and $\sqsupset_{\tau \Rightarrow \rho}$ do point-wise comparisons.

□

Next, the following lemmas provide basic properties of these functions (and how they interact with each other).

Lemma 4.2.3. For all types σ , for all $x \in \langle \sigma \rangle$:

1. $\text{addc}_{\sigma}(0, x) = x$;
2. for all $n, m \in \mathbb{N}$: $\text{addc}_{\sigma}(n, \text{addc}_{\sigma}(m, x)) = \text{addc}_{\sigma}(n + m, x)$;
3. if $n > 0$ then $\text{addc}_{\sigma}(n, x) \sqsupset_{\sigma} x$;
4. if $y \in \langle \sigma \rangle$ is such that $x \sqsupset_{\sigma} y$ then $x \sqsupseteq_{\sigma} \text{addc}_{\sigma}(1, y)$; and
5. for all $n \in \mathbb{N}$: $\text{costof}_{\sigma}(\text{addc}_{\sigma}(n, x)) = n + \text{costof}_{\sigma}(x)$.

Proof. We prove each item of the lemma by induction on the structure of σ .

- (1) In the base case, we assume σ is a base type $\iota \in \mathbb{B}$, then

$$\text{addc}_\sigma(0, x) = \langle 0 + x_1, x_2, \dots, x_{K(\iota)} \rangle = \langle x_1, \dots, x_{K(\iota)} \rangle = x$$

and the result follows.

For the inductive case, we have $\sigma = \tau \Rightarrow \rho$. Therefore, for $F \in \langle \tau \Rightarrow \rho \rangle$, $\text{addc}_\sigma(0, F) = d \mapsto \text{addc}_\rho(0, F(d))$ and by induction hypothesis $\text{addc}_\rho(0, F(d)) = F(d)$. Hence, we get

$$\begin{aligned} \text{addc}_\sigma(0, F) &= d \mapsto \text{addc}_\rho(0, F(d)) \\ &= d \mapsto F(d) \\ &= F \end{aligned}$$

wich proves the result.

- (2) If $\sigma = \iota \in \mathbb{B}$, then

$$\text{addc}_\sigma(n, \text{addc}_\sigma(m, x)) = \langle n + m + x_1, x_2, \dots, x_{K(\iota)} \rangle = \text{addc}_\sigma(n + m, x)$$

If $\sigma = \tau \Rightarrow \rho$, then

$$\begin{aligned} \text{addc}_\sigma(n, \text{addc}_\sigma(m, x)) &= d \mapsto \text{addc}_\sigma(n, \text{addc}_\sigma(m, x(d))) \\ &\stackrel{(IH)}{=} d \mapsto \text{addc}_\sigma(n + m, x(d)) \\ &= \text{addc}_\sigma(n + m, x) \end{aligned}$$

- (3) For the base case, we assume σ is a base type $\iota \in \mathbb{B}$. So

$$\begin{aligned} \text{addc}_\iota(n, \langle x_1, x_2, \dots, x_{K(\iota)} \rangle) &= \langle n + x_1, x_2, \dots, x_{K(\iota)} \rangle \\ &\sqsubset_\iota \langle x_1, x_2, \dots, x_{K(\iota)} \rangle \end{aligned}$$

In the inductive step, we assume $\sigma = \tau \Rightarrow \rho$. So taking $F \in \langle \sigma \rangle$ we get $\text{addc}_\sigma(n, F) = d \mapsto \text{addc}_\rho(n, F(d))$ by the (IH) we get $\text{addc}_\rho(n, F(d)) \sqsubset_\rho F(d)$. Thus, for any $d \in \langle \tau \rangle$

$$\begin{aligned} \text{addc}_\sigma(n, F) &= d \mapsto \text{addc}_\rho(n, F(d)) \\ &\sqsubset_\sigma d \mapsto F(d) \\ &= F \end{aligned}$$

- (4) Let us take $x \sqsubset_\sigma y$. In the base case, σ is a base type $\iota \in \mathbb{B}$, then $x = \langle x_1, \dots, x_{K(\iota)} \rangle$ and $y = \langle y_1, \dots, y_{K(\iota)} \rangle$, and $x \sqsubset_\iota y$ implies that $x_1 > y_1$ and each $x_i \geq y_i$. Then,

$x_1 \geq 1 + y_1$, so

$$\begin{aligned} x &= \langle x_1, \dots, x_{K(\iota)} \rangle \\ &\sqsupseteq_{\iota} \langle 1 + y_1, y_2, \dots, y_{K(\iota)} \rangle \\ &= \text{addc}_{\iota}(1, y) \end{aligned}$$

In the inductive case, we have $\sigma = \tau \Rightarrow \rho$, then $x = d \mapsto x(d)$ and similarly $y = d \mapsto y(d)$. Notice that $x \sqsupseteq_{\sigma} y$ implies that $x(d) \sqsupseteq_{\rho} y(d)$ for all $d \in \llbracket \tau \rrbracket$. By the induction hypothesis, $x(d) \sqsupseteq_{\rho} \text{addc}_{\rho}(1, y(d))$ for all d , and therefore

$$\begin{aligned} x &= d \mapsto x(d) \\ &\sqsupseteq_{\tau \Rightarrow \rho} d \mapsto \text{addc}_{\rho}(1, y(d)) \\ &= \text{addc}_{\tau \Rightarrow \rho}(1, y) \end{aligned}$$

(5) If $\sigma = \iota \in \mathbb{B}$, then

$$\begin{aligned} \text{costof}_{\sigma}(\text{addc}_{\sigma}(n, x)) &= \text{costof}_{\sigma}(\langle n + x_1, x_2, \dots, x_{K(\iota)} \rangle) \\ &= n + x_1 \\ &= n + \text{costof}_{\sigma}(x) \end{aligned}$$

In the inductive case, $\sigma = \tau \Rightarrow \rho$, then

$$\begin{aligned} \text{costof}_{\sigma}(\text{addc}_{\sigma}(n, x)) &= \text{costof}_{\sigma}(d \mapsto \text{addc}_{\rho}(n, x(d))) \\ &= \text{costof}_{\rho}(\text{addc}_{\rho}(n, x(\mathbf{0}_{\tau}))) \\ &\stackrel{(IH)}{=} n + \text{costof}_{\rho}(x(\mathbf{0}_{\tau})) \\ &= n + \text{costof}_{\tau \Rightarrow \rho}(x) \end{aligned}$$

□

Lemma 4.2.4. Let σ and τ be types, F a function in $\llbracket \sigma \Rightarrow \tau \rrbracket$, x an element of $\llbracket \sigma \rrbracket$, and $n \in \mathbb{N}$, then $F(\text{addc}_{\sigma}(n, x)) \sqsupseteq_{\sigma} \text{addc}_{\tau}(n, F(x))$.

Proof. By induction on n . If $n = 0$, then $F(\text{addc}_{\sigma}(0, x)) = F(x) = \text{addc}_{\tau}(0, F(x))$ by Lemma 4.2.3(1). If $n = i + 1$, then by Lemma 4.2.3(2), $\text{addc}_{\sigma}(n, x) = \text{addc}_{\sigma}(1, \text{addc}_{\sigma}(i, x))$ which in turn is $\sqsupseteq_{\sigma} \text{addc}_{\sigma}(i, x)$ by Lemma 4.2.3(3). Thus, by strong monotonicity of F , we have

$$F(\text{addc}_{\sigma}(n, x)) \sqsupseteq_{\tau} F(\text{addc}_{\sigma}(i, x)) \quad (4.1)$$

The induction hypothesis gives $F(\text{addc}_{\sigma}(i, x)) \sqsupseteq_{\tau} \text{addc}_{\tau}(i, F(x))$, which combined with Equation (4.1) gives us $F(\text{addc}_{\sigma}(n, x)) \sqsupseteq_{\tau} \text{addc}_{\tau}(i, F(x))$. Furthermore, Lemma 4.2.3(4)

implies $F(\text{addc}_\sigma(n, x)) \sqsupseteq_\tau \text{addc}_\tau(1, \text{addc}_\tau(i, F(x)))$. By Lemma 4.2.3(2) we thus have $F(\text{addc}_\sigma(n, x)) \sqsupseteq_\tau \text{addc}_\tau(i + 1, F(x)) = \text{addc}_\sigma(n, F(x))$. \square

Lemma 4.2.5. For all types σ and all $x \in \llbracket \sigma \rrbracket$: $x \sqsupseteq_\sigma \text{addc}_\sigma(\text{costof}_\sigma(x), \mathbf{0}_\sigma)$.

Proof. By induction on the structure of σ . For the base case, σ is a base type $\iota \in \mathbb{B}$. So

$$\begin{aligned} x &= \langle x_1, x_2, \dots, x_{K(\iota)} \rangle \\ &\sqsupseteq_\iota \langle x_1, 0, \dots, 0 \rangle \\ &= \text{addc}_\iota(x_1, \langle 0, \dots, 0 \rangle) \\ &= \text{addc}_\iota(\text{costof}_\iota(x), \mathbf{0}_\iota) \end{aligned}$$

For the inductive step, we consider the case $\sigma = \tau \Rightarrow \rho$. In this case, $x = d \mapsto x(d)$ (extensionally), which by the induction hypothesis is $\sqsupseteq_{\tau \Rightarrow \rho} d \mapsto \text{addc}_\rho(\text{costof}_\rho(x(d)), \mathbf{0}_\rho)$. On the other hand,

$$\begin{aligned} \text{addc}_\sigma(\text{costof}_\sigma(x), \mathbf{0}_\sigma) &= d \mapsto \text{addc}_\rho(\text{costof}_\sigma(x), \mathbf{0}_\sigma(d)) \\ &= d \mapsto \text{addc}_\rho(\text{costof}_\sigma(x), \text{addc}_\rho(\text{costof}_\tau(d), \mathbf{0}_\rho)) \\ &= d \mapsto \text{addc}_\rho(\text{costof}_\sigma(x) + \text{costof}_\tau(d), \mathbf{0}_\rho) \end{aligned}$$

where the last equality follows from Lemma 4.2.3(2). Notice that by monotonicity of addc_ρ , Lemma 4.2.2(4), it suffices to prove that, for all d ,

$$\text{costof}_\rho(x(d)) \geq \text{costof}_\sigma(x) + \text{costof}_\tau(d)$$

To see this, note that by the induction hypothesis, $d \sqsupseteq_\tau \text{addc}_\tau(\text{costof}_\tau(d), \mathbf{0}_\tau)$. Therefore, $x(d) \sqsupseteq_\rho x(\text{addc}_\tau(\text{costof}_\tau(d), \mathbf{0}_\tau))$ by monotonicity of x . By Lemma 4.2.4 we have $x(d) \sqsupseteq_\rho \text{addc}_\rho(\text{costof}_\tau(d), x(\mathbf{0}_\tau))$. Hence, by monotonicity of costof_ρ given by Lemma 4.2.2(3), $\text{costof}_\rho(x(d)) \geq \text{costof}_\rho(\text{addc}_\rho(\text{costof}_\tau(d), x(\mathbf{0}_\tau)))$. Note that by Lemma 4.2.3(5),

$$\text{costof}_\rho(\text{addc}_\rho(\text{costof}_\tau(d), x(\mathbf{0}_\tau))) = \text{costof}_\tau(d) + \text{costof}_\rho(x(\mathbf{0}_\tau))$$

which in turn is equal to $\text{costof}_\tau(d) + \text{costof}_\sigma(x)$. Finally we have obtained the required inequality $\text{costof}_\rho(x(d)) \geq \text{costof}_\sigma(x) + \text{costof}_\tau(d)$. \square

Lemma 4.2.6. For $F \in \llbracket \sigma \Rightarrow \tau \rrbracket$ and $x \in \llbracket \sigma \rrbracket$ we have: $\text{costof}_\tau(F(x)) \geq \text{costof}_\sigma(x)$.

Proof. Let $n := \text{costof}_\sigma(x)$. By Lemma 4.2.5, we get $x \sqsupseteq_\sigma \text{addc}_\sigma(\text{costof}_\sigma(x), \mathbf{0}_\sigma) = \text{addc}_\sigma(n, \mathbf{0}_\sigma)$. Hence, by monotonicity of F , $F(x) \sqsupseteq_\tau F(\text{addc}_\sigma(n, \mathbf{0}_\sigma))$. By Lemma 4.2.4, this implies that $F(x) \sqsupseteq_\tau \text{addc}_\tau(n, F(\mathbf{0}_\sigma))$. Since costof_τ is strict in its first argument by Lemma 4.2.2(3), we thus have $\text{costof}_\tau(F(x)) \sqsupseteq \text{costof}_\sigma(\text{addc}_\tau(n, F(\mathbf{0}_\sigma)))$, which is $\sqsupseteq n$ by Lemma 4.2.3(5). \square

4.2.2 Making a MakeSM

We can use these functions to for instance create candidates for $\text{MakeSM}_{\sigma,\tau}$. While many suitable definitions are possible, we will particularly consider the following:

Definition 4.2.7. For types σ, τ , and F a weakly monotonic function in $\langle\sigma\rangle \longrightarrow \langle\tau\rangle$, let:

$$\Phi_{\sigma,\tau}(F) = \begin{cases} d \mapsto \text{addc}_{\sigma \Rightarrow \tau}(1, F(d)) & \text{if } F \text{ is in } \langle\sigma \Rightarrow \tau\rangle \\ d \mapsto \text{addc}_{\sigma \Rightarrow \tau}(\text{costof}_{\sigma}(d) + 1, F(d)) & \text{otherwise} \end{cases}$$

Then $\Phi_{\sigma,\tau}$ is a (σ, τ) -monotonicity function. To see this, the most challenging part is proving that $\Phi_{\sigma,\tau}(F) \sqsupset \Phi_{\sigma,\tau}(G)$ if $F \sqsupset G$ and $F \in \langle\sigma \Rightarrow \tau\rangle$ while G is a constant function. We can prove this using the result that $x \sqsupset y$ implies $\text{addc}(1, x) \sqsupseteq y$ for all x, y .

Lemma 4.2.8. Let σ, τ be simple types. Then $\Phi_{\sigma,\tau}$ is a (σ, τ) -monotonicity function.

Proof. First, we must see that $\Phi_{\sigma,\tau}$ maps each element of $C_{\sigma,\tau}$ to an element of $\langle\sigma, \tau\rangle$. Thus, let $F \in C_{\sigma,\tau}$. There are two cases:

- F is a constant function in $\langle\sigma\rangle \longrightarrow \langle\tau\rangle$.

Then $\Phi_{\sigma,\tau}(F)$ is the function $d \mapsto \text{addc}_{\tau}(\text{costof}_{\sigma}(d) + 1, F(d))$. Since $F \in \langle\sigma\rangle \longrightarrow \langle\tau\rangle$ we have $F(d) \in \langle\tau\rangle$ so $\text{addc}_{\tau}(\text{costof}_{\sigma}(d) + 1, F(d)) \in \langle\tau\rangle$ by Lemma 4.2.2(2); hence, $d \mapsto \text{addc}_{\tau}(\text{costof}_{\sigma}(d) + 1, F(d)) \in \langle\sigma\rangle \longrightarrow \langle\tau\rangle$.

It remains to be seen that this function is (a) weakly monotonic, and (b) strict in its only argument. We show only the latter; the former is very similar.

Let $x, y \in \langle\sigma\rangle$ with $x \sqsupset_{\sigma} y$. Then by Lemma 4.2.2(3), $\text{costof}_{\sigma}(x) > \text{costof}_{\sigma}(y)$, which implies $\text{costof}_{\sigma}(x) + 1 > \text{costof}_{\sigma}(y) + 1$ as well. Moreover, since F is constant, we have $F(x) = F(y)$, so certainly $F(x) \sqsupseteq_{\tau} F(y)$. Thus, by Lemma 4.2.2(4), we have $\text{addc}_{\tau}(\text{costof}_{\sigma}(x) + 1, F(x)) \sqsupset_{\tau} \text{addc}_{\tau}(\text{costof}_{\sigma}(y) + 1, F(y))$.

- F is a function in $\langle\sigma \Rightarrow \tau\rangle$; that is, a strongly monotonic function in $\langle\sigma\rangle \longrightarrow \langle\tau\rangle$. Then $\Phi_{\sigma,\tau}(F)$ is the function $d \mapsto \text{addc}_{\tau}(1, F(d))$. By Lemma 4.2.2(2) this function is indeed in $\langle\sigma\rangle \longrightarrow \langle\tau\rangle$. To see that it is monotonic, suppose that $x \sqsupset_{\sigma} y$; the case for $x \sqsupseteq_{\sigma} y$ is similar. Then $F(x) \sqsupset_{\tau} F(y)$ by strong monotonicity of F . By Lemma 4.2.2(4), $\text{addc}_{\tau}(1, F(x)) \sqsupset_{\tau} \text{addc}_{\tau}(1, F(y))$ as required.

Second, we prove that $\Phi_{\sigma,\tau}$ is strongly monotonic. That is, for $F, G \in C_{\sigma,\tau}$: (a) if $F(x) \sqsupset_{\tau} G(x)$ for all $x \in \langle\sigma\rangle$ then $\Phi_{\sigma,\tau}(F) \sqsupset_{\sigma \Rightarrow \tau} \Phi_{\sigma,\tau}(G)$; (b) if $F(x) \sqsupset_{\tau} G(x)$ for all $x \in \langle\sigma\rangle$ then $\Phi_{\sigma,\tau}(F) \sqsupset_{\sigma \Rightarrow \tau} \Phi_{\sigma,\tau}(G)$. We will only show (b); the proof of (a) is parallel. There are four cases to consider:

- F, G are both constant functions. Then $\Phi_{\sigma,\tau}(F) = d \mapsto \text{addc}_{\tau}(\text{costof}_{\sigma}(d) + 1, F(d)) \sqsupset_{\sigma \Rightarrow \tau} d \mapsto \text{addc}_{\tau}(\text{costof}_{\sigma}(d) + 1, G(d)) = \Phi_{\sigma,\tau}(G)$ by Lemma 4.2.2(4) and because $F(d) \sqsupset_{\tau} G(d)$.

- F, G are both in $(\sigma \Rightarrow \tau)$. Then we must see that $d \mapsto \text{addc}_\tau(1, F(d)) \sqsupset_\tau d \mapsto \text{addc}_\tau(1, G(d))$, so that $\text{addc}_\tau(1, F(d)) \sqsupset_\tau \text{addc}_\tau(1, G(d))$ for all d . This holds by Lemma 4.2.2(4) because $F(d) \sqsupset_\tau G(d)$ (by definition of $F \sqsupset G$).
- F is in $(\sigma \Rightarrow \tau)$ and G is constant. Then we must see that for all $d \in (\sigma)$ we have: $\text{addc}_\tau(1, F(d)) \sqsupset_\tau \text{addc}_\tau(\text{costof}_\sigma(d) + 1, G(d))$. By monotonicity of addc_τ (Lemma 4.2.2(4)) and by Lemma 4.2.3(2) it suffices to show that $F(d) \sqsupset_\tau \text{addc}_\tau(\text{costof}_\sigma(d), G(d))$.

So consider a fixed d . By Lemma 4.2.5, $d \sqsupset_\sigma \text{addc}_\sigma(\text{costof}_\sigma(d), \mathbf{0}_\sigma)$. Hence, by monotonicity of F we have $F(d) \sqsupset_\tau F(\text{addc}_\sigma(\text{costof}_\sigma(d), \mathbf{0}_\sigma))$. By Lemma 4.2.4, then $F(\text{addc}_\sigma(\text{costof}_\sigma(d), \mathbf{0}_\sigma)) \sqsupset_\tau \text{addc}_\tau(\text{costof}_\sigma(d), F(\mathbf{0}_\sigma))$ by Lemma 4.2.2(4). By assumption, $F(\mathbf{0}_\sigma) \sqsupset_\tau G(\mathbf{0}_\sigma)$, and since G is a constant function, $G(\mathbf{0}_\sigma) = G(d)$. Hence, $F(d) \sqsupset_\tau \text{addc}_\tau(\text{costof}_\sigma(d), G(d))$.

- F is a constant function and G is strongly monotonic. In fact, this cannot happen. To see this, let $m := \text{costof}_\tau(F(\mathbf{0}_\sigma))$. Note that $F(\mathbf{0}_\sigma) = F(\text{addc}_\sigma(m, \mathbf{0}_\sigma))$ since F is constant, $\sqsupset_\tau G(\text{addc}_\sigma(m, \mathbf{0}_\sigma))$ since $F \sqsupset G$, which $\sqsupset_\tau \text{addc}_\tau(m, G(\mathbf{0}_\sigma))$ by Lemma 4.2.4. Hence, $F(\mathbf{0}_\sigma) \sqsupset_\tau \text{addc}_\tau(m, G(\mathbf{0}_\sigma))$, so by Lemma 4.2.2(3) we have $m = \text{costof}_\tau(F(\mathbf{0}_\sigma)) > \text{costof}_\tau(\text{addc}_\tau(m, G(\mathbf{0}_\sigma))) = m + \text{costof}_\tau(G(\mathbf{0}_\sigma)) \geq m$ by Lemma 4.2.3(5). This gives the required contradiction. \square

4.2.3 Orienting Beta and Eta

Lemma 4.2.9. If $\text{MakeSM}_{\sigma, \tau} = \Phi_{\sigma, \tau}$ then $\llbracket (\lambda x. s) t \rrbracket \sqsupset_\tau \llbracket s[x := t] \rrbracket$, for $s : \tau, t : \sigma, x \in \mathbb{X}_\sigma$.

Proof. We have either

1. $\llbracket (\lambda x. s) t \rrbracket_\alpha^{\mathcal{J}} = \text{addc}_\tau(\text{costof}_\sigma(\llbracket t \rrbracket_\alpha) + 1, \llbracket s \rrbracket_{\alpha[x := \llbracket t \rrbracket]})$; or
2. $\llbracket (\lambda x. s) t \rrbracket_\alpha^{\mathcal{J}} = \text{addc}_\tau(1, \llbracket s \rrbracket_{\alpha[x := \llbracket t \rrbracket]})$

By Lemma 4.2.3(3) we have $\llbracket (\lambda x. s) t \rrbracket_\alpha^{\mathcal{J}} \sqsupset_\tau \llbracket s \rrbracket_{\alpha[x := \llbracket t \rrbracket]}$ in both cases. By Lemma 4.1.12, $\llbracket s \rrbracket_{\alpha[x := \llbracket t \rrbracket]}} = \llbracket s[x := b] \rrbracket_\alpha^{\mathcal{J}}$. This completes the proof. \square

In examples in the remainder of this paper, we will assume that $\text{MakeSM}_{\sigma, \tau} = \Phi_{\sigma, \tau}$. With this choice, we do not only orient the β -rule (and thus satisfy Item 1 of the compatibility conditions), but also the η -reduction rules mentioned in Remark 2.1.16.

Lemma 4.2.10. If $\text{MakeSM}_{\sigma, \tau} = \Phi_{\sigma, \tau}$ then for any $F \in \mathbb{X}_{\sigma \Rightarrow \tau}$ we have: $\llbracket \lambda x. F x \rrbracket \sqsupset_{\sigma \Rightarrow \tau} \llbracket F \rrbracket$.

Proof. Since $F \neq x$, we have that $d \mapsto \llbracket Fx \rrbracket_{\alpha[x:=d]} = d \mapsto \alpha(F)(d)$, which by extensionality is exactly $\alpha(F)$. Since $\alpha(F)$ is monotonic by assumption on α , we have

$$\begin{aligned} \llbracket \lambda x. Fx \rrbracket_{\alpha} &= \Phi_{\sigma, \tau}(d \mapsto \llbracket Fx \rrbracket_{\alpha[x:=d]}) \\ &= \Phi_{\sigma, \tau}(d \mapsto \llbracket Fd \rrbracket_{\alpha}) \\ &= \Phi_{\sigma, \tau}(\alpha(F)) \\ &= \text{addc}_{\sigma, \tau}(1, \alpha(F)) \\ &\sqsupset_{\sigma \Rightarrow \tau} \alpha(F) \\ &= \llbracket F \rrbracket_{\alpha}, \end{aligned}$$

where the inequality step follows from Lemma 4.2.3(3). \square

4.3 Creating strongly monotonic interpretation functions

We can use Theorem 4.1.13 to obtain bounds on the derivation heights of given terms. However, to achieve this, we must find an interpretation function \mathcal{J} , and prove that each \mathcal{J}_f is in (σ) whenever $f : \sigma$. We now explore ways to construct such strongly monotonic functions. It turns out to be useful to also consider *weakly* monotonic functions. In the following, we will write “ f is $\text{wm}(A_1, \dots, A_k; B)$ ” as a shorthand for f is a weakly monotonic function in $A_1 \Rightarrow \dots \Rightarrow A_k \Rightarrow B$.

Lemma 4.3.1. Let x^1, \dots, x^k be variables ranging over $(\sigma_1), \dots, (\sigma_k)$ respectively; we shortly denote this sequence by \vec{x} . We let $\overrightarrow{(\sigma)}$ denote the sequence $(\sigma_1), \dots, (\sigma_k)$. Then:

1. if $F(\vec{x}) = x^i$ then F is $\text{wm}(\overrightarrow{(\sigma)}; (\sigma_i))$, and F is strict in argument i ;
2. if $F(\vec{x}) = x^i(F_1(\vec{x}), \dots, F_n(\vec{x}))$, $\sigma_i = \tau_1 \Rightarrow \dots \Rightarrow \tau_n \Rightarrow \rho$, and each F_j is in the set $\text{wm}(\overrightarrow{(\sigma)}; (\tau_j))$ then F is $\text{wm}(\overrightarrow{(\sigma)}; (\rho))$ and for all $p \in \{1, \dots, k\}$: F is strict in argument p if $p = i$ or some F_j is strict in argument p ;
3. if $F(\vec{x}) = \langle G_1(\vec{x}), \dots, G_{K(\iota)}(\vec{x}) \rangle$ and each G_j is $\text{wm}(\overrightarrow{(\sigma)}; \mathbb{N})$ then F is $\text{wm}(\overrightarrow{(\sigma)}; (\iota))$, and for all $p \in \{1, \dots, k\}$: F is strict in argument p if G_1 is.

The last result uses functions mapping to \mathbb{N} ; these can be constructed using the observations:

4. if $G(\vec{x}) = n$ for some $n \in \mathbb{N}$ then G is $\text{wm}(\overrightarrow{(\sigma)}; \mathbb{N})$;
5. if $G(\vec{x}) = x_j^i$ and $\sigma_i = \iota \in \mathbb{B}$ and $1 \leq j \leq K(\iota)$, then G is $\text{wm}(\overrightarrow{(\sigma)}; \mathbb{N})$, and G is strict in argument i if $j = 1$;

6. if $G(\vec{x}) = f(G_1(\vec{x}), \dots, G_n(\vec{x}))$ and all G_j are $\text{wm}(\overrightarrow{(\|\sigma\|)}; \mathbb{N})$ and f is $\text{wm}(\mathbb{N}, \dots, \mathbb{N}; \mathbb{N})$, then G is $\text{wm}(\overrightarrow{(\|\sigma\|)}; \mathbb{N})$, and for all $p \in \{1, \dots, k\}$: G is strict in argument p if, for some $j \in \{1, \dots, n\}$: G_j is strict in argument p and f is strict in argument j ;
7. if $G(\vec{x}) = F(\vec{x})_j$ and F is $\text{wm}(\overrightarrow{(\|\sigma\|)}; (\|\iota\|))$ and $1 \leq j \leq K(\iota)$ then G is $\text{wm}(\overrightarrow{(\|\sigma\|)}; \mathbb{N})$ and if $j = 1$ then for all $p \in \{1, \dots, k\}$: G is strict in argument p if F is.

Proof Sketch. We easily see that in each case, F or G is in the given function space. To show weak monotonicity, assume given both \vec{x} and \vec{y} such that each $x^i \sqsupseteq y^i$; we then check for all cases that $F(\vec{x}) \sqsupseteq F(\vec{y})$, or $G(\vec{x}) \geq G(\vec{y})$. For the strictness conditions, we assume that $x^p \sqsupset y^p$ and similarly check all cases. \square

The reader may recognise items (4–6): these largely correspond to the sufficient conditions for a weakly monotonic function S in Lemma 3.3.4. For the function f in item (6), we could for instance choose $+$, $*$ or \max , where $+$ is strict in all arguments. However, we can get beyond Lemma 3.3.4 by using the other items; for example, applying variables to each other.

Now, if a function f is $\text{wm}(\overrightarrow{(\|\sigma\|)}; (\|\tau\|))$ and f is strict in all its arguments, then we easily see that the function $d_1 \mapsto \dots \mapsto d_k \mapsto f(d_1, \dots, d_k)$ is an element of $(\|\sigma_1 \Rightarrow \dots \Rightarrow \sigma_k \Rightarrow \tau\|)$. To illustrate how this can be used in practice, we show monotonicity of \mathcal{J}_{map} of Example 4.1.14:

Example 4.3.2. Suppose

$$\mathcal{J}_{\text{map}}(F, q) = (F(\langle q_c, q_m \rangle)_c + q_l * F(\langle q_c, q_m \rangle)_c + q_l + 1, q_l, F(\langle q_c, q_m \rangle)_l)$$

By (5), the functions $(F, q) \mapsto q_i$ are $\text{wm}(\|\text{nat} \Rightarrow \text{nat}\|, (\|\text{list}\|); \mathbb{N})$ for $i \in \{c, l, m\}$ and moreover, $(F, q) \mapsto q_c$ is strict in argument 2. Hence, by (3), $(F, q) \mapsto \langle q_c, q_m \rangle$ is $\text{wm}(\|\text{nat} \Rightarrow \text{nat}\|, (\|\text{list}\|); (\|\text{nat}\|))$ and strict in argument 2. Therefore, by (2), $(F, q) \mapsto F(\langle q_c, q_m \rangle)$ is $\text{wm}(\|\text{nat} \Rightarrow \text{nat}\|, (\|\text{list}\|); (\|\text{nat}\|))$ and strict in both arguments. Hence, by (7), $(F, q) \mapsto F(\langle q_c, q_m \rangle)_c$ and $(F, q) \mapsto F(\langle q_c, q_m \rangle)_l$ are $\text{wm}(\|\text{nat} \Rightarrow \text{nat}\|, (\|\text{list}\|); \mathbb{N})$ and the former is strict in both arguments. Continuing like this, it is not hard to see how we can iteratively prove that

$$(F, q) \mapsto (F(\langle q_c, q_m \rangle)_c + q_l * F(\langle q_c, q_m \rangle)_c + q_l + 1, q_l, F(\langle q_c, q_m \rangle)_l)$$

is $\text{wm}(\|\text{nat} \Rightarrow \text{nat}\|, (\|\text{list}\|); (\|\text{list}\|))$ and strict in both arguments, which immediately gives $\mathcal{J}_{\text{map}} \in (\|\text{nat} \Rightarrow \text{nat}\| \Rightarrow \text{list} \Rightarrow \text{list})$.

In practice, it is usually not needed to write such an elaborate proof: Lemma 4.3.1 essentially tells us that if a function is built exclusively using variables and variable applications, projections $F(\vec{x})_j$, constants, and weakly monotonic operators over the

natural numbers, then that function is weakly monotonic; we only need to check that the cost component indeed increases if one of the variables x^i is increased.

Unfortunately, while Lemma 4.3.1 is useful for rules like the ones for `map`, it is not enough to handle functions like `foldl`, where the same function is repeatedly applied on a term. As `foldl`-like functions occur more often in higher-order rewriting, we should also address this.

To handle iteration, we define: for a function $Q \in A \rightarrow A$ and natural number n , let $Q^n(a)$ indicate repeated function application; that is, $Q^0(a) = a$ and $Q^{n+1}(a) = Q^n(Q(a))$.

Lemma 4.3.3. Suppose F is $\text{wm}(\overrightarrow{\langle\sigma\rangle}, \langle\tau \Rightarrow \tau\rangle)$ and G is $\text{wm}(\overrightarrow{\langle\sigma\rangle}; \mathbb{N})$. Suppose that for all $u_1 \in \langle\sigma_1\rangle, \dots, u_k \in \langle\sigma_k\rangle$ and $v \in \langle\tau\rangle$ we have $F(u_1, \dots, u_k, v) \sqsupseteq_\tau v$. Under these assumptions, the function: $(x_1, \dots, x_k) \mapsto F(x_1, \dots, x_k)^{G(x_1, \dots, x_k)}$ is $\text{wm}(\overrightarrow{\langle\sigma\rangle}, \langle\tau \Rightarrow \tau\rangle)$.

Proof. Let Q indicate the function $(x_1, \dots, x_k, y) \mapsto F(x_1, \dots, x_k)^{G(x_1, \dots, x_k)}(y)$. We first note that the image of Q is contained in $\langle\tau \Rightarrow \tau\rangle$. So let $u_1 \in \langle\sigma_1\rangle, \dots, u_k \in \langle\sigma_k\rangle$. Since $F \in \langle\tau \Rightarrow \tau\rangle \subseteq \langle\tau\rangle \rightarrow \langle\tau\rangle$, by definition of repeated function application the function $F(u_1, \dots, u_k)^{G(u_1, \dots, u_k)}$ is in $\langle\tau\rangle \rightarrow \langle\tau\rangle$ as well. We must show that for all $v, v' \in \langle\tau\rangle$, if $v \sqsupseteq_\tau v'$ then

$$\begin{aligned} Q(u_1, \dots, u_k, v) &= F(u_1, \dots, u_k)^{G(u_1, \dots, u_k)}(v) \\ &\sqsupseteq_\tau F(u_1, \dots, u_k)^{G(u_1, \dots, u_k)}(v') \\ &= Q(u_1, \dots, u_k, v') \end{aligned}$$

We will show this by induction on the natural number $G(u_1, \dots, u_k)$.

- If $G(u_1, \dots, u_k) = 0$ then $F(u_1, \dots, u_k)^{G(u_1, \dots, u_k)}(v) = v \sqsupseteq_\tau v'$ by assumption, which $= F(u_1, \dots, u_k)^{G(u_1, \dots, u_k)}(v')$.
- If $G(u_1, \dots, u_k) = n + 1$ then note that, because $F(u_1, \dots, u_k) \in \langle\tau \Rightarrow \tau\rangle$ (so this defines a strongly monotonic function), we have $F(u_1, \dots, u_k, v) \sqsupseteq_\tau F(u_1, \dots, u_k, v')$. Hence,

$$F(u_1, \dots, u_k)^{G(u_1, \dots, u_k)}(v) = F(u_1, \dots, u_k)^n(F(u_1, \dots, u_k, v))$$

(by definition), $\sqsupseteq_\tau F(u_1, \dots, u_k)^n(F(u_1, \dots, u_k, v'))$ by the induction hypothesis. This suffices, as this equals $F(u_1, \dots, u_k)^{G(u_1, \dots, u_k)}(v')$.

It remains to be shown that Q is weakly monotonic in its first k arguments. So suppose $u'_1 \in \langle\sigma_1\rangle, \dots, u'_k \in \langle\sigma_k\rangle$ and each $u_i \sqsupseteq_{\sigma_i} u'_i$. We must show that $Q(u_1, \dots, u_k) \sqsupseteq_{\tau \Rightarrow \tau} Q(u'_1, \dots, u'_k)$. We achieve this by proving that

$$\text{for all } n, m \text{ with } n \geq m, F(u_1, \dots, u_k)^n \sqsupseteq_{\tau \Rightarrow \tau} F(u'_1, \dots, u'_k)^m \quad (4.2)$$

Then $Q(u_1, \dots, u_k) \sqsupseteq_{\tau \Rightarrow \tau} Q(u'_1, \dots, u'_k)$ follows because $G(u_1, \dots, u_k) \geq G(u'_1, \dots, u'_k)$ is a consequence of the weak monotonicity of G .

We prove Statement 4.2 by induction on n . We have the following cases.

1. If $n = 0$, then also $m = 0$. For all $v \in \langle \tau \rangle$ we have $F(u_1, \dots, u_k)^n(v) = v = F(u'_1, \dots, u'_k)^m$.
2. For the inductive case, we let $n = i + 1 = m$ and consider $v \in \langle \tau \rangle$. We must show

$$F(u_1, \dots, u_k)^i(F(u_1, \dots, u_k, v)) \sqsupseteq_{\tau} F(u'_1, \dots, u'_k)^i(F(u'_1, \dots, u'_k, v))$$

But we know that $F(u_1, \dots, u_k, v) \sqsupseteq_{\tau} F(u'_1, \dots, u'_k, v)$; this inequality holds because F is in $\text{wm}(\overrightarrow{\langle \sigma \rangle}; \langle \tau \Rightarrow \tau \rangle)$. Since we have already seen that, for all i , $F(u_1, \dots, u_k)^i \in \langle \tau \Rightarrow \tau \rangle$ and is therefore also a weakly monotonic function, therefore $F(u_1, \dots, u_k)^i(F(u_1, \dots, u_k, v)) \sqsupseteq_{\tau} F(u_1, \dots, u_k)^i(F(u'_1, \dots, u'_k, v))$. By the induction hypothesis, we get $F(u_1, \dots, u_k)^i \sqsupseteq_{\tau \Rightarrow \tau} F(u'_1, \dots, u'_k)^i$. By definition, this means that we have

$$\begin{aligned} F(u_1, \dots, u_k)^i(F(u'_1, \dots, u'_k, v)) &\sqsupseteq_{\tau} F(u'_1, \dots, u'_k)^i(F(u'_1, \dots, u'_k, v)) \\ &= F(u'_1, \dots, u'_k)^m(v) \end{aligned}$$

The proof is complete by transitivity of \sqsupseteq_{τ} .

3. Now we consider the case where $n = i + 1$ and $i \geq m$. As before, we take $v \in \langle \tau \rangle$. We must show

$$F(u_1, \dots, u_k)^i(F(u_1, \dots, u_k, v)) \sqsupseteq_{\tau} F(u'_1, \dots, u'_k)^m(v)$$

By assumption, $F(u_1, \dots, u_k, v) \sqsupseteq_{\tau} v$. As we saw before, $F(u_1, \dots, u_k)^i$ also weakly monotonic, hence the following inequality holds:

$$F(u_1, \dots, u_k)^i(F(u_1, \dots, u_k, v)) \sqsupseteq_{\tau} F(u_1, \dots, u_k)^i(v)$$

By the induction hypothesis, we finally get $F(u_1, \dots, u_k)^i(v) \sqsupseteq_{\tau} F(u'_1, \dots, u'_k)^m(v)$.

□

With this in hand, we can orient the foldl rules of Example 2.1.11.

Example 4.3.4. For $F \in \langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$ and $x, y \in \langle \text{nat} \rangle$, let `Helper` be defined by:

$$\text{Helper}(F, y, x) = \langle F(x, y)_{\text{c}}, \max(x_{\text{s}}, F(x, y)_{\text{s}}) \rangle$$

Then Helper is $\text{wm}(\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle, \langle \text{nat} \rangle, \langle \text{nat} \rangle; \langle \text{nat} \rangle)$ and strict in its third argument by Lemma 4.3.1 (1, 2, 3, 6, 7). Hence, Helper is a member of

$$\text{wm}(\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle, \langle \text{nat} \rangle; \langle \text{nat} \Rightarrow \text{nat} \rangle).$$

Since, in general, $\text{costof}_{\text{nat}}(F(x, y)) \geq \text{costof}_{\text{nat}}(x)$ by Lemma 4.2.6, we have that $\text{Helper}(F, y, x) \sqsupseteq_{\text{nat}} x$. Therefore, by using Lemma 4.3.3, we see that the function

$$(F, z, q) \mapsto \text{Helper}(F, \langle q_c, q_m \rangle)^{q_l}(z)$$

is weakly monotonic, and strict in its second argument. This ensures that the following function is strongly monotonic:

$$\mathcal{J}_{\text{foldl}} = \lambda F z q. \text{Helper}(F, \langle q_c, q_m \rangle)^{q_l}(\langle 1 + q_c + q_l + F(\mathbf{0}_{\text{nat}}, \mathbf{0}_{\text{nat}})_c + z_c, z_s \rangle)$$

Now we can show that this interpretation function is compatible with the rules for foldl in Example 2.1.11.

1. First, we have

$$\begin{aligned} \llbracket \text{foldl } F \ z \ [] \rrbracket &= \langle 1 + F(\mathbf{0}_{\text{nat}}, \mathbf{0}_{\text{nat}})_c + z_c, z_s \rangle \\ &\sqsupseteq_{\text{nat}} \langle z_c, z_s \rangle \\ &= z, \end{aligned}$$

which orients the first rule.

2. For the second rule, we use the general property that

$$F(\text{addc}(n, x), y) \sqsupseteq \text{addc}(n, F(x, y)) \tag{4.3}$$

provided by Lemma 4.2.4. We define A and B as the following

$$A := \langle x_c + q_c, \max(x_s, q_m) \rangle \quad B := 1 + q_c + q_l + F(\mathbf{0}_{\text{nat}}, \mathbf{0}_{\text{nat}})_c + z_c$$

Then we have $\llbracket \text{foldl}(F, z, x : q) \rrbracket = \text{Helper}(F, A)^{q_l+1}(\langle B + x_c + 1, z_s \rangle)$, which:

$$\begin{aligned}
& \sqsupset_{\text{nat}} \text{Helper}(F, A)^{q_l}(\text{Helper}(F, A, \langle B, z_s \rangle)) \\
& \quad \text{because } \langle B + x_c + 1, z_s \rangle \sqsupset_{\text{nat}} \langle B, z_s \rangle \\
& \sqsupset_{\text{nat}} \text{Helper}(F, A)^{q_l}(F(\langle B, z_s \rangle, A)) \\
& \quad \text{because } \text{Helper}(F, n, m) \sqsupset_{\text{nat}} F(m, n) \\
& \sqsupset_{\text{nat}} \text{Helper}(F, \langle q_c, q_m \rangle)^{q_l}(F(\langle B, z_s \rangle, x)) \\
& \quad \text{because } A \sqsupset_{\text{nat}} \langle q_c, q_m \rangle \text{ and } A \sqsupset_{\text{nat}} x \\
& \sqsupset_{\text{nat}} \text{Helper}(F, \langle q_c, q_m \rangle)^{q_l}(\text{add}_{\text{nat}}(1 + q_c + q_l + F(\mathbf{0}_{\text{nat}}, \mathbf{0}_{\text{nat}})_c, F(z, x))) \\
& \quad \text{by Equation (4.3)} \\
& = \llbracket \text{foldl}(F, (F z x), q) \rrbracket.
\end{aligned}$$

The interpretation in Example 4.3.4 may *seem* too convoluted for practical use: it does not obviously tell us something like “ F is applied a linear number of times on terms whose size is bounded by n ”. However, its value becomes clear when we plug in specific bounds for F .

Example 4.3.5. The function `sum`, defined in Example 3.1.1, could alternatively be defined in terms of `foldl`: let $\text{sum}(q) \rightarrow \text{foldl}(\lambda xy. (\text{add } x \ y), 0, q)$. To find an interpretation for this function, we use the interpretation functions for `0`, `s`, `[]`, `cons` and `add` from Example 3.3.5. Then $\llbracket \lambda xy. (\text{add } x \ y) \rrbracket = d, e \mapsto (d_c + e_c + e_s + 3, d_s + e_s)$. We easily see that $\text{Helper}(\llbracket \lambda xy. (\text{add } x \ y) \rrbracket, \langle q_c, q_m \rangle, z) = \langle z_c + q_c + q_m + 3, z_s + q_m \rangle$. Importantly, the iteration variable z is used in a very innocent way: although its size is increased, this increase is by the same number (q_m) in every iteration step. Moreover, the length of z does not affect the evaluation cost. Hence, we can choose $\llbracket \text{sum}(q) \rrbracket = \langle 5 + q_c + q_l + q_l * (q_c + q_m + 3), q_l * q_m \rangle$. This is close to the interpretation from Example 3.3.5 but differs both in a small overhead for the β -reductions, and because our interpretation of `foldl` slightly overestimates the true cost.

This approach can be used to obtain bounds for any function that may be defined in terms of `foldl`, which includes many first-order functions. For example, with a small change to the signature of `foldl`, we could let $\text{rev}(q) = \text{foldl}(\lambda xy. (y : x), [], q)$; however, this would necessitate corresponding changes in the interpretation of `foldl`.

4.4 Finding Higher-Order Complexity Bounds

A key notion in complexity analysis of first-order rewriting is runtime complexity, which we studied in Chapter 3. In this section, we define a conservative notion of runtime complexity for higher-order term rewriting and show how higher-order strongly monotonic interpretations can be used to find runtime complexity bounds.

It is not obvious how this notion translates to the higher-order setting. It may be tempting to directly apply the definition to a TRS, but a “ground constructor term” (or perhaps “closed constructor term”) is not a natural concept in higher-order rewriting; it does not intuitively capture data. Moreover, we would like to create a *robust* notion which can be extended to simple functional programming languages, so which is not subject to minor language difference like whether partial application of function symbols is allowed.

Instead, there are two obvious ways to capture the idea of input in higher-order rewriting:

- *closed irreducible terms*: this includes all ground constructor terms, but also for instance $\lambda x. \text{add } 0 \ x$ (but not $\lambda x. \text{add } x \ 0$, since this can be rewritten following the rules in Example 3.1.1);
- *data*: this includes only ground constructor terms with no higher-order subterms.

As we observed in Example 4.1.1, the size of a higher-order term does not capture its behavior. Hence, a notion of runtime complexity using closed irreducible terms is not obviously meaningful—and might be closer to *derivational* complexity due to defined symbols inside abstractions. Therefore, we here take the conservative choice and consider *data terms*, as in Definition 2.1.13.

Recall that, in practice, a sort is defined by its data constructors. For example, `nat` is defined by `0` and `s`, and `list` by `[]` and `cons`. In typical examples of first- and higher-order term rewriting systems, rules are defined to exhaustively pattern match on all constructors for a sort.

With this definition, we can conservatively extend the original notion of runtime complexity, as in Section 3.1, to be applicable to both first- and higher-order term rewriting.

Note that if $f(d_1, \dots, d_k)$ is a basic term, then $f : \iota_1 \Rightarrow \dots \Rightarrow \iota_k \Rightarrow \tau$ with all ι_i base types. Hence, higher-order runtime complexity considers the same (first-order) notion of basic terms as the first-order case; terms such as $\text{map}(F, s)$ or even $\text{map}(\lambda x. s(x), [])$ are not basic. One might reasonably question whether such a first-order notion is useful when studying the complexity of *higher-order* term rewriting. However, we argue that it is: runtime complexity aims to address the length of computations that begin at a typical starting point. When performing a *full program* analysis of a TRS, the computation will still typically start in a basic term, for instance; the entry-point symbol `main` applied to some user input d_1, \dots, d_k .

Example 4.4.1 (Extrec). We consider a TRS from the Termination Problem Database, v11.0 [41].

$$\begin{array}{ll}
 x + 0 & \rightarrow x & \text{rec } 0 \ y \ F & \rightarrow y \\
 x + s(y) & \rightarrow s(x + y) & \text{rec } s(x) \ y \ F & \rightarrow F \ x \ (\text{rec } x \ y \ F) \\
 & & x \times y & \rightarrow \text{rec } y \ 0 \ (\lambda n. \lambda m. x + m)
 \end{array}$$

Here, $\text{rec} : \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}) \Rightarrow \text{nat}$. The only basic terms have the form $s^n 0 + s^m(0)$ or $s^n(0) \times s^m(0)$. Using our method, we obtain cubic runtime complexity; to be precise: $\mathcal{O}(m^2 * n)$.

$$\begin{array}{l}
 \mathcal{J}_0 = \langle 0, 0 \rangle \\
 \mathcal{J}_s = \lambda x. \langle x_c, x_s + 1 \rangle \\
 \mathcal{J}_+ = \lambda x y. \langle x_c + y_c + y_s + 1, x_s + y_s \rangle \\
 \mathcal{J}_\times = \lambda x y. \langle 1 + y_s * (x_c + y_c + x_s * (y_s + 1)) / 2 + 3, x_s * y_s \rangle \\
 \mathcal{J}_{\text{rec}} = \lambda x y F. \text{Helper}(x, F)^{x_s}(\langle 1 + x_c + y_c + x_s + F(\mathbf{0}_{\text{nat}}, \mathbf{0}_{\text{nat}})_c, y_s \rangle) \\
 \text{Helper}(x, F) = z \mapsto \langle F(x, z)_c, \max(z_s, F(x, z)_s) \rangle
 \end{array}$$

Checking compatibility conditions for this interpretation is analogous to the foldl case, see [69, Section A.3].

In the extra examples below we show that our technique can be used to derive general information about the complexity of higher-order systems (and additionally prove their termination, as we have seen in this chapter with the compatibility theorem).

Example 4.4.2 (Filter). The next example also comes from the Termination Problem Database, version 11.0 [41].

$$\begin{array}{ll}
 \text{rand}(x) & \rightarrow x & \text{filter}(F, []) & \rightarrow [] \\
 \text{rand}(s(x)) & \rightarrow \text{rand}(x) & \text{filter}(F, x : q) & \rightarrow \text{consif}(F \ x, x, \text{filter}(F, q)) \\
 \text{bool}(0) & \rightarrow \text{false} & \text{consif}(\text{true}, x, q) & \rightarrow x : q \\
 \text{bool}(s(0)) & \rightarrow \text{true} & \text{consif}(\text{false}, x, q) & \rightarrow q
 \end{array}$$

We let $\llbracket \text{nat} \rrbracket = \mathbb{N}^2$ and $\llbracket \text{list} \rrbracket = \mathbb{N}^3$ as before, and additionally let $\llbracket \text{boolean} \rrbracket = \mathbb{N}$ (so only a cost component and no size components). We let:

$$\begin{aligned}
\llbracket \text{true} \rrbracket &= \langle 0 \rangle & \llbracket \text{s}(x) \rrbracket &= \langle x_c, x_s + 1 \rangle \\
\llbracket \text{bool}(x) \rrbracket &= \langle x_c + 1 \rangle & & \\
\llbracket \text{false} \rrbracket &= \langle 0 \rangle & \llbracket [] \rrbracket &= \langle 0, 0, 0 \rangle \\
\llbracket \text{rand}(x) \rrbracket &= \langle 1 + x_c + x_s, x_s \rangle & & \\
\llbracket 0 \rrbracket &= \langle 0, 0 \rangle & \llbracket x : q \rrbracket &= \langle x_c + q_c, q_l + 1, \max(x_s, q_m) \rangle \\
\llbracket \text{consif}(z, x, q) \rrbracket &= \langle z_c + x_c + q_c + 1, q_l + 1, \max(x_s, q_m) \rangle & & \\
\llbracket \text{filter}(F, q) \rrbracket &= \langle 1 + (q_l + 1) * (2 + q_c + F(\langle q_c, q_m \rangle)_c), q_l, q_m \rangle . & &
\end{aligned}$$

It is easy to see that monotonicity requirements are satisfied. Notice that whenever q is a data term, then applying $\text{filter}(F)$ on q does not increase the list's size or maximum element. Therefore, we might think of the complexity of this operation as roughly $O(q_l * F(q_m))$. In Chapter 6, we deal with a more formal notion of "higher-order polynomial time".

Example 4.4.3. Our final example also comes from the termination problem database. This example seems to be designed to calculate a function's derivative. It is worth noting that all symbols other than der are constructors.

$$\begin{aligned}
\text{der}(\lambda x. y) &\rightarrow \lambda z. 0 \\
\text{der}(\lambda x. \text{sin}(x)) &\rightarrow \lambda z. \text{cos}(z) \\
\text{der}(\lambda x. x) &\rightarrow \lambda z. 1 \\
\text{der}(\lambda x. \text{cos}(x)) &\rightarrow \lambda z. \text{min}(\text{cos}(z)) \\
\text{der}(\lambda x. \text{add}(F x, G x)) &\rightarrow \lambda z. \text{add}(\text{der}(F) z, \text{der}(G) z) \\
\text{der}(\lambda x. \text{times}(F x, G x)) &\rightarrow \lambda z. \text{add}(\text{times}(\text{der}(F) z, G z), \text{times}(F z, \text{der}(G) z)) \\
\text{der}(\lambda x. \text{ln}(F x)) &\rightarrow \lambda z. \text{div}(\text{der}(F) z, F z)
\end{aligned}$$

With $\text{der} : (\text{real} \Rightarrow \text{real}) \Rightarrow \text{real} \Rightarrow \text{real}$. We let $\llbracket \text{real} \rrbracket = \mathbb{N}^3$ where the first component indicates cost, and the second and third component roughly indicate the number of plus/times/ln occurrences and the number of times/ln occurrences respectively. We will denote x_s for x_2 , and x_\star for x_3 . We use the following interpretation:

$$\begin{aligned}
\llbracket 0 \rrbracket &= \langle 0, 0, 0 \rangle & \llbracket \text{add}(x, y) \rrbracket &= \langle x_c + y_c, x_s + y_s + 1, x_\star + y_\star \rangle \\
\llbracket 1 \rrbracket &= \langle 0, 0, 0 \rangle & \llbracket \text{times}(x, y) \rrbracket &= \langle x_c + y_c, x_s + y_s + 1, x_\star + y_\star + 1 \rangle \\
\llbracket \text{cos}(x) \rrbracket &= x & \llbracket \text{ln}(x) \rrbracket &= \langle x_c, x_s + 1, x_\star + 1 \rangle \\
\llbracket \text{sin}(x) \rrbracket &= x & &
\end{aligned}$$

Finally, we set the following interpretation for `der`, by setting $H(z) := 1 + F(z)_c + 2 * F(z)_s + F(z)_\star * F(z)_c$

$$\llbracket \text{der}(F) \rrbracket = z \mapsto \langle H(z), F(z)_s * (F(z)_\star + 1), F(z)_\star * (F(z)_\star + 1) \rangle$$

Notice that since the only defined symbol in this system is `der`, its runtime complexity is actually $\mathcal{O}(1)$. No “start” term is included in the system. However, this interpretation does show the termination of the system, and given a specific function of interest F built from the basic functions `cos`, `sin`, \dots we can utilize our interpretation to bound the derivation height of terms of the shape `der(F)`.

Notice that with this notion of runtime complexity the results we proved in Lemmas 3.3.9 and 3.6.3 easily carry over to this higher-order setting.

4.5 Conclusion

In this chapter, we have introduced tuple interpretations for higher-order term rewriting. This includes providing a new definition of strongly monotonic algebras, a compatibility theorem, a function `MakeSM` that orients β - and η -reductions, and several lemmas to prove monotonicity of interpretation functions. We also show that for certain restrictions on interpretation functions, we find linear, polynomial or exponential bounds on runtime complexity (for a simple but natural definition of higher-order runtime complexity).

Our type-based, semantical approach allows us to relate various “size” notions (e.g., list length, tree depth, term size, etc.) to reduction cost, and thus offers a more fine-grained analysis than traditional notions like runtime complexity. Most importantly, we can express the complexity of a higher-order function in terms of the behavior of its (function) arguments. In the future, we hope that this could be used towards a truly higher-order complexity notion.

A clear weakness we discovered was that our method can only handle “plain function-passing” systems [75]. That is, we typically do not succeed on systems where a variable of function type occurs inside a subterm of base type, and occurs outside this subterm in the right-hand side. Examples of such systems are `ordrec`, which has a rule `ordrec (lim F) x G H → H F (λn. ordrec (F n) x G H)` with `lim : [nat ⇒ ord] ⇒ ord`, and `apply`, which has a rule `lapply(x, fcons(F, q)) → F lapply(x, q)` with `fcons : [(a ⇒ a) ⇒ listf] ⇒ listf`.

Chapter 5

Higher-Order Tuple Interpretations for Call-by-Value

5.1 Call-by-Value Higher-order Rewriting

In Chapter 4, we considered full higher-order term rewriting, so without an evaluation strategy. However, this is not a very realistic setting, especially to eventually extend the methodology to various functional programming languages. In practice, program evaluation is deterministic, i.e., it follows a specific strategy such as call-by-value evaluation. Reduction below a λ -binder is also not usually allowed. The difference can be substantial: for instance for a pair of rules $f\ x\ 0 \rightarrow x$, $f\ x\ (s\ y) \rightarrow f\ (\text{pair } x\ x)\ y$, if x is instantiated by a term that is not in normal form, the complexity is linear if we evaluate call-by-value, and exponential with an arbitrary evaluation strategy. Additionally, in complexity analysis of first-order term rewriting, considering innermost evaluation is commonplace [88, 89].

In this chapter, our goal is to extend our techniques from Chapter 4 to *weak call-by-value* reduction. To our knowledge, this is the first complexity method for higher-order term rewriting with an evaluation strategy. While the restriction of the strategy leads to tighter complexity bounds, the definitions needed to obtain these bounds are much more intricate, largely due to the potential for rules and β -redexes of higher type. We believe that this will bring the method of weakly monotonic algebras closer to the reality of functional program analysis.

We are interested in a restricted evaluation strategy, which limits reduction to terms whose immediate subterms that are *values*:

Definition 5.1.1. Let \mathbb{R} be a term rewriting system. A term s is a *value* whenever s is:

- of the form $f\ v_1 \dots v_n$, with each v_i a value and there is no rule $f\ \ell_1 \dots \ell_k \rightarrow r$ with $k \leq n$;

- an abstraction, i.e., $s = \lambda x. t$.

Example 5.1.2. In this example, we collect some common higher-order functions encoded as rules: `map` applies a function to each element of a list; `comp` composes two functions, `app` is the application functional, and `rec` encodes primitive recursion. Their monomorphic signature is defined as expected with functional arguments of type $\text{nat} \Rightarrow \text{nat}$ and lists having type `list`.

$$\begin{array}{ll} \text{map } F [] \rightarrow [] & \text{comp } F G \rightarrow \lambda x. F (G x) \\ \text{map } F (\text{cons } x q) \rightarrow \text{cons } (F x) (\text{map } F q) & \text{app } F \rightarrow \lambda x. F x \\ \text{rec } 0 y F \rightarrow y & \text{rec } (s x) y F \rightarrow F x (\text{rec } x y F) \end{array}$$

Example 5.1.3. Some first-order functions over natural numbers:

$$\begin{array}{lll} \text{dbl } 0 \rightarrow 0 & \text{add } x 0 \rightarrow 0 & \text{mult } x 0 \rightarrow 0 \\ \text{dbl } (s x) \rightarrow s(s (\text{dbl } x)) & \text{add } x (s y) \rightarrow s (\text{add } x y) & \text{mult } x (s y) \rightarrow \text{add } x (\text{mult } x y) \end{array}$$

Notice that by definition ground constructor terms are values, since there is no rule $c \ell_1 \dots \ell_k \rightarrow r$ for any k if $c \in \Sigma^{\text{con}}$. More complex values include partially applied functions and lambda-terms; for example, `add 0` or a list of functions `[add 0; $\lambda x. x$; mult 0; dbl]`. In the weak call-by-value reduction strategy defined below, we shall not reduce under abstractions.

Definition 5.1.4. The **higher-order weak call-by-value rewrite relation** \rightarrow_v induced by \mathbb{R} is defined as follows:

- $f(\ell_1 \gamma) \dots (\ell_k \gamma) \rightarrow_v r \gamma$, if $f \ell_1 \dots \ell_k \rightarrow r \in \mathbb{R}$ and each $\ell_i \gamma$ is a value;
- $(\lambda x. s) v \rightarrow_v s[x := v]$, if v is a value;
- $s t \rightarrow_v s' t$ if $s \rightarrow_v s'$; and $s t \rightarrow_v s t'$ if $t \rightarrow_v t'$.

Notice that when instantiating rules we use *value substitutions*, that is, their image for any nontrivial variable is always a value. All reductions in this chapter are weak call-by-value. So we drop the v from the arrow, and $s \rightarrow t$ should be read as $s \rightarrow_v t$. We use explicit notation whenever confusion may arise.

5.2 Cost–Size Overview

In this section we sketch the broad idea of the methodology, focusing on intuition.

To start, every term is associated with a *size*. For a closed term of base type, this size could for instance be the number of symbols in its normal form; or a pair of integers, or

a set of terms (e.g., the set of all normal forms of the term). We only require that each base type is associated with a *quasi-ordered set*. For a term of higher-order type, the size is a *weakly monotonic function*, which provides a bound for applications of the term.

Example 5.2.1. In the signature of 5.1.2 and 2.1.19, we may let $\mathit{Size}(0) = 1$ and $\mathit{Size}(s\ t) = 1 + \mathit{Size}(t)$; intuitively, the size of a ground constructor term of type nat is the number of function symbols in it. For lists, we could let $\mathit{Size}([]) = (0, 0)$ and $\mathit{Size}(\mathit{cons}\ s\ t) = (\mathit{Size}(t)_1 + 1, \max(\mathit{Size}(s), \mathit{Size}(t)_2))$; intuitively, the size of a list of numbers is the pair (list length, size of its greatest element). We could let $\mathit{Size}(\mathit{add}\ s)$ be the function that maps n to $\mathit{Size}(s) + n$, and $\mathit{Size}(\mathit{map})$ the function that takes a (weakly monotonic) function F and a pair (l, m) , and returns $(l, F(m))$; intuitively, if F bounds the size of the first argument, and we are given a list with maximum element of size m and length l , then applying map to these arguments yields a list which has length l , and elements have sizes bounded by $F(m)$.

Aside from a size, we need to calculate a *cost* for each term to associate a bound on the number of steps that can be taken from a given starting term. Aside from associating a natural number bounding this cost to each term, terms of higher type have computational content even in normal form; hence, we should associate a *cost function* to such terms: a weakly monotonic function that indicates the cost of applying this term to a value.

Example 5.2.2 (First idea for costs). Intuitively, the number of steps to evaluate $\mathit{add}\ s\ t$ is bounded by the cost of evaluating the arguments, plus $\mathit{Size}(s)$ (as we easily see by inspecting the rules defining add). Hence, we would let $\mathit{Cost}(\mathit{add}\ s\ t) = \mathit{Cost}(s) + \mathit{Cost}(t) + \mathit{Size}(s)$, and could define $\mathit{Cost}(\mathit{add}) = \lambda(c_1, s_1), (c_2, s_2). c_1 + c_2 + s_1$. Note that the cost function takes a *pair* of values for each argument: respectively, the cost and size of the argument.

For map , the number of steps to evaluate $\mathit{map}\ s\ t$ depends heavily on s , even if both s and t are values: $\mathit{map}\ (\lambda x.\mathit{add}\ x\ 0)\ t$ will take substantially fewer steps than evaluating $\mathit{map}\ (\lambda x.\mathit{mult}\ x\ x)\ t$. Hence, we should take the cost function for s into account as well as its size. This yields $\mathit{Cost}(\mathit{map}) = \lambda(F_{\mathit{cost}}, F_{\mathit{size}}), (q_{\mathit{cost}}, (l, m)). q_{\mathit{cost}} + l + 1 + l * F_{\mathit{cost}}(0, m)$: the number of steps to evaluate $\mathit{map}\ s\ t$ is bounded by the cost of evaluating t first, then applying s (*length of list*) times to the largest element of t , plus the $1 + \langle \mathit{length of list} \rangle$ steps for the evaluation of map itself. Note that since we use a call-by-value strategy, the list q is evaluated to a value *before* the map rule fires, which is why F_{cost} is given a zero argument.

The cost for constructor applications $c\ s_1 \cdots s_m$ is always just $\mathit{Cost}(s_1) + \cdots + \mathit{Cost}(s_m)$, since applying a constructor to terms does not lead to a further computation being done.

Examples 5.2.1 and 5.2.2 sketch an idea where $\mathit{Size}(s\ t) = \mathit{Size}(s)(\mathit{Size}(t))$ and $\mathit{Cost}(s\ t) = \mathit{Cost}(s)(\mathit{Cost}(t), \mathit{Size}(t))$. Unfortunately, while this idea works well for *sizes*,

it has some issues for *costs*; most importantly, that the computational content of terms of higher types is ignored. Although a term $\lambda x.s$ cannot be reduced, a term such as $\text{add}(\text{dbl } 0)$ can be, and the cost for the $\text{dbl } 0$ reduction should be included. Moreover, terms of higher type can also reduce directly even when their subterms are values; e.g., $\text{comp } s t$ or $(\lambda x.s) t$ of type $\text{nat} \Rightarrow \text{nat}$.

Hence, we will instead consider a *pair* of costs: each term has a *cost number* (a bound on the number of steps to reduce this term to normal form), and a *cost function* (which bounds the cost of applying this normal form to a value, or is unit for base-type terms).

Unfortunately, this choice necessarily imposes a more complicated definition, since a pair cannot be applied like a function can; e.g., if the cost of s is $(12, \lambda(x_{\text{cost}}, x_{\text{size}}). x_{\text{cost}} + x_{\text{size}})$, then when computing the cost for $s t$, we cannot just apply the function and forget the 12. Hence, we will define (formally in Definition 5.3.8) an alternative interpretation of application, so that, for $s : \sigma \Rightarrow \tau$ and $t : \sigma$, $\text{Cost}(s t) = (\text{CostNum}(s) + \text{CostNum}(t) + c, \text{fun})$, where $\text{CostFun}(s)(\text{CostFun}(t), \text{Size}(t))$ is the pair (c, fun) .

Example 5.2.3 (Cost pairs). We let $\text{Cost}(\text{add}) = (0, \lambda(u_1, n). (0, \lambda(u_2, m). n))$: the first 0 is the “cost number” for add , which is 0 because add is in normal form; and the function $\lambda(u_1, n). (0, \lambda(u_2, m). n)$ takes a unit element and the size of a value, and returns a new pair. With the rough definition of application above, we have $\text{Cost}(\text{add } s) = (\text{CostNum}(s), \lambda(u, n). (\text{Size}(s), u))$. This matches the intuition that the number of steps needed to reduce $\text{add } s$ to normal form is just the number of steps needed to reduce s , and the result is a value of function type which, if applied to a value with size n , can be normalized in $\text{Size}(s)$ steps. We obtain $\text{Cost}(\text{add } s t) = \text{Cost}(s) + \text{Cost}(t) + \text{Size}(s)$ as expected.

The notation is rather cumbersome but is needed for the formal definition. In practice, we can identify $\text{unit} \times A$ and $A \times \text{unit}$ with A for any set, and use $(x_1, \dots, x_n) \mapsto \varphi$ as shorthand for $(0, \lambda x_1. (0, \lambda x_2. \dots \varphi))$. Then we can use more palatable notation such as $\text{Cost}(\text{add}) = (n, m) \mapsto n$, or $\text{Cost}(\text{comp}) = ((F_{\text{cost}}, F_{\text{size}}), (G_{\text{cost}}, G_{\text{size}})) \mapsto (2, \lambda x_{\text{size}}. G_{\text{cost}}(x_{\text{size}}) + F_{\text{cost}}(G_{\text{size}}(x_{\text{size}})))$ for the symbol comp which admits a rule of higher type $\text{nat} \Rightarrow \text{nat}$.

With these definitions, if we can show that $(\text{Cost}(\ell), \text{Size}(\ell)) > (\text{Cost}(r), \text{Size}(r))$ for all *value instances* of rules, then $\text{CostNum}(s)$ defines a bound on the number of steps that can be taken to reduce s to normal form. We can use this to define bounds on the runtime complexity of the rewriting system – that is, on the number of steps that can be done when starting in certain kinds of terms of a given size (as we will discuss in Section 5.5).

Example 5.2.4. We choose $\text{Size}([])$, $\text{Size}(\text{cons})$ and $\text{Size}(\text{map})$ following Example 5.2.1, and let $\text{Cost}([]) = 0$, $\text{Cost}(\text{cons}) = (n, m) \mapsto 0$ and

$$\text{Cost}(\text{map}) = ((F_{\text{cost}}, F_{\text{size}}), (l, m)) \mapsto l * F_{\text{cost}}(m) + l + 1$$

Then, for a list $\text{cons } h \ t$ with $\text{Size}(t) = (l, m)$, we have

$$\begin{aligned} \text{Size}(\text{map } F \ (\text{cons } h \ t)) &= (l + 1, \text{Size}(F)(\max(\text{Size}(h), m))) \\ &= (l + 1, \max(\text{Size}(F)(\text{Size}(h)), \text{Size}(F)(m))) \\ &= \text{Size}(\text{cons } (F \ h) \ (\text{map } F \ t)) \end{aligned}$$

by weak monotonicity of $\text{Size}(F)$. Taking into account that if F , h and t are values, then they all have a cost number of 0, we also have:

$$\begin{aligned} \text{Cost}(\text{map } F \ (\text{cons } h \ t)) &= (l + 1) * \text{CostFun}(F)(\max(\text{Size}(h), m)) + l + 2 \\ &> \text{CostFun}(F)(\text{Size}(h)) + l * \text{CostFun}(F)(m) + l + 1 \\ &= \text{Cost}(\text{cons } (F \ h) \ (\text{map } F \ t)) \end{aligned}$$

Hence, all value instantiations of the left-hand side of this rule both have greater cost, and greater-than-or-equal size, to the right-hand sides. If the other rules are similarly oriented, we can conclude that $\text{CostNum}(s)$ provides a bound on the reduction cost of s .

In the rest of this chapter, the ideas above will be formally defined and their correctness proven. We will not use the elaborate names CostNum , Size , etc., but rather define interpretations as tuples that contain all these components.

5.3 Cost–Size Semantics for Simple Types

In this section we build a set-theoretical cost–size semantics to the simple types in $\mathbb{T}(\mathbb{B})$. The goal is to define a function $\llbracket \cdot \rrbracket$ that maps each type $\sigma \in \mathbb{T}(\mathbb{B})$ to a well-founded set $\llbracket \sigma \rrbracket$, the cost–size interpretation of σ . We start by formally defining what we mean by cost–size sets. Recall that these notions were also introduced in Chapter 3 in the context of first-order rewriting.

Definition 5.3.1. Given a well-founded set $(C, >, \succeq)$, called the *cost set*, and a quasi-ordered set (S, \sqsupseteq) , called the *size set*, we call $C \times S$ the **cost–size product** of $(C, >, \succeq)$ and (S, \sqsupseteq) , and its elements *cost–size tuples*.

Given a cost–size product $C \times S$, the well-foundedness of C and quasi-ordering on S naturally induce an order structure on the product $C \times S$ as follows.

Definition 5.3.2 (Product Order). Let $(C, >, \succeq) \times (S, \sqsupseteq)$ be a cost–size product. Then we define the relations $>, \succcurlyeq$ over $C \times S$ as follows: for all $\langle x, y \rangle$ and $\langle x', y' \rangle$ in $C \times S$,

- $\langle x, y \rangle > \langle x', y' \rangle$ iff $x > x'$ and $y \sqsupseteq y'$, and
- $\langle x, y \rangle \succcurlyeq \langle x', y' \rangle$ iff $x \succeq x'$ and $y \sqsupseteq y'$.

Next, we show that the triple $(C \times S, >, \succcurlyeq)$ is well-founded.

Lemma 5.3.3. The triple $(C \times S, >, \succcurlyeq)$ defined in Definition 5.3.2 is a well-founded set.

Proof. It follows immediately from Definition 5.3.2 that $>, \succcurlyeq$ are transitive and \succcurlyeq is reflexive. To show well-foundedness of $>$ we note that the existence of an infinite chain $\langle x_1, y_1 \rangle > \langle x_2, y_2 \rangle > \dots$ would imply $x_1 > x_2 > \dots$, which cannot be the case since $>$ is well-founded. We still need to check that \succcurlyeq is compatible with $>$.

- Suppose $\langle x, y \rangle > \langle x', y' \rangle$. Since $x > x'$ implies $x \succcurlyeq x'$, we have $\langle x, y \rangle \succcurlyeq \langle x', y' \rangle$.
- Suppose $\langle x, y \rangle > \langle x', y' \rangle \succcurlyeq \langle x'', y'' \rangle$. Since $x > x' \succcurlyeq x''$ implies $x > x''$ and \sqsupseteq is transitive, we have $\langle x, y \rangle > \langle x'', y'' \rangle$. \square

We shall use product orders to induce well-founded ordering on cost–size sets. Let us define next the requirements for the sets used for size interpretations.

Definition 5.3.4 (Type Interpretation Key). Let \mathbb{B} be a set of base types. An **interpretation key** for \mathbb{B} , denoted $\mathcal{J}_{\mathbb{B}}$, is a function that maps each base type $\iota \in \mathbb{B}$ to a quasi-ordered set $(\mathcal{J}_{\mathbb{B}}(\iota), \sqsupseteq)$.

Example 5.3.5 (Cost–Size Tuples over natural numbers). A first example of an interpretation key is that of tuples over \mathbb{N} . For each $\iota \in \mathbb{B}$, $\mathcal{J}_{\mathbb{B}/\mathbb{N}}(\iota)$ is a set of the form $(\mathbb{N}^{K(\iota)}, \sqsupseteq)$, with $K(\iota) \geq 1$ and $(x_1, \dots, x_{K(\iota)}) \sqsupseteq (y_1, \dots, y_{K(\iota)})$ iff $x_i \geq y_i$ for all $1 \leq i \leq K(\iota)$. Notice that $(\mathbb{N}^{K(\iota)}, \sqsupseteq)$ is quasi-ordered for any choice of $K(\iota)$ and $\mathcal{J}_{\mathbb{B}/\mathbb{N}}$ is completely determined by a function mapping each $\iota \in \mathbb{B}$ to $K(\iota) \in \mathbb{N}$.

The definition below formalizes our intuition for cost and size from Section 5.2. Given an interpretation key $\mathcal{J}_{\mathbb{B}}$ we inductively interpret the elements of $\mathbb{T}(\mathbb{B})$ as cost–size products.

Definition 5.3.6 (Interpretation of Types). Let $\mathcal{J}_{\mathbb{B}}$ be an interpretation key. We define for each type σ the **cost–size tuple interpretation** of σ as the set $\llbracket \sigma \rrbracket = C_{\sigma} \times S_{\sigma}$ where C_{σ} and S_{σ} are defined as follows (mutually with the set \mathcal{F}_{σ}^c):

$$\begin{aligned} C_{\sigma} &= \mathbb{N} \times \mathcal{F}_{\sigma}^c & S_{\iota} &= \mathcal{J}_{\mathbb{B}}(\iota) \\ \mathcal{F}_{\iota}^c &= \text{unit} & S_{\sigma \Rightarrow \tau} &= S_{\sigma} \Longrightarrow S_{\tau} \\ \mathcal{F}_{\sigma \Rightarrow \tau}^c &= (\mathcal{F}_{\sigma}^c \times S_{\sigma}) \Longrightarrow C_{\tau} \end{aligned}$$

The set $\llbracket \sigma \rrbracket$ is ordered as follows:

- $\langle (n, f_1), f_2 \rangle > \langle (m, g_1), g_2 \rangle$ if $n > m$, $f_1 \succcurlyeq g_1$ and $f_2 \sqsupseteq g_2$, and
- $\langle (n, f_1), f_2 \rangle \succcurlyeq \langle (m, g_1), g_2 \rangle$ if $n \geq m$, $f_1 \succcurlyeq g_1$ and $f_2 \sqsupseteq g_2$.

We say a function f is a *cost (size) function* whenever $f \in \mathcal{F}_{\sigma}^c$ ($f \in S_{\sigma}$), for some type σ .

Lemma 5.3.7. For any type σ , $(C_\sigma, >, \succeq)$ is well-founded and (S_σ, \sqsupseteq) is quasi-ordered. Therefore, (σ) is a cost-size product.

Proof. When σ is a base type, $C_\sigma = \mathbb{N} \times \text{unit} \cong \mathbb{N}$ and $S_\sigma = \mathcal{J}_{\mathbb{B}}(\sigma)$, so the statement is trivially true. Let $\sigma = \tau \Rightarrow \rho$, then by induction hypothesis S_τ and S_ρ are quasi-ordered. Quasi-ordering of $(S_{\tau \Rightarrow \rho}, \sqsupseteq)$ follows from the induced point-wise comparison. Well-foundedness of $(C_\sigma, >, \succeq)$ follows from Lemma 5.3.3 by showing that $\mathcal{F}_{\tau \Rightarrow \rho}^c$ is quasi-ordered. \square

To map each term $s : \sigma$ to an element of (σ) (Definition 5.4.4), we need a notion of application for cost-size tuples. More precisely, assume given a type $\sigma \Rightarrow \tau$ and cost-size tuples $f \in (\sigma \Rightarrow \tau)$ and $x \in (\sigma)$. We define the application of f to x , denoted $f \cdot x$, as follows.

Definition 5.3.8. Let $\sigma \Rightarrow \tau$ be an arrow type, $f = \langle (n, f^c), f^s \rangle \in (\sigma \Rightarrow \tau)$, and $x = \langle (m, x^c), x^s \rangle \in (\sigma)$. The **semantic application** of f to x , denoted $f \cdot x$, is defined by:

$$\text{let } f^c(x^c, x^s) = (k, h); \text{ then } \langle (n, f^c), f^s \rangle \cdot \langle (m, x^c), x^s \rangle = \langle (n + m + k, h), f^s(x^s) \rangle$$

We set the semantic application to be left-associative, so $f \cdot g \cdot h$ denotes $(f \cdot g) \cdot h$.

Example 5.3.9. Let us illustrate semantic application with a concrete example: consider the type $\sigma = (\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{list} \Rightarrow \text{list}$, which is the type of map defined in Example 5.1.2. The function map takes as argument a function $F : \text{nat} \Rightarrow \text{nat}$ and list q and applies F to each element of q . This formalizes the cost and size ideas in Examples 5.2.1 and 5.2.2. Hence, the cost-size interpretation of map is an element $\langle (n, f^c), f^s \rangle$ of (σ) . Its cost component (n, f^c) is in $C_\sigma = \mathbb{N} \times \mathcal{F}_\sigma^c$ which is composed of a numeric and functional component. The numeric component n carries the cost of partial application. Meanwhile, the functional component in \mathcal{F}_σ^c is parametrized by functional arguments carrying the cost and size information of F . Indeed, Definition 5.3.6 gives us $f^c : \mathcal{F}_{\text{nat} \Rightarrow \text{nat}}^c \times S_{\text{nat} \Rightarrow \text{nat}} \Rightarrow C_{\text{list} \Rightarrow \text{list}}$, which can be written explicitly as:

$$\begin{array}{c} \text{the functional cost of map} \\ \overbrace{\left(\underbrace{(\text{unit} \times \mathbb{N} \Rightarrow \mathbb{N} \times \text{unit})}_{\text{cost of } F} \times \underbrace{(S_{\text{nat}} \Rightarrow S_{\text{nat}})}_{\text{size of } F} \right)} \Rightarrow \left(\mathbb{N} \times \left[\underbrace{\text{unit}}_{q^c} \times \underbrace{S_{\text{list}}}_{q^s} \Rightarrow \mathbb{N} \times \text{unit} \right] \right) \end{array}$$

The set for the size function is somewhat simpler with $f^s : (S_{\text{nat}} \Rightarrow S_{\text{nat}}) \Rightarrow S_{\text{list}} \Rightarrow S_{\text{list}}$.

Therefore, we apply f to a cost-size tuple x of the form $\langle (m, x^c), x^s \rangle$ where x^c is the cost of computing F (so an element of $\mathcal{F}_{\text{nat} \Rightarrow \text{nat}}^c$) and x^s is the size of F , so an element of $S_{\text{nat} \Rightarrow \text{nat}}$. We proceed by applying the respective functions so $f^c(x^c, x^s) = (k, h)$ belongs

to $C_{\text{list} \Rightarrow \text{list}}$ and $f^s(x^s)$ is in $\mathcal{S}_{\text{list} \Rightarrow \text{list}}$. We put everything together and add the numeric components to obtain: $f \cdot x = \langle (n + m + k, h), f^s(x^s) \rangle$. Notice that this gives us a new cost–size tuple with the cost component in $\mathbb{N} \times (C_{\text{list}} \Rightarrow C_{\text{list}})$ and size component in $\mathcal{S}_{\text{list}} \Rightarrow \mathcal{S}_{\text{list}}$, which is a tuple in $(\text{list} \Rightarrow \text{list})$.

Observe that our intention with Definition 5.3.8 is that the semantic application conforms with a form of “application typing rule”. A straightforward analysis on Definition 5.3.8 shows that this is indeed the case. This is summarized in the lemma below.

Lemma 5.3.10. If $f \in (\sigma \Rightarrow \tau)$ and $x \in (\sigma)$, then $f \cdot x$ belongs to (τ) .

Definition 5.3.6 gives us a family of cost–size sets $\mathcal{T} = \{(\sigma)\}_{\sigma \in \mathbb{T}(\mathbb{B})}$ indexed by $\mathbb{T}(\mathbb{B})$, and combined with Definition 5.3.8 we get a family of *application operators*

$$(\mathcal{T}, \cdot) = \left(\{(\sigma)\}_{\sigma \in \mathbb{T}(\mathbb{B})}, \{\cdot_{\sigma, \tau}\}_{\sigma, \tau \in \mathbb{T}(\mathbb{B})} \right), \text{ with } \cdot_{\sigma, \tau} : (\sigma \Rightarrow \tau) \times (\sigma) \longrightarrow (\tau)$$

We call the pair (\mathcal{T}, \cdot) the cost–size type structure generated by the interpretation key $\mathcal{J}_{\mathbb{B}}$. Indeed, in the next Lemma we show that such structure preserves the orderings $>$ and \succcurlyeq on cost–size tuples.

Lemma 5.3.11. The application operator is strongly monotonic in both arguments.

Proof. We need to prove the following: (i) if $f > g$ and $x \succcurlyeq y$, then $f \cdot x > g \cdot y$; (ii) if $f \succcurlyeq g$ and $x > y$, then $f \cdot x > g \cdot y$; (iii) if $f \succcurlyeq g$ and $x \succcurlyeq y$, then $f \cdot x \succcurlyeq g \cdot y$. Consider cost–size tuples $f, g \in (\sigma \Rightarrow \tau)$ and $x, y \in (\sigma)$. Let $f = \langle (n, f^c), f^s \rangle$, $g = \langle (m, g^c), g^s \rangle$, $x = \langle (j, x^c), x^s \rangle$, and $y = \langle (j', y^c), y^s \rangle$. We proceed to show (i) and observe that (ii) and (iii) follow similar reasoning. Indeed, if $f > g$ and $x \succcurlyeq y$ we have that $n > m$, $f^c \succeq g^c$, $f^s \sqsupseteq g^s$, $j \geq j'$, $x^c \succeq y^c$, and $x^s \sqsupseteq y^s$. Let $f^c(x^c, x^s) = (k, h)$ and $g^c(y^c, y^s) = (k', h')$, we get:

$$f \cdot x = \langle (n + j + k, h), f^s(x^s) \rangle > \langle (m + j' + k', h'), g^s(y^s) \rangle = g \cdot y$$

□

Remark 5.3.12. Notice that the type structure (\mathcal{T}, \cdot) is nonstandard. Indeed, the intended standard semantics given to arrow types is usually a functional space [14, Chapter 3]. So inhabitants of functional types are interpreted as functions. Since our intention with defining *cost–size type structures* as above is to capture the complexity-wise behavior of functions (defined by rewriting rules) and a cost component associated with the computational environment, this non-standardness is expected. In the next sections we show that even though our interpretations do not give rise to a standard semantic of simple types, we can still prove classical lemmata for substitution and compatibility.

Example 5.3.13. In Examples 5.1.2 and 5.1.3 we have two examples of base types: `nat` and `list`. Values of type `nat` are built using the constructors $0 : \text{nat}$ and $s : \text{nat} \Rightarrow \text{nat}$. Similarly, for `list` we have $[] : \text{list}$ and $\text{cons} : \text{nat} \Rightarrow \text{list} \Rightarrow \text{list}$.

Let us give a cost–size type structure over \mathbb{N} (Example 5.3.5) for $\mathbb{B} = \{\text{nat}, \text{list}\}$. Essentially, we need to choose the numbers $K(\text{nat}), K(\text{list})$ associated with `nat` and `list`, respectively. To do so we take the intended size semantic of `nat, list` into account. Let us set $K(\text{nat}) = 1$ and $K(\text{list}) = 2$. This exactly gives the size sets we used in Section 5.2, and allows us to use “number of symbols” as a notion of size in a unary representation of numbers, and $(\text{length}, \text{maximum element size})$ as a size notion for lists. Intuitively, since a list is a container-like data structure we want to be able to simultaneously give upper bounds to “the size of the container” (which is *length* for lists) and “the size of its elements”. This choice of $\mathcal{J}_{\mathbb{B}/\mathbb{N}}$ affects the shape of interpretations for symbols in Σ , as we will see in Example 5.4.2.

Even though we have manually chosen the size tuples for $\mathcal{J}_{\mathbb{B}/\mathbb{N}}$ above, an automated procedure can still be devised to determine the number $K(\iota)$, for $\iota \in \mathbb{B}$. In fact, this is implemented in Hermes. See Section 3.7.

5.4 Cost–Size Semantics for Terms

In the previous section, we established a cost–size semantics for the simple types in $\mathbb{T}(\mathbb{B})$. Our goal in this section is to interpret terms as elements of those sets.

An interpretation of a signature $\mathbb{F} = (\mathbb{B}, \Sigma, \text{typeOf})$ interprets the base types in \mathbb{B} and each $f \in \Sigma$ of arity $\text{typeOf}(f) = \sigma$ as an element of $\langle \sigma \rangle$ which is constructed by Definition 5.3.6. This is formally stated in the definition below.

Definition 5.4.1. A **cost–size tuple interpretation** \mathcal{F} for a signature $\mathbb{F} = (\mathbb{B}, \Sigma, \text{typeOf})$ consists of a pair of functions $(\mathcal{J}_{\mathbb{B}}, \mathcal{J}_{\Sigma})$ where

- $\mathcal{J}_{\mathbb{B}}$ is a type interpretation key (Definition 5.3.4),
- \mathcal{J}_{Σ} is an *interpretation of symbols* in Σ which maps each $f \in \Sigma$ with $\text{typeOf}(f) = \sigma$ to a cost–size tuple in $\langle \sigma \rangle$, where $\langle \sigma \rangle$ is built using $\mathcal{J}_{\mathbb{B}}$ in Definition 5.3.6.

In what follows we slightly abuse notation by writing \mathcal{J}_f for $\mathcal{J}_{\Sigma}(f)$ and just \mathcal{J} for \mathcal{J}_{Σ} .

Example 5.4.2. As a first example of interpretation, let us interpret the data signature from Example 5.3.13. Recall that $0 : \text{nat}$, $s : \text{nat} \Rightarrow \text{nat}$ are the constructors for `nat` and $K(\text{nat}) = 1$.

$$\mathcal{J}_0 = \left\langle (0, \mathbf{u}), 1 \right\rangle \quad \mathcal{J}_s = \left\langle (0, \lambda x.(0, \mathbf{u})), \lambda x.x + 1 \right\rangle$$

The highlighted cost components for the constructors are filled with zeroes. That is because in the rewriting cost model data values do not fire rewriting sequences. In the language of Section 5.2: the *cost number* for 0 is 0, (because it is a value), the *cost function* is the unit element u (because it is of base type), and *size component* is 1 (since we chose a notion of size for terms of type nat to mean “number of symbols”). The cost number for s is 0, the cost function is the constant function mapping to 0, and the size component is the function $\lambda x. x + 1$ in $\mathcal{S}_{\text{nat} \Rightarrow \text{nat}}$. We interpret the constructors for list, i.e., $[]$ and cons , following the same principle, with $K(\text{list}) = 2$. We write a size tuple q in $\mathcal{S}_{\text{list}}$ as (q_1, q_m) since the first component is the length of the list and the second is a bound on the size of its elements.

$$\mathcal{J}_{[]} = \left\langle (0, u), (0, 0) \right\rangle \quad \mathcal{J}_{\text{cons}} = \left\langle (0, \lambda x.(0, \lambda q.(0, u))), \lambda x q.(q_1 + 1, \max(x, q_m)) \right\rangle$$

The highlighted cost components are filled with zeroes for lists as well. Size components are interpreted as expected, and exactly following Example 5.2.1.

The next step is to extend the interpretation of a signature \mathbb{F} to the set of terms. But first, we define *valuation functions* to interpret the variables in $x : \sigma$ as elements of $\langle \sigma \rangle$.

Definition 5.4.3. A **cost–size valuation** α is a function that maps each $x : \sigma$ to a cost-size tuple in $\langle \sigma \rangle$ such that:

- $\alpha(x) = \langle (0, u), x^s \rangle$, for all $x \in \mathbb{X}$ of base type, and
- $\alpha(F) = \langle (0, F^c), F^s \rangle$ when $F :: \sigma \Rightarrow \tau$.

Notice that, in this definition, the cost component of $\alpha(x)$ has the form $(0, u)$, if $x : \iota$. This interpretation is motivated by Definition 5.1.4, where a matching substitution γ (i.e., a substitution such that $\ell\gamma \rightarrow_v r\gamma$) must map each $x : \iota$ to a value of base type. Those can only have the form $c(v_1, \dots, v_m)$ with $c \in \Sigma^{\text{con}}$. Variables of arrow type still have a cost number 0; however, they can be instantiated to values that carry *indirect* computational content: a partial application or abstraction. For instance, a variable of type $F : \text{nat} \Rightarrow \text{nat}$ can be instantiated with $\text{add } 0$, which is a value that produces a cost as soon as it is applied to the next argument. We use the notation F^c/F^s to denote the cost/size component of $\alpha(F)$.

Definition 5.4.4. Assume given a signature $\mathbb{F} = (\mathbb{B}, \Sigma, \text{typeOf})$ and its cost–size tuple interpretation $\mathcal{F} = (\mathcal{J}_{\mathbb{B}}, \mathcal{J})$ together with a valuation α . The **term interpretation** $\llbracket s \rrbracket_{\alpha}^{\mathcal{J}}$ of s under \mathcal{J} and α is defined by induction on the structure of s as follows:

$$\begin{aligned} \llbracket x \rrbracket_{\alpha}^{\mathcal{J}} &= \alpha(x) & \llbracket f \rrbracket_{\alpha}^{\mathcal{J}} &= \mathcal{J}_f & \llbracket s t \rrbracket_{\alpha}^{\mathcal{J}} &= \llbracket s \rrbracket_{\alpha}^{\mathcal{J}} \cdot \llbracket t \rrbracket_{\alpha}^{\mathcal{J}} \\ \llbracket \lambda x. s \rrbracket_{\alpha}^{\mathcal{J}} &= \left\langle \left(0, \lambda d. (1 + \pi_{11}(\llbracket s \rrbracket_{[x:=d]_{\alpha}}^{\mathcal{J}}), \pi_{12}(\llbracket s \rrbracket_{[x:=d]_{\alpha}}^{\mathcal{J}})) \right), \lambda d^s. \pi_2(\llbracket s \rrbracket_{[x:=(0,d)]_{\alpha}}^{\mathcal{J}}) \right\rangle, \end{aligned}$$

where π_i is the projection on the i th-component and π_{ij} is the composition $\pi_j \circ \pi_i$, and $\underline{0}$ is a cost function of the form $\lambda x_1.(0, \lambda x_2 \dots (0, u) \dots)$. If $d = (d^c, d^s)$, the notation $[x := d]\alpha$ denotes the valuation that maps x to $\langle (0, d^c), d^s \rangle$ and every other variable y to $\alpha(y)$.

We write $\llbracket s \rrbracket$ for $\llbracket s \rrbracket_\alpha^{\mathcal{J}}$ whenever α and \mathcal{J} are universally quantified or clear from the context.

The interpretation for abstractions may seem baroque, but can be understood as follows: an abstraction is a value, so its cost number is 0. The cost of applying that abstraction on a value v is 1 plus the cost number for $s[x := v]$ – which is obtained by evaluating $\llbracket s \rrbracket_{[x:=d]\alpha}^{\mathcal{J}}$ if d is the cost function/size pair for v . The cost *function* of this application is exactly the cost function of $s[x := v]$. The *size* of an abstraction $\lambda x.s$ is exactly the function that takes a size and maps it to the size interpretation of s where x is mapped to that size. Technically, to obtain the size component of $\llbracket s \rrbracket_{[x:=d]\alpha}^{\mathcal{J}}$ we also need a cost component, but by definition, this component does not play a role, so we can safely choose an arbitrary pair $\underline{0}$ in the right set.

Example 5.4.5. We continue with Example 5.4.2 by interpreting ground constructor terms fully. A ground constructor term d of type nat is of the form $s(s \dots (s 0) \dots)$ where the number $n \in \mathbb{N}$ is represented by n successive applications of s to 0. Recall that $\ulcorner n \urcorner$ is our shorthand notation for such terms. Similarly, for ground constructor terms of type list , we write $[\ulcorner n^1 \urcorner; \dots; \ulcorner n^k \urcorner]$ for the term $\text{cons}^{\ulcorner n^1 \urcorner} \dots (\text{cons}^{\ulcorner n^k \urcorner} [])$. The empty list constructor is written as $[]$ in this notation, as usual. Hence, the cost–size interpretation of $\ulcorner 3 \urcorner : \text{nat}$ is given by:

$$\llbracket \ulcorner 3 \urcorner \rrbracket = \llbracket s(s(s 0)) \rrbracket = \llbracket s \rrbracket \cdot (\llbracket s \rrbracket \cdot (\llbracket s \rrbracket \cdot \llbracket 0 \rrbracket)) = \left\langle (0, u), 4 \right\rangle.$$

Consider, for instance, the list $[\ulcorner 1 \urcorner; \ulcorner 7 \urcorner; \ulcorner 9 \urcorner]$. Its cost–size interpretation is given by:

$$\llbracket [\ulcorner 1 \urcorner; \ulcorner 7 \urcorner; \ulcorner 9 \urcorner] \rrbracket = \llbracket \text{cons}^{\ulcorner 1 \urcorner} (\text{cons}^{\ulcorner 7 \urcorner} (\text{cons}^{\ulcorner 9 \urcorner} [])) \rrbracket = \left\langle (0, u), (3, 10) \right\rangle.$$

The important information we can extract from such interpretations is their size component. Indeed, $\llbracket \ulcorner 3 \urcorner \rrbracket^s = 4$ counts the number of constructor symbols in the term representation $\ulcorner 3 \urcorner$ and $\llbracket [\ulcorner 1 \urcorner; \ulcorner 7 \urcorner; \ulcorner 9 \urcorner] \rrbracket^s = (3, 10)$ gives us the length and an upper bound on the size of each element in $[\ulcorner 1 \urcorner; \ulcorner 7 \urcorner; \ulcorner 9 \urcorner]$. The size interpretation for the constructors of nat and list correctly capture our notion of “size” given in Example 5.3.13.

The next lemma expresses the soundness of term interpretation, that is, the interpretation of terms preserves the type structure:

Lemma 5.4.6 (Type Soundness). If $s : \sigma$ then $\llbracket s \rrbracket \in \langle \sigma \rangle$.

Proof. The proof is by induction on the structure of s . The base cases follow directly from Definitions 5.4.1 and 5.4.3. We use Lemmas 5.3.10 and 5.3.11 in the application case. The abstraction case follows from the induction hypothesis and weak monotonicity of π_i . \square

Up to now, we have given cost–size semantics for types and terms. Observe that Definition 5.4.1 only requires that we interpret function symbols as cost–size tuples in the correct domain. For instance, we might interpret all function components as constant functions. This is a valid, but not so useful, interpretation of terms. So we move on to the next component of our interpretation framework: we want to interpret terms in such a way that $\llbracket s \rrbracket > \llbracket t \rrbracket$ whenever $s \rightarrow t$, for any pair of terms s, t .

Definition 5.4.7. Consider a signature $\mathbb{F} = (\mathbb{B}, \Sigma, \text{typeOf})$. A **cost–size call-by-value termination model** for a term rewriting system (\mathbb{F}, \mathbb{R}) consists of the following ingredients:

- a cost–size interpretation $(\mathcal{J}_{\mathbb{B}}, \mathcal{J}_{\Sigma})$ (Definition 5.4.1), such that $\pi_{11}(\llbracket v \rrbracket) = 0$ for all values v

such that the following **compatibility conditions** hold:

- for all value substitutions γ and all terms s and t , $\llbracket s\gamma \rrbracket > \llbracket t\gamma \rrbracket$ whenever $\llbracket s \rrbracket > \llbracket t \rrbracket$;
- for every term s and value v , $\llbracket (\lambda x. s) v \rrbracket > \llbracket s[x := v] \rrbracket$;
- for all terms s and t ,
 - $\llbracket s t \rrbracket > \llbracket s' t \rrbracket$ whenever $\llbracket s \rrbracket > \llbracket s' \rrbracket$, and $\llbracket s t \rrbracket > \llbracket s t' \rrbracket$ whenever $\llbracket t \rrbracket > \llbracket t' \rrbracket$;
- for all rules $\ell \rightarrow r \in \mathbb{R}$, we have $\llbracket \ell \rrbracket > \llbracket r \rrbracket$.

Roughly speaking, a call-by-value termination model is an interpretation of types and terms that is compatible with each rule in \mathbb{R} , the call-by-value beta rule and the formation of terms, and which is closed under value substitutions. By a straightforward induction on the reduction $s \rightarrow_v t$, we can establish the following result.

Theorem 5.4.8. Let (\mathbb{F}, \mathbb{R}) be a TRS. If we have a termination model of (\mathbb{F}, \mathbb{R}) , then the higher-order call-by-value rewriting relation \rightarrow_v is strongly normalizing.

Hence, termination models collect sufficient conditions for strong normalization. The lemmata below are to show that cost–size interpretations satisfy some of the compatibility conditions for termination models. Let us first prove closure under substitutions.

Definition 5.4.9. Given a substitution γ and valuation α , we define the γ -**extension** of α as the valuation defined by $\alpha^\gamma = \llbracket \cdot \rrbracket_\alpha^{\mathcal{J}} \circ \gamma$.

Lemma 5.4.10. If $x \notin \text{fv}(s)$ then $\llbracket s \rrbracket_{[x:=d]\alpha} = \llbracket s \rrbracket_{\alpha}$. Consequently, if x is not free in $y\gamma$ for any variable y , then $([x := d]\alpha)^{\gamma} = [x := d]\alpha^{\gamma}$.

Lemma 5.4.11 (Substitution Lemma). For any value substitution γ and valuation α , we have that $\llbracket s\gamma \rrbracket_{\alpha} = \llbracket s \rrbracket_{\alpha^{\gamma}}$.

Proof. Let us work out the abstraction case $s = \lambda x. t$. Since we assume that the application of substitution is capture-avoiding, we can assume that x does not occur free in any term in the range of γ . Hence,

$$\begin{aligned} & \llbracket \lambda x. (t\gamma) \rrbracket_{\alpha} \\ &= \left\langle \left(0, \lambda d. (1 + \pi_{11}(\llbracket t\gamma \rrbracket_{[x:=d]\alpha}^{\mathcal{J}}), \pi_{12}(\llbracket t\gamma \rrbracket_{[x:=d]\alpha}^{\mathcal{J}})) \right), \lambda d^{\mathbf{s}}. \pi_2(\llbracket t\gamma \rrbracket_{[x:=\langle 0, d \rangle]\alpha}^{\mathcal{J}}) \right\rangle \\ &\stackrel{IH}{=} \left\langle \left(0, \lambda d. (1 + \pi_{11}(\llbracket t \rrbracket_{([x:=d]\alpha)^{\gamma}}^{\mathcal{J}}), \pi_{12}(\llbracket t \rrbracket_{([x:=d]\alpha)^{\gamma}}^{\mathcal{J}})) \right), \lambda d^{\mathbf{s}}. \pi_2(\llbracket t \rrbracket_{([x:=\langle 0, d \rangle]\alpha)^{\gamma}}^{\mathcal{J}}) \right\rangle \\ &= \left\langle \left(0, \lambda d. (1 + \pi_{11}(\llbracket t \rrbracket_{[x:=d]\alpha^{\gamma}}^{\mathcal{J}}), \pi_{12}(\llbracket t \rrbracket_{[x:=d]\alpha^{\gamma}}^{\mathcal{J}})) \right), \lambda d^{\mathbf{s}}. \pi_2(\llbracket t \rrbracket_{[x:=\langle 0, d \rangle]\alpha^{\gamma}}^{\mathcal{J}}) \right\rangle \\ &= \llbracket \lambda x. t \rrbracket_{\alpha^{\gamma}}. \end{aligned}$$

□

As a consequence of the substitution lemma, if $\llbracket s \rrbracket_{\alpha}^{\mathcal{J}} > \llbracket t \rrbracket_{\alpha}^{\mathcal{J}}$ for all α , then $\llbracket s\gamma \rrbracket_{\alpha}^{\mathcal{J}} > \llbracket t\gamma \rrbracket_{\alpha}^{\mathcal{J}}$ for all α . Consequently, the first compatibility condition is valid for any interpretation. The second compatibility requirement is for β reductions.

Lemma 5.4.12. The call-by-value beta rule scheme $(\lambda x. s)v \rightarrow_v s[x := v]$ is strictly decreasing for any cost–size interpretation.

Proof. The proof reduces to checking $\llbracket (\lambda x. s)v \rrbracket > \llbracket s[x := v] \rrbracket$. Let $\llbracket v \rrbracket = \langle (0, v^c), v^s \rangle$, and denote V for the pair (v^c, v^s) . Then we have the following:

$$\begin{aligned} & \llbracket (\lambda x. s)v \rrbracket = \llbracket \lambda x. s \rrbracket \cdot \llbracket v \rrbracket \\ &= \left\langle \left(0, \lambda d. (1 + \pi_{11}(\llbracket s \rrbracket_{[x:=d]\alpha}^{\mathcal{J}}), \pi_{12}(\llbracket s \rrbracket_{[x:=d]\alpha}^{\mathcal{J}})) \right), \lambda d^{\mathbf{s}}. \pi_2(\llbracket s \rrbracket_{[x:=\langle 0, d^{\mathbf{s}} \rangle]\alpha}^{\mathcal{J}}) \right\rangle \cdot \llbracket v \rrbracket \\ &= \left\langle \left(0 + 0 + 1 + \pi_{11}(\llbracket s \rrbracket_{[x:=V]\alpha}^{\mathcal{J}}), \pi_{12}(\llbracket s \rrbracket_{[x:=V]\alpha}^{\mathcal{J}}) \right), \pi_2(\llbracket s \rrbracket_{[x:=\langle 0, v^{\mathbf{s}} \rangle]\alpha}^{\mathcal{J}}) \right\rangle \\ &> \left\langle \left(\pi_{11}(\llbracket s \rrbracket_{[x:=V]\alpha}^{\mathcal{J}}), \pi_{12}(\llbracket s \rrbracket_{[x:=V]\alpha}^{\mathcal{J}}) \right), \pi_2(\llbracket s \rrbracket_{[x:=V]\alpha}^{\mathcal{J}}) \right\rangle \\ &= \llbracket s[x := v] \rrbracket_{\alpha}. \end{aligned}$$

In the second-to-last step, we use that the size component of $\llbracket s \rrbracket_{\alpha}^{\mathcal{J}}$ does not regard any cost component in α , so $\pi_2(\llbracket s \rrbracket_{[x:=\langle 0, v^{\mathbf{s}} \rangle]\alpha}^{\mathcal{J}}) = \pi_2(\llbracket s \rrbracket_{[x:=V]\alpha}^{\mathcal{J}})$. In the last step, we use the substitution lemma. □

Compatibility over applicative terms is a consequence of Lemma 5.3.11. Notice that the results above do not depend on a particular interpretation. Hence, to establish a

termination model for a TRS, only the last compatibility condition remains to be checked, i.e., $\llbracket \ell \rrbracket > \llbracket r \rrbracket$ for all rules $\ell \rightarrow r$ in \mathbb{R} . We collect this fact below, which is a consequence of Theorem 5.4.8 and the lemmas above.

Corollary 5.4.13. Let \mathbb{R} be a TRS that admits a cost–size interpretation $(\mathcal{J}_{\mathbb{B}}, \mathcal{J}_{\Sigma})$. If $\llbracket \ell \rrbracket > \llbracket r \rrbracket$ for all rules $\ell \rightarrow r$ in \mathbb{R} , then \mathbb{R} is a termination model, and consequently strongly normalizing.

5.5 Complexity Analysis of Call-by-Value Rewriting

In the previous section, we showed that cost–size tuples can be used to establish termination of call-by-value rewriting. In this section, we concentrate on a quantitative analysis of such termination proofs. Hence, the goal is not merely to find tuple interpretations that prove termination but also ones that establish “good” upper bounds on the complexity of reducing terms to normal form. To start, we will extend the notion of *derivation height* to our setting:

Definition 5.5.1. The weak call-by-value **derivation height** of a term s , notation $\text{dh}_{\mathbb{R}}(s)$, is the largest number n such that $s \rightarrow_v s_1 \rightarrow_v \dots \rightarrow_v s_n$.

This notion is defined for all terms when the TRS is finitely branching and terminating. We will simply refer to the weak call-by-value derivation height as “derivation height”.

The methodology of weakly monotonic algebras offers a systematic way to derive bounds for the derivation height of a given term:

Lemma 5.5.2. If $\llbracket s \rrbracket = \langle (n, F^c), F^s \rangle$, then $\text{dh}_{\mathbb{R}}(s) \leq n$.

Proof. By the lemmas in Section 5.4 we see that $\llbracket s \rrbracket > \llbracket t \rrbracket$ whenever $s \rightarrow t$. Since this implies $\pi_{11}(s) > \pi_{11}(t)$, the lemma follows. \square

As an illustration of how this is used, we present the formalized examples of Section 5.2 and complete the interpretation of Examples 5.1.2 and 5.1.3.

Let us start with the system \mathbb{R}_{add} which intuitively defines addition over nat. We will use the type and constructor interpretations as given in Example 5.4.2. The rules $\text{add } x \ 0 \rightarrow 0$ and $\text{add } x \ (s \ y) \rightarrow s \ (\text{add } x \ y)$ suggest the following cost–size interpretation:

$$\mathcal{J}_{\text{add}} = \left\langle (0, \lambda x.(0, \lambda y.(y^s, u))), \lambda xy.x + y \right\rangle. \quad (5.1)$$

Notice that the (highlighted) cost component of \mathcal{J}_{add} suggests a linear cost measure for computing with add . We also set the intermediate numeric components in the cost tuple to zero. The reason for this choice is that in a cost tuple $C_{\sigma} = \mathbb{N} \times \mathcal{F}_{\sigma}^c$, the numeric component \mathbb{N} captures the cost of partially applying terms, which is 0 in this

case. Using the shorthand notation of Example 5.2.3, we could alternatively write $\mathcal{J}_{\text{add}} = \langle (x^s, y^s) \mapsto y^s, \lambda x^s y^s . x^s + y^s \rangle$.

Now, consider the partially applied term $s = \text{add} \ulcorner 2 \urcorner \ulcorner 3 \urcorner$ (of type $\text{nat} \Rightarrow \text{nat}$). Intuitively, the cost of reducing this term to normal form, is the cost of reducing the subterm $\text{add} \ulcorner 2 \urcorner \ulcorner 3 \urcorner$ to $\ulcorner 5 \urcorner$, since the partially applied term $\text{add} \ulcorner 5 \urcorner$ cannot be reduced. Hence, $\text{dh}_{\mathbb{R}}(s) = 4$. This is also the bound we find through interpretation:

$$\begin{aligned} \llbracket s \rrbracket &= \llbracket \text{add} \rrbracket \cdot (\llbracket \text{add} \rrbracket \cdot \llbracket \ulcorner 2 \urcorner \rrbracket \cdot \llbracket \ulcorner 3 \urcorner \rrbracket) \\ &= \llbracket \text{add} \rrbracket \cdot \langle (4, u), 7 \rangle \\ &= \langle (4, \lambda y . (y^s, u)) , \lambda y . 7 + y \rangle. \end{aligned}$$

While in this case the bound we find is tight, this is not always the case; for instance $\llbracket \text{add} \ulcorner 0 \urcorner (\text{add} \ulcorner 0 \urcorner \ulcorner 0 \urcorner) \rrbracket = \langle (3, u), 3 \rangle$, even though $\text{dh}_{\mathbb{R}}(\text{add} \ulcorner 0 \urcorner (\text{add} \ulcorner 0 \urcorner \ulcorner 0 \urcorner)) = 2$. We could obtain a tight bound by choosing a different interpretation, but this is also not always possible.

Remark 5.5.3. Intuitively, we think of the numeric component of a partially applied term $f s_1 \dots s_n$ that cannot be reduced at the root as the “environment cost” of computing functional arguments to values. This plays an important role in the complexity analysis in our setting. Namely, when interpreting terms this is what allows us to limit interest to value substitutions, since the cost of reducing arguments to values is captured implicitly by the \cdot operator. This assumption consequently allows us to limit the class of cost functions to *weakly* monotonic functions as used in Definition 5.3.6, as opposed to the *strongly* monotonic functionals used in the full rewriting setting [69, 92].

In complexity analysis of term rewriting, it is common to consider bounds on the derivation height for terms of a given size. However, it is useful to impose some limitations. Consider for example a TRS consisting *only* of the two `add` rules. Then, we might construct a term $(\lambda x . \text{add } x x) ((\lambda x . \text{add } x x) (\dots (s 0) \dots))$, with n occurrences of $(\lambda x . \text{add } x x)$. The size of this term is linear in n , but its derivation height is exponential, since each contraction of a λ essentially duplicates the number of `s` occurrences. Hence, the traditional notion of *derivational complexity* (which maps a natural number n to the largest derivation height a term of size at most n can have) is arguably not so useful in a setting with λ .

Instead, we will consider the *runtime complexity* of a TRS. Following the definition in Chapter 4 for *full* higher-order runtime complexity, we recall the following notions.

Definition 5.5.4. The **weak call-by-value runtime complexity** of a TRS is the function $n \mapsto \text{rc}_{\mathbb{R}}(n)$ that maps each natural number n to the largest number h with $\text{dh}_{\mathbb{R}}(s) = h$ for some basic term s of size at most n .

Note that for instance lists of functions are not data terms, and therefore not considered as viable inputs in the notion of runtime complexity. As discussed in Chapter 4, this arguably makes the notion somewhat first-order, but it can still be used to analyze higher-order programs or modules (so long as they, for instance, have a rule $\text{start } x \rightarrow r$ where x has base type, and r is allowed to use abstractions, partial application or calls to higher-order functions).

Example 5.5.5. Let us collect the interpretation for `dbl` and `mult` from Example 5.1.3.

$$\begin{aligned} \mathcal{J}_{\text{dbl}} &= \left\langle (0, \lambda x.(x^s, u)) , \lambda x.2x \right\rangle \\ \mathcal{J}_{\text{mult}} &= \left\langle (0, \lambda x.(0, \lambda y.2x^s y^s, u)) , \lambda xy.xy \right\rangle \end{aligned}$$

In the TRS of Example 5.1.3, the only basic terms have the form `add` $v_1 v_2$ or `dbl` v or `mult` $v_1 v_2$. Since by Equation (5.1) we have $\llbracket s^n 0 \rrbracket^s = n + 1$, Lemma 5.5.2 allows us to conclude that $\text{rc}_{\mathbb{R}}(n) < n^2$.

Now, the size bound for data constructors introduced in Example 5.4.2 is well-behaved. However, suppose we had defined $\mathcal{J}_0 = \langle (0, u), 1 \rangle$ and the interpretation of `s` as $\mathcal{J}_s = \langle (0, \lambda x.(0, u)), \lambda x.2x + 1 \rangle$. In this case, for a data term $\ulcorner n \urcorner = s^n 0$, we would have $\llbracket \ulcorner n \urcorner \rrbracket^s = 2^n + n \geq 2^n$. As a result, we would only be able to derive exponential runtime complexity. Notice that this choice is compatible with \mathbb{R}_{add} , and hence proves its termination; however, it induces an exponential overhead on the cost tuple of `add`, whose actual runtime complexity is linearly bounded as we saw in Example 5.5.5. Such a huge overestimation is not desirable in a complexity analysis setting. This behavior suggests an upper bound to the interpretation of data constructors; namely, we seek to bound the constructor's size interpretations *additively*.

Let `c` be a data constructor of type $\sigma = \iota_1 \Rightarrow \dots \Rightarrow \iota_m \Rightarrow \kappa$. The size component of (σ) is $\mathcal{S}_\sigma = \mathbb{N}^{K(\iota_1)} \Rightarrow \dots \Rightarrow \mathbb{N}^{K(\iota_m)} \Rightarrow \mathbb{N}^{K(\kappa)}$. The size tuple \mathcal{J}_c^s when fully applied can be written in terms of its functional components. Hence, $\mathcal{J}_c^s(x_1, \dots, x_m) = \langle f_1^s(x_1, \dots, x_m), \dots, f_{K(\kappa)}^s(x_1, \dots, x_m) \rangle$.

Definition 5.5.6. If `c` : σ is a data constructor as above, we say \mathcal{J}_c^s is **additive** if there is a constant $a \in \mathbb{N}$ such that $\sum_{l=1}^{K(\kappa)} f_l^s(x_1, \dots, x_m) \leq a + \sum_{i=1}^m \sum_{j=1}^{K(\iota_i)} x_{ij}$.

It is easy to show that size components for `nat` and `list` in 5.4.2 are additive.

If data constructors are additive, and there are only finitely many of them, then there exists a constant a such that, for every data term d of size at most n : $\llbracket d \rrbracket^s \leq an$. Hence, for instance the following result from [69], which we proved in Chapter 4, also extends to our setting:

Lemma 5.5.7 (From [69]; Corollary 33). Let \mathbb{R} be a TRS. If all interpretations for data constructors are additive and the interpretations for all defined symbols are polynomially bounded, then the weak call-by-value runtime complexity of \mathbb{R} is polynomially bounded.

This result provides us with a systematic approach to establishing bounds to the runtime complexity of weak call-by-value systems. The difficulty now lies in developing techniques to find suitable interpretation shapes. For instance, a first example of a higher-order function over lists is that of `map`. We studied the structure of its cost-size tuples in Example 5.3.9 to illustrate semantical application. We give a concrete cost-size interpretation for `map` below:

$$\mathcal{J}_{\text{map}} = \left\langle (0, \lambda F.(0, \lambda q.(q_l + F^c(\mathbf{u}, q_m)q_l + 1, \mathbf{u}))) , \lambda Fq.(q_l, F(q_m)) \right\rangle,$$

The highlighted cost component accounts for q_l possible β steps, the cost of applying the higher-order argument F over the list q is bounded by $F^c(\mathbf{u}, q_m)q_l$ since F^c is assumed to be weakly monotonic, and the unitary component is for dealing with the empty list case.

Finding such interpretations for higher-order systems can become quite challenging. In the example below we collect basic weakly monotonic combinators in order to generate more complex cost/size interpretations.

Example 5.5.8. We list the following weakly monotonic combinators. Here, sets X, Y, Z are used generically to denote cost/size sets:

- for any X and $a \in Y$, there is a constant functional $\lambda x.a$ in $X \Longrightarrow Y$;
- for $f : X \Longrightarrow Y$ and $g : Y \Longrightarrow Z$, we write $g \circ f : X \Longrightarrow Z$ as the composition of f and g .
- the projection function on the i th coordinate, $\pi_i : X_1 \times \cdots \times X_k \Longrightarrow X_i$;
- given $f : X \Longrightarrow Y$ and $g : X \Longrightarrow Z$, we have a function $\langle f, g \rangle : X \Longrightarrow Y \times Z$ which is defined by $\langle f, g \rangle(x) = \langle f(x), g(x) \rangle$;
- given $f : Y \times X \Longrightarrow Z$, we get a function $\lambda_f : X \Longrightarrow (Y \Longrightarrow Z)$. For each $x \in X$ and $y \in Y$, we define $(\lambda_f(x))(y) = f(y, x)$;
- given $f : X \Longrightarrow (Y \Longrightarrow Z)$ and $g : X \Longrightarrow Y$, we obtain $f . g : X \Longrightarrow Z$, which is defined as $(f . g)(x) = f(x)(g(x))$;
- given $f : X \Longrightarrow Y$ and $x \in X$, we have an element application functional with domain $\mathbf{app}_x : (X \Longrightarrow Y) \Longrightarrow Y$ which sends f to $f(x)$, i.e., $\mathbf{app}_x(f) = f(x)$.

Notice that we can use the combinators above with the usual monotonic functionals and operators over \mathbb{N} to produce new monotonic functionals and pointwise operators over sets $X \Longrightarrow Y$. For instance, we can utilize $+, *, \lfloor \cdot \rfloor, \max, \log(\lfloor \cdot \rfloor)$, and so forth.

These basic combinators provide the building blocks for cost–size interpretations.

Example 5.5.9. The higher-order functions in Example 5.1.2 admit the following interpretations:

$$\begin{aligned} \mathcal{J}_{\text{app}} &= \left\langle (0, \lambda F.(2, \lambda x.(F^c(\mathbf{u}, x^s), \mathbf{u}))) , \lambda Fx.F(x) \right\rangle \\ \mathcal{J}_{\text{comp}} &= \left\langle (0, \lambda F.(0, \lambda G.(2, \lambda x.(F^c(\mathbf{u}, G^s(x^s)) + G^c(\mathbf{u}, x^s), \mathbf{u})))) , \lambda FGx.F(G(x)) \right\rangle \\ \mathcal{J}_{\text{rec}} &= \left\langle (0, \lambda x.(0, \lambda y.(0, \lambda F.(x^s + H^c(x, y, F), \mathbf{u})))) , \lambda xyF.H^s(x, y, F) \right\rangle \end{aligned}$$

In the cost component for \mathcal{J}_{rec} , the term x^s computes the total number of rewriting steps using the `rec` symbol. Meanwhile, H^c is an auxiliary symbol computing the total cost of recursively applying the higher-order argument F . It can be defined as follows

$$H^c(x, y, F) = \sum_{i=1}^{x^s-1} \pi_1(F^c((\mathbf{u}, i), (\mathbf{u}, H^s(i, y^s, F^s))))$$

with the size helper function H^s given as a weakly monotonic variant of the recursor over \mathbb{N} :

$$H^s(x, y, F) = \begin{cases} y & \text{if } x \leq 1 \\ \max(y, F(x-1, H^s(x-1, y, F))) & \text{if } x > 1 \end{cases}$$

5.6 Conclusions and Future Work

In this chapter, we built an interpretation method for higher-order rewriting with weak call-by-value reduction. In this approach, we build on existing work defining tuple interpretations [69, 109], but restrict the evaluation strategy, and define a cost–size semantics for types and terms which generates a whole new class of cost–size termination models that can be used to reason about both termination and complexity of weak call-by-value systems. We showed that cost–size tuples correctly capture call-by-value termination and allow us to bound both the cost (number of steps to reach normal forms) and a variety of size notions for different data types. A second advantage of the call-by-value approach compared to full rewriting in Chapter 4 is that the cost functionals are now weakly rather than strongly monotonic functionals, which simplifies the search for cost interpretations.

This is foundational work in the research direction of transposing the methodology and tools from (higher-order) term rewriting to program analysis. A first step for future work is to consider more expressive type theories, so we can capture more programs. For instance, the power of the techniques developed here would be greatly improved if polymorphic types are taken into account. A second step is to expand other complexity

methods for innermost/call-by-value rewriting to the higher-order setting, such as dependency tuples [89] or polynomial path orders [8]. Also for termination analysis, it would be interesting to combine tuple interpretations with a higher-order variant of innermost dependency pairs [5], similar to what was done for full rewriting with tuple interpretations in [66].

Finally, we plan to implement this work, to automatically derive bounds to the derivation height of individual terms, as well as provide bounds for both full and call-by-value runtime complexity of higher-order term rewriting systems. The automation approach could build on the strategy for higher-order polynomial interpretations for full rewriting (not using tuples) in [38, Section 5]. While the search for tuple interpretations has more unknowns (because it can happen that $K(t) > 1$) than the search for interpretations to \mathbb{N} , and will therefore likely take longer, we expect that the overall methodology can stay largely unchanged at least when it comes to an unrestricted evaluation strategy. Adapting to weak call-by-value rewriting may require some additional study, however.

Chapter 6

A Rewriting Characterization of Higher-Order Feasibility

6.1 Higher-Order Feasibility

Computational complexity classes, and in particular those relating to polynomial time and space [27, 49] capture the concept of a feasible problem, and as such have been scrutinized with great care by the scientific community in the last fifty years. The fact that even apparently simple problems, such as establishing nontrivial separations between those classes, remain open today has highlighted the need for a comprehensive study aimed at investigating the deep nature of computational complexity. The so-called implicit computational complexity [12, 33, 60, 78, 90] fits well into this picture and is concerned with characterizations of complexity classes based on tools from mathematical logic and the theory of programming languages.

After the introduction of $FPTIME$, it became clear that the study of computational complexity also applies to *higher-order functionals* which — as we have seen in previous chapters — are those functions that take not only data but also other functions as inputs. The pioneering work of Constable [29], Mehlhorn [79], and Kapron and Cook [60] laid the foundations of the so-called higher-order complexity theory which remains a prolific research area to this day. Motivations for this line of work can be found e.g. in computable analysis [62], NP search problems [16], and programming language theory [35].

There have been several proposals for a class of second order (type-two) functionals that correctly generalizes $FPTIME$. However, the most widely accepted one is the class BFF of *Basic Feasible Functionals*. The class BFF was then the object of study by the research community, which over the years has introduced a variety of characterizations, e.g., in terms of programming languages with restricted recursion schemes [35, 56], typed imperative languages [44, 45], and restricted forms of iteration in OTMs [61]. An investigation of higher-order complexity classes employing the higher-order interpretation

method (in the context of a pure higher-order functional language) was also proposed in [46]. However, this paper does not provide a characterization of the standard BFF class. Instead, it characterizes a newly proposed class SFF_2 (Safe Feasible Functionals) which is defined as the restriction of BFF to argument functions in $FPTIME$ (see Sect. 4.2 and the conclusion in [46]).

The studies cited above present structurally complex programming languages and logical systems, precisely due to the presence of higher-order functions. It is not currently known whether it is possible to give a characterization of BFF in terms of mainstream concepts of rewriting theory, although the latter has long been known to provide tools for the modeling and analysis of functional programs with higher-order functions [63].

In this chapter, we go precisely in this direction by showing that the interpretation method we introduced in Chapters 4 and 5 provides the right tools to characterize BFF via a higher-order rewriting setting. More precisely, we consider a class of higher-order rewriting systems admitting cost–size tuple interpretations (with some mild upper-bound conditions on their cost and size components) and show that this class contains exactly the functionals in BFF. Such a characterization could not have been obtained employing classical integer interpretations as e.g. in [21] because BFF crucially relies on some simultaneous conditions over both notions of time and size. This is the main contribution of this chapter, and it is formally stated in Theorem 6.5.9.

We believe that a benefit of this characterization is that it opens the way to effectively handling programs or executable specifications implementing BFF functions, in full generality. For instance, we expect that such a characterization could be integrated into rewriting-based tools for complexity analysis of rewriting systems such as e.g. [9, 42].

6.2 Basic Feasible Functionals

As stated before in the thesis, \mathbb{N} denotes the set of natural numbers. In this chapter we also identify $n \in \mathbb{N}$ with its dyadic representation over $\{0, 1\}$. With that we have for instance $0 \equiv \epsilon$, $1 \equiv 0$, $2 \equiv 1$, $3 \equiv 00$, etc. We freely switch between $n \in \mathbb{N}$ as a number and its binary representation. The function $|\cdot| : \mathbb{N} \rightarrow \mathbb{N}$ maps each n to the length of its dyadic representation. It is well-known that this function can be defined as $|x| = \lceil \log_2(x + 1) \rceil$. For complexity analysis, it is useful to consider the binary successors S_0, S_1 with semantics $S_0(x) = 2x$ and $S_1(x) = 2x + 1$. We also write $[x] = 2^{|x|}$.

We call a function from \mathbb{N} to \mathbb{N} a type-1 function. As usual in this thesis, $\mathbb{N} \rightarrow \mathbb{N}$ denotes the set of all functions from \mathbb{N} to \mathbb{N} . We say a type-2 function is a mapping $\Psi : (\mathbb{N} \rightarrow \mathbb{N})^k \rightarrow \mathbb{N}^l \rightarrow \mathbb{N}$, for some $k, l \geq 1$. We call this mapping a type-2 functional of rank (k, l) . For type-1 functions, the notion of feasibility is well understood in the literature. Its first characterization is due to Cobham [27]. In this germinal work, Cobham defines the class \mathcal{L} of feasible functions in terms of *limited recursion on notation*.

Definition 6.2.1. We say that a function $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ is defined from $h : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$, $g_0 : \mathbb{N}^n \rightarrow \mathbb{N}$, and $g_1 : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$ by **limited recursion on notation** if and only if f is given by the following equations:

$$\begin{aligned} f(0, \vec{y}) &= g_0(\vec{y}) \\ f(x, \vec{y}) &= g_1(f(\lfloor x/2 \rfloor, \vec{y}), x, \vec{y}), \text{ if } x > 0 \\ |f(x, \vec{y})| &\leq |h(x, \vec{y})| \end{aligned}$$

With this Cobham [27] defined the class \mathcal{L} as follows.

Definition 6.2.2 (Cobham's \mathcal{L}). Let \mathcal{L} be the smallest collection of functions that contains the initial functions: $\lambda x.0$, $\lambda x.2x$, $\lambda x.2x + 1$, $\lambda xy.2^{|x||y|}$, and the projection functions $\lambda x_0 \dots x_n.x_j$ (one such function for all n and all $j \leq n$); and that is closed under composition and limited recursion on notation.

Example 6.2.3. Let us define the following type-1 function $f : \mathbb{N} \rightarrow \mathbb{N}$ by limited recursion on notation:

$$\begin{aligned} f(0) &= 1 & h(x) &= 2^{2^{|x|}} \\ f(x) &= S_0(S_0(f(\lfloor x/2 \rfloor))), \text{ if } x > 0 \end{aligned}$$

This function is of quadratic growth. In fact, $f(x) = \lfloor x \rfloor^2$.

In [27] the Cobham argues that \mathcal{L} is exactly the class of *feasible* functions, which today we call polytime functions. A variety of characterizations of this class appeared over the years. A low-level definition of polytime is understood as the set of functions computable in polynomial time by a Turing machine.

Definition 6.2.4. We formally define FPTIME as the class of functions contained in $\cup_{k>0} (\mathbb{N}^k \rightarrow \mathbb{N})$ such that a function $f : \mathbb{N}^k \rightarrow \mathbb{N}$ is in FPTIME if and only if there exists a deterministic Turing machine M and a polynomial p such that for each input x_1, \dots, x_k :

1. the machine M outputs $f(\vec{x})$, and
2. M runs within $p(|x_1|, \dots, |x_k|)$ steps.

Theorem 6.2.5 (Cobham's Characterization of Polytime). A function is in FPTIME if and only if it is in \mathcal{L} .

Example 6.2.6. Consider the type-1 function f as in Example 6.2.3. Let us define a new type-1 function g in terms of f as follows:

$$\begin{aligned} g(0) &= 2 \\ g(x) &= f(g(\lfloor x/2 \rfloor)), \text{ if } x > 0 \end{aligned}$$

This recursion leads to exponential growth, indeed, $g(x) = 2^{\lfloor x \rfloor}$. Even though there is a function h bounding f , there cannot be such h function in \mathcal{L} . Therefore, when we say \mathcal{L} is closed by limited recursion on notation it is important to note that the bounding function should also be a member of \mathcal{L} .

The question of higher-order analogue classes for polytime computability is long standing in complexity theory. It was initially posed by Constable [29]. Following Constable's work, Mehlhorn [79], defined a higher-order analogue of Cobham's syntactic characterization of FPTIME, which Mehlhorn called $\mathcal{L}()$. This class is a relativization¹ of Cobham's syntactic characterization of polytime via \mathcal{L} . Mehlhorn [79] provided some evidence that this class is a sensible type-2 analogue to FPTIME from the fact that $\mathcal{L}()$ is a natural type-2 extension of the limited recursion on notation recursion schemata.

Definition 6.2.7. Given a functional F we say that

- F is defined from H, G_1, \dots, G_l by **functional composition** if for all \vec{f} and \vec{x} ,

$$F(\vec{f}, \vec{x}) = H(\vec{f}, G_1(\vec{f}, \vec{x}), \dots, G_l(\vec{f}, \vec{x})).$$

- F is defined from G by **expansion** if for all $\vec{f}, \vec{g}, \vec{x}$, and \vec{y} ,

$$F(\vec{f}, \vec{g}, \vec{x}, \vec{y}) = G(\vec{f}, \vec{x}).$$

- F is defined from G, H , and K by **limited recursion on notation** (LRN) if for all \vec{f}, \vec{x} , and y ,

$$\begin{aligned} F(\vec{f}, \vec{x}, 0) &= G(\vec{f}, \vec{x}) \\ F(\vec{f}, \vec{x}, y) &= H(\vec{f}, \vec{x}, y, F(\vec{f}, \vec{x}, \lfloor y/2 \rfloor)), \text{ if } y > 0, \\ |F(\vec{f}, \vec{x}, y)| &\leq |K(\vec{f}, \vec{x}, y)|. \end{aligned}$$

Definition 6.2.8. The class BFF of **Basic Feasible Functionals** is defined as the smallest class of type-2 functionals containing FPTIME, the application functional (i.e., $\lambda f x. f(x)$), and it is closed under composition, expansion, and limited recursion on notation.

¹A relativization of a complexity class C is, intuitively, the replacement of the usage of Turing machines occurring in the machine-based definition of C by *oracle Turing machines*. For instance, we relativize the definition of NP as follows. We say that a language L is in NP^A iff there is a relation R and a polynomial q such that membership in R can be checked in polynomial time by an oracle Turing machine A , and such that $x \in L \Leftrightarrow \exists y. |y| \leq p(|x|) \wedge (x, y) \in R$. A subtle but important note is that relativization is not an operation that is applied to a complexity class, such as NP, and an oracle A to get the relativized class NP^A . Indeed, if relativization was an operation on complexity classes, then we would have that $\text{P} = \text{NP}$ would imply $\text{P}^A = \text{NP}^A$ for any oracle A , which is not the case. See [91, Section 14.3].

This is the definition we will consider for higher-order feasibility. For an overview of characterizations of this class, we refer the reader to [57]. Recently, a new characterization of BFF also appears in [45]. Taking such works as inspiration — in particular the work done in [46, 60] — we set out to provide a rewriting-based characterization of higher-order feasibility via our notion of tuple interpretations. The characterization we will provide makes usage of *Oracle Turing Machines* and uses the characterization of Kapron and Cook as in Theorem 6.4.1.

Simply Typed Rewriting Systems. In this chapter, we consider “terms without lambdas” in order to simplify our proofs. Hence, we consider the set $T(\mathbb{F}, \mathbb{X})$ by removing the abstraction rule in Definition 2.1.4. The definitions for substitutions, rewriting rules, and so on, remain the same but without the abstraction case. The interpretation theory we use is that of Chapter 5. So reduction follows a call-by-value strategy, and the rewriting relation is given by Definition 5.1.4. The interpretation of terms is from Definition 5.4.4, again, with terms without abstractions. So the abstraction case is never used in this chapter. We use the acronym (STRS) for such rewriting systems as this restriction essentially coincides with the formalism introduced by Kusakari [73]. An orthogonality condition will be required in this chapter, and it is defined as usual. In this chapter, we work with STRSs assuming orthogonality holds. Orthogonality is required for efficiency of graph rewriting, which we provide in Section 6.6.2.

Moreover, for the rest of this chapter, rules have base types. Hence, to satisfy the requirement that values have zero cost component (Definition 5.4.7) \mathcal{J}_f must be as in

$$\mathcal{J}_f = \langle (0, \lambda x_1.(0, \dots, \lambda x_m.(C, u))), \lambda x_1 \dots x_m.S \rangle$$

whenever $\text{typeOf}(f) = \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \iota$. We will then write \mathcal{J}_f^c for the function C and \mathcal{J}_f^s for the function S . In this way, \mathcal{J}_f^c and \mathcal{J}_f^s are functions such that

$$\mathcal{J}_f = \langle (0, \lambda x_1.(0, \dots, \lambda x_m.(\mathcal{J}_f^c(x_1, \dots, x_m), u))), \lambda x_1 \dots x_m.\mathcal{J}_f^s(x_1, \dots, x_m) \rangle$$

Additionally, for terms of a base type, we will often suggestively write $\llbracket s \rrbracket^c$ and $\llbracket s \rrbracket^s$ to denote their numeric cost component (so without u) and size components, respectively.

Restrictions to BFF. While BFFs are in principle defined with the potential for multiple functional and numeric parameters, in the formal definitions in this chapter, we follow what is done in Kapron and Cook [60] and limit proofs to single-oracle machines, so they do not become cumbersome. We will also assume only a single input of type-0 and another of type-1 (so $k = l = 1$). These definitions and proofs generalize naturally to the multi-tape setting. We choose to proceed with a restricted assumption in the proofs (and the formal definition below) to avoid bureaucracy that would obfuscate intuition.

6.3 Oracle Turing Machines

We assume familiarity with Turing Machines and their various equivalent extensions, see [91, Definition 2.1]. The notion of Turing machine used to define Basic Feasible Functionals is that of *deterministic multi-tape Turing machines*. This is, conceptually, a machine consisting of a finite set of internal control states and one or more (but a fixed number of) right-infinite *tapes* divided into cells. Each tape is equipped with a tape head that scans the symbols on the tape and may write on it. The head can move to the left or right. The terminology “right-infinite tape” means that it has no rightmost cell but does have a leftmost one. When the head is reading the leftmost cell, it is not allowed to move left.

Consider $f : \mathbb{N} \rightarrow \mathbb{N}$. An *oracle Turing machine* (OTM) over f , written M_f , is a deterministic 3-tape Turing machine, that has the following designated components:

- a starting state;
- one input tape;
- two distinguished working tapes, which we call the **query** tape and **answer** tape of the oracle f ;
- additional distinct distinguished states: a **query state** query and a **answer state** answer

This is formally expressed in the definition below.

Definition 6.3.1. An **Oracle Turing machine** M with oracle function f — written M_f — is a tuple $(Q, \Gamma, \text{start}, \text{end}, \text{query}, \text{answer}, f, \mathcal{T})$ consisting of the following components:

1. $Q \supseteq \{\text{start}, \text{end}, \text{query}, \text{answer}\}$ is a finite set of *states* such that start, end, query, and answer are all pairwise distinct;
2. $\Gamma \supseteq \{0, 1, B\}$ is a finite set of *tape symbols*;
3. f is a function in $\mathbb{N} \rightarrow \mathbb{N}$;
4. \mathcal{T} is a finite set of *transitions* (i, r, t, v, d, j) such that
 - (a) $i \in Q \setminus \{\text{query}, \text{end}\}$ is the *original state*,
 - (b) $r \in \Gamma$ is the *read symbol*,
 - (c) $t \in \{1, 2, 3\}$ is the *tape of interest*,²

²Commonly in definitions of multi-tape Turing Machines, a machine can simultaneously move each one of its head in one step of its computation. However, in order to facilitate a less cumbersome rewriting encoding in Section 6.7, we assume that at each transition the machine moves only one head, and looks only at one tape. This assumption does not pose any threat to the generality of our results since it is in practice a sequentialization of the parallel moves of the standard multi-tape machine. Indeed, it is easy to see that these two models can simulate each other with polynomial overhead.

- (d) $v \in \Gamma$ is the *written symbol*,
- (e) $d \in \{L, R\}$ is the *direction*,
- (f) $j \in Q$ is the *result state*.

We only consider deterministic machines in this chapter, so we put the following restrictions on the set \mathcal{T} of transitions:

1. for all $i \in Q \setminus \{\text{query}, \text{end}\}$ and $r \in \Gamma$ there is exactly one transition (i, r, t, v, d, j) in \mathcal{T} , and
2. for all $i \in Q \setminus \{\text{query}, \text{end}\}$: if both $(i, r, t, v, d, j) \in \mathcal{T}$ and $(i, r', t', v', d', j') \in \mathcal{T}$, then $t = t'$; that is, a given state only looks at one tape.

Now that we formally defined our notion of oracle Turing machines, we need to formally capture what we mean by *tape*. Recall that intuitively we idealize a machine's tape consisting of an infinite number of *cells*. At each cell, the machine may write a symbol from its alphabet. Hence, if n is a natural number and the word $w_0w_1 \dots w_m$ is its binary representation, it can be written on a machine's tape as in the figure below.

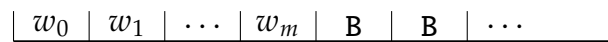


Fig. 6.1 Illustration of a tape.

Notice that words written on the tape are finite, but since the tape is infinite we assume all the other cells are filled in with a special symbol B.

Definition 6.3.2. A **tape** is a function $h : \mathbb{N} \rightarrow \Gamma$ such that $\{n \mid h(n) \neq B\}$ is finite.

If w is a nonempty word $w_0w_1 \dots w_n$ over Γ^+ , we denote $\langle w \rangle$ for the tape function mapping i to w_i if $0 \leq i \leq n$, and to B otherwise. Intuitively, $\langle w \rangle$ is the tape $w_1 \dots w_n \text{BBB} \dots$. This allows us to formally express "writing a word w to a tape". Similarly, to *read* from a tape, let us denote $h|_p$ for the binary string starting at position p ; that is, if $h(p+i) \in \{0, 1\}$ for $0 \leq i < n$, and $h(p+n) \notin \{0, 1\}$, then $h|_p$ is the number whose binary representation corresponds with $h(p)h(p+1) \dots h(p+n-1)$. If $n = 0$ then $w = \epsilon$, so we let $h|_p := 0$. Note that for any tape h , the number $h|_p$ is always defined and $\langle w \rangle$ is a tape for any string w as above.

We have not formally defined how a machine computes yet. However, let us once again invoke intuition and recall how we conceptualize its computation. Indeed, suppose that at a certain moment of its computation the machine is at some state q and its head is reading the symbol w_i from a tape. The figure below describes this scenario.

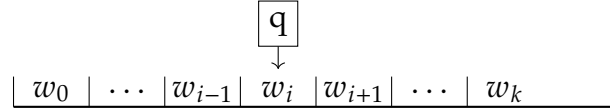


Fig. 6.2 The machine is reading w_i on this tape and is at state q .

Then, according to its internal controller (which by Definition 6.3.1 is a transition tuple in \mathcal{T}), it changes to a new state s , writes a new symbol w'_i at this tape's position, and moves the head to the left or right. We pick a movement to the right as an example:

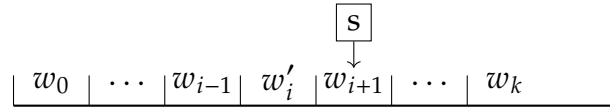


Fig. 6.3 Illustration of a tape.

Now, the machine is in the state s and reading the symbol w_{i+1} on the tape. The Section 6.3 above describe the machine's *configuration* for a tape before and after a computational step, respectively. As such, so they serve as a snapshot of what the machine is doing. This is valid for one single tape in the oracle machine. However, we want to express this configuration of the machine for all tapes.

Definition 6.3.3. Let $M_f = (Q, \Gamma, \text{start}, \text{end}, \text{query}, \text{answer}, f, \mathcal{T})$ be an oracle Turing machine. A **configuration** of M_f is a tuple $(s, h_1, h_2, h_3, p_1, p_2, p_3)$ where s is a state in Q , h_1, h_2, h_3 are valid tapes, and $p_1, p_2, p_3 \in \mathbb{N}$ are positions in those tapes, respectively.

So a configuration $(s, h_1, h_2, h_3, p_1, p_2, p_3)$ tells us the current state of the machine, the contents of each one of its tapes, and the positions of the respective tape heads. This definition is good for a formal representation. In practice, however, we write configurations as words in order to ease notation and evoke intuition. So we may write a configuration using the special symbol $\# \notin \Gamma$ to indicate the position of each tape head as follows:

$$(q, w_0 \dots w_{p_1-1} \# w_{p_1} \dots w_k, u_0 \dots u_{p_2-1} \# u_{p_2} \dots u_l, v_0 \dots v_{p_3-1} \# v_{p_3} \dots v_m)$$

This configuration that uses words represents the same information as the more formal one given in Definition 6.3.3. The only difference is that here we accept that after the end of a word written on the tape, there are only blanks, so we can ignore them. This is particularly useful for examples and our rewriting encoding of OTMs in Section 6.7. Notice however that for formal proofs Definition 6.3.3 is used.

6.3.1 Computing with Oracle Turing Machines

Based on our previous discussions, it is natural to expect the computation of a machine to be expressed as stepwise transformations on the configuration tuples that respect the transition tuples in \mathcal{T} . This is the case indeed, and we formalize this intuition in the definition below.

Definition 6.3.4. Let $M_f = (Q, \Gamma, \text{start}, \text{end}, \text{query}, \text{answer}, f, \mathcal{T})$ be an oracle Turing machine. We say a configuration (s, \vec{h}, \vec{p}) **steps** to the configuration (q, \vec{g}, \vec{o}) — which we denote by $(s, \vec{h}, \vec{p}) \rightsquigarrow (q, \vec{g}, \vec{o})$ — if one of the following holds:

1. **(Transition Step).** There exists a transition $(s, r, t, v, d, q) \in \mathcal{T}$ such that
 - (a) $h_t(p_t) = r$ and $g_t(p_t) = v$, and $h_t(n) = g_t(n)$ for $n \in \mathbb{N} \setminus \{p_t\}$;
(The symbol read by the machine at tape t and position p_t is r . It is replaced by v in the next configuration. All other symbols remain unaltered.)
 - (b) if $d = R$ then $o_t = p_t + 1$; and if $d = L$ then $o_t = p_t - 1$
(The position of the head at the relevant tape is changed following d .)
 - (c) For each $t' \in \{1, 2, 3\} \setminus \{t\}$, we have $g_{t'} = h_{t'}$ and $o_{t'} = p_{t'}$.
(The tapes other than the relevant one are unaltered, and their respective head does not move.)
2. **(Query Step).** If $s = \text{query}$ and $q = \text{answer}$ then
 - $g_1 = h_1$ and $o_1 = p_1$
(The input/output tape is unaltered, and its head does not move.)
 - $g_2 = i \mapsto B$ and $o_2 = 0$
(The query tape is erased, and its head is moved to the start of the tape.)
 - $g_3 = \langle f(h_3|p_3) \rangle$ and $o_3 = 0$
(The new answer tape is obtained by reading the binary word starting at the head of the query tape, then executing f on this word and printing the result in binary at the start of the answer tape, while moving its head to the start of this answer.)

Definition 6.3.5. An OTM *halts* whenever it enters the final state end or whenever a transition to the left, i.e., $o_t = p_t - 1$, is undefined.

Notice that it is always possible to move to the right, but moving to the left from the leftmost position of a type (so $p_t = 0$) would cause $o_t = p_t - 1 = -1$ which is not in \mathbb{N} . So this is the case when the machine halts via an invalid transition.

Definition 6.3.4 above formalizes how one step of a machine computation occurs. In essence, computation proceeds in a standard way for non-query states. However, if at some step of the computation the machine wants to know the value of some f on a word x , it writes the argument x on the query tape, moves the read head to the beginning of that tape, and enters the special query state `query`. Then **in one step**:

- the contents of the query tape are read, let x be the corresponding number;
- the contents of the answer tape are changed to the image of f on x ;
- the head of the answer tape gets moved to its first symbol;
- the machine transitions to the answer state `answer`.

We observe that when the machine transitions to the answer state, `answer`, the contents of the answer tape are changed to $f(x)$ in one step; regardless of the length of x or $f(x)$, or even the computational complexity of determining the value of f on x . After this, computation resumes as usual. In this thesis, we only consider machines that halt on all inputs, so such computations are always finite.

Remark 6.3.6. It is important to note that the requirement of having one tape for calling the oracle and another distinct one to get its answer is strictly necessary. The reason is that due to this restriction iterated calls to the oracle, i.e., $f(f(\dots f(x)\dots))$, can only be computed by copying each intermediate result of the application from the answer tape to the query tape. Dropping this restriction would make the machine substantially more powerful, and potentially bring us outside of BFF.

We are usually interested in a full computation of the machine, that is, a computation starting with an input x and ending with an output y .

Definition 6.3.7. Let $x \in \mathbb{N}$. The **initial configuration** $\text{initial}(x)$ to execute M_f on x is $(\text{start}, \langle x \rangle, 0, i \mapsto \text{B}, 0, i \mapsto \text{B}, 0)$. A configuration of the form $(\text{end}, \vec{h}, \vec{p})$ is a **final configuration**, and it *yields* y if $h_1 | p_1 = y$.

A computation of M_f on an input x proceeds by starting with the initial configuration and taking \rightsquigarrow steps until a final configuration is reached.

Definition 6.3.8. Let M_f be an OTM and x a natural number. A **computation** of M_f on x is a finite sequence of configuration steps such that $\text{initial}(x) \rightsquigarrow \dots \rightsquigarrow (\text{end}, \vec{h}, \vec{p})$.

We observe that a computation with an OTM M_f requires the oracle function f to be previously fixed. However, $Q, \Gamma, \text{start}, \text{end}, \text{query}, \text{answer}, \mathcal{T}$ do not depend on f . Hence, given an OTM M_f , we can let f range over $\mathbb{N} \rightarrow \mathbb{N}$ to obtain a map that sends each oracle function f to the OTM M_f . Henceforth, we will call both the OTM M_f and the functional $f \mapsto M_f$ "Oracle Turing Machines". Notwithstanding, we write M for this

functional and M_f for its image under f . This allows us to write M for an OTM and say that it receives as input a type-1 argument f and a type-0 argument x . We then write $M_f(x)$ for the computation of M given f and x as input.

A Machine-Based Computational Model for Type-2 Functionals. In type-1 computability, we say a Turing machine *computes* a function $f : \mathbb{N} \rightarrow \mathbb{N}$ whenever for all x , with $\langle x \rangle$ written on its input tape, the machine halts in a final configuration that yields $\langle f(x) \rangle$. We say this notion of computability is finitary in the sense that all arguments to the computation are finite, that is, of type-0. Kapron and Cook [60] then extend type-1 computability by adding to it infinitary objects, i.e., the functional parameters.

Definition 6.3.9 (Type-2 Computability). Let $\Psi : (\mathbb{N} \rightarrow \mathbb{N}) \times \mathbb{N} \rightarrow \mathbb{N}$ be a type-2 functional. We say the oracle Turing machine M **computes** Ψ if and only if for all type-1 f and for all type-0 x : if M_f is started with initial configuration $\text{initial}(x)$, it halts with a final configuration that yields $\Psi(f, x)$.

6.3.2 Complexity of Oracle Turing Machines

Recall that the running time of a Turing machine is just the number of steps that it executes before halting. This cost model is natural as each step of a Turing machine is *atomic* in the sense that at each step it can only change a single bit of information on the tape. Hence, *running time* and *number of steps* coincide. For oracle Machines, however, that is not the case. Indeed, whenever an OTM queries the oracle it creates more than a bit worth of information in one single step. The following question then arises: “what is a reasonable *cost model* for OTMs?”

There are two possibilities. The first, which is usually called *unit cost model* is to charge the machine exactly one unit of time for calling the oracle. This cost model reflects the intuition that oracles are all-powerful and external to the system. So querying an oracle only costs the time of “calling it”. Observe that in this model we still take into account the time needed to write the query string in the query tape; and similarly, the time required to read from the answer tape. The second approach is to charge the length of the oracle’s answer. This cost model reflects the fact that the value returned by the oracle must still be written down on the answer tape and read by the machine. So even though the oracle is like a subroutine with infinite power, its answer must still be read by the caller.

In this thesis, we follow Kapron and Cook [60] and choose the latter. More precisely, if we query the oracle f on input x , the associated cost is $\max(1, |f(x)|)$. As usual, a non-query step in an OTM computation costs 1 unit of time. This cost model is easier to work with since queries to the oracle have an explicit size component which, as we shall see later in this chapter, plays nicely with tuple interpretations. In [79] Mehlhorn works

with the unit cost model, and Kapron and Cook [60] observed that the two models are equivalent.

Definition 6.3.10. The **running time** of an OTM with a given input is the sum of the costs of the steps on its execution. We denote by $\text{TIME}_M(f, x)$ the running time of M on inputs f and x .

Notice that in a Turing machine computation the number of steps from an initial configuration $\text{initial}(x)$ to a final configuration (end, h, p) is exactly the time measure. However, because of the cost model we have chosen for oracle calls, the number of steps in an oracle machine computation is not equal to its running time, as oracle calls are atomic steps with nonunit cost.

As usual in complexity theory, we would like to be able to supply upper bounds for the running time of an OTM in terms of the size of its inputs. To do this, we must define what the size of an input means. This is straightforward for elements of \mathbb{N} (i.e., the type-0 arguments): we let $|x|$ be its *size* (recall that $|x| = \lceil \log_2(x + 1) \rceil$). For functions, it would be of no surprise that we use a *size functional*, in the same fashion of size interpretations we studied in the rewriting setting.³ Therefore, we define the *length* of a type-1 functional f as the function below.

Definition 6.3.11. For any type-1 $f : \mathbb{N} \rightarrow \mathbb{N}$ its **length** is the function defined by

$$|f| = \lambda x. \max_{|y| \leq x} |f(y)|$$

Second-Order Polynomials. In order to majorize the running time of OTMs we define the class of second-order polynomials as follows.

Definition 6.3.12. Let $\{x_1, \dots, x_l\}$ be a set of *type-0 variables* and $\{F_1, \dots, F_k\}$ be a set of *type-1 variables*. The set $\text{Pol}_{\mathbb{N}}^2[F_1, \dots, F_k; x_1, \dots, x_l]$ of **second-order polynomials** over \mathbb{N} with indeterminates $F_1, \dots, F_k; x_1, \dots, x_l$ is generated by the grammar:

$$P, Q := n \mid x \mid P + Q \mid P * Q \mid F(Q)$$

where $n \in \mathbb{N}$, $x \in \{x_1, \dots, x_l\}$, and $F \in \{F_1, \dots, F_k\}$.

Example 6.3.13. Examples of polynomial expressions are $P = F(x + 1) + x^2 + 3$ and $Q = (x + 1) * F(y) + 3$.

Let type-0 variables range over elements of \mathbb{N} and type-1 variables range over elements of $\mathbb{N} \rightarrow \mathbb{N}$. We then interpret $+, *$ as addition and multiplication over

³Indeed, as we shall see in this chapter, the innovation brought by tuple interpretations, i.e., the ability to explicitly split cost and size, makes reasoning about those size functionals easier. Yes, of course, we are going to use tuple interpretations and higher-order rewriting to capture BFF. I am a *rewriter* after all.

\mathbb{N} , respectively. Therefore, polynomial expressions can be interpreted as type-2 functionals in $(\mathbb{N} \rightarrow \mathbb{N})^k \rightarrow \mathbb{N}^l \rightarrow \mathbb{N}$. If P is a second-order polynomial in $\text{Pol}_{\mathbb{N}}^2[F_1, \dots, F_k; x_1, \dots, x_l]$, we write $\lambda F_1 \dots F_k x_1 \dots x_l. P$ to denote the associated type-2 functional. We also write $\lambda \vec{F} \vec{x}. P$ as shorthand notation for such type-2 functionals.

Example 6.3.14. The polynomials in Example 6.3.13 can be viewed as the type-2 functional $P = \lambda F x. F(x + 1) + x^2 + 3$ and $Q = \lambda F x y. (x + 1) * F(y) + 3$, respectively. When those are applied to a concrete argument we may write $P(F, x)$ and $Q(F, x)$.

Lemma 6.3.15. If P is a second-order polynomial in some $\text{Pol}_{\mathbb{N}}^2[F_1, \dots, F_k; x_1, \dots, x_l]$, then the associated type-2 functional $\lambda \vec{F} \vec{x}. P$ is weakly monotonic on each one of its arguments.

Proof. By induction on the structure of P and observing that addition, multiplication, and functional application over \mathbb{N} are all weakly monotonic. \square

6.4 Kapron and Cook's Characterization of BFF

In [60], Kapron and Cook characterized BFF as those functionals that can be computed by OTMs whose running times are majorized by second-order polynomials in terms of the size of their arguments. This is stated as the theorem below.

Theorem 6.4.1. A type-2 functional Ψ is in BFF if and only if there exists an oracle Turing machine M and a second-order polynomial P such that M computes Ψ and for all f and x , $\text{TIME}_M(f, x) \leq P(|f|, |x|)$.

6.5 From Higher-Order Rewriting to BFF and Back Again

The main result of this chapter is to show that BFFs are captured by a class of STRSs with second-order polynomially bounded interpretations. To formally state this result in Theorem 6.5.9, we must first define what it means for an STRS to compute a higher-order function.

6.5.1 Type-2 Computability via Higher-Order Rewriting

We start by representing a natural number in \mathbb{N} as a term over a signature \mathbb{F} .

Definition 6.5.1. Let $C_{\mathcal{N}}$ be a set of data constructors. This set **defines** \mathbb{N} if there is a bijection $\mathbb{N} \rightarrow \mathcal{N}$, with $\mathcal{N} \subseteq \text{T}(C_{\mathcal{N}})$. That is, each $n \in \mathbb{N}$ has a unique data term representation $\lfloor n \rfloor$.

Remark 6.5.2. There are multiple ways of representing natural numbers as terms. In the earlier chapters, we have used for instance unary encoding. In this chapter, however, the term encoding of \mathbb{N} will be mainly on binary numbers. That choice makes our proofs more sensible for the efficiency of the encoding and the fact that we also deal with low-level Turing machines that in practice use binary encoding. Therefore, to differentiate from the previous chapters we write such terms as $\lfloor n \rfloor$. Similarly, we call the base type of such encoding terms bnat .

Example 6.5.3 (Encoding Binary Numbers as Terms). In order to encode the binary representation of numbers in \mathbb{N} as terms, we let $\text{bit}, \text{bnat} \in \mathbb{B}$ and introduce symbols $\text{o}, \text{i} : \text{bit}$ and $[\] : \text{bnat}, :: : \text{bit} \Rightarrow \text{bnat} \Rightarrow \text{bnat}$. Then for instance 001 is encoded as the term $:: \text{o} (:: \text{o} (:: \text{i} [\]))$. We use the cleaner list-like notation $[\text{o}; \text{o}; \text{i}]$ in practice.

We also fix the following interpretation for these data constructors:

$$\mathcal{J}_\text{o} = \langle 0, 0 \rangle \quad \mathcal{J}_\text{i} = \langle 0, 0 \rangle \quad \mathcal{J}_{::} = \left\langle (\lambda x q. 0) , \lambda x q. 1 + q \right\rangle$$

So, the size of such encoding is the length of the corresponding binary word.

Example 6.5.4 (Implementing Binary Addition). Let us implement binary addition. For this purpose, we consider binary sequences written in *little-endian* format, i.e., the most significant digit is at the head of the list. So the binary number 110 (in *big-endian* notation) is written as 011 in little-endian notation. As a data term, this is equivalent to reversing the list $[\text{i}; \text{i}; \text{o}]$. We need the following logical operations on bit symbols.

$$\begin{array}{llll} \text{o xor o} \rightarrow \text{o} & \text{i xor i} \rightarrow \text{o} & \text{o xor i} \rightarrow \text{i} & \text{i xor o} \rightarrow \text{i} \\ \text{o and o} \rightarrow \text{o} & \text{i and i} \rightarrow \text{i} & \text{o and i} \rightarrow \text{o} & \text{i and o} \rightarrow \text{o} \\ \text{o or o} \rightarrow \text{o} & \text{i or i} \rightarrow \text{i} & \text{o or i} \rightarrow \text{i} & \text{i or o} \rightarrow \text{i} \end{array}$$

We can interpret the rules above as follows. The cost–size interpretation for each constructor i, o is $\langle 0, 0 \rangle$. The cost component of $\text{xor}, \text{and}, \text{or}$ are given by $\lambda xy. 1$ and size component by $\lambda xy. 0$.

The rules defining $\text{aux} : \text{bnat} \Rightarrow \text{bnat} \Rightarrow \text{bit} \Rightarrow \text{bnat}$ below compute the bitwise addition and carrying value recursively on the size of the input lists. We define such a

function using case distinction on the shape of the two input numbers.

$$\begin{aligned}
& \text{aux } [] [] \text{o} \rightarrow [] \\
& \text{aux } [] [] i \rightarrow i :: [] \\
& \text{aux } (a :: []) (b :: []) \text{acc} \rightarrow ((a \text{ xor } b) \text{ xor } \text{acc}) :: \text{aux } [] [] (((a \text{ xor } b) \text{ and } \text{acc}) \text{ or } (a \text{ and } b)) \\
& \text{aux } (a :: []) (b :: b' :: bs) \text{acc} \rightarrow \\
& \quad ((a \text{ xor } b) \text{ xor } \text{acc}) :: \text{aux } (\text{o} :: []) (b' :: bs) (((a \text{ xor } b) \text{ and } \text{acc}) \text{ or } (a \text{ and } b)) \\
& \text{aux } (a :: a' :: as) (b :: []) \text{acc} \rightarrow \\
& \quad ((a \text{ xor } b) \text{ xor } \text{acc}) :: \text{aux } (a' :: as) (\text{o} :: []) (((a \text{ xor } b) \text{ and } \text{acc}) \text{ or } (a \text{ and } b)) \\
& \text{aux } (a :: a' :: as) (b :: b' :: bs) \text{acc} \rightarrow \\
& \quad ((a \text{ xor } b) \text{ xor } \text{acc}) :: \text{aux } (a' :: as) (b' :: bs) (((a \text{ xor } b) \text{ and } \text{acc}) \text{ or } (a \text{ and } b))
\end{aligned}$$

We then set the following interpretation:

$$\mathcal{J}_{\text{aux}} = \left\langle (\lambda x y a.3 + 6 \max(x, y)) , \lambda x y a.1 + \max(x, y) \right\rangle$$

Finally, we write the addition of binary numbers as the rule $x +_{\text{B}} y \rightarrow \text{aux } x y \text{o}$.

Definition 6.5.5. A set of rules \mathbb{R}_f over $C_{\mathcal{N}} \cup \{\mathbf{S}_f : \text{bnat} \Rightarrow \text{bnat}\}$ **defines** a function $f : \mathbb{N} \rightarrow \mathbb{N}$ by way of the symbol \mathbf{S}_f if and only if for each $n, m \in \mathbb{N}$ such that $m = f(n)$, the rule $\mathbf{S}_f [n] \rightarrow [m]$ is in \mathbb{R}_f and there is no other other rule $\ell \rightarrow r$ in \mathbb{R}_f with $\ell = \mathbf{S}_f [n]$.

Intuitively, this infinite set of rules simulates an oracle f , which, in one step of computation provides us with the value $[f(x)]$. In the next definition, we extend a finite TRS with such an oracle. Henceforth, we assume given that our STRS (\mathbb{F}, \mathbb{R}) at hand is such that \mathbb{F} contains $\text{o}, i, [], ::$ typed as above and a distinguished symbol $\mathbf{F} : (\text{bnat} \Rightarrow \text{bnat}) \Rightarrow \text{bnat} \Rightarrow \text{bnat}$.

Definition 6.5.6. Let a distinguished function symbol $\mathbf{F} \in \Sigma$ of type $(\text{bnat} \Rightarrow \text{bnat}) \Rightarrow \text{bnat} \Rightarrow \text{bnat}$ be given, and assume given a type-1 function $f : \mathbb{N} \rightarrow \mathbb{N}$, and fresh symbols $\mathbf{G}, \mathbf{S}_f : \text{bnat} \Rightarrow \text{bnat}$ not in Σ . We write $\mathbb{R}_{\mathbf{F}, f, \mathbf{G}}$ for the infinite TRS

$$\mathbb{R} \cup \mathbb{R}_f \cup \{\mathbf{G} x \rightarrow \mathbf{F} \mathbf{S}_f x\}.$$

This definition allows us to state what it means for an STRS to compute a second-order function. As it is done with higher-order computability for OTMs in Definition 6.3.9, we start with a finitary notion of computability.

Definition 6.5.7 (Type-1 Computability). Let (\mathbb{F}, \mathbb{R}) be a TRS and $\mathbf{f} \in \Sigma$. We say that the symbol \mathbf{f} **computes** a type-1 function $f : \mathbb{N} \rightarrow \mathbb{N}$ whenever $\mathbf{f} [n] \xrightarrow{+} [m]$ if and only if $f(n) = m$.

Then we extend this notion by adding to it the infinitary parameters.

Definition 6.5.8 (Type-2 Computability). We say that in a finite TRS \mathbb{R} the function symbol $F : (\text{bnat} \Rightarrow \text{bnat}) \Rightarrow \text{bnat} \Rightarrow \text{bnat}$ **computes** the type-2 functional $\Psi : (\mathbb{N} \longrightarrow \mathbb{N}) \longrightarrow \mathbb{N} \longrightarrow \mathbb{N}$ if and only if for every type-1 function f in $\mathbb{N} \longrightarrow \mathbb{N}$, the TRS $\mathbb{R}_{F,f,G}$ is such that the symbol G computes $\Psi(f)$.

This finally allows us to state the main result:

Theorem 6.5.9 (BFF Characterization Theorem). A type-2 functional Ψ is in BFF if and only if there exists an orthogonal STRS (Σ, \mathbb{R}) such that $F \in \Sigma$ computes Ψ , and which admits a second-order polynomially-bounded interpretation such that

1. there exist constants $c \geq 1$ and $d \geq 0$ such that for all $n \in \mathbb{N}$ we have $|n| \leq \llbracket \lfloor n \rfloor \rrbracket^s \leq c * |n| + d$, and
2. \mathcal{J}_F^c and \mathcal{J}_F^s are polynomially bounded.

In this chapter, we make a slight abuse of nomenclature and say that such interpretations are polynomially bounded to mean that the function of interest F is polynomially bounded. This comes from the fact that we can in practice interpret the size components of other data types — which are not `bnat` — using sets that are not numeric. Indeed, the only required numeric notion of size should be given to `bnat` in order to even talk about polynomially bounded interpretations.

In the following sections, we will prove this result. We will refer to an interpretation that satisfies the property that always $|n| \leq \llbracket \lfloor n \rfloor \rrbracket^s \leq c * |n| + d$ as a *bnat-size reflecting interpretation*. Note that if constructor interpretations are additive, we can always find c, d such that $\llbracket \lfloor n \rfloor \rrbracket^s \leq c * |n| + d$. However, the property also restricts us from, for instance, letting $\llbracket \lfloor n \rfloor \rrbracket^s$ be the number of 0s in the binary representation of n . This is needed to ensure that the interpretation can be extended to include S_f .

Proving Theorem 6.5.9 requires non-trivial work. We split this proof in two major properties that in the literature are called *soundness* and *completeness* of the characterization. This first property is to ensure that whenever a type-2 functional can be computed by a STRS satisfying our conditions it is a member of BFF. The second is to ensure that STRSs satisfying our requirements capture all type-2 functionals. In essence, soundness states that we are correct in our conditions to model elements of BFF and completeness state that we do not “miss” functionals, i.e., the entire class is captured.

Example 6.5.10. Let us consider the type-2 functional defined by $\Psi := \lambda f x. \sum_{i < |x|} f(i)$.

Notice that Ψ adds all $f(i)$ over each word $i \in \mathbb{N}$ whose value (as a natural number) is smaller than the length of x . This functional was proved to lie in BFF in [56], where the authors utilized an encoding of Ψ as a $\text{BTL}P_2$ program. We can encode Ψ as an STRS as

follows. Let us consider ancillary symbols $\text{lengthOf} : \text{bnat} \Rightarrow \text{nat}$ and $\text{toBin} : \text{nat} \Rightarrow \text{bnat}$. The former computes the length of a given word and the latter converts a number from unary to binary representation. We also consider rules for addition on binary words, i.e., $+_B : \text{bnat} \Rightarrow \text{bnat} \Rightarrow \text{bnat}$, which we write using infix notation below.

$$\begin{aligned} & \text{compute } F \ x \ 0 \ acc \rightarrow acc \\ & \text{compute } F \ x \ (s \ i) \ acc \rightarrow \text{compute } F \ x \ i \ (acc \ +_B \ F(\text{toBin } i)) \\ & \text{start } F \ x \rightarrow \text{compute } F \ x \ (\text{lengthOf } x) \ [] \end{aligned}$$

Now, if we want to compute $\Psi(f, x)$ we simply normalize the term $\text{start } S_f \ [x]$.

6.6 Soundness

In this section, we prove soundness, that is, *if* a type-2 functional Ψ is computed by an orthogonal STRS that admits a suitable interpretation, *then* it is in BFF. The proof of this takes roughly the following form:

- If the STRS that computes Ψ admits a suitable second-order polynomially-bounded interpretation, then so does the extended STRS $\mathbb{R}_{F,f,G}$. The restriction that the interpretation is bnat-size reflecting implies that both $\llbracket G \ [n] \rrbracket^c$ and $\llbracket G \ [n] \rrbracket^s$ are bounded by second-order polynomials over $|f|, |n|$. Since $\llbracket s \rrbracket^c > \llbracket t \rrbracket^c$ whenever $s \rightarrow_{\mathbb{R}} t$, the former bounds the number of steps we can do, starting in $G \ [n]$, by a polynomial. The latter bounds the size of the normal form.
- The cost polynomial also restricts the size of any input that the function variable F is applied to (since, for instance, a cost bound of $3 + F^c(\llbracket n \rrbracket^s)$ implies that F is never called on a term with size interpretation $> \llbracket n \rrbracket^s$).
- By using the observation above, and using sharing to handle data duplication (for instance by a rule $f \ x \rightarrow g \ x \ x$), we can represent terms without excessive space blowup, and compute each reduction step in only polynomially many steps on an oracle Turing machine.

In the remainder of this section, we assume given a fixed STRS (Σ, \mathbb{R}) , and a second-order polynomially-bounded bnat-size reflecting interpretation that orients \mathbb{R} .

6.6.1 Interpreting the extended TRS

To start, we will extend the interpretation function to the system $\mathbb{R}_{F,f,G}$, which has infinitely many rules defining f by S_f . The first question regarding this extended system is the following: “how do we interpret the additional symbols S_f and G ”?

Since S_f basically represents a call for an oracle to compute f in a single step, we can set its cost component to 1. So we get

$$\mathcal{J}_{S_f}^c = \lambda x.1 \quad (6.1)$$

Meanwhile, the size interpretation of S_f is given by the length of the oracle output, just like in Definition 6.3.11. We can hence define $\llbracket S_f \rrbracket^s$ in terms of $|f|$ as follows: let $c \geq 1, d \geq 0$ be the integers such that always $|n| \leq \llbracket [n] \rrbracket^s \leq c * |n| + d$. Then

$$\mathcal{J}_{S_f}^s = \lambda x.c * |f|(x) + d \quad (6.2)$$

This is weakly monotonic because $|f|$ is, and we can interpret the infinitely many rules defining f by way of S_f as follows:

$$\begin{aligned} \llbracket S_f [n] \rrbracket &= \left\langle (0, \lambda x.(1, u)) , \lambda x.c * |f|(x) + d \right\rangle \cdot \langle 0, i \rangle , \\ &\quad \text{for some } i \in \{|n|, \dots, c * |n| + d\} \\ &= \langle 1, c * |f|(i) + d \rangle \\ &\succcurlyeq \langle 1, c * |f(n)| + d \rangle \quad (\text{by definition of } |f|, \text{ since } i \geq |n|) \\ &> \langle 0, c * |f(n)| + d \rangle \\ &\succcurlyeq \llbracket [f(n)] \rrbracket \end{aligned}$$

This orients each S_f rule added to the system for any function $f \in \mathbb{N} \rightarrow \mathbb{N}$.

To orient the rule $G x \rightarrow F S_f x$, we use the fact that all rules in the original system are already oriented. With this, we get

$$\mathcal{J}_G = \left\langle \left(0, \lambda x. \left(1 + \mathcal{J}_F^c \left(\left\langle \mathcal{J}_{S_f}^c, \mathcal{J}_{S_f} \right\rangle, x \right) \right) \right), \lambda x. \mathcal{J}_F^s(\mathcal{J}_{S_f}^s, x) \right\rangle$$

Notice that in the cost component of \mathcal{J}_G we just sum 1 to the total numeric component of $F S_f x$. This is to encode that exactly one step is needed to eliminate the symbol G . So we can write its interpretation as follows:

$$\llbracket G x \rrbracket = \langle 1 + \llbracket F S_f x \rrbracket^c, \llbracket F G x \rrbracket^s \rangle$$

Indeed, the rule $G x \rightarrow F S_f x$ does this in exactly one step. Sizes can remain the same, so we can choose the same size component for G as the right-hand side of the rule, which we interpreted above. The compatibility conditions can be checked by combining

the two previous equations as follows:

$$\begin{aligned}
\llbracket G x \rrbracket &= \left\langle \left(0, \lambda x. \left(1 + \mathcal{J}_F^c \left(\left\langle \mathcal{J}_{S_f}^c, \mathcal{J}_{S_f} \right\rangle, x \right) \right) \right), \lambda x. \mathcal{J}_F^s(\mathcal{J}_{S_f}^s, x) \right\rangle \cdot \langle 0, x \rangle \\
&= \left\langle 0 + 1 + \llbracket FG x \rrbracket^c, \llbracket FS_f x \rrbracket^s \right\rangle \\
&> \left\langle \llbracket FG x \rrbracket^c, \llbracket FS_f x \rrbracket^s \right\rangle \\
&= \llbracket F \rrbracket \cdot \llbracket S_f \rrbracket \cdot \llbracket x \rrbracket \\
&= \llbracket FS_f x \rrbracket
\end{aligned}$$

Which orients the rule $G x \rightarrow F S_f x$. We collect this result in the lemma below.

Lemma 6.6.1. Suppose (\mathbb{F}, \mathbb{R}) is a TRS compatible with a bnat-size reflecting cost-size tuple interpretation and $F : (\text{bnat} \Rightarrow \text{bnat}) \Rightarrow \text{bnat} \Rightarrow \text{bnat}$ is a second-order symbol in Σ . Then for any type-1 function f , the extended TRS $\mathbb{R}_{F,f,G}$ is also compatible with a bnat-size reflecting cost-size tuple interpretation.

Next, we reason about the polynomial bounds to the extended TRSs computing a type-2 functional.

Theorem 6.6.2. Let (\mathbb{F}, \mathbb{R}) be a finite TRS such that the symbol $F : (\text{bnat} \Rightarrow \text{bnat}) \Rightarrow \text{bnat} \Rightarrow \text{bnat} \in \Sigma$ computes the type-2 functional $\Psi : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rightarrow \mathbb{N}$. Suppose \mathbb{R} is compatible with a polynomially-bounded bnat-size reflecting cost-size interpretation. Then we can define second-order polynomials C, S such that, for any oracle function f and argument $x \in \mathcal{N}$, the derivation height of $G x$ is bounded by $C(|f|, |x|)$ and the size of its normal form by $S(|f|, |x|)$.

Proof. First, note that \mathcal{J}_F is polynomially bounded, so there exist second-order polynomials P^c, P^s such that $P^c \geq \mathcal{J}_F^c$ and $P^s \geq \mathcal{J}_F^s$. Then $P^c : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ (taking a cost function, a size function, and the size of an input argument, and returning a cost bound) and $P^s : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ (taking a size function and the size of an input argument, and returning a size bound). We let:

$$\begin{aligned}
C(F, z) &= 1 + P^c(\lambda x. 1, \lambda x. c * F(x) + d, c * z + d) \\
S(F, z) &= P^s(\lambda x. c * F(x) + d, c * z + d)
\end{aligned}$$

Then indeed, recall that

$$\llbracket G x \rrbracket = \left\langle 1 + \llbracket FS_f x \rrbracket^c, \llbracket FS_f x \rrbracket^s \right\rangle$$

Let us first “compress” the cost component of this interpretation, see Definition 5.4.4, hence we can write this interpretation as follows

$$\llbracket Gx \rrbracket^c = 1 + \mathcal{J}_F^c \left(\langle \mathcal{J}_{S_f}^c, \mathcal{J}_{S_f}^s \rangle, \llbracket x \rrbracket^s \right)$$

Observe that this shaded component is the component bounding the derivation height of Gx since this system respects compatibility. We continue by plugging in Equations (6.1) and (6.2) and using P^c as defined above to get the following.

$$\begin{aligned} \llbracket Gx \rrbracket^c &= 1 + \mathcal{J}_F^c \left(\langle (0, \lambda x. (1, u)), \lambda x. c * |f|(x) + d \rangle, \llbracket x \rrbracket^s \right) \\ &\leq P^c(\lambda y. 1, \lambda y. c * |f|(y) + d, c * |x| + d) \\ &\leq C(|f|, |x|) \end{aligned}$$

As the derivation height of a term is bounded by its cost interpretation, clearly C is a polynomial bound to the number of steps — in terms of $|f|$ and $|x|$ — we need to reduce Gx to normal form.

We treat the size component similarly. Indeed, we get the following:

$$\begin{aligned} \llbracket Gx \rrbracket^s &= \llbracket F S_f x \rrbracket^s \\ &= \mathcal{J}_F^s(\mathcal{J}_{S_f}^s, \llbracket x \rrbracket^s) \\ &\leq P^s(\lambda y. c * |f|(y) + d, c * |x| + d) \\ &= S(|f|, |x|) \end{aligned}$$

As for the size of the normal form we observe that if $Gx \xrightarrow{+} s$ with $s \in \mathcal{N}$ then $\llbracket Gs \rrbracket^s \geq \llbracket s \rrbracket^s \geq |s|$ since the interpretation is bnat-size reflecting. Additionally, if $x \in \mathcal{N}$ then necessarily the normal form of Gx is in \mathcal{N} since F computes Ψ by assumption. \square

Notice that this theorem does not imply Ψ is in BFF. It only guarantees that there is a polynomial bound to the runtime complexity of such systems. However, it does not immediately follow that the number of rewriting steps is a reasonable upper bound for the actual computational cost of simulating a reduction on a Turing machine. Consider for example a rule $f(s n) t \rightarrow f n (c t t)$. Every step doubles the size of the term, and in a polynomial number of steps we can create exponentially large terms. Therefore, in order to establish the soundness result, we need to show how to realize a reasonable implementation of term rewriting w.r.t. OTMs; in essence, we show that duplication is not an issue with a suitable representation of rewriting. We achieve this in the next section using term graph rewriting.

6.6.2 Term Graph Rewriting

In this section, we adapt the development of [77] to our higher-order setting. We also adapt notions from [15]. Let us start by defining our main object, i.e., labeled graphs.

Term Graphs

Definition 6.6.3. A **labeled graph** G over a set of symbols Σ is a triple $(V, \text{label}, \text{succ})$ that consists of the following components:

1. V is a finite nonempty set of vertices;
2. $\text{label} : V \rightarrow \Sigma \cup \{\text{@}\}$ is a partial function, and @ is a symbol not occurring in Σ ;
3. $\text{succ} : V \rightarrow V^*$ is a total function such that
 - (a) $\text{succ}(v) = v_1v_2$ whenever $\text{label}(v) = \text{@}$ and $\text{succ}(v) = \varepsilon$ otherwise,
 - (b) for every $v \in V$, $\text{succ}(v)$ does not contain v .

Definition 6.6.4. A **term graph** is a quadruple $(V, \text{label}, \text{succ}, \Lambda)$ such that the tuple $(V, \text{label}, \text{succ})$ is a labeled graph and Λ is a vertex in V . We call such vertex the **root** of the term graph. A vertex labeled with @ is an **application vertex**, otherwise, it is a **terminal vertex**.

Example 6.6.5. Consider G over Σ , where Σ contains the symbols from Example 5.1.2, with vertices $V = \{\Lambda, v_1, v_2, v_3, v_4\}$, $\text{label} = [v_0 \mapsto \text{@}, v_1 \mapsto \text{@}, v_2 \mapsto \text{app}]$, successors $\text{succ} = [\Lambda \mapsto v_1v_4, v_1 \mapsto v_2v_3, v_2 \mapsto \varepsilon, v_3 \mapsto \varepsilon, v_4 \mapsto \varepsilon]$, and the root symbol is Λ .

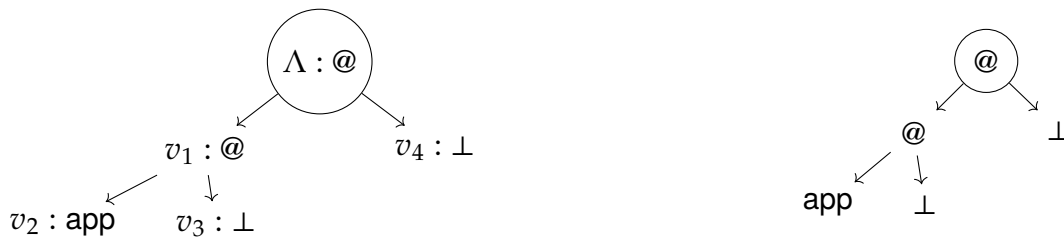


Fig. 6.4 Example of term graph

The picture on the left completely describes G with the circled vertex representing the root and the left-to-right positioning of the outgoing arrows representing the order of the successors in each application node. In the picture on the right, we simplify it by not writing the vertices names. We use the symbol \perp for the vertices where the labeling function is undefined. Each such vertex represents distinct variable names (in the corresponding term). Hence, the graph term in Example 6.6.5 represents a term of the form $\text{app } F x$. We consider those up to variable names so $\text{app } G y$ is also valid.

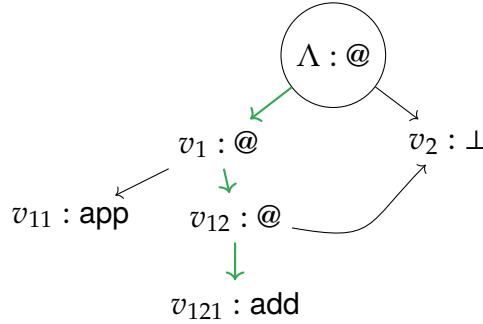
Example 6.6.6. In the following examples some vertices are *shared*. We say a vertex is shared whenever its in-degree is greater than one.



Definition 6.6.7. A (v_0, v_m) -**path** in a graph G is a sequence of vertices $v_0v_1 \dots v_m$ such that $v_{k+1} = \pi_i(\text{succ}(v_k))$, for $i = 1, 2$. The **length** of a (v_0, v_m) -path $v_0v_1 \dots v_m$ is m . A **cyclic path** in G is a (v, v) -path in G . A term graph is **acyclic** if it contains no cyclic path.

The set $\mathbb{G}(\Sigma)$ collects the acyclic term graphs over Σ . Essentially, an acyclic term graph G over Σ is a directed acyclic graph where the nodes are labeled with $@$ and some leaves may be labeled with symbols from Σ .

Example 6.6.8. A (Λ, v_{121}) -path is highlighted in the term graph below:



Definition 6.6.9. Let $G = (V, \text{label}, \text{succ}, \Lambda)$ be a term graph and $v \in V$. The **subgraph** of G rooted at v is the term graph $G|_v = (V|_v, \text{label}', \text{succ}', v)$ where $V|_v$ is the subset of V whose elements are those vertices $v' \in V$ such that there is a (v, v') -path in G . The labeling and successor functions label' , succ' are the respective restrictions of label , succ to $V|_v$.

Definition 6.6.10. A **homomorphism** between term graphs G given by the tuple $(V_G, \text{label}_G, \text{succ}_G, \Lambda_G)$ and H given by the tuple $(V_H, \text{label}_H, \text{succ}_H, \Lambda_H)$ is a function $\phi : V_G \rightarrow V_H$ that preserves the labeled graph structure. That is, it satisfies the following conditions:

$$\text{label}_H(\phi(v)) = \text{label}_G(v) \qquad \text{succ}_H(\phi(v)) = \phi^*(\text{succ}_G(v)),$$

where ϕ^* is the homomorphic extension of ϕ to V_G^* , i.e., $\phi^*(\varepsilon) = \varepsilon$ and $\phi^*(v_1 \dots v_k) = \phi(v_1) \dots \phi(v_k)$.

Term Graph Rules and Rewrite Relation

As is the case with term rewriting, in the graph rewriting case we also express reducibility as a binary relation on the set of objects: in this case $\mathbb{G}(\Sigma)$.

Definition 6.6.11. A **graph rewrite rule** ρ is a triple (G, ℓ, r) such that G is a term graph and ℓ, r are vertices of G called, respectively, the left and right root of ρ .

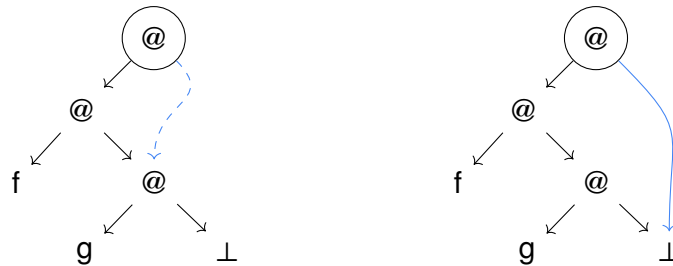
Definition 6.6.12. A **redex** in G is a pair (ρ, ϕ) consisting of a graph rule $\rho = (P, \ell, r)$ and a homomorphism $\phi : P|_{\ell} \rightarrow G$.

We define redirection as the operation in a term graph that replaces every reference to v_1 with references to v_2 .

Definition 6.6.13. Let $G = (V_G, \text{label}_G, \text{succ}_G, \Lambda_G)$ be a term graph and v_1, v_2 vertices in G . The term graph $G[v_1 \gg v_2]$ is the term graph $(V_G, \text{label}_G, \text{succ}_{G'}, \Lambda_G)$ with

$$\text{succ}_{G'}(v)_i = \begin{cases} v_2, & \text{if } \text{succ}_G(v)_i = v_1 \\ \text{succ}_G(v)_i, & \text{otherwise} \end{cases}$$

Example 6.6.14. The redirected edge is dashed in the example below:



Definition 6.6.15. Let G be a term graph and (ρ, ϕ) be a redex in G , with $\rho = (P, \ell, r)$. The **contraction** of (ρ, ϕ) in G is the term graph produced after the following steps: H (the build phase), I (the redirection phase), and J (the garbage collection phase).

1. The term graph H is built as follows:

(a) its vertex set is given by $V_H := V_G \uplus (V_{P|r} \setminus V_{P|\ell})$

(b) $\text{label}_H(v) := \begin{cases} \text{label}_G(v), & \text{if } v \in V_G \\ \text{label}_P(v), & \text{if } v \in V_{P|r} \setminus V_{P|\ell} \end{cases}$

(c) $\text{succ}_H(v)_i := \begin{cases} \text{succ}_G(v)_i, & \text{if } v \in V_G \\ \text{succ}_P(v)_i, & \text{if } v, \text{succ}_P(v)_i \in V_{P|r} \setminus V_{P|\ell} \\ \phi(\text{succ}_P(v)_i), & \text{if } v \in V_{P|r} \setminus V_{P|\ell} \text{ and } \text{succ}_P(v)_i \in V_{P|\ell} \end{cases}$

(d) its root is Λ_G .

2. In the redirection phase, we redirect in H all references to $\phi(\ell)$ by references to r . Notice that if $V_{P|r} \setminus V_{P|\ell}$ is empty, then $H = G$ in the previous step. The redirection phase is done by replacing every reference to $\phi(\ell)$ by a reference to $\phi(r)$. Therefore, we have:

$$I := \begin{cases} H[\phi(\ell) \gg r], & \text{if } V_{P|r} \setminus V_{P|\ell} \text{ is empty,} \\ H[\phi(\ell) \gg \phi(r)], & \text{otherwise} \end{cases}$$

3. Finally, we take the subgraph of I which is accessible from its root, i.e., $J := I|_{\Lambda_I}$.

We then write $G \rightsquigarrow J$ in one step, and $G \rightsquigarrow^n J$ for the n -step reduction.

We illustrate this with two examples. First, we aim to rewrite the graph of Figure 6.5a with a rule $\text{add } 0 \ y \rightarrow y$ at vertex v . Since the right-hand side is a variable, the building phase does nothing. The result of the redirection phase is given in Figure 6.5b, and the result of the garbage collection in Figure 6.5c.

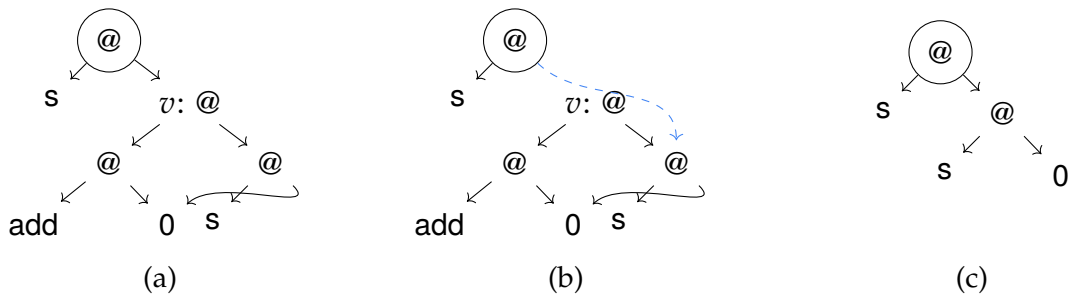


Fig. 6.5 Reducing a graph with the rule $\text{add } 0 \ y \rightarrow y$

Second, we consider a reduction by $\text{mult}(s \ x) \ y \rightarrow \text{add } y \ (\text{mult } x \ y)$. Figure 6.6a shows the result of the building phase, with the vertices and edges added during this phase in red. Redirection sets the root to the squared node (the root of the right-hand side), and the result after garbage collection is in Figure 6.6b.

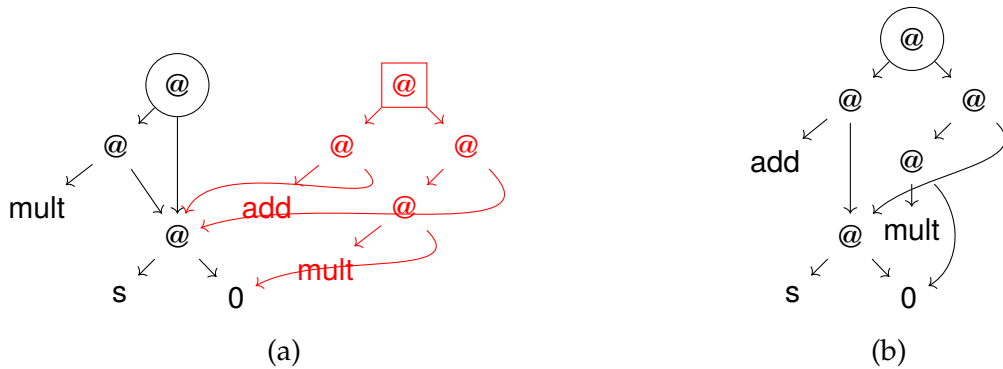


Fig. 6.6 Reducing a term graph with substantial sharing

From Term Rewriting to Graph Rewriting and Back Again

We need to simulate term rewriting steps using graph rewriting. We start with the observation that each term s has a natural representation as a term tree.

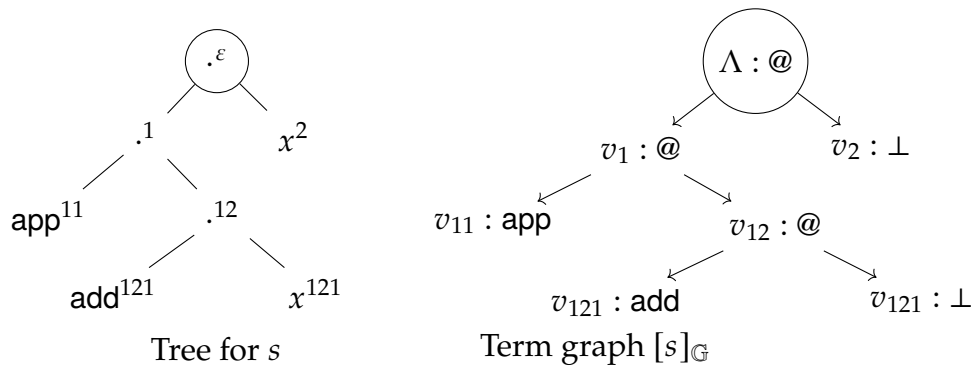
Definition 6.6.16. Given a term s , we define the term graph $[s]_{\mathbb{G}} = (V, \text{label}, \text{succ}, \Lambda)$ as follows:

1. we take as vertices the set of positions of s , that is $V = \text{pos}(s)$;
2. we construct the function label as follows:

$$\text{label}(v) = \begin{cases} @ & \text{if } s|_v \text{ is an application} \\ f & \text{if } s|_v = f \\ \perp & \text{if } s|_v \in \mathbb{X} \end{cases}$$

3. for the successor function, we recall that in applicative terms the only positions with direct successors are the applicative ones, hence if $s|_v = s_1 s_2$ then $\text{succ}(v) = (1v)(2v)$ and $\text{succ}(v) = \varepsilon$ otherwise.
4. the root of the term graph is the vertex Λ , which is equivalent to the root position in the term s .

Example 6.6.17. Consider the term $s = \text{app}(\text{add } x)x$. In the figure on the left, we informally write a term tree for s annotating each position. On the right, we picture the formal term graph representation of s .



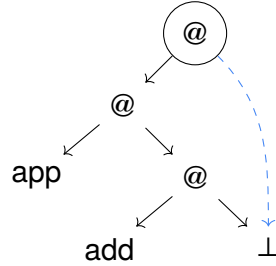
Essentially, $[s]_{\mathbb{G}}$ maintain the positioning structure of s and forgets variable names.

Our next goal is to translate rewriting rules $\ell \rightarrow r$ to a graph rewriting rule such that multiple occurrences of the same variable are shared.

Definition 6.6.18 (Maximum Sharing of Variables). Let s be a term. For each variable x with multiple occurrences in s we collect the set of positions $\text{o}(x) := \{p \in \text{pos}(s) \mid s|_p = x\}$. We choose one $p \in \text{o}(x)$. For each $p' \in \text{o}(x) \setminus \{p\}$, we redirect all references to p' to

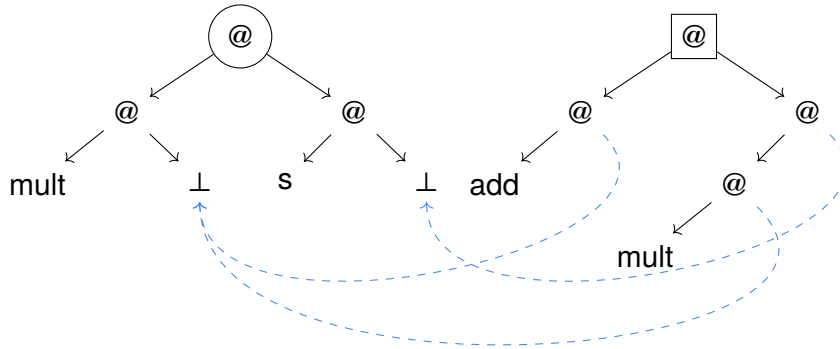
references to p in $[s]_{\mathbb{G}}$. Formally, $[s]_{\mathbb{G}}[p' \gg p]$. Finally, we remove from the resulting graph all vertices in $\text{o}(x) \setminus \{p\}$. The result of this process is a graph such that all occurrences of a variable in s are shared in one position.

Example 6.6.19. The graph below is the result of sharing the variable x in the term graph $[\text{app}(\text{add } x) x]_{\mathbb{G}}$ from Example 6.6.17.



Definition 6.6.20. Let \mathbb{R} be a TRS. For each rule $\ell \rightarrow r$, the graph $[\ell \rightarrow r]_{\mathbb{G}}$ is the graph rewriting rule (P, p_ℓ, p_r) defined as the disjoint union of $[\ell]_{\mathbb{G}}$ and $[r]_{\mathbb{G}}$ after sharing variables.

Example 6.6.21. Consider the rule $\text{mult } x (\text{s } y) \rightarrow \text{add } x (\text{mult } x y)$ from Example 2.1.19. We get the following graph rewriting rule where p_ℓ is the circled node and p_r is the squared node.



Consider a term s and the term graph $[s]_{\mathbb{G}}$. Since all variables in $[s]_{\mathbb{G}}$ are maximally shared, each variable vertex \perp in $[s]_{\mathbb{G}}$ represents a different variable from s . Consequently, we can reconstruct s from $[s]_{\mathbb{G}}$ as follows: for each vertex v_i labeled with \perp in $[s]_{\mathbb{G}}$ we choose a variable x_i ; next, we go through the structure of $[s]_{\mathbb{G}}$ and build a new term taking into account those edges pointing to v_i , which we use the variable name x_i . This gives rise to a function $[\cdot]_{\mathbb{G}}^{-1}$ from term graphs to terms such that $[s]_{\mathbb{G}}^{-1} = s'$ where $s' = s\gamma$ and γ is a variable renaming substitution. Notice that if G is a term graph isomorphic to $[s]_{\mathbb{G}}$ then $[G]_{\mathbb{G}}^{-1}$ is equal to s up to renaming of variables.

Simulating Call-by-Value Term Rewriting by Term Graph Rewriting

We need to simulate a trace of computation (so reductions) with an STRS as appropriate reductions in Term Graph rewriting. For this purpose, the following definition of rewriting is more useful than that of Definition 5.1.4, from a computation point of view.

Definition 6.6.22. Let \mathbb{R} be a TRS. We say s **reduces** to t if and only if there exist a position p in $\text{pos}(s)$, a value substitution γ , and a rule $\rho : \ell \rightarrow r \in \mathbb{R}$ such that

$$s|_p = \ell\gamma \quad \text{and} \quad t = s[r\gamma]_p.$$

Recall that such reductions will take place whenever the immediate subterms $\ell_i\gamma$ of $\ell\gamma = f(\ell_1\gamma) \dots (\ell_k\gamma)$ are values. With this information we can write $s \rightarrow^{p,\rho} t$ to say that s reduces to t at position p using the rule ρ . In this case, whenever s reduces in n steps to its normal form, we can trace the reduction sequence exactly:

$$s = s_0 \rightarrow^{p_0,\rho_0} s_1 \rightarrow^{p_1,\rho_1} \dots \rightarrow^{p_m,\rho_m} s_m = t$$

Hence, we can write a sequence consisting of pairs of positions and labels for rules $p_0\rho_0 \dots p_m\rho_m$, we call such sequence a *trace* for the reduction of s . This trace gives us exactly the positions where the reductions can be applied on the term side. To simulate this as graph rewriting, we apply graph reductions at the same positions. We note, however, that orthogonality is necessary here. Otherwise, overlapping applications of rules can destroy redexes [77]. This observation is collected in the following lemma.

Lemma 6.6.23 (Graph Rewriting Simulation). Let (\mathbb{F}, \mathbb{R}) be an orthogonal term rewriting system. Suppose $s \rightarrow^{p,\rho} t$ and that $[G]_{\mathbb{G}}^{-1} = s$. Then there exists a graph H such that $G \rightsquigarrow H$ and $[H]_{\mathbb{G}}^{-1} = t$.

The Efficiency of Graph Rewriting

In this section, we deal with the reasonability of graph rewriting w.r.t OTMs.

Lemma 6.6.24 (Subterm Lemma). Let (\mathbb{F}, \mathbb{R}) be a term rewriting system compatible with a polynomially-bounded bnat-size reflecting cost-size interpretation. Then there is a second-order polynomial interpretation B such that for every type-1 functional $f : \mathbb{N} \rightarrow \mathbb{N}$, data term $[n] : \text{bnat}$, and context C : if $\mathbb{F} S_f [n] \xrightarrow{+} C[S_f [m]]$ then $\| [m] \| \leq P(\|f\|, \| [n] \|)$.

Proof. Let us consider a polynomial interpretation for the distinguished function symbol $\mathbb{F} : (\text{bnat} \Rightarrow \text{bnat}) \Rightarrow \text{bnat} \Rightarrow \text{bnat}$ over the extended system $\mathbb{R}_{\mathbb{F},f,\mathbb{G}}$. By assumption, we have that the cost interpretation of \mathbb{F} is polynomially bounded. Hence,

$$\mathcal{J}_{\mathbb{F}}^c \leq \lambda F^c F^s x.P^c$$

where P^c is a polynomial expression (Definition 6.3.12) over $\text{Pol}_{\mathbb{N}}^2[F^c, F^s, x]$. Since P^c is a second-order polynomial expression, each occurrence of a sub-expression $F^c(e)$ in P^c is a second-order polynomial, so each argument e given to F^c is a second-order polynomial expression of base type as well. Let us enumerate these arguments as e_1, \dots, e_n . We can form the new polynomial B , defined as:

$$B := \sum_i e_i \quad \text{where occurrences of } F^c(e'_j) \text{ inside } e_i \text{ are replaced by } 1$$

Remark 6.6.25. As a concrete example of this construction take for instance $\mathcal{J}_F = \lambda F^c F^s x^s. x^s F^c(3 + F^s(9x^s)) + F^c(12) * F^c(3 + x^s * F^c(2)) + 5$. Then we would define $Q' = \lambda F^c F^s x. 3 + F^s(9x) + 12 + 3 + x * 1 + 2 = \lambda F^c F^s x. 20 + F^s(9x) + x$; we then have $B(G, y) = 20 + c * G^s(9 * c * y + 9 * d) + c * y + 2 * d$, where $x := c * y + d$. This comes from the fact that terms `bnat` typed terms are `bnat`-size reflecting. Note that if F^c does not occur in P^c , then $Q = 0$. Additionally, note that for a variable of base type, its cost component is zero (due to call-by-value). Hence, x^s is mapped to the variable x in B .

The proof idea is to show that if we can reduce $F S_f [n]$ to a context $C[F [m]]$ such that $[m]$ is bigger than $B(|f|, |n|)$, then we can construct an alternative term $S'_f [n]$ in place of our oracle S_f , so that $F(S'_f [n])[n]$ will need more than $\llbracket F(S'_f [n])[n] \rrbracket^c$ steps to reduce to normal form. This gives a contradiction with the compatibility theorem.

To implement the above idea, let us start by defining $N(n) = \llbracket F S_f [n] \rrbracket^c$. Thus, n is a function mapping natural numbers to natural numbers. We let $\underline{N}(n)$ denote the unary encoding `s (s ... (s 0) ...)` (with $N(n)$ successor symbols) of this number.

Consider a TRS $\mathbb{R}_{F, f', G}$ such that f' is the following oracle:

$$f'(x, y) := f(y)$$

Now consider a set of rules where, instead of the oracle S_f , we define:

$$\begin{aligned} S_{f'} [x] [y] &\rightarrow [f(y)] && \text{if } |y| \leq B(|f|, |x|) \\ S_{f'} [x] [y] &\rightarrow \text{helper } \underline{N}(x) [f(y)] && \text{otherwise} \end{aligned}$$

There are infinitely many rules both of the first kind and of the second kind. Additionally, we add to this new system the rules defining the symbol `helper`.

$$\begin{aligned} \text{helper } 0 y &\rightarrow y \\ \text{helper } (s x) y &\rightarrow \text{helper } x y \end{aligned}$$

Notice that by their definition, the rules for $S_{f'}$ will produce $[f(y)]$ in one step if $|y| \leq B(|f|, |x|)$, but they will take $N(x) + 1$ steps otherwise. Also, observe that S_f behaves exactly as $(S_{f'} [n]) : \text{bnat} \Rightarrow \text{bnat}$, where $[n]$ is the fixed but arbitrary input we

give as the argument to the initial term $s_0 = F S_f [n]$. That is, $S_{f'} [n] [m]$ and $S_f [m]$ have the same normal form. Hence, the normal forms $(F S_f [n])\downarrow$ and $(F (S_{f'} [n]) [n])\downarrow$ are equal for any initial $[n]$.

We interpret this new oracle symbol as follows:

$$\mathcal{J}_{S_{f'}}^c = \lambda x y. \begin{cases} 1 & \text{if } |y| \leq B(|f|, |x|) \\ N(x) + 1 & \text{if } |y| > B(|f|, |x|) \end{cases} \quad \mathcal{J}_{S_{f'}}^s = \lambda x y. \mathcal{J}_{S_f}^s(y)$$

Meanwhile, we interpret the helper symbol and constructors for unary numbers as follows:

$$\mathcal{J}_{\text{helper}}^c = \lambda x y. x + 1 \quad \mathcal{J}_{\text{helper}}^s = \lambda x y. y \quad \mathcal{J}_0^s = 0 \quad \mathcal{J}_s^s = \lambda x. x + 1$$

We easily see that this indeed orients the rules for $S_{f'}$ and helper. Moreover, $\llbracket F S_f [n] \rrbracket = \llbracket F (S_{f'} [n]) [n] \rrbracket$ since the size interpretations of S_f and $S_{f'} [n]$ are equal, and $\llbracket S_f e \rrbracket^c = \llbracket S_{f'} [n] e \rrbracket^c$ for any e that occurs as an argument to F^c in P^c .

Hence, if we replace the rule $G x \rightarrow F S_f x$ by the rule $G x \rightarrow F (S_{f'} x) x$, this new rule is still oriented by our interpretation, without having to alter \mathcal{J}_G .

To finally prove the Lemma we reason as follows. Suppose that $F S_f [n] \rightarrow^k C[S_f [m]]$ and k is the least number such that the context C is of this shape but $|m| > B(|f|, |n|)$.

Remark 6.6.26. It worth noting that if, for such minimum k , we can choose more than one context, i.e., k is the minimum number such that $F S_f [n] \rightarrow^k s$ and s can be written as $s = C_1[S_f [m]_1], s = C_2[S_f [m]_2], \dots$ we may pick any of the positions.

Now, we argue that in the extended system $\mathbb{R}_{F, f', G}$, that uses the alternative oracle f' and corresponding function $S_{f'}$, we have

$$F S_{f'} [n] \rightarrow^k C[(S_{f'} [n]) [m]]$$

So both $\mathbb{R}_{F, f, G}$ and $\mathbb{R}_{F, f', G}$ take the same number of steps to reduce $G [n]$.

All in all, we have the following reduction chain:

$$G [n] \rightarrow F S_f [n] \rightarrow^k C[S_f [m]] \rightarrow C[\llbracket f(m) \rrbracket] \rightarrow^l [q]$$

However, due to the use of the alternative oracle function S'_f and our assumption that $|m| > B(|f|, |n|)$, the reduction chain in the second extended system works as follows:

$$\begin{aligned}
G[n] &\rightarrow F(S'_f[n])[n] \\
&\rightarrow^k C[(S'_f[n])[m]] \\
&\rightarrow C[\text{helper } \underline{N(n)}[f(m)]] \\
&\rightarrow^{N(n)+1} C[[f(m)]] \\
&\rightarrow^{l'} [q]
\end{aligned}$$

In this reduction, the number $l' \geq l$. Indeed, this comes from the fact that k is minimal and there can be calls for oracles on intermediate terms after the k -th position. Hence, reducing $G[n]$ term to normal form in $\mathbb{R}_{F, S'_f, G}$ requires more than $\llbracket G[n] \rrbracket$ number of steps. This is impossible due to the compatibility theorem.

Hence, by contradiction, we conclude that while reducing $G[n]$, we do not ever encounter subterms $S_f[m]$ with $|m| > B(|f|, |n|)$. \square

With this lemma in hand, we can finally prove the soundness of our approach, which we state as the following theorem.

Theorem 6.6.27 (Soundness). Let (\mathbb{F}, \mathbb{R}) be an orthogonal STRS compatible with a polynomially-bounded bnat -size reflecting cost-size interpretation. If $F \in \Sigma$ computes the type-2 functional Ψ , then $\Psi \in \text{BFF}$.

Proof. Let f be a type-1 functional. Consider the oracle Turing machine M_f that, on input $n \in \mathbb{N}$, evaluates $G[n]$ according to the extended system $\mathbb{R}_{F, f, G}$ using graph rewriting⁴. So this machine implements the steps described in Definition 6.6.15. Rewriting steps that do not call the oracle are performed exactly as their term rewriting counterparts.

Now, observe that each call to the oracle, which we implement in rewriting by way of S_f , Definition 6.5.1, is resolved by the OTM M_f by querying the oracle and adjoining the answer into the graph. We first notice that the size of any intermediary graph is polynomially bounded by a second-order polynomial. Indeed, the total number of rewriting steps is bounded polynomially due to Theorem 6.6.2, and as a consequence of Lemma 6.6.24, each rewriting step can increase the size of the corresponding graph by at most a second-order polynomial factor. Therefore, along the evaluation, graph sizes are bounded polynomially. Secondly, there is a cost to M_f for performing each graph reduction step. The contraction algorithm we defined in Definition 6.6.15 is clearly

⁴Notice that we do not give an explicit definition of this OTM. The major limitation of low-level machine descriptions of algorithms is that no sensible person wants to write or read a non-trivial machine program. Hence, we appeal to the reader's intuition and observation that graph rewriting can be implemented as such machines. Indeed, it is certainly possible to give a concrete implementation of Definition 6.6.15 in a high-level programming language.

feasible and by the previous observation graph sizes are polynomially bounded. So the machine takes no more than a polynomial number of steps to perform each graph reduction step. Finally, the total cost of the machine M running on f and n as inputs is bounded by a second-order polynomial in terms of $|n|$ and $|f|$. Additionally, since M_f simulates $\mathbb{R}_{F,f,G}$ via graph rewriting and $\mathbb{R}_{F,f,G}$ computes Ψ , we have that M also computes Ψ . So it computes Ψ in time bounded by a second-order polynomial in terms of $|f|$ and $|n|$. By Theorem 6.4.1, Ψ is in BFF. \square

6.7 Completeness

In this section, we prove the other side of Theorem 6.5.9: if a given type-2 functional Ψ is in BFF, then there exists an orthogonal STRS that computes Ψ and admits a polynomially bounded interpretation. We prove this by providing an encoding of OTMs as STRSs that admit a polynomially bounded interpretation.

The encoding is divided into three steps. In Section 6.7.1, we will define the function symbols that will allow us to encode any possible machine configuration as terms. In Section 6.7.2, we will encode transitions as reduction rules that rewrite configuration terms. Lastly, we will design an STRS to simulate a complete execution of an OTM in polynomially many steps. Achieving this polynomial bound is non-trivial and is done in Sections 6.7.3–6.7.4.

Henceforth, we assume given a fixed OTM M , and a second-order polynomial P_M , such that M operates in time P_M . For simplicity, we assume the machine has only three tapes (one input/output tape, one query tape, one answer tape); that each non-oracle transition only operates on one tape (i.e., reading/writing and moving the tape head); and that we only have tape symbols $\{0, 1, B\}$.

6.7.1 Constructors

To start, we define the types and data constructors to represent configurations and \mathcal{N} .

Encoding the tape alphabet. For all $a \in \Gamma$, we introduce a fresh symbol $Sa : \text{symbol}$. We also introduce a constant $\text{nil} : \text{symblist}$, and $:: : \text{symbol} \Rightarrow \text{symblist} \Rightarrow \text{symblist}$. For example, for machines with $\Gamma = \{0, 1, B\}$, we have:

$S0 : \text{symbol}$	$\text{nil} : \text{symblist}$
$S1 : \text{symbol}$	$:: : \text{symbol} \Rightarrow \text{symblist} \Rightarrow \text{symblist}$
$SB : \text{symbol}$	

The list constructor $::$ is denoted in right-infix notation as usual; we use $[a_1; \dots; a_n]$ as syntactic sugar for $a_1 :: a_2 :: \dots :: a_n :: \text{nil}$. We can represent the contents of any tape using the signature above. Indeed, recall that a tape necessarily has a form $w_0 w_1 \dots w_k \text{BBB} \dots$, where the blank after the end of the content is repeated indefinitely. Consider for instance the tape with the following content $01101\text{BB} \dots$. This may be represented by the data term: $[\text{S0}; \text{S1}; \text{S1}; \text{S0}; \text{S1}]$ of type `symlist`. We may also add an arbitrary number of blanks in the representation; e.g., $[\text{S0}; \text{S1}; \text{S1}; \text{S0}; \text{S1}; \text{SB}; \text{SB}]$, and retain a representation of the same tape.

Lemma 6.7.1. Let \mathcal{N} be the set of ground terms of type `symlist` built only from `S0`, `S1`, `nil`, `::`, which do not have a form $\text{S0} :: t$. There exists a bijection from \mathbb{N} to \mathcal{N} .

Proof. Let $n \in \mathbb{N}$. If $n = 0$ let $[n] := \text{nil}$. Otherwise, let $w_1 \dots w_k$ be the binary representation of n ; then $k > 0$ and $w_1 = 1$. We define $[n] := \phi(w_1 \dots w_k)$, where $\phi(w)$ is defined by induction on $|w|$, as follows:

$$\phi(w) = \begin{cases} \text{S0} :: \text{nil}, & \text{if } |w| = 1 \text{ and } w = 0 \\ \text{S1} :: \text{nil}, & \text{if } |w| = 1 \text{ and } w = 1 \\ \text{S0} :: \phi(w'), & \text{if } w = 0w' \\ \text{S1} :: \phi(w'), & \text{if } w = 1w' \end{cases}$$

Now it is easy to see that this function is indeed bijective. □

We choose the following interpretations for these constructors:

$$\llbracket \text{nil} \rrbracket^s = 0 \quad \llbracket x :: y \rrbracket^s = 1 + \llbracket y \rrbracket^s \quad \llbracket \text{Sa} \rrbracket^s = 0 \text{ for all symbols } a$$

That is, the size of a symbol list is the length of the list.

Encoding tape configurations. Recall that, by our definitions, we view a tape as a function $h \in \Gamma^{\mathbb{N}}$ along with the position $p \in \mathbb{N}$ of its head. Now that we can express the contents of a tape, the next step is to also capture this position of the head. In practice, we may think of a tape with the tape head at position p as a finite word $w_1 \dots w_{p-1} \# w_p w_{p+1} \dots w_k$ (followed by an infinite number of blanks). This means the tape's head is currently reading the symbol w_p of the tape. So we can split this tape into three components: left is the word $w_1 \dots w_{p-1}$, the symbol currently being read is w_p , and the right word is $w_{p+1} \dots w_k$.

We introduce the following new constructors.

$$\text{L} : \text{symlist} \Rightarrow \text{left} \quad \text{R} : \text{symlist} \Rightarrow \text{right} \quad \text{split} : \text{left} \Rightarrow \text{right} \Rightarrow \text{tape}$$

Here, L, R hold the content of the left and right split of the tape, respectively. For convenience in rewriting transitions, later on, we will encode the left side of the split in reverse order. Specifically, we encode $w_1 \dots w_{p-1} \# w_p w_{p+1} \dots w_k$ as

$$\text{split}(\text{L}[w_{p-1}; \dots; w_2; w_1]) (\text{R}[w_p w_{p+1}; \dots; w_{k-1}; w_k])$$

Notice that here we are using the type system to correctly encode the configuration word $w_1 \dots w_{i-1} \# w_i w_{i+1} \dots w_k$. In the term encoding of this split, we say that the symbol currently being read is the first element of the list that is the argument of R. In the case of R nil, the symbol currently being read is B.

Let us consider a concrete example: the state of the input tape for an initial configuration of a machine M on the word 10110 is represented as #10110. So in the initial configuration, the head is reading the first symbol on the input tape. As such, we construct the following term:

$$\text{split}(\text{L nil}) (\text{R}[S1; S0; S1; S1; S0])$$

There is no split yet, so the first argument of split is an empty word, L nil. The whole content of the tape is at the right of the tape's head, which is encapsulated by the constructor R. Notice that its immediate subterm is [S1; S0; S1; S1; S0], whose first element is S1. This denotes the symbol currently being read.

Now, suppose a few transitions later the machine has changed the contents of the tape for instance to 1B0#10. The representation of this is given by:

$$\text{split}(\text{L}[S0; SB; S1]) (\text{R}[S1; S0])$$

We shall see later how to emulate these transitions using rewriting rules. The lemma below states that this encoding is correct.

Lemma 6.7.2. There exists an injection from the set of valid tape configurations, that is, the set $\{h : \mathbb{N} \rightarrow \Gamma \mid \exists N \forall i > N, h(i) = B\} \times \mathbb{N}$, to the set of ground terms of type tape.

Proof. Let $h \in \Gamma^{\mathbb{N}}$, $p \in \mathbb{N}$ and $N \in \mathbb{N}$ such that $h(i) = B$ for all $i > N$. Let $w_l = h(p-1)h(p-2) \dots h(0)$ and $w_r = h(p)h(p+1) \dots h(N)$. Consider the translation function ϕ constructed in the proof of Lemma 6.7.1. We map (h, p) to the term $\text{split}(\text{L} \phi(w_l)) (\text{R} \phi(w_r))$. Clearly, if $h \neq h'$ or $p \neq p'$, then (h', p') is mapped to a different term, so this function is injective. \square

Note that this is not a bijection, since we may choose different numbers N for the same tape. This technically yields a different term. However, in practice, these terms will be treated the same.

We conclude that terms of type `tape` faithfully encode tape configurations. Notice that the only way to build a ground inhabitant of this type is by using the constructor `split`, which will enforce a correct encoding for tape configurations. We choose the following interpretations for these constructors:

$$\llbracket L x \rrbracket^s = \llbracket x \rrbracket^s \quad \llbracket R x \rrbracket^s = \llbracket x \rrbracket^s \quad \llbracket \text{split } x y \rrbracket^s = \llbracket x \rrbracket^s + \llbracket y \rrbracket^s$$

That is, the size of a tape is the total number of known symbols on the tape (both to the left and to the right of the tape head). Those are all constructor symbols, so their cost interpretation is the constant function returning zero on all components.

Encoding machine configurations. Recall that — since our notion of OTM has three tapes — a configuration of a machine at any moment is a tuple of the form $(q, h_1, h_2, h_3, p_1, p_2, p_3)$. Hence, given an oracle machine M with components given by the tuple $(Q, \Gamma, \text{start}, \text{end}, \text{query}, \text{answer}, f, \mathcal{T})$. For each state q in Q , we introduce one constructor given by the signature:

$$q : \text{tape} \Rightarrow \text{tape} \Rightarrow \text{tape} \Rightarrow \text{config}$$

The constructor symbol indicates the state; the three arguments are the term representations of the three tape configurations.

Example 6.7.3. Let us write the term encoding of the initial configuration of a machine M_f on input n . Recall that this is a tuple $(\text{start}, \langle n \rangle, 0, i \mapsto B, 0, i \mapsto B, 0)$. It is encoded as follows: `start (split (L nil) (R [n])) (split (L nil) (R nil)) (split (L nil) (R nil))`.

Lemma 6.7.4. There exists an injection from the set of configuration tuples for M to the set of ground terms of type `config`.

For all states q we choose the following interpretation:

$$\mathcal{J}_q^s = \lambda x y. x + y$$

That is, the size of a configuration is the size of its first and second tapes combined. We do *not* include the third tape, because it does not directly affect either the result yielded by a final configuration nor the size of a number that the function f is applied on.

Unary numbers. The role of the type `bnat` in Definition 6.5.1 — so the type of elements of \mathcal{N} — is filled by the type `symblist`. In addition, it proves useful to define a set of unary numbers, to represent *lengths* and the polynomial $P_M(|f|, |w|)$. For this, we let $0 : \text{unat}$ and $s : \text{unat} \Rightarrow \text{unat}$. We can now represent a number n in unary notation as $s^n 0$. Notice

that we use the new base type `unat` here to differentiate it from the `nat` type we used in previous chapters. We chose the following size interpretations:

$$\mathcal{J}_0^s = (0, 0) \qquad \mathcal{J}_s^s = \lambda x.(x_1 + 1, x_2 + 1)$$

That is, we map `unat` to $(\mathbb{N}, \geq) \times (\mathbb{N}, \leq)$; the size of a unary number `sn 0` is just (n, n) . We refer to the first size component of $\llbracket n \rrbracket$ as $\llbracket n \rrbracket^s$ and to the second size component as $\llbracket n \rrbracket^n$.

6.7.2 Executing the machine

Having encoded our configurations, we will define rules to encode the transitions. To start, we introduce a function symbol `step` : $(\text{symlist} \Rightarrow \text{symlist}) \Rightarrow \text{config} \Rightarrow \text{config}$ designed to execute a single step of the machine. The first argument, of type $\text{symlist} \Rightarrow \text{symlist}$, is to capture the usage of the oracle function, so it can be used when the machine calls the oracle. The second argument represents the current configuration. Hence, we model the computation of OTMs as rules that simulate the small step semantics given for the machine.

Encoding transitions as rules. Recall from Definition 6.3.4 that a (non-oracle) transition for a configuration (q, \vec{h}, \vec{p}) is only allowed to operate in one of the tape configurations (h_i, p_i) . We work on the first tape configuration in the construction below and emphasize that the same process applies to all tapes. So assume given a transition $(q_i, a, 1, c, d, q_o) \in \mathcal{T}$. We have two possibilities for d .

Moving the tape's head to the left. If $d = L$, there are three cases to consider.

1. In the first case, there is no symbol to the left of the tape's head. This only happens when the head is at the leftmost position of the tape. If there is a transition to the left then the machine halts, so no rewriting rule is added for this case. Note that this should not arise in practice, since we assumed that the machine maps all valid input to a valid final configuration.
2. In the second case, there are symbols to the left and right of the tape head. So the tape configuration steps from $w_1 \dots w_{i-1} w_i \# a w_{i+2} \dots w_k$ to the new configuration $w_1 \dots w_{i-1} \# w_i c w_{i+2} \dots w_k$. Note that w_i is moved to the right side of the tape configuration in this case. Hence, this transition is modeled by the rule:

$$\text{step } F(q_i (\text{split } (L(x :: y)) (R(Sa :: z)))) u v \rightarrow q_o (\text{split } (L y) (R(x :: Sc :: z))) u v$$

3. In the third case, there are symbols to the left but not to the right of the tape head. Hence, if $a = B$, we must also add the following rule:

$$\text{step } F (q_i (\text{split } (L (x :: y)) (R \text{ nil})) u v) \rightarrow q_o (\text{split } (L y) (R (x :: Sc :: \text{nil}))) u v$$

This is because our tapes are infinite, even though we encode only a finite number of symbols. This rule handles the case where the tape head is at the end of the represented symbols: $w_1 \dots w_k \#$. Hence, the rule reads the (implicit) blank at the reader's head and reduces this state to $w_1 \dots w_{k-1} \# w_k c$.

Moving the tape's head to the right. If $d = R$, the tape configuration changes from $w_1 \dots w_i \# a w_{i+2} \dots w_k$ to $w_1 \dots w_i c \# w_{i+2} \dots w_k$. This gives rise to the following rule:

$$\text{step } F (q_i (\text{split } (L y) (R (Sa :: z))) u v) \rightarrow q_o (\text{split } (L (Sc :: y)) (R z)) u v$$

As before, if $a = B$ we also handle the case when the tape's head is at the end of the represented input symbols:

$$\text{step } F (q_i (\text{split } (L y) (R \text{ nil})) u v) \rightarrow q_o (\text{split } (L (Sc :: y)) (R \text{ nil})) u v$$

Rules that encode oracle calls. We also need to encode the machine's call to the oracle. Recall that in order to query the machine for the value of f at u , we first write u on the second tape (the query tape), move its head to the leftmost position, and enter the query state `query`. Then, in one step the content of the second tape is erased and the image of f over u is written in the third tape (the answer tape). Visually, this step can be represented as follows:

$$(\text{query}, w, \#[n] a \dots, v) \rightsquigarrow (\text{answer}, w, \#B, \#[f(n)])$$

where $a \in \Gamma \setminus \{0, 1\}$. This suggests the following rule:

$$\text{step } F (\text{query } w (\text{split } x (R y)) v) \rightarrow \text{answer } w (\text{split } e (R \text{ nil})) (\text{split } e (R (F (\text{clean } y))))),$$

where $e = (L \text{ nil})$. Here, `clean`, `clean2` : `symlist` \Rightarrow `symlist` are new function symbols defined by the rules:

$$\begin{array}{ll} \text{clean } (S0 :: x) \rightarrow \text{clean } x & \text{clean } (Sa :: x) \rightarrow \text{nil} \text{ for } a \in \Gamma \setminus \{0, 1\} \\ \text{clean } (S1 :: x) \rightarrow S1 :: (\text{clean2 } x) & \text{clean nil} \rightarrow \text{nil} \\ \text{clean2 } (S0 :: x) \rightarrow S0 :: (\text{clean2 } x) & \text{clean2 } (Sa :: x) \rightarrow \text{nil} \text{ for } a \in \Gamma \setminus \{0, 1\} \\ \text{clean2 } (S1 :: x) \rightarrow S1 :: (\text{clean2 } x) & \text{clean2 nil} \rightarrow \text{nil} \end{array}$$

This function cleans up the content of the query tape (which may for instance have more than one word on it) to return an element of \mathcal{N} . If $\text{split}(L s)(R t)$ encodes a tape configuration (h, p) , then $\text{clean } t$ exactly reduces to $h|p$.

We observe that the various step rules as well as the clean-rules are non-overlapping, due to our requirement that the OTM is deterministic (and that there are no transitions defined from the query state). They are also left-linear. We choose the following interpretations:

$$\mathcal{J}_{\text{clean}} = \left\langle (0, \lambda x.(x, u)) , \lambda x.x + 1 \right\rangle \quad \mathcal{J}_{\text{clean2}} = \left\langle (0, \lambda x.(x + 1, u)) , \lambda x.x \right\rangle$$

One can check by (somewhat boring) calculations and easily see that this interpretation orients all eight rules.

Properties of the step relation. For step, we choose the following interpretation:

$$\mathcal{J}_{\text{step}} = \left\langle (0, \lambda Fx.(2 + x + F^c(x), u)) , \lambda Fx.x + 1 \right\rangle$$

We notice that non-query transition steps can be oriented directly by this interpretation. Let us work out the oracle steps below. For the numeric component of the cost interpretation, we get:

$$\begin{aligned} & \pi_1(\llbracket \text{step } F(\text{query } w(\text{split } x(R y)) v) \rrbracket) \\ &= 2 + \llbracket \text{query } w(\text{split } x(R y)) v \rrbracket^s + F^c(\llbracket \text{query } w(\text{split } x(R y)) v \rrbracket^s) \\ &= 2 + (\llbracket w \rrbracket^s + \llbracket x \rrbracket^s + \llbracket y \rrbracket^s) + F^c(\llbracket w \rrbracket^s + \llbracket x \rrbracket^s + \llbracket y \rrbracket^s) \\ &> 1 + \llbracket y \rrbracket^s + F^c(\llbracket w \rrbracket^s + \llbracket x \rrbracket^s + \llbracket y \rrbracket^s) \\ &\geq (\llbracket y \rrbracket^s + 1) + F^c(\llbracket y \rrbracket^s) \\ &= \pi_1(\llbracket \text{clean } y \rrbracket) + F^c(\llbracket \text{clean } y \rrbracket^s) \\ &= \pi_1(\llbracket \text{answer } w(\text{split}(L \text{nil})(R \text{nil}))(\text{split}(L \text{nil})(R(F(\text{clean } y)))) \rrbracket) \end{aligned}$$

For the size interpretation, we have:

$$\begin{aligned} & \llbracket \text{step } F(\text{query } w(\text{split } x(R y)) v) \rrbracket^s \\ &= \llbracket \text{query } w(\text{split } x(R y)) v \rrbracket^s + 1 \\ &= \llbracket w \rrbracket^s + \llbracket x \rrbracket^s + \llbracket y \rrbracket^s + 1 \\ &\geq \llbracket w \rrbracket^s \\ &= \llbracket w \rrbracket^s + \llbracket (\text{split}(L \text{nil})(R \text{nil})) \rrbracket^s \\ &= \llbracket \text{answer } w(\text{split}(L \text{nil})(R \text{nil}))(\text{split}(L \text{nil})(R(F(\text{clean } y)))) \rrbracket^s \end{aligned}$$

since the third tape is not included in the size of the configuration. From the definition we proposed to step rules, we collect the following lemma.

Lemma 6.7.5. Let M_f be an OTM and C, C' be machine configurations of M_f such that $C \rightsquigarrow C'$. Then $\text{step } S_f [C] \xrightarrow{+} [C']$, where $[C]$ is the encoding of the configuration C .

We also introduce a fresh symbol $\text{extract} : \text{tape} \Rightarrow \text{symlist}$, defined by the rule

$$\text{extract} (\text{split} (L x) (R y)) \rightarrow \text{clean } y$$

Then if t encodes a tape configuration (h, p) , the term $\text{extract } t$ reduces to $[h|p]$. This helper function will allow us to avoid pattern matching on configurations going forward. This rule is oriented by using the interpretation:

$$\mathcal{J}_{\text{extract}} = \left\langle (0, \lambda x.(x + 2, u)) , \lambda x.x \right\rangle$$

6.7.3 A bound on the number of steps

To generalize from performing a single step of the machine to tracing a full computation on the machine level, the natural idea would be to introduce a defined symbol $\text{execute} : (\text{symlist} \Rightarrow \text{symlist}) \Rightarrow \text{config} \Rightarrow \text{symlist}$ and rules

$$\begin{aligned} \text{execute } F (q x y z) &\rightarrow \text{execute } F (\text{step}(q x y z)) \quad \text{for } q \neq \text{end} \\ \text{execute } F (\text{end} (\text{split} (L x) (R w)) y z) &\rightarrow \text{clean } w \end{aligned}$$

Having this, and letting i_n denote the term encoding of the initial state $\text{initial}(n)$, we certainly have $\text{execute } S_f i_n \xrightarrow{+} [m]$, for m the result of executing the OTM M_f on n . However, we cannot find a general interpretation for execute which orients the rule above. Indeed, it is very possible that the system with this rule is non-terminating, for instance if there is an (unreachable) state q with a transition $(q, B, 1, B.R, q)$.

Instead, a common idea is to give execute an additional argument of type unat , and replace the first rule above by

$$\text{execute } F (s m) (q x y z) \rightarrow \text{execute } F m (\text{step}(q x y z)) \quad \text{for } q \neq \text{end}$$

The ancillary argument of type unat is used to compute the total number of steps we are allowed to take in the computation, and is what ensures termination. If we can start the computation with $\text{execute } S_f (s^N 0) i_n$ for some $N \geq P_M(|f|, |n|)$, we will certainly reduce to the desired return value.

Unfortunately, we cannot: to compute $P_M(|f|, |n|)$ from f and n in general, we need to have access to $|f|$ — but it is not in general possible to compute $|f|$ from f in polynomial time, that is the length of a type-1 parameter itself is not in BFF. This was

proved by Kapron and Cook [60]. Indeed, since $|f|(x) = \max_{y \leq x} f(y)$ computing the value $|f|(x)$ depends on exponentially many potential choices for y . It turns out that we do not actually need all such values to compute $|f|$. Rather, it suffices to know a bound for the size of $f(x)$ for only those x on which the oracle is actually questioned. That is: for a set $A \subseteq \mathbb{N}$, we define $|f|_A$ as the function mapping n to $\max\{|f|(x) \mid x \in A \wedge |x| \leq n\}$.

Lemma 6.7.6. Let M be an oracle Turing machine operating in time bounded by the second-order polynomial P_M . Let $x \in \mathbb{N}$ and $f \in \mathbb{N} \rightarrow \mathbb{N}$. Suppose $\text{initial}(x) \rightsquigarrow^* C$, and during this execution, the oracle is only called on numbers in the set $A \subseteq \mathbb{N}$. Then the cost of the execution $\text{initial}(x) \rightsquigarrow^* C$ is bounded by $|f|_A$.

Proof. Let f' be the function

$$n \mapsto \begin{cases} f(n) & \text{if } n \in A \\ 0 & \text{otherwise} \end{cases}$$

Since the machine acts exactly the same on every oracle query for f and f' , clearly also $\text{initial}(x) \rightsquigarrow^* C$ in $M_{f'}$, at the same cost. This cost is bounded by $P_M(|f'|, |x|)$. But $|f'|$ is exactly $|f|_A$. \square

This observation was also made by Kapron and Cook in [60, Proposition 2.3].

Representing $|f|_A$ in term rewriting. We will now stepwise define a term rewriting system to compute $P(|f|_A, |x|)$ including its interpretation.

Let $\text{length} : \text{symlist} \Rightarrow \text{unat}$ be defined by the rules:

$$\text{length nil} \rightarrow 0 \quad \text{length}(x :: y) \rightarrow s(\text{length } y)$$

Then $\text{length } a \xrightarrow{+} s^{|a|} 0$: this function computes the length of a list. We choose:

$$\mathcal{J}_{\text{length}} = \left\langle (0, \lambda x.(x_1 + 1, u)) , \lambda x.(x_1, 0) \right\rangle$$

It is easy to see that this interpretation orients both rules.

Next, let $\text{limit} : \text{symlist} \Rightarrow \text{unat} \Rightarrow \text{symlist}$ be defined by the rules:

$$\text{limit}(x :: y)(s n) \rightarrow x :: (\text{limit } y n) \quad \text{limit nil } n \rightarrow \text{nil} \quad \text{limit}(x :: y) 0 \rightarrow \text{nil}$$

Then $\text{limit } a (s^n 0)$ reduces to a if $|a| \leq n$ and to its first n symbols otherwise. We choose:

$$\mathcal{J}_{\text{limit}} = \left\langle (0, \lambda x.(0, \lambda n.(n_1 + 1, u))) , \lambda x n.n_1 \right\rangle$$

Here, it is straightforward to show that this interpretation orients both rules. Next, let $\text{checkbound} : \text{symlist} \Rightarrow \text{unat} \Rightarrow \text{symlist} \Rightarrow \text{symlist}$ be defined by the rules:

$$\begin{aligned} \text{checkbound } (x :: y) (\mathbf{s} \ n) \ z &\rightarrow \text{checkbound } y \ n \ z & \text{checkbound nil } n \ z &\rightarrow z \\ \text{checkbound } (x :: y) \ 0 \ z &\rightarrow \text{nil} \end{aligned}$$

Then $\text{checkbound } a (\mathbf{s}^n \ 0) \ b$ reduces to b if $|a| \leq n$ and to $[]$ otherwise. We choose:

$$\llbracket \text{checkbound } x \ n \ z \rrbracket^{\mathbf{s}} = \llbracket z \rrbracket^{\mathbf{s}} \quad \llbracket \text{checkbound } x \ n \ z \rrbracket^{\mathbf{c}} = \llbracket n \rrbracket^{\mathbf{s}} + 1$$

which is once more easily seen to orient these rules.

Now we can put those three symbols (and their meanings) together. Let $\text{tryapply} : (\text{symlist} \Rightarrow \text{symlist}) \Rightarrow \text{symlist} \Rightarrow \text{unat} \Rightarrow \text{unat}$ be defined by the rule:

$$\text{tryapply } F \ a \ n \rightarrow \text{length}(\text{checkbound } a \ n \ (F(\text{limit } a \ n)))$$

We now see that $\text{tryapply } F \ a \ (\mathbf{s}^n \ 0)$ reduces to $|F(a)|$ if $|a| \leq n$, and to 0 otherwise. That is, it exactly returns $|F|_{\{a\}}(n)$. We choose the following interpretation:

$$\begin{aligned} \llbracket \text{tryapply } F \ a \ n \rrbracket^{\mathbf{s}} &= \llbracket F \rrbracket^{\mathbf{s}}(\llbracket n \rrbracket^{\mathbf{s}}) & \llbracket \text{tryapply } F \ a \ n \rrbracket^{\mathbf{n}} &= 0 \\ \llbracket \text{tryapply } F \ a \ n \rrbracket^{\mathbf{c}} &= \llbracket F \rrbracket^{\mathbf{c}}(\llbracket n \rrbracket^{\mathbf{s}}) + \llbracket F \rrbracket^{\mathbf{s}}(\llbracket n \rrbracket^{\mathbf{s}}) + 2 * \llbracket n \rrbracket^{\mathbf{s}} + 4 \end{aligned}$$

By writing out $\llbracket \text{length}(\text{checkbound } a \ n \ (F(\text{limit } a \ n))) \rrbracket$ we can see that this orients the rule.

Next, let $\text{max} : \text{unat} \Rightarrow \text{unat} \Rightarrow \text{unat}$ be defined by the rules:

$$\text{max } (\mathbf{s} \ n) (\mathbf{s} \ m) \rightarrow \mathbf{s}(\text{max } n \ m) \quad \text{max } 0 \ m \rightarrow m \quad \text{max } (\mathbf{s} \ n) \ 0 \rightarrow \mathbf{s} \ n$$

Then clearly $\text{max } (\mathbf{s}^n \ 0) (\mathbf{s}^m \ 0)$ reduces to $\mathbf{s}^{\max(n,m)} \ 0$. These rules are oriented by the following interpretation:

$$\mathcal{J}_{\text{max}} = \left\langle (0, \lambda n.(0, \lambda y.(n_1 + 1, u))) , \lambda nm.(\text{max}(n_1, m_1), 0) \right\rangle$$

Note that this interpretation is allowed because we are polynomially *bounded*; there is no requirement that all interpretation functions are polynomial. Clearly $(n, m) \mapsto \text{max}(n, m)$ is bounded by $(n, m) \mapsto n + m$.

Now, at last, we can define the rules to compute $|f|_A$. For this, let $\emptyset : \text{set}$ and $\text{setcons} : \text{symlist} \Rightarrow \text{set} \Rightarrow \text{set}$ be new constructors — used to express a set of symbol lists — and $\text{tryall} : (\text{symlist} \Rightarrow \text{symlist}) \Rightarrow \text{set} \Rightarrow \text{unat} \Rightarrow \text{unat}$, be a new function symbol,

defined by the rules:

$$\begin{aligned} \text{tryall } F \emptyset n &\rightarrow 0 \\ \text{tryall } F (\text{setcons } a \text{ tl}) n &\rightarrow \max(\text{tryapply } F a n) (\text{tryall } F \text{ tl } n) \end{aligned}$$

Taking into account the meanings of `tryapply` and `max`, we see that if A is the encoding of a set $\{a_1, \dots, a_k\}$, then `tryapply` $S_f A$ ($s^n 0$) computes $\max_{1 \leq i \leq k} \{ |f|_{\{a_i\}} \}$, which is exactly $|f|_{\{a_1, \dots, a_k\}}$. We choose the following interpretation:

$$\begin{aligned} \llbracket \emptyset \rrbracket^s &= 0 & \llbracket \text{setcons } x \ y \rrbracket^s &= \llbracket y \rrbracket^s + 1 \\ \llbracket \text{tryall } F a \ n \rrbracket^s &= \llbracket F \rrbracket^s (\llbracket n \rrbracket^s) & \llbracket \text{tryall } F a \ n \rrbracket^n &= 0 \\ \llbracket \text{tryall } F a \ n \rrbracket^c &= 1 + \llbracket a \rrbracket^s * (\llbracket F \rrbracket^c (\llbracket n \rrbracket^s) + 2 * \llbracket F \rrbracket^s (\llbracket n \rrbracket^s) + 2 * \llbracket n \rrbracket^s + 6) \end{aligned}$$

This indeed orients both `tryall` rules (and is monotonic).

Computing $P_M(|A|, |x|)$. Now, for a given second-order polynomial P , we would like to find its value as a term of type `unat`. To do this, we use additional function symbols `add`, `mult` : `unat` \Rightarrow `unat` \Rightarrow `unat`, defined by the rules:

$$\begin{aligned} \text{add } x \ 0 &\rightarrow x & \text{mult } x \ 0 &\rightarrow 0 \\ \text{add } x \ (s \ y) &\rightarrow s (\text{add } x \ y) & \text{mult } x \ (s \ y) &\rightarrow \text{add} (\text{mult } x \ y) \ x \end{aligned}$$

These clearly implement addition and multiplication, and are oriented by the following interpretations:

$$\begin{aligned} \llbracket \text{add } x \ y \rrbracket^s &= \llbracket x \rrbracket^s + \llbracket y \rrbracket^s & \llbracket \text{mult } x \ y \rrbracket^s &= \llbracket x \rrbracket^s * \llbracket y \rrbracket^s \\ \llbracket \text{add } x \ y \rrbracket^n &= 0 & \llbracket \text{mult } x \ y \rrbracket^n &= 0 \\ \llbracket \text{add } x \ y \rrbracket^c &= \llbracket y \rrbracket^s + 1 & \llbracket \text{mult } x \ y \rrbracket^c &= \llbracket x \rrbracket^s * \llbracket y \rrbracket^s + 2 * \llbracket y \rrbracket^s + 1 \end{aligned}$$

Then, for any higher-order polynomial $P(f, x)$ and terms $F : \text{unat} \Rightarrow \text{symbolist}$, $a : \text{set}$, and $z : \text{unat}$ we define $\Theta_p^{F;a;z}$ inductively:

$$\begin{aligned} \Theta_n^{F;a;z} &= s^n 0 & \Theta_{P_1+P_2}^{F;a;z} &= \text{add } \Theta_{P_1}^{F;a;z} \ \Theta_{P_2}^{F;a;z} \\ \Theta_x^{F;a;z} &= z & \Theta_{P_1*P_2}^{F;a;z} &= \text{mult } \Theta_{P_1}^{F;a;z} \ \Theta_{P_2}^{F;a;z} \\ \Theta_{f(P_1)}^{F;a;z} &= \text{tryall } F a \ \Theta_{P_1}^{F;a;z} \end{aligned}$$

It is clear that if $z = s^{|x|} 0$ and a represents the set of symbol lists A , then $\Theta_p^{S_f;a;z}$ reduces to $s^{P(|f|_A, |x|)} 0$. Moreover, by induction on the size of P we quickly see that $\llbracket \Theta_p^{F;a;z} \rrbracket^s = P(\llbracket F \rrbracket^s, \llbracket z \rrbracket^s)$ for any P — so we retain the originally intended size bound. We also see that we can find higher-order polynomials $A_P(f^c, f^s, z^s)$, $B_P(f^c, f^s, z^c, z^s)$

such that $\llbracket \Theta_P^{F;a;z} \rrbracket^c \leq \llbracket a \rrbracket^s * A_P(\llbracket F \rrbracket^c, \llbracket F \rrbracket^s, \llbracket z \rrbracket^c, \llbracket z \rrbracket^s) + B_P(\llbracket F \rrbracket^c, \llbracket F \rrbracket^s, \llbracket z \rrbracket^c, \llbracket z \rrbracket^s)$. That is, the use of $\llbracket a \rrbracket$ is limited.

Rules for unary subtraction. For our eventual execution rules (in Section 6.7.4), it will prove very useful to also be able to subtract unary numbers. For this, we use the symbol $\text{minus} : \text{unat} \Rightarrow \text{unat} \Rightarrow \text{unat}$ and rules:

$$\text{minus}(\text{s } x)(\text{s } y) \rightarrow \text{minus } x \ y \quad \text{minus } x \ 0 \rightarrow x \quad \text{minus } 0(\text{s } y) \rightarrow 0$$

We see that $\text{minus}(\text{s}^n 0)(\text{s}^m 0)$ reduces to $\text{s}^k 0$ where $k = \max(n - m, 0)$. We orient the rules with the following interpretation:

$$\begin{aligned} \llbracket \text{minus } x \ y \rrbracket^s &= \max(\llbracket x \rrbracket^s - \llbracket y \rrbracket^n, 0) & \llbracket \text{minus } x \ y \rrbracket^n &= 0 \\ \llbracket \text{minus } x \ y \rrbracket^c &= \llbracket x \rrbracket^s + 1 \end{aligned}$$

Note that this interpretation is monotonic despite the use of the minus function because the second size argument of unat is in (\mathbb{N}, \leq) . If $\llbracket y \rrbracket^n \leq \llbracket y' \rrbracket^n$ then indeed $\max(\llbracket x \rrbracket^s - \llbracket y \rrbracket^n, 0) \geq \max(\llbracket x \rrbracket^s - \llbracket y' \rrbracket^n, 0)$.

6.7.4 Finalizing execution

Now, at last, we have all the preparations to define execution in a way that can be bounded by a polynomial interpretation. We let $\text{execute} : (\text{symlist} \Rightarrow \text{symlist}) \Rightarrow \text{unat} \Rightarrow \text{unat} \Rightarrow \text{unat} \Rightarrow \text{set} \Rightarrow \text{config} \Rightarrow \text{symlist}$, and will define rules to reduce expressions $\text{execute } F \ n \ m \ z \ a \ c$ where

1. F is the function to be used in oracle calls
2. $n - 1$ is a bound on the number of steps that can be done before the next oracle call (or the end of the machine's execution)
3. m counts the number of steps that has been done so far
4. z is a unary representation of $|x|$, where x is the input number
5. a is a set with all the inputs the oracle has been executed on so far
6. c is the current configuration

We also introduce $F : (\text{symlist} \Rightarrow \text{symlist}) \Rightarrow \text{symlist} \Rightarrow \text{symlist}$, as well as the helper symbols $F' : (\text{symlist} \Rightarrow \text{symlist}) \Rightarrow \text{unat} \Rightarrow \text{config} \Rightarrow \text{symlist}$ and $\text{execute}' : (\text{symlist} \Rightarrow \text{symlist}) \Rightarrow \text{unat} \Rightarrow \text{unat} \Rightarrow \text{unat} \Rightarrow \text{set} \Rightarrow \text{config} \Rightarrow \text{symlist}$. We provide

the following rules, to execute the oracle Turing machine M operating in time bounded by P_M .

$$\begin{aligned}
& F F n \rightarrow F' F (\text{length } n) (\text{start } (\text{split}(\text{L nil}) (\text{R } n)) (\text{split}(\text{L nil}) (\text{R nil})) (\text{split}(\text{L nil}) (\text{R nil}))) \\
& F' F z c \rightarrow \text{execute } F \Theta_{P_{M+1}}^{F;\emptyset;z} 0 z \emptyset c \\
& \text{execute } F (s n) m z a (q t_1 t_2 t_3) \rightarrow \text{execute } F n (s m) z (\text{step } F (q t_1 t_2 t_3)), \\
& \text{for } q \notin \{\text{query, end}\} \\
& \text{execute } F (s n) m z a (\text{query } t_1 t_2 t_3) \rightarrow \\
& \quad \text{execute } F n (s m) z (\text{setcons } (\text{extract } t_2) a) (\text{query } t_1 t_2 t_3) \\
& \text{execute } F n m z a c \rightarrow \text{execute } F (\text{minus } \Theta_{P_{M+1}}^{F;a;z} m) m z a (\text{step } F c) \\
& \text{execute } F n m z a (\text{end } t_1 t_2 t_3) \rightarrow \text{extract } t_1
\end{aligned}$$

Before explaining these rules, let us revisit the shape of a machine execution: if $\text{initial}(n) \rightsquigarrow^* C$ in the oracle Turing machine M_f , and $\{a_1, \dots, a_k\}$ are the numbers on which the oracle is called during the transition up to configuration C , then let M be the number of \rightsquigarrow steps, and $N := P_M(|f|_{\{a_1, \dots, a_k\}}, |\underline{n}|) + 1 - M$. By Lemma 6.7.6, $N > 0$ (since the cost of a sequence of transition steps is at least the number of steps), and there must be strictly less than N steps before the next oracle call, or the final state must be reached instead. Writing \underline{i} for a unary number $s^i 0$ and $\#a_k \dots a_1\#$ for the data term $\text{setcons } [a_p] (\text{setcons } [a_{p-1}] (\dots (\text{setcons } [a_1] \emptyset) \dots))$, we let $[\rightsquigarrow^* C]$ denote the term $\text{execute } S_f N M \underline{|\underline{n}|} \#a_k \dots a_1\# [C]$. We will prove the following result:

Lemma 6.7.7. Let M_f be an OTM running in time bounded by P_M . Let C be a machine configuration such that $\text{initial}(n) \rightsquigarrow^* C$. Then $F S_f \underline{|\underline{n}|} \xrightarrow{+} [\rightsquigarrow^* C]$. Moreover, if C is a final configuration that yields m , then $[\rightsquigarrow^* C] \xrightarrow{+} \underline{|\underline{m}|}$.

Thus, we use F as the distinguished function symbol of Definition 6.5.6.

Proof. The application $F S_f \underline{|\underline{n}|}$ first computes $z := \underline{|\underline{n}|}$, and the initial configuration $[\text{initial}(n)]$. Both are passed to the helper function F' , which reduces to $[\rightsquigarrow^* \text{initial}(n)]$; that is: $\text{execute } S_f \underline{P_M(|f|_{\emptyset}, |\underline{n}|) + 1} \underline{0} \emptyset [\text{initial}(n)]$.

So suppose that $F S_f \underline{|\underline{n}|} \xrightarrow{+} [\rightsquigarrow^* C]$ and $C \rightsquigarrow C'$. We can always write $[\rightsquigarrow^* C] = \text{execute } S_f \underline{N} \underline{M} \underline{|\underline{n}|} \#a_k \dots a_1\# [C]$ There are two possibilities:

- The step is by a normal transition. In that case, necessarily $N \geq 2$ (since $N - 1$ must be at least 1 since it bounds the number of steps that can be done before the next oracle call), and $[\rightsquigarrow^* C] \xrightarrow{+} \text{execute } \underline{N - 1} \underline{M + 1} \underline{|\underline{n}|} \#a_k \dots a_1\# [C']$ by Lemma 6.7.5. Note that since the next oracle step or the end of the computation is now one step closer, indeed $(N - 1) - 1$ bounds this number of steps for the new configuration.

- The step is by an oracle call. In that case, $N \geq 1$ since $N - 1 \geq 0$, so we have the right shape to apply the second execute rule. Hence,

$$[\rightsquigarrow^* C] \xrightarrow{+} \text{execute}' S_f \underline{N} \underline{M + 1} \underline{||n||} \#a_{k+1}a_k \cdots a_1\#[C]$$

where a_{k+1} is the number to be read from the query tape, and this reduces to

$$\text{execute} S_f \underline{P_M(|f|_{\{a_1, \dots, a_{k+1}\}}, ||n||) + 1 - (M + 1)} \underline{M + 1} \underline{||n||} \#a_{k+1}a_k \cdots a_1\#[C']$$

by Lemma 6.7.5, and the observations on $\Theta_P^{F;a;z}$ from Section 6.7.3. This is exactly $[\rightsquigarrow^* C']$: clearly an extra step has been made, so M is increased by one, and the set of arguments the oracle has been applied to has been extended with the new element a_{k+1} . The counter is $P_M(|f|_{\{a_1, \dots, a_{k+1}\}}, ||n||) + 1 - (M + 1)$ is also exactly as expected, and indeed bounds the number of steps before the next oracle call as explained before the lemma.

This provides the first part of the lemma. As for the second part: if C is a final configuration, note that the second argument $N > 0$ as observed above the lemma, so the last execute rule applies, and $[\rightsquigarrow^* C]$ reduces to $\text{extract} \langle \text{second tape} \rangle$, which exactly reduces to the number yielded by this configuration. \square

We orient the execution rules by the interpretation below. Here we present each component separately since each component is large: $\llbracket \cdot \rrbracket^c$ is used to represent the numeric cost component of interpretations. The full interpretation is given below.

$$\begin{aligned}
\llbracket FF n \rrbracket^s &= \llbracket n \rrbracket^s + P_M(\llbracket F \rrbracket^s, \llbracket n \rrbracket^s) + 1 \\
\llbracket F' F z c \rrbracket^s &= \llbracket c \rrbracket^s + P_M(\llbracket F \rrbracket^s, \llbracket z \rrbracket^s) + 1 \\
\llbracket \text{execute } F n m z a c \rrbracket^s &= \llbracket c \rrbracket^s + \theta_{F,z,n,m} \\
\llbracket \text{execute}' F n m z a c \rrbracket^s &= \llbracket c \rrbracket^s + 1 + \theta_{F,z,n,m} \\
\llbracket FF n \rrbracket^c &= (P_M(\llbracket F \rrbracket^s, \llbracket n \rrbracket^s) + 1) * (\\
&\quad 8 + 3 * P_M(\llbracket F \rrbracket^s, \llbracket n \rrbracket^s) + 2 * \llbracket n \rrbracket^s + \\
&\quad \llbracket F \rrbracket^c (P_M(\llbracket F \rrbracket^s, \llbracket n \rrbracket^s) + \llbracket n \rrbracket^s + 1) + \\
&\quad \text{POLY}_{F,z}[P_M(\llbracket F \rrbracket^s, \llbracket n \rrbracket^s) + 1] \\
&\quad) + 6 + 2 * \llbracket n \rrbracket^s + \text{POLY}_{F,z}[0] \\
\llbracket F' F z c \rrbracket^c &= (P_M(\llbracket F \rrbracket^s, \llbracket z \rrbracket^s) + 1) * (\\
&\quad 8 + 3 * P_M(\llbracket F \rrbracket^s, \llbracket z \rrbracket^s) + 2 * \llbracket c \rrbracket^s + \\
&\quad \llbracket F \rrbracket^c (P_M(\llbracket F \rrbracket^s, \llbracket z \rrbracket^s) + 1 + \llbracket c \rrbracket^s) + \\
&\quad \text{POLY}_{F,z}[P_M(\llbracket F \rrbracket^s, \llbracket z \rrbracket^s) + 1] \\
&\quad) + 4 + \llbracket c \rrbracket^s + \text{POLY}_{F,z}[0] \\
\llbracket \text{execute } F n m z a c \rrbracket^c &= \theta_{F,z,n,m} * (\\
&\quad 5 + \\
&\quad 2 * (\theta_{F,z,n,m} + \llbracket c \rrbracket^s) + \\
&\quad \llbracket F \rrbracket^c (\theta_{F,z,n,m} + \llbracket c \rrbracket^s) + \\
&\quad \text{POLY}_{F,z}[\theta_{F,z,n,m} + \llbracket a \rrbracket^s] + \\
&\quad P_M(\llbracket F \rrbracket^s, \llbracket z \rrbracket^s) \\
&\quad) + 3 + \theta_{F,z,n,m} + \llbracket c \rrbracket^s \\
\llbracket \text{execute}' F n m z a c \rrbracket^c &= (\theta_{F,z,n,m} + 1) * (\\
&\quad 5 + \\
&\quad 2 * (\theta_{F,z,n,m} + \llbracket c \rrbracket^s + 1) + \\
&\quad \llbracket F \rrbracket^c (\theta_{F,z,n,m} + \llbracket c \rrbracket^s + 1) + \\
&\quad \text{POLY}_{F,z}[\theta_{F,z,n,m} + \llbracket a \rrbracket^s] + P_M(\llbracket F \rrbracket^s, \llbracket z \rrbracket^s) \\
&\quad) + 1
\end{aligned}$$

Where $\theta_{F,z,n,m} := \max(P_M(\llbracket F \rrbracket^s, \llbracket z \rrbracket^s) + 1 - \llbracket m \rrbracket^n, \llbracket n \rrbracket^s)$ and

$$\text{POLY}_{F,z}[x] := x * A_{P_M+1}(\llbracket F \rrbracket^c, \llbracket F \rrbracket^s, \llbracket z \rrbracket^s) + B_{P_M+1}(\llbracket F \rrbracket^c, \llbracket F \rrbracket^s, \llbracket z \rrbracket^s)$$

so the polynomial bounding $\llbracket \Theta_{P_M}^{F,a;z} \rrbracket^c$ if $\llbracket a \rrbracket^s = x$.

To see that these interpretations are correct, we first observe:

$$\begin{aligned}
\theta_{F,z,n,m} &= \max(P_M(\llbracket F \rrbracket^s, \llbracket z \rrbracket^s) + 1 - \llbracket m \rrbracket^n, \llbracket n \rrbracket^s + 1) \\
&= \max(P_M(\llbracket F \rrbracket^s, \llbracket z \rrbracket^s) + 1 - (\llbracket m \rrbracket^n + 1), \llbracket n \rrbracket^s) + 1 \\
&= \theta_{F,z,n,m} + 1
\end{aligned}$$

(Because $\max(a + 1, b + 1) = \max(a, b) + 1$.) We also have, for all a :

$$\begin{aligned}
\theta_{F,z,n,m} &= \max(P_M(\llbracket F \rrbracket^s, \llbracket z \rrbracket^s) + 1 - \llbracket m \rrbracket^n, \llbracket n \rrbracket^s) \\
&\geq \max(P_M(\llbracket F \rrbracket^s, \llbracket z \rrbracket^s) + 1 - \llbracket m \rrbracket^n, 0) \\
&= \max(P_M(\llbracket F \rrbracket^s, \llbracket z \rrbracket^s) + 1 - \llbracket m \rrbracket^n, \max(P_M(\llbracket F \rrbracket^s, \llbracket z \rrbracket^s) + 1 - \llbracket m \rrbracket^n), 0) \\
&= \theta_{F,z,\text{minus}} \Theta_{P_{M+1},m}^{F;a;z}
\end{aligned}$$

The cost inequalities now follow by writing out definitions.

We observe:

Lemma 6.7.8. Let $\Psi : (\mathbb{N} \rightarrow \mathbb{N}) \times \mathbb{N} \rightarrow \mathbb{N}$ be a type-2 functional, which is computed by the oracle Turing machine M in time P . Then for all $f \in \mathbb{N} \rightarrow \mathbb{N}$ and $n \in \mathbb{N}$, $\text{FS}_f \text{Size}_f \llbracket n \rrbracket \xrightarrow{+} \llbracket f(n) \rrbracket$ in the STRS $\mathbb{R} \cup \mathbb{R}'_f$ defined in this section.

Finally, we conclude with the completeness result.

Corollary 6.7.9 (Completeness). If a type-2 functional Ψ is in BFF, then there exists an orthogonal STRS (Σ, \mathbb{R}) that computes Ψ , and which admits a second-order polynomially-bounded interpretation such that a constant c exists with, for all $\llbracket n \rrbracket \in \mathcal{N}$: $|\llbracket n \rrbracket| = \llbracket \llbracket n \rrbracket \rrbracket^s$.

6.8 Conclusions and Future Work

In this chapter, we have shown that BFF can be characterized through second-order term rewriting systems admitting polynomially bounded cost–size interpretations. This is arguably the first characterization of the basic feasible functionals purely in terms of rewriting theoretic concepts.

For the purpose of presentation, we have imposed some mild restrictions that we believe are not essential in practice. In future extensions, we can eliminate these restrictions, such as allowing lambda-abstraction, non-base type rules, and higher-order functions (assuming that F is still second-order). We can also allow arbitrary inductive data structures as input.

Another direction we definitely wish to explore is the characterization of polynomial time complexity for functionals of order strictly higher than two. It is well known that the underlying theory in this case becomes less robust than in type-2 complexity. As such, it is not clear which of the existing proposals for complexity classes of higher-order polytime complexity we can hope to capture within our framework.

Chapter 7

Certification of Higher-Order Polynomial Interpretations

7.1 Certifying Termination Tools

Up to now in this thesis, we have been working mainly with complexity-related questions. In this chapter, we look at the certification (defined precisely below) problem for termination tools. Termination of the term rewriting system at hand guarantees that the derivational complexity function is well-defined, i.e., every term can be mapped to a natural number. For practical purposes, automating the termination proof is of major interest. Therefore, automatically proving termination is an important problem in term rewriting, and numerous tools have been developed for this purpose, such as AProVE [40], NaTT [110], MatchBox [104], MU-TERM [43], SOL [47], $\mathbb{T}\mathbb{T}_2$ [72] and Wanda [65], which compete against each other in an annual termination competition [41]. Aside from basic (first-order) term rewriting, this includes tools analyzing for instance string, conditional, and higher-order term rewriting.

Developing termination tools is a difficult and error-prone endeavor. On the one hand, the termination techniques that are implemented may contain errors. This is particularly relevant in higher-order term rewriting, where the proofs are often very intricate due to partial application, type structure, beta-reduction, and techniques often not transferring perfectly between different formalisms of higher-order rewriting. Hence, it should come as no surprise that errors have been found in published papers on higher-order rewriting. On the other hand, it is very easy for a tool developer to accidentally omit a test whether some conditions to apply specific termination techniques are satisfied, or to incorrectly translate a method between higher-order formalisms.

To exacerbate this issue, termination proofs are usually complex and technical in nature, which makes it hard to assess the correctness of a prover's output by hand. Not only do many benchmarks contain hundreds of rules, modern termination tools make

use of various proof methods that have been developed for decades. Indeed, a single termination proof might, for instance, make use of a combination of dependency pairs [5, 39, 68], recursive path orders [19, 71], rule removal, and multiple kinds of interpretations [38, 69, 85, 109]. This makes bugs very difficult to find. Hence, there is a need to formally certify the output of termination provers, ideally automatically. There are two common engineering strategies to provide such certification. In the first, one builds the certifier as a library in a proof assistant along with tools to read the prover’s output and construct a formal proof, which we call *proof script*. The proof script is then verified by a proof assistant. Examples of this system design are the combinations Coccinelle/CiME3 [31] and CoLoR/Rainbow [20]. In the second, the formalization includes certified algorithms for checking the correctness of the prover’s output. This allows for the whole certifier to be extracted, using code extraction, and be used as a standalone program. Hence, the generation of proof scripts by a standalone tool is not needed in this approach, but it comes with a higher formalization cost. IsaFoR/CeTA [101] utilizes this approach.

When it comes to higher-order rewriting, however, the options are limited. Both Coccinelle [31] and IsaFoR/CeTA [101] only consider first-order rewriting. Meanwhile, CoLoR/Rainbow [20] does include a formalization of an early definition of HORPO [71]. Since here we use a different term formalism compared to that of [71], our results are not directly compatible. See for instance [3, 95] for more formalization results in rewriting.

In this chapter, we introduce a new combination Nijn/ONijn for the certification of higher-order rewriting termination proofs. We follow the first aforementioned system design: Nijn is a Coq library providing a formalization of the underlying higher-order rewriting theory and ONijn is a proof script generator that given a minimal description of a termination proof (which we call *proof trace*), outputs a Coq proof script. The proof script then utilizes results from Nijn for checking the correctness of the traced proof. The schematic below depicts the basic steps to produce proof certificates using Nijn/ONijn.

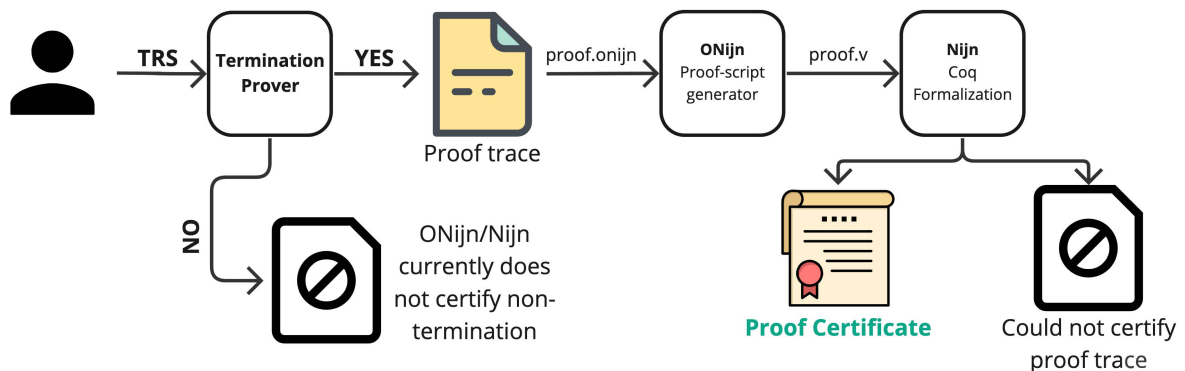


Fig. 7.1 Nijn/ONijn schematics

While Nijn is the certified core part of our tool since it is checked by Coq, the proof script generation implemented in OCaml (ONijn) is not currently certified and must be

trusted. For this reason, we deliberately keep ONijn as simple as possible and minimal checking or computation is done by it. The only task delegated to ONijn is that of parsing the proof trace given by the termination prover to a Coq proof script. Additionally, checking the correctness of polynomial termination proofs in Coq is an inherently incomplete task, since it would require a method to solve inequalities over arbitrary polynomials, which is undecidable in general.

Technical Overview. This chapter orbits Nijn, a Coq library formalizing higher-order rewriting. It can be found in the following repository:

<https://zenodo.org/records/7913023>

The formalization is based on intensional dependent type theory extended with two axioms: *function extensionality* and *uniqueness of identity proofs* [55]. Currently, the termination criterion formalized in the library is *the higher-order polynomial method*, introduced in [38]. The tool `coqwc` counts the following number of lines of code:

spec	proof	comments	
5497	1985	272	total

The higher-order interpretation method roughly works as follows. First, types are interpreted as well-ordered structures (Definition 7.3.4), compositionally. For instance, we interpret base types as natural numbers (with the usual ordering). Then we interpret a functional type $\sigma \Rightarrow \tau$ as the set of weakly monotonic functions from $\langle\sigma\rangle$ to $\langle\tau\rangle$ where $\langle\sigma\rangle, \langle\tau\rangle$ denote the interpretations of σ, τ respectively. The second step is to map inhabitants of a type σ to elements of $\langle\sigma\rangle$, which is expressed here by Definition 7.3.10.

This interpretation, called *extended monotonic algebras* in [38], alone does not suffice for termination. To guarantee termination, we interpret both term application (Definition 7.4.6) and function symbols as strongly monotonic functionals. In addition, terms must be interpreted in such a way that the rules of the system are strictly oriented, i.e., $\llbracket \ell \rrbracket > \llbracket r \rrbracket$, for all rules $\ell \rightarrow r$. This means that whenever a rewriting is fired in a term, the interpretation of that term strictly decreases. As such, termination is guaranteed. Here we use *termination models* (Definition 7.3.11) to collect these necessary conditions.

The main result establishing the correctness of this technique in the higher-order case is expressed by Theorem 7.3.12. To the reader familiar with *the interpretation method* in first-order rewriting, Theorem 7.4.7 would be no surprise. It is essentially the combination of the Manna–Ness criterion with higher-order polynomials and the additional technicalities that are needed for the higher-order case.

7.2 The Basics of Higher-Order Rewriting in Coq

In this section, we introduce the basic coq constructs needed to formalize important aspects of higher-order rewriting theory, mainly some notions from Chapter 2 like types, terms, substitutions, and rewriting rules. We end the section with an exposition on how to express termination constructively in Coq.

7.2.1 Terms and Rewrite Rules

Let us start by defining *simple types*.

Definition 7.2.1 (ty). **Simple types** over a type B are defined as follows:

```
Inductive ty (B : Type) : Type :=
| Base : B → ty B
| Fun  : ty B → ty B → ty B.
```

Elements of B are called base types, as before. Every inhabitant $b : B$ gives rise to a simple type $\text{Base } b$ and if A_1, A_2 are simple types then so is $\text{Fun } A_1 A_2$. We write $A_1 \rightarrow A_2$ for $\text{Fun } A_1 A_2$.

In Chapter 2 we defined terms with *named* variables and identified them up to α -equivalence. In contrast, for the formalization, we work with the so-called *de Bruijn indices*, also called the *nameless representation* of terms. In this representation, we do not use variable names but refer to their position in a context. We need (*variable*) *contexts* in order to type terms that may contain free variables. Conceptually, a context is a list of variables with their respective types. For instance, $[x_0 : \sigma_0; \dots; x_n : \sigma_n]$ is the context with variables x_0 of type σ_0, \dots, x_n of type σ_n . However, we use nameless variables in our development, so we do not keep track of their names. Consequently, a context is represented by a list of types. Then we only consider the list $[\sigma_0, \dots, \sigma_n]$. However, we still need to refer to the free variables in terms. In order to do so, we represent them through indexing positions in the context. For instance, in the context $[\sigma_0; \dots; \sigma_n]$ we have $n + 1$ position indexes $0, 1, \dots, n$, which we use as variables.

Definition 7.2.2 (con). The type of **variable contexts** over a type B is defined as follows.

```
Inductive con (B : Type) : Type :=
| Empty : con B
| Extend : ty B → con B → con B.
```

We write \bullet for Empty and $A \bullet C$ for $\text{Extend } A C$.

Definition 7.2.3 (var). We define the type $\text{var } C A$ of **variables** of type A in context C as

```
Inductive var {B : Type} : con B → ty B → Type :=
| Vz : forall {C : con B} {A : ty B}, var (A ● C) A
| Vs : forall {C : con B} {A1 A2 : ty B}, var C A2 → var (A1 ● C) A2.
```

Let us consider an example of a context and some variables. Suppose that we have a base type denoted by b . Then we can form the context

$$\text{Base } b \text{ ,, Base } b \longrightarrow \text{Base } b \text{ ,, Empty.}$$

In this context, we have two variables. The first one, which is Vz , has type $\text{Base } b$, and the second variable, which is $Vs Vz$, has type $\text{Base } b \longrightarrow \text{Base } b$. The context that we discussed corresponds to $[x_0 : b; x_1 : b \longrightarrow b]$. The variable Vz represents x_0 , while $Vs Vz$ represents x_1 .

In Definition 7.2.4 below we define the notion of *well-typed terms-in-context* which consists of those expressions such that there is a typing derivation. We use dependent types to ensure well-typedness of such expressions. The type of terms depends on a simple type $A : \text{ty } B$ (which represents the object-level type of the expression) and context $C : \text{con } B$ that carries the types of all free variables in the term. We also need to type function symbols. So we require a type $F : \text{Type}$ of function symbols and $\text{ar} : F \rightarrow \text{ty } B$, which maps $f : F$ to a simple type $\text{ar } f$.

Definition 7.2.4 (tm). We define the type of **well-typed terms** as follows

```
Inductive tm {B : Type} {F : Type} (ar : F → ty B) (C : con B) : ty B → Type :=
| BaseTm : forall (f : F), tm ar C (ar f)
| TmVar : forall {A : ty B}, var C A → tm ar C A
| Lam : forall {A1 A2 : ty B}, tm ar (A1 ,, C) A2 → tm ar C (A1 → A2)
| App : forall {A1 A2 : ty B}, tm ar C (A1 → A2) → tm ar C A1 → tm ar C A2.
```

For every function $f : F$ we have a term $\text{BaseTm } f$ of type $\text{ar } f$. Every variable v gives rise to a term $\text{TmVar } v$. For λ -abstractions, given a term $s : \text{tm ar } (A1 \text{ ,, } C) A2$, there is a term $\lambda s : \text{tm ar } C (A1 \longrightarrow A2)$, namely $\text{Lam } s$. The last constructor represents term application. If we have a term $s : \text{tm ar } C (A1 \longrightarrow A2)$ and a term $t : \text{tm ar } C A1$, we get a term $s \cdot t : \text{tm ar } C A2$, which is defined to be $\text{App } s \ t$.

While it may be more cumbersome to write down terms using de Bruijn indices, it does have several advantages. Most importantly, it eliminates the need for α -equivalence, so that determining equality between terms is reduced to a simple syntactic check.

Our notion of rewriting rules deviates from the presentation in [38]. Mainly, we do not impose the pattern restriction on the left-hand side of rules nor that free variables on the right-hand side occur on the left-hand side. This choice simplifies the formalization effort because when defining a concrete TRS, one does not need to check this particular condition. Note that in *IsaFoR* [101] the authors also do not impose any variable conditions when defining rewriting rules.

Definition 7.2.5 (*rewriteRule*). The type of **rewrite rules** is defined as follows:

```
Record rewriteRule {B : Type} {F : Type} (ar : F → ty B) :=
  make_rewrite {
    vars_of : con B ;
    tar_of : ty B ;
    lhs_of : tm ar vars_of tar_of ;
    rhs_of : tm ar vars_of tar_of
  }.
```

The context `vars_of` carries the variables used in the rule, and the type `tar_of` is used to guarantee that both the `lhs_of` and `rhs_of` are terms of the same type.

Definition 7.2.6 (*afs*). The type of **algebraic functional systems**¹ is defined as follows

```
Record afs (B : Type) (F : Type) :=
  make_afs {
    arity : F → ty B ;
    list_of_rewriteRules : list (rewriteRule arity)
  }.
```

As usual, every AFS induces a rewrite relation on the set of terms, which we denote by \sim . The formal definition is found in [RewritingSystem.v](#). The rewrite relation \sim is defined to be the closure of the one-step relation under transitivity and compatibility with the term constructors. In Coq, we use an inductive type to define this relation. Each rewrite step is represented by a constructor. More specifically, we have a constructor for rewriting the left-hand and the right-hand side of an application, we have a constructor for β -reduction, and we have a constructor for the rewrite rules of the AFS.

Example 7.2.7 (*map_afs*). Let us encode \mathbb{R}_{map} in Coq. It is composed of two rules: $\text{map } F [] \rightarrow []$ and $\text{map } F (\text{cons } x q) \rightarrow \text{cons } (F x) (\text{map } F q)$. We start with base types.

```
Inductive base_types := TBtype | TList.
```

```
Definition Btype : ty base_types := Base TBtype.
```

```
Definition List : ty base_types := Base TList.
```

The abbreviations `Btype` and `List` is to smoothen the usage of the base types. There are three function symbols in this system:

```
Inductive fun_symbols := TNil | TCons | TMap.
```

The arity function `map_ar` maps each function symbol in `fun_symbols` to its type.

¹The nomenclature *algebraic functional systems* used here is an unfortunate legacy of old Nijn code. So it is not supposed to model algebraic functional systems as defined in [58] but the formalism we defined in Chapter 2. Major renamings in the formalization will come in the next major version of Nijn.

```

Definition map_ar f : ty base_types
:= match f with
  | TNil  $\Rightarrow$  List
  | TCons  $\Rightarrow$  Btype  $\longrightarrow$  List  $\longrightarrow$  List
  | TMap  $\Rightarrow$  (Btype  $\longrightarrow$  Btype)  $\longrightarrow$  List  $\longrightarrow$  List
end.

```

So, TNil is a list and given an inhabitant of Btype and List, the function symbol TCons gives a List. Again we introduce some abbreviations to simplify the usage of the function symbols.

```

Definition Nil {C} : tm map_ar C _ := BaseTm TNil.

```

```

Definition Cons {C} x xs : tm map_ar C _ := BaseTm TCons · x · xs.

```

```

Definition Map {C} f xs : tm map_ar C _ := BaseTm TMap · f · xs.

```

The first rule, $\text{map } F [] \rightarrow []$, is encoded as the following Coq construct:

```

Program Definition map_nil :=
  make_rewrite
  (_ ,, •) _
  (let f := TmVar Vz in Map · f · Nil)
  Nil.

```

Notice that we only defined the *pattern* of the first two arguments of `make_rewrite`, leaving the types in the context `(_ ,, •)` and the type of the rule unspecified. Coq can fill in these holes automatically, as long as we provide a context pattern of the correct length. In this particular rewrite rule, there is only one free variable. As such, the variable `TmVar Vz` refers to the only variable in the context. In addition, we use iterated `let`-statements to imitate variable names. For every position in the context, we introduce a variable in Coq, which we use in the left- and right-hand sides of the rule. This makes the rules more human-readable. Indeed, the lhs $\text{map } F []$ of this rule is represented as `Map · f · Nil` in code. The second rule for `map` is encoded following the same ideas.

```

Program Definition map_cons :=
  make_rewrite
  (_ ,, _ ,, _ ,, •) _
  (let f := TmVar Vz in
    let x := TmVar (Vs Vz) in
      let xs := TmVar (Vs (Vs Vz)) in
        Map · f · (Cons · x · xs))
  (let f := TmVar Vz in
    let x := TmVar (Vs Vz) in
      let xs := TmVar (Vs (Vs Vz)) in
        Cons · (f · x) · (Map · f · xs)).

```


Putting this all together, we obtain an AFS, which we call `map_afs`.

Definition `map_afs` :=
`make_afs map_ar (map_nil :: map_cons :: List.nil)`.

7.2.2 Termination

Strong normalization is usually defined as the absence of infinite rewrite sequences. Such a definition is sufficient in a classical setting where the law of excluded middle holds. However, we work in a constructive setting, and thus we are interested in a stronger definition. Therefore, we need a constructive predicate, formulated positively, which implies there are no infinite rewrite sequences. This idea is captured by the following definition

Definition 7.2.8 (WellfoundedRelation.v). The **well-foundedness predicate** for a relation R is defined as follows

Inductive `isWf` { X : Type} (R : $X \rightarrow X \rightarrow$ Type) (x : X): Prop :=
| `acc`: (`forall` (y : X), R x $y \rightarrow$ `isWf` R y) \rightarrow `isWf` R x .

A relation is **well-founded** if the well-foundedness predicate holds for every element.

Definition `Wf` { X : Type} (R : $X \rightarrow X \rightarrow$ Type) :=
`forall` (x : X), `isWf` R x .

Note that this definition has been considered numerous times before, for example in [18] and in CoLoR [20]. An element x is well-founded if all y such that R x y are well-founded. Note that if there is no y such that R x y , then x is vacuously well-founded. From the rewriting perspective, this definition properly captures the notion of strong normalization. Indeed, a term s is strongly normalizing iff every s' such that s rewrites to s' is strongly normalizing.

Well-foundedness contradicts the existence of infinite rewrite sequences, even in a constructive setting. As such, it indeed gives a stronger condition.

Proposition 7.2.9 (no_infinite_chain). If R is well-founded, then there is no infinite sequence s_0, s_1, \dots such that $R(s_n, s_{n+1})$, for all n .

Next, we define strong normalization using well-founded predicates.

Definition 7.2.10 (is_SN). An algebraic functional system is **strongly normalizing** if for every context C and every type A the rewrite relation for terms of type A in context C is well-founded. We formalize that as follows:

Definition `isSN` { B F : Type} (X : `afs` B F): Prop :=
`forall` (C : `con` B) (A : `ty` B),
`Wf` (`fun` (t_1 t_2 : `tm` X C A) \Rightarrow $t_1 \sim t_2$).

7.3 Higher-Order Interpretation Method in Coq

In this section, we formalize the method of weakly monotonic algebras for algebraic functional systems. We proceed by providing type-theoretic semantics for the syntactic constructions introduced in the last section and a sufficient condition for which such semantics can be used to establish strong normalization.

7.3.1 Interpreting types and terms

In weakly monotonic algebras, types are interpreted as sets along with a well-founded ordering and a quasi-ordering [38, 92]. For that reason, we start by defining *compatible relations*. Intuitively, these are the domain for our semantics.

Definition 7.3.1 (*CompatibleRelation.v*). **Compatible relations** are defined as follows

```
Record CompatRel := {
  carrier :> Type ;
  gt : carrier → carrier → Prop ;
  ge : carrier → carrier → Prop
}.
```

We write $x > y$ and $x \geq y$ for $gt\ x\ y$ and $ge\ x\ y$ respectively.

The record `CompatRel` consists of the data needed to express compatibility between $>$ and \geq . The conditions it needs to satisfy, are in the type class `isCompatRel`, defined below.

```
Class isCompatRel (X : CompatRel) := {
  gt_trans : forall {x y z : X}, x > y → y > z → x > z ;
  ge_trans : forall {x y z : X}, x ≥ y → y ≥ z → x ≥ z ;
  ge_refl  : forall (x : X), x ≥ x ;
  compat   : forall {x y : X}, x > y → x ≥ y ;
  ge_gt    : forall {x y z : X}, x ≥ y → y > z → x > z ;
  gt_ge    : forall {x y z : X}, x > y → y ≥ z → x > z
}.
```

Note that the field `gt_trans` in `isCompatRel` follows from `compat` and `ge_gt`. The type `nat` of natural numbers with the usual orders is the first example of data that satisfies `isCompatRel`. We denote this one by `nat_CompatRel`.

This type class essentially models the notion of extended well-founded set introduced in Chapter 2. An **extended well-founded set** is a set together with compatible orders $>$, \geq such that $>$ is well-founded and \geq is a quasi-ordering. This compatibility requirement corresponds to the axiom `compat` in the type class `isCompatRel`. However, since we do not require $>$ to be well-founded in this definition, we instead call it a compatible relation.

More specifically, x is a compatible relation if it is of type `CompatRel` and satisfies the constraints in the type class `isCompatRel`.

In order to interpret simple types (Definition 7.2.1), we start by fixing a type $B : \text{Type}$ of base types and an interpretation $\text{semB} : B \rightarrow \text{CompatRel}$ such that each $\text{semB } b$ is a compatible relation. Whenever semB satisfies such property we call it an **interpretation key** for B , like in Definition 5.3.4. Notice that here we do not formalize call-by-value rewriting, like in Chapter 5. An interpretation key in this chapter is just the function that maps each base type to a compatible relation type. We interpret arrow types as functional compatible relations, i.e., compatible relations such that the inhabitants of their carrier are functional. The class of functionals we are interested in is that of *weakly-monotone maps*.

Definition 7.3.2 (MonotonicMaps.v). **Weakly monotone maps** are defined as follows

```
Class weakMonotone {X Y : CompatRel} (f : X → Y) :=
  map_ge : forall (x y : X), x >= y → f x >= f y.
```

```
Record weakMonotoneMap (X Y : CompatRel) :=
  make_monotone {
    fun_carrier :> X → Y ;
    is_weak_monotone : weakMonotone fun_carrier
  }.
```

The class `weakMonotone` says when a function is weakly monotonic, and an inhabitant of the record `weakMonotoneMap` consists of a function together with proof of its weak monotonicity. Then we define `fun_CompatRel` which is of type `CompatRel` and represents the **functional compatible relations** from X to Y . It is defined as follows:

```
Definition fun_CompatRel (X Y : CompatRel) : CompatRel :={ |
  carrier := weakMonotoneMap X Y ;
  gt f g := forall (x : X), f x > g x ;
  ge f g := forall (x : X), f x >= g x
| }.
```

In what follows, we write $X \rightarrow_{\text{wm}} Y$ for `fun_CompatRel X Y`. The semantics for a type is parametrized by an interpretation key semB .

Here, the idea is the same: an interpretation key maps base types to elements of the interpretation's domain. In our case, this is a `CompatRel`.

Definition 7.3.3. In the file `OrderInterpretation.v` we introduce the following Coq-level context to define the notion of **interpretation key** over a type B .

```
Context {B : Type}
  (semB : B → CompatRel)
  '{ forall (b : B), isCompatRel (semB b)}.
```

This Coq context makes explicit the information that the output Now we proceed to define the interpretation of types.

Definition 7.3.4 (sem_Ty). Assume $A : \text{ty } B$ and $\text{sem}B$ is an interpretation key for B . Then

```
Fixpoint sem_Ty (A : ty B) : CompatRel :=
  match A with
  | Base b    => semB b
  | A1 → A2  => sem_Ty A1 →wm sem_Ty A2
  end.
```

We also show how to interpret contexts, and to do so, we need to interpret the empty context and context extension. For those, we define the unit and product of compatible relations.

Definition 7.3.5 (Examples.v). The **unit** compatible relations

Definition unit_CompatRel :

```
CompatRel := { |
  carrier := unit ;
  gt _ _ := False ;
  ge _ _ := True
  |}.
```

The product compatible relation is implemented as follows:

Definition prod_CompatRel (X Y : CompatRel) :

```
CompatRel := { |
  carrier := X * Y ;
  gt x y := fst x > fst y ∧ snd x > snd y ;
  ge x y := fst x >= fst y ∧ snd x >= snd y
  |}.
```

Note that `unit_CompatRel` is the compatible relation on the type with only one element, for which the ordering is trivial. In addition, `prod_CompatRel` is the compatible relation on the product, for which we compare elements coordinate-wise. We write $X * Y$ for `prod_CompatRel X Y`.

Definition 7.3.6 (sem_Con). Contexts are interpreted as follows

Fixpoint sem_Con (C : con B) : CompatRel :=

```
  match C with
  | •      => unit_CompatRel
  | A ,, C => sem_Ty A * sem_Con C
  end.
```

Next, we give semantics to variables and terms. The approach we use here is slightly different from what is usually done in higher-order rewriting. In [38, 69, 92], for instance, context information is lifted to the meta-level and variables are interpreted using the notion of valuations. In contrast, in our setting, the typing context lives at the syntactic level and variables are interpreted as weakly monotonic functions. Consequently, to every term $t : \text{tm } C \ A$, we assign a map from $\text{sem_Con } C$ to $\text{sem_Ty } A$. In the remainder, we need the following weakly monotonic functions.

Definition 7.3.7 (Examples.v). We define the following weakly monotonic functions.

- Given $y : Y$, we write $\text{const_wm } y : X \rightarrow_{\text{wm}} Y$ for the constant function.
- Given $f : X \rightarrow_{\text{wm}} Y$ and $g : Y \rightarrow_{\text{wm}} Z$, we define $g \circ_{\text{wm}} f : X \rightarrow_{\text{wm}} Z$ to be their composition.
- We have the first projection $\text{fst_wm} : X * Y \rightarrow_{\text{wm}} X$, which sends a pair (x, y) to x , and the second projection $\text{snd_wm} : X * Y \rightarrow_{\text{wm}} Y$, which sends (x, y) to y .
- Given $f : X \rightarrow_{\text{wm}} Y$ and $g : X \rightarrow_{\text{wm}} Z$, we have a function $\langle f, g \rangle : X \rightarrow_{\text{wm}} (Y * Z)$. For $x : X$, we define $\langle f, g \rangle x$ to be $(f x, g x)$.
- Given $f : Y * X \rightarrow_{\text{wm}} Z$, we get $\lambda_{\text{wm}} f : X \rightarrow_{\text{wm}} (Y \rightarrow_{\text{wm}} Z)$. For every $x : X$ and $y : Y$, we define $\lambda_{\text{wm}} f y x$ to be $f (y, x)$.
- Given $f : X \rightarrow_{\text{wm}} (Y \rightarrow_{\text{wm}} Z)$ and $x : X \rightarrow_{\text{wm}} Y$, we obtain $f \cdot_{\text{wm}} x : X \rightarrow_{\text{wm}} Z$, which sends every $a : X$ to $f a (x a)$.
- Given $x : X$, we have $\text{apply_el_wm } x : (X \rightarrow_{\text{wm}} Y) \rightarrow_{\text{wm}} Y$ which is a weakly monotonic function that sends $f : X \rightarrow_{\text{wm}} Y$ to $f x$.

Recall that variables are represented by positions in a context which in turn is interpreted as a weakly monotonic product (Definition 7.3.6). This allows us to interpret the variable at a position in a context as the corresponding interpretation of the type in that position.

Definition 7.3.8 (sem_Var). We interpret variables with the following function

```
Fixpoint sem_Var {C : con B} {A : ty B} (v : var C A) : sem_Con C →wm sem_Ty A
:= match v with
| Vz   ⇒ fst_wm
| Vs v ⇒ sem_Var v ◦wm snd_wm
end.
```

We need the following data in order to provide semantics to terms. An arity function $\text{ar} : F \rightarrow \text{ty } B$, together with its interpretation $\text{sem}_F : \text{forall } (f : F), \text{sem_Ty } (\text{ar } f)$, and an *application operator* given by

```
semApp : forall (A1 A2 : ty B),
  (sem_Ty A1 →wm sem_Ty A2) * sem_Ty A1 →wm sem_Ty A2
```

to interpret term application.

Remark 7.3.9. A first, but incorrect, guess to interpret application would have been by interpreting the application of $f : \text{sem_Ty } A1 \rightarrow_{\text{wm}} \text{sem_Ty } A2$ to $x : \text{sem_Ty } A1$ by $f \ x$. However, there is a significant disadvantage of this interpretation. Ultimately, we want to deduce strong normalization from the interpretation, and the main idea is that if we have a rewrite $x \sim x'$, then we have $\text{semTm } x > \text{semTm } x'$. This requirement would not be satisfied if we interpret term application as functional application. Indeed, if we have $x < x'$, then one is not guaranteed that we also have $f \ x < f \ x'$, because f is only weakly monotone.

There are two ways to deal with this. One way is by interpreting function types as strictly monotonic maps [69]. In this approach, this interpretation of application is valid. However, it comes at a price, because the interpretation of lambda abstraction becomes more difficult.

Another approach, which we use here, is also used in [38]. We add a parameter to our interpretation method, namely `semApp`, which abstractly represents the interpretation of application. To deduce strong normalization in this setting, we add requirements about `semApp` in Section 7.3.2. As a result, in concrete instantiations of this method, we need to provide an actual definition for `semApp`. We see this in Section 7.4.2.

Definition 7.3.10 (sem_Tm). Given a function `semF : forall (f : F), sem_Ty (ar f)`, the semantics of terms is given by

```
Fixpoint sem_Tm {C : con B} {A : ty B} (t : tm ar C A) : sem_Con C →wm sem_Ty A :=
  match t with
  | BaseTm f   => const_wm (semF f)
  | TmVar v    => sem_Var v
  | λ f        => λ_wm (sem_Tm f)
  | f · t      => semApp _ _ ◦wm ⟨sem_Tm f, sem_Tm t⟩
  end.
```

Notice that we could have chosen a fixed way of interpreting term application. We follow the same approach used by Fuhs and Kop [38] in our formalization and leave `semApp` abstract. This choice is essential if we want to use the interpretation method for both *rule removal* and the *dependency pair* approach. See [64, Chapters 4 and 6] for more detail.

7.3.2 Termination Models

Now we have set up everything that is necessary to define the main notion of this section: *termination models*. From a termination model of an algebraic functional system,

one obtains an interpretation of the types and terms. In addition, every rewrite rule is satisfied in this interpretation that is the interpretation of the left-hand side of rules is strictly greater than the interpretation of the right-hand side.

Definition 7.3.11 (Interpretation). Let \mathbb{R} be an algebraic functional system with base type B and function symbols F . A **termination model** of \mathbb{R} consists of the following:

1. an interpretation key sem_B ;
2. a function $\text{sem}_F : \text{forall } (f : F), \text{ sem_Ty } (\text{ar } f)$;
3. a function

$$\begin{aligned} \text{semApp} : & \text{forall } (A1 A2 : \text{ty } B), \\ & (\text{sem_Ty } A1 \rightarrow_{\text{wm}} \text{sem_Ty } A2) * \text{sem_Ty } A1 \rightarrow_{\text{wm}} \text{sem_Ty } A2 \end{aligned}$$

such that the following axioms are satisfied

- each $\text{sem}_B b$ is well-founded and inhabited;
- if $f > f'$, then $\text{semApp } _ _ (f, x) > \text{semApp } _ _ (f', x)$;
- if $x > x'$, then $\text{semApp } _ _ (f, x) > \text{semApp } _ _ (f, x')$;
- we have $\text{semApp } _ _ (f, x) \geq f x$ for all f and x ;
- for every rewrite rule r , substitution s , and element x , we have

$$\text{semTm } (\text{lhs } r [s]) x > \text{semTm } (\text{rhs } r [s]) x.$$

Notice that given a termination model for \mathbb{R} , the fact that the left-hand side of every rewrite rule is greater than its right-hand side only orients those rewriting sequences that do not have any β -steps. In contrast with our development in Chapter 4 in which we showed how to interpret the β rule scheme. In the setting of Kop [64], termination models alone are not enough to conclude strong normalization. Kop [64] solves this issue by employing *rule removal* to show that strong normalization of all rewriting sequences follows from the strong normalization of β -reduction, which is a famous theorem proven by Tait [99]. The strong normalization of β -reduction has been formalized numerous times and an overview can be found in [1]. Now we are ready to state and prove the main theorem of this section.

Theorem 7.3.12 (afs_is_SN_from_Interpretation). Let \mathbb{R} be an algebraic functional system. If we have a termination model of X , then X is strongly normalizing.

7.4 The Higher-Order Polynomial Method in Coq

7.4.1 Polynomials

In this section, we instantiate the material of Section 7.3 to a concrete instance, namely *the polynomial method* [38]. For that reason, we define the notation of *higher-order polynomial*.

Definition 7.4.1 (Polynomial.v). We define the type `base_poly` of **base polynomials** and `poly` of **higher-order polynomials** by mutual induction as follows:

```

Inductive base_poly {B : Type} : con B → Type :=
| P_const : forall {C : con B},
  nat → base_poly C
| P_plus : forall {C : con B},
  (P1 P2 : base_poly C) → base_poly C
| P_mult : forall {C : con B},
  (P1 P2 : base_poly C) → base_poly C
| from_poly : forall {C : con B} {b : B},
  poly C (Base b) → base_poly C

with poly {B : Type} : con B → ty B → Type :=
| P_base : forall {C : con B} {b : B},
  base_poly C → poly C (Base b)
| P_var : forall {C : con B} {A : ty B},
  var C A → poly C A
| P_app : forall {C : con B} {1A 2A : ty B},
  poly C (1A → 2A) →
  poly C 1A →
  poly C 2A
| P_lam : forall {C : con B} {1A 2A : ty B},
  poly (1A ,, C) 2A → poly C (1A → 2A).

```

We can make expressions of base polynomials using `P_const` (constants), `P_plus` (addition), and `P_mult` (multiplication). In addition, `from_poly` takes an inhabitant of `poly C (Base b)` and returns a base polynomial in context `C`. Using `P_base`, we can turn a base polynomial into a polynomial of any base type. The constructors, `P_var`, `P_app`, and `P_lam`, are reminiscent of the simply typed lambda calculus. We get variables from `P_var`, λ -abstraction from `P_lam`, and application from `P_app`. Note that combining `from_poly` and `P_var`, we can use variables in base polynomials.

Let us make some remarks about the design choices we made and how they affected the definition of polynomials. One of our requirements is that we are able to add and multiply polynomials on different base types. This is frequently used in actual examples, such as Example 7.4.2. Function symbols might use arguments from different base types, and we would like to use both of them in polynomial expressions.

Note that our definition of higher-order polynomials is rather similar to the one given by Fuhs and Kop [38, Definition 4.1]. They define a set $\text{Pol}(X)$, which consists of polynomial expressions, and for every type A a set $\text{Pol}^A(X)$. The set $\text{Pol}^A(X)$ is defined by recursion: for base types, it is the set of polynomials over X and for function types $A_1 \rightarrow A_2$, it consists of expressions $\Lambda(y : A_1).P$ where P is a polynomial of type A_2 using an extra variable $y : A_1$. Our `base_poly C` and `poly C A` correspond to $\text{Pol}(X)$ and $\text{Pol}^A(X)$ respectively. However, there are some differences. First of all, Fuhs and Kop require variables to be fully applied, whereas we permit partially applied variables. Secondly, Fuhs and Kop define polynomials in such a way that for every two base types b_1, b_2 the types $\text{Pol}^{b_1}(X)$ and $\text{Pol}^{b_2}(X)$ are equal. This is not the case in our definition: instead we use constructors `from_poly` and `P_base` to relate `base_poly C` and `poly C (Base b)`.

In the polynomial method, the interpretation key sends every base type to a type `nat_CompatRel`, and in what follows, we write $\llbracket C \rrbracket_{\text{con}}$ and $\llbracket A \rrbracket_{\text{ty}}$ for the interpretation of contexts and types respectively. Note that every polynomial $P : \text{poly C A}$ gives rise to a weakly monotonic function $\text{sem_poly } P : \llbracket C \rrbracket_{\text{con}} \rightarrow_{\text{wm}} \llbracket A \rrbracket_{\text{ty}}$ and that every base polynomial $P : \text{base_poly C}$ gives rise to $\text{sem_base_poly } P : \llbracket C \rrbracket_{\text{con}} \rightarrow_{\text{wm}} \text{nat_CompatRel}$. These two functions are defined using mutual recursion and every constructor is interpreted in the expected way: `sem_poly`.

In order to actually use `base_poly C` and `poly C A`, we provide convenient notations for operations on polynomials. More concretely, we define notations $+$, $*$, and $\cdot P$ that represent addition, multiplication, and application respectively. These operations must be overloaded since we need to be able to add polynomials of different types. To do so, we similarly use type classes as in `MathClasses` [97]. For details, we refer the reader to the formalization.

Example 7.4.2 (`map_fun_poly`). We continue with Example 7.2.7 and provide a polynomial interpretation to the system `map_afs` as follows:

```

Definition map_fun_poly fn_symbols : poly •(arity trs fn_symbols) :=
  match fn_symbols with
  | Tnil  $\Rightarrow$  to_Poly (P_const 3)
  | Tcons  $\Rightarrow$   $\lambda P \lambda P$  let y1 := P_var Vz in
    to_Poly (P_const 3 + P_const 2 * y1)
  | Tmap  $\Rightarrow$   $\lambda P$  let y0 := P_var (Vs Vz) in  $\lambda P$  let G1 := P_var Vz in
    to_Poly (P_const 3 * y0 + P_const 3 * y0 * (G1 · P (y0)))
  end.

```

Informally, the interpretation of `nil` is the constant 3. The interpretation of `cons` is the function that sends $y_1 : \mathbb{N}$ to $3 + 2y_1$, and `map` is interpreted as the function that sends $y_0 : \mathbb{N}$ and $G_1 : \mathbb{N} \Rightarrow \mathbb{N}$ to $3y_0 + 3y_0G_1(y_0)$.

7.4.2 Polynomial Interpretation

Using polynomials, we deduce strong normalization under certain circumstances using Theorem 7.3.12. Suppose that for all function symbols f we have a polynomial $J : \text{poly} \bullet (\text{arity } X \ f)$, and now we need to provide the interpretation for application. Following Fuhs and Kop [38], we use a general method to interpret application. We start by constructing a minimal element in the interpretation of every type.

Definition 7.4.3 (`min_el_ty`). For every simple type A we define a minimal element of $\llbracket A \rrbracket_{\text{ty}}$ as follows

```
Fixpoint min_el_ty (A : ty B) : minimal_element  $\llbracket A \rrbracket_{\text{ty}}$ 
  := match A with
    | Base _  $\Rightarrow$  nat_minimal_element
    | A1  $\longrightarrow$  A2  $\Rightarrow$  min_el_fun_space (min_el_ty A2)
  end.
```

Here `nat_minimal_element` is defined to be 0, and `min_el_fun_space (min_el_ty A2)` is the constant function on $(\text{min_el_ty } A2)$.

In order to define the semantics of application, we need several operations involving $\llbracket A \rrbracket_{\text{ty}}$. First, we consider *lower value functions*.

Definition 7.4.4 (`lvf`). We define the **lower value function** as follows

```
Fixpoint lvf {A : ty B} :  $\llbracket A \rrbracket_{\text{ty}} \rightarrow_{\text{wm}} \text{nat\_CompatRel} :=
  match A with
    | Base _  $\Rightarrow$  id_wm
    | A1  $\longrightarrow$  A2  $\Rightarrow$  lvf  $\circ_{\text{wm}}$  apply_el_wm (min_el_ty A1)
  end.$ 
```

Note that we construct `lvf` directly as a weakly monotonic function. In addition, we reuse the combinators defined in Definition 7.3.7. As such, we do not need to prove separately that this function is monotonic.

In Fuhs and Kop [38], this definition is written down in a different, but equivalent, way. Instead of defining lvf_σ recursively, they look at full applications, which would be more complicated in our setting. More specifically, since we are working with simple types, we must have that $\sigma = \sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \tau$. Then they define $\text{lvf}_\sigma(f) := f(\perp_{\sigma_1}, \dots, \perp_{\sigma_n})$, where \perp_σ is the minimum element of $\llbracket \sigma \rrbracket$, the interpretation of σ . Next, we define two addition operations on $\llbracket A \rrbracket_{\text{ty}}$.

Definition 7.4.5 (`plus_ty_nat`). Addition of natural numbers and elements on $\llbracket A \rrbracket_{\text{ty}}$ is defined as follows

```

Fixpoint plus_ty_nat {A : ty B} : [[ A ]]ty * nat_CompatRel →wm [[ A ]]ty
:= match A with
  | Base _ ⇒ plus_wm
  | A1 → A2 ⇒
  let f := fst_wm ◦wm snd_wm in
  let x := fst_wm in
  let n := snd_wm ◦wm snd_wm in
  λwm (plus_ty_nat ◦wm ⟨f ·wm x , n ⟩)
end.

```

The function `plus_ty_nat` allows us to add arbitrary natural numbers to elements of the interpretation of types. Note that there are two cases in Definition 7.4.5. First of all, the type `A` could be a base type. In that case, we are adding two natural numbers, and we use the usual addition operation. In the second case, we are working with a functional type `A1 → A2`. The resulting function is defined using pointwise addition with the relevant natural number. Now we have everything in place to define the interpretation of application.

Definition 7.4.6 (`p_app`). Application is interpreted as the following function

```

Definition p_app {A1 A2 : ty B}
: [[ A1 → A2 ]]ty * [[ A1 ]]ty →wm [[ A2 ]]ty
:= let f := fst_wm in
  let x := snd_wm in
  plus_ty_nat ◦wm ⟨f ·wm x , lvf ◦wm x ⟩.

```

If both `A1` and `A2` are base types, then `p_app (f , x)` reduces to `f x + x`. Note that `p_app` satisfies the requirements from Theorem 7.3.12. Hence, we obtain the following.

Theorem 7.4.7 (`poly_Interpretation`). Let \mathbb{R} be an AFS. Suppose that for every function symbol `f` we have a polynomial `p_fun_sym f` such that for all rewrite rules $l \rightsquigarrow r$ in \mathbb{R} we have $\text{semTm } l \ x > \text{semTm } r \ x$ for all x . Then \mathbb{R} has a termination model.

7.4.3 Constraint Solving Tactic

Notice that in order to formally verify a proof of termination of a system using Theorem 7.4.7, we need to provide a polynomial interpretation and show that $[[\ell]] > [[r]]$ holds for all rules $\ell \rightarrow r$. This will introduce inequality proof-goals into the Coq context that must be solved. Let us consider a concrete example.

Example 7.4.8. We use the polynomials given in Example 7.4.2 to show strong normalization of Example 7.2.7. This example introduces two inequalities, one for each rule. Let $G_0 : \mathbb{N} \Rightarrow \mathbb{N}$ be weakly monotonic. For rule `map_ni1`, we need to prove

that for all G_0 , the constraint $12 + G_0(0) + 9G_0(3) > 3$ holds. For the second rule, `map_cons`, the constraint is: $12 + 4y_0 + 12y_1 + G_0(0) + (3y_0 + 9y_1 + 9)G_0(3 + y_0 + 3y_1) > 3 + y_0 + 12y_1 + 3G_0(0) + G_0(y_0) + 9y_1G_0(y_1)$, for all $y_0, y_1 \in \mathbb{N}$ and G_0 .

Manually finding witnesses for such inequalities is tedious, and we would like to automate this task. For that reason, we developed a tactic (`solve_poly`) that automatically solves the inequalities coming from Theorem 7.4.7. Essentially, this tactic tries to mimic how one would solve those goals in a pen-and-paper proof, and the same method is used by Wanda.

Example 7.4.9. We show how to solve the constraint arising from `map_cons` mentioned in Example 7.4.8. The first step is to find matching terms on both sides of the inequality and subtract them. In our example, $3 + y_0 + 12y_1 + G_0(0)$ occurs on both sides, and after subtraction, we obtain the following constraint:

$$9 + 3y_0 + 9y_1 + (3y_0 + 9y_1 + 9)G_0(3 + y_0 + 3y_1) > 2G_0(0) + G_0(y_0) + 9y_1G_0(y_1).$$

The second step is combining the arguments for the higher-order variable G_0 using its monotonicity. Note that each of 0 , y_0 , and y_1 is less than or equal to $3 + y_0 + 3y_1$, because they are natural numbers. Since G_0 is weakly monotonic, we get

$$2G_0(0) + G_0(y_0) + 9y_1G_0(y_1) \leq (9y_1 + 3)G_0(3 + y_0 + 3y_1).$$

Now we can simplify our original constraint to

$$9 + 3y_0 + 9y_1 + (3y_0 + 9y_1 + 9)G_0(3 + y_0 + 3y_1) > (9y_1 + 3)G_0(3 + y_0 + 3y_1).$$

Since $3y_0 + 9y_1 + 9 \geq 9y_1 + 3$, we have

$$(3y_0 + 9y_1 + 9)G_0(3 + y_0 + 3y_1) \geq (9y_1 + 3)G_0(3 + y_0 + 3y_1).$$

This is sufficient to conclude that the constraints for `map_cons` are satisfied.

The tactic `solve_poly` (`solve_poly`) follows the steps described above. Note that we use the tactic `nia`, which is a tactic in Coq that can solve inequalities and equations in nonlinear integer arithmetic. More specifically, `solve_poly` works as follows:

- First, we generate a goal for every rewrite rule, and we destruct the assumptions so that each variable in the context is either a natural number or a function.
- For every variable f that has a function type, we look for pair (x, y) such that $f(x)$ on the left hand side and $f(y)$ occurs on the right-hand side. We try using `nia` whether we can prove $x < y$ from our assumptions. If so, we add $x < y$ to the assumptions, and otherwise, we continue.

- The resulting goals with the extra assumptions are solved using `nia`.

Note that `solve_poly` is not complete, because `nia` is incomplete. As such, if a proof using this tactic is accepted by Coq, then that proof is correct. However, if the proof is not accepted, then it does not have to be the case that the proof is false. With the material discussed in this section, we can write down the polynomials given in Example 7.4.2, and the tactic is able to verify strong normalization.

7.5 Generating Proof Scripts

In this section, we discuss the practical aspects of our verification framework. In principle one can manually encode rewrite systems as Coq files and use the formalization we provide to verify their own termination proofs. However, this is cumbersome to do. Indeed, in Example 7.2.7 we used abbreviations to make the formal description of \mathbb{R}_{map} more readable. A rewrite system with many more rules would be difficult to encode manually. Additionally, to formally establish termination we also need to encode proofs. We did this in Example 7.4.2. The full formal encoding of \mathbb{R}_{map} and its termination proof is found in the file `Map.v`.

7.5.1 Proof traces for polynomial interpretation

This difficulty of manual encoding motivates the usage of proof traces. A proof trace is a human-friendly encoding of a TRS and the essential information needed to reconstruct the termination proof as a Coq script. Let us again consider \mathbb{R}_{map} as an example. The proof trace for this system starts with YES to signal that we have a termination proof for it. Then we have a list encoding the signature and the rules of the system.

YES

Signature: [

```
  cons : a -> list -> list ;
  map  : list -> (a -> a) -> list ;
  nil  : list
```

]

Rules: [

```
  map nil F => nil ;
  map (cons X Y) G => cons (G X) (map Y G)
```

]

Notice that the free variables in the rules do not need to be declared nor their typing information provided. Coq can infer this information automatically. The last section of the proof trace describes the interpretation of each function symbol in the signature.

```

Interpretation: [
  J(cons) = Lam[y0;y1].3 + 2*y1;
  J(map)  = Lam[y0;G1].3*y0 + 3*y0 * G1(y0);
  J(nil)  = 3
]

```

We can fully reconstruct a formal proof of termination for \mathbb{R}_{map} , which uses Theorem 7.4.7, with the information provided in the proof trace above. The full description of proof traces can be found at <https://github.com/deividrvale/nijn-coq-script-generation>, the API for ONijn. Proof traces are not Coq files. So we need to further compile them into a proper Coq script. The schematics in Figure 7.1 describe the steps necessary for it. We use ONijn to compile proof traces to Coq script. It is invoked as follows:

```
onijn path/to/proof/trace.onijn -o path/to/proof/script.v
```

Here, the first argument is the file path to a proof trace file and the `-o` option requires the file path to the resulting Coq script. The resulting Coq script can be verified by Nijn as follows:

```
coqc path/to/proof/script.v
```

Instructions on how to locally install ONijn/Nijn can be found in the following repository:

<https://github.com/deividrvale/nijn-coq-script-generation>

7.5.2 Verifying Wanda's Polynomial Interpretations

It is worth noticing that the termination prover is abstract in our certification framework. This means that we are not bound to a specific termination tool. So we can verify any termination tool that implements the interpretation method described here and can output proof traces in ONijn format.

Since Wanda [65] is a termination tool that implements the interpretation method in [38], it is our first candidate for verification. We added to Wanda the runtime argument `--formal` so it can output proof traces in ONijn format. In [65] one can find details on how to invoke Wanda. For instance, we illustrate below how to run Wanda on the map AFS.

```
./wanda.exe -d rem --formal Mixed_HO_10_map.afs
```

The setting `-d rem` sets Wanda to disable rule removal. The option `--formal` sets Wanda to only use polynomial interpretations and output proofs to ONijn proof traces. Running Wanda with these options gives us the proof trace we used for \mathbb{R}_{map} above. The latest version of Wanda, which includes this parameter, is found in the following repository: <https://github.com/hezzel/wanda>.

The table below describes our experimental evaluation on verifying Wanda’s output with the settings above. The benchmark set consists of those 46 TRSs for which Wanda outputs YES while using only polynomial interpretations and no rule removal. The time limit for certification of each system is set to 60 seconds.

Technique	Wanda			Nijn/ONijn		
	# YES	Pct.	Avg. Time	# Cert.	Perc.	Avg. Time
Poly, no rule removal	46	23%	0.07s	46	100%	4.06s

Table 7.1 Experimental Results

The experiment was run in a machine with M1 Pro 2021 processor with 16GB of RAM. Memory usage of Nijn during certification ranges from 400MB to 750MB. We provide the experimental benchmarks at the link below

<https://github.com/deividrvale/nijn-coq-script-generation>.

Hence, we can certify all TRSs proven strongly normalizing by Wanda while using only polynomial interpretations.

7.6 Conclusions and Future Work

We presented a formalization of the polynomial method in higher-order rewriting. This not only included the basic notions, such as algebraic functional systems, but also the interpretation method and the instantiation of this method to polynomials. In addition, we showed how to generate Coq scripts from the output of termination provers. This allowed us to certify their output and construct a formal proof of strong normalization. We also applied our tools to a concrete instance, namely to check the output of Wanda.

There are numerous ways to extend this work. First, one could formalize more techniques from higher-order rewriting, such as tuple interpretations [69] and dependency pairs [68, 74]. One could also integrate HORPO into our framework [71]. Second, in the current formalization, the interpretation of application is fixed for every instance of the polynomial method. One could also provide the user with the option to select their own interpretation. Third, currently, only Wanda is integrated with our work. This could be extended so that there is direct integration for other tools as well.

Chapter 8

Conclusions

The beauty — and oftentimes the root of all evil — in higher-order rewriting is that a lot of one’s expectations on how higher-order systems should behave are usually broken. This is especially true in the complexity analysis setting as even defining a complexity function for such systems was not initially obvious nor trivial. If abstractions are added to the play, β reductions introduce new complications not present in the first-order counterpart. Let us not even take note of the fact that accidentally encoding the untyped λ -calculus and “proving” its termination is commonplace for novices in the field. I do not emphasize this as a message of despair, however. Instead, it should be taken as a message of encouragement. Higher-order rewriting introduces a fairly simple framework to model higher-order computations, but it is complex enough to provide very interesting research opportunities. After all, higher-order functions are pervasive in both mathematics and computer science.

On The Interpretation Method. In this thesis, we focused mainly on providing a semantical treatment of higher-order systems (in the form of curried functional systems) with emphasis on their complexity-related properties. We developed a class of interpretations called tuple interpretations that allowed us to split complexity into two components: cost and size. This idea gave rise to a fairly robust complexity analysis framework capable of subsuming interpretation methods such as higher-order polynomial and matrix interpretations.

In Chapter 3, we focused on first-order systems, and consider both full and innermost evaluation strategies. We showed that tuple interpretations are amenable to implementation and provided a tool implementing a search algorithm for it, Hermes. In Chapter 4, we considered full rewriting and strongly monotonic tuple interpretations. In this setting β -reductions introduced an interesting problem: the expected interpretation of abstraction is not necessarily strongly monotonic. This phenomenon happens exactly when interpreting an abstraction term $\lambda x. s$ such that x does not occur free in s . Indeed, if abstraction is interpreted naively, we get $\llbracket \lambda x. s \rrbracket = d \mapsto \llbracket s \rrbracket_{[d:=x]\alpha}$ which is a constant

function. To deal with this case we introduced a `MakeSM` functional capable of taking the costs of such tuples into account and adding them back into the cost of the whole interpretation. In Chapter 5 we moved to a more “realistic” setting by considering the common evaluation strategy of weak call-by-value. Here, we introduced a notion of values and do not allow reductions to take place under a lambda. This development is the first step towards a rewriting-based complexity analysis framework that can capture call-by-value functional programs more naturally.

On Higher-Order Feasibility. On Chapter 6 we turned our attention to the problem of characterizing some notion of higher-order computational feasibility. This problem is interesting from both a theoretical and practical point of view. In the former, it shows that our interpretation method can be used to capture exactly BFF in a nicer language than that of oracle Turing machines. In the latter, this result opens the way to a rewriting-based analysis of the complexity of programs that would give meaningful complexity results.

On Certifying Termination Tools. Lastly, in Chapter 7, we focus on the certification problem for termination provers. We have provided `Nijn` which is a Coq library containing a formalization of higher-order rewriting theory. The focus here are polynomial interpretations. This opens the way for other formalizations based on semantics, like tuple interpretations. In this work, we not only formalize the theory but also provide a set of tooling so that `Nijn` can be used to certify the output of termination provers. That is where `ONijn` comes into play. Termination provers can describe their proof informally via a *proof trace*, and `ONijn` translates it to a Coq script, which can be checked by Coq.

On Future Work. The study of semantical properties of higher-order term rewriting systems is by no means complete. In fact, the work presented here can be extended in numerous ways. From a termination point of view, tuple interpretations were developed by Yamada [109] to show the termination of first-order TRSs. An obvious direction for further research is to adapt our tuple interpretations to show termination. This would mostly require tuples to be combined with other termination proof methods. Static Dependency Pairs would be my first attempt.

If one is interested in complexity, the first direction for future work would be a proper implementation of the theoretical development we introduce here. We could provide implementations for first-order complexity namely the `Hermes` tool presented in Chapter 3. In a higher-order setting, however, automation becomes more complicated. More research is needed to properly develop heuristics and better strategies to find useful interpretations automatically.

I believe that using tuple interpretations in a practical setting would require two main tasks to be completed. First, we must extend it to richer type systems. Simple types are “too simple” to model proper real-world programs. This seems like a promising direction and the first candidate would be types a la Milner [80]. By considering more expressive type systems, finding interpretations automatically becomes even more difficult and more research is needed for such extensions. The second problem is that tuple interpretations are not easily modularized. Further research in the direction of Kop [66] may provide an excellent starting point.

Finally, formalizing such theory is an exciting direction for future work. The base of such formalization is already in Nijn. One can have access to a term library, which by design formalizes the same curried functional systems we used in this thesis. Formalizations for termination properties as well. The formalization direction can focus on either complexity or termination. In both cases, one needs to formalize cost-size tuples, rewriting strategies (if call-by-value is the goal), and their respective compatibility theorems. With that comes the practical aspects like defining new notions of proof certificates and extending ONijn to support those.

References

- [1] Andreas Abel et al. “POPLMark reloaded: Mechanizing proofs by logical relations”. In: *J. Funct. Program.* 29 (2019), e19. doi: [10.1017/S0956796819000170](https://doi.org/10.1017/S0956796819000170).
- [2] Beniamino Accattoli and Ugo Dal Lago. “(Leftmost-Outermost) Beta Reduction is Invariant, Indeed”. In: *Log. Methods Comput. Sci.* 12.1 (2016). doi: [10.2168/LMCS-12\(1:4\)2016](https://doi.org/10.2168/LMCS-12(1:4)2016).
- [3] Ariane Alves Almeida and Mauricio Ayala-Rincón. “Formalizing the dependency pair criterion for innermost termination”. In: *Sci. Comput. Program.* 195 (2020), p. 102474. doi: [10.1016/j.scico.2020.102474](https://doi.org/10.1016/j.scico.2020.102474).
- [4] Sandra Alves, Delia Kesner, and Daniel Ventura. “A Quantitative Understanding of Pattern Matching”. In: *25th International Conference on Types for Proofs and Programs, TYPES 2019, June 11-14, 2019, Oslo, Norway*. Ed. by Marc Bezem and Assia Mahboubi. Vol. 175. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 3:1–3:36. doi: [10.4230/LIPIcs.TYPES.2019.3](https://doi.org/10.4230/LIPIcs.TYPES.2019.3).
- [5] Thomas Arts and Jürgen Giesl. “Termination of term rewriting using dependency pairs”. In: *Theor. Comput. Sci.* 236.1-2 (2000), pp. 133–178. doi: [10.1016/S0304-3975\(99\)00207-8](https://doi.org/10.1016/S0304-3975(99)00207-8).
- [6] Martin Avanzini and Ugo Dal Lago. “Automating sized-type inference for complexity analysis”. In: vol. 1. ICFP. 2017, 43:1–43:29. doi: [10.1145/3110287](https://doi.org/10.1145/3110287).
- [7] Martin Avanzini, Ugo Dal Lago, and Georg Moser. “Analysing the complexity of functional programs: higher-order meets first-order”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, 2015, pp. 152–164. doi: [10.1145/2784731.2784753](https://doi.org/10.1145/2784731.2784753).
- [8] Martin Avanzini and Georg Moser. “Complexity Analysis by Rewriting”. In: *Functional and Logic Programming, 9th International Symposium, FLOPS 2008, Ise, Japan, April 14-16, 2008. Proceedings*. Ed. by Jacques Garrigue and Manuel V. Hermenegildo. Vol. 4989. Lecture Notes in Computer Science. Springer, 2008, pp. 130–146. doi: [10.1007/978-3-540-78969-7_11](https://doi.org/10.1007/978-3-540-78969-7_11).
- [9] Martin Avanzini, Georg Moser, and Michael Schaper. “TcT: Tyrolean Complexity Tool”. In: *Proceedings of TACAS 2016 conference*. Ed. by Marsha Chechik and Jean-François Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 407–423. doi: [10.1007/978-3-662-49674-9_24](https://doi.org/10.1007/978-3-662-49674-9_24).
- [10] Mauricio Ayala-Rincón, Maribel Fernández, Daniele Nantes-Sobrinho, and Deivid Vale. “Nominal Equational Problems”. In: *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*. Ed. by Stefan Kiefer and Christine Tasson. Vol. 12650. Lecture Notes in Computer Science. Springer, 2021, pp. 22–41. doi: [10.1007/978-3-030-71995-1_2](https://doi.org/10.1007/978-3-030-71995-1_2).

- [11] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. DOI: [10.1017/CBO9781139172752](https://doi.org/10.1017/CBO9781139172752).
- [12] Patrick Baillot, Erika De Benedetti, and Simona Ronchi Della Rocca. “Characterizing polynomial and exponential complexity classes in elementary lambda-calculus”. In: *Inf. Comput.* 261 (2018), pp. 55–77. DOI: [10.1016/J.IC.2018.05.005](https://doi.org/10.1016/J.IC.2018.05.005).
- [13] Patrick Baillot and Ugo Dal Lago. “Higher-order interpretations and program complexity”. In: *Inf. Comput.* 248 (2016), pp. 56–81. DOI: [10.1016/j.ic.2015.12.008](https://doi.org/10.1016/j.ic.2015.12.008).
- [14] H. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013. DOI: [10.1017/CBO9781139032636](https://doi.org/10.1017/CBO9781139032636).
- [15] Hendrik Pieter Barendregt et al. “Term Graph Rewriting”. In: *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Eindhoven, The Netherlands, June 15-19, 1987, Proceedings*. Ed. by J. W. de Bakker, A. J. Nijman, and Philip C. Treleaven. Vol. 259. Lecture Notes in Computer Science. Springer, 1987, pp. 141–158. DOI: [10.1007/3-540-17945-3_8](https://doi.org/10.1007/3-540-17945-3_8).
- [16] Paul Beame et al. “The Relative Complexity of NP Search Problems”. In: *J. Comput. Syst. Sci.* 57.1 (1998), pp. 3–19. DOI: [10.1006/JCSS.1998.1575](https://doi.org/10.1006/JCSS.1998.1575).
- [17] Ralph Benzinger. “Automated higher-order complexity analysis”. In: *Theor. Comput. Sci.* 318.1-2 (2004), pp. 79–103. DOI: [10.1016/j.tcs.2003.10.022](https://doi.org/10.1016/j.tcs.2003.10.022).
- [18] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq’Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. DOI: [10.1007/978-3-662-07964-5](https://doi.org/10.1007/978-3-662-07964-5).
- [19] Frédéric Blanqui, Jean-Pierre Jouannaud, and Albert Rubio. “The computability path ordering”. In: *Log. Methods Comput. Sci.* 11.4 (2015). DOI: [10.2168/LMCS-11\(4:3\)2015](https://doi.org/10.2168/LMCS-11(4:3)2015).
- [20] Frédéric Blanqui and Adam Koprowski. “CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates”. In: *Math. Struct. Comput. Sci.* 21.4 (2011), pp. 827–859. DOI: [10.1017/S0960129511000120](https://doi.org/10.1017/S0960129511000120).
- [21] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. “Algorithms with polynomial interpretation termination proof”. In: *J. Funct. Program.* 11.1 (2001), pp. 33–53. DOI: [10.1017/s0956796800003877](https://doi.org/10.1017/s0956796800003877).
- [22] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. “On Lexicographic Termination Ordering with Space Bound Certifications”. In: *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk, Russia, July 2-6, 2001, Revised Papers*. Ed. by Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin. Vol. 2244. Lecture Notes in Computer Science. Springer, 2001, pp. 482–493. DOI: [10.1007/3-540-45575-2_46](https://doi.org/10.1007/3-540-45575-2_46).
- [23] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. “End-to-end verification of stack-space bounds for C programs”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom - June 09 - 11, 2014*. Ed. by Michael F. P. O’Boyle and Keshav Pingali. ACM, 2014, pp. 270–281. DOI: [10.1145/2594291.2594301](https://doi.org/10.1145/2594291.2594301).
- [24] Ahlem Ben Cherifa and Pierre Lescanne. “Termination of Rewriting Systems by Polynomial Interpretations and Its Implementation”. In: *Sci. Comput. Program.* 9.2 (1987), pp. 137–159. DOI: [10.1016/0167-6423\(87\)90030-X](https://doi.org/10.1016/0167-6423(87)90030-X).

- [25] Ezgi Çiçek, Deepak Garg, and Umut A. Acar. “Refinement Types for Incremental Computational Complexity”. In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer, 2015, pp. 406–431. doi: [10.1007/978-3-662-46669-8_17](https://doi.org/10.1007/978-3-662-46669-8_17).
- [26] Adam Cichon and Pierre Lescanne. “Polynomial Interpretations and the Complexity of Algorithms”. In: *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*. Ed. by Deepak Kapur. Vol. 607. Lecture Notes in Computer Science. Springer, 1992, pp. 139–147. doi: [10.1007/3-540-55602-8_161](https://doi.org/10.1007/3-540-55602-8_161).
- [27] Alan Cobham. “The Intrinsic Computational Difficulty of Functions”. In: *Logic, Methodology and Philosophy of Science: Proceedings of the 1964 International Congress (Studies in Logic and the Foundations of Mathematics)*. Ed. by Yehoshua Bar-Hillel. North-Holland Publishing, 1965, pp. 24–30.
- [28] Michael Codish et al. “SAT-based termination analysis using monotonicity constraints over the integers”. In: *Theory Pract. Log. Program.* 11.4-5 (2011), pp. 503–520. doi: [10.1017/S1471068411000147](https://doi.org/10.1017/S1471068411000147).
- [29] Robert L. Constable. “Type Two Computational Complexity”. In: *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*. Ed. by Alfred V. Aho et al. ACM, 1973, pp. 108–121. doi: [10.1145/800125.804041](https://doi.org/10.1145/800125.804041).
- [30] Evelyne Contejean, Claude Marché, Ana Paula Tomás, and Xavier Urbain. “Mechanically Proving Termination Using Polynomial Interpretations”. In: *J. Autom. Reason.* 34.4 (2005), pp. 325–363. doi: [10.1007/s10817-005-9022-x](https://doi.org/10.1007/s10817-005-9022-x).
- [31] Evelyne Contejean et al. “Automated Certified Proofs with CiME3”. In: *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*. Ed. by Manfred Schmidt-Schauß. Vol. 10. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011, pp. 21–30. doi: [10.4230/LIPIcs.RTA.2011.21](https://doi.org/10.4230/LIPIcs.RTA.2011.21).
- [32] Pierre Courtieu, Gladys Gbedoand, and Olivier Pons. “Improved Matrix Interpretation”. In: *SOFSEM 2010: Theory and Practice of Computer Science*. Ed. by Jan van Leeuwen et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 283–295. doi: [10.1007/978-3-642-11266-9_24](https://doi.org/10.1007/978-3-642-11266-9_24).
- [33] Ugo Dal Lago and Martin Hofmann. “Realizability models and implicit complexity”. In: *Theor. Comput. Sci.* 412.20 (2011), pp. 2029–2047. doi: [10.1016/J.TCS.2010.12.025](https://doi.org/10.1016/J.TCS.2010.12.025).
- [34] Norman Danner, Daniel R. Licata, and Ramyaa. “Denotational cost semantics for functional languages with inductive types”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, 2015, pp. 140–151. doi: [10.1145/2784731.2784749](https://doi.org/10.1145/2784731.2784749).
- [35] Norman Danner and James S. Royer. “Adventures in time and space”. In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. Ed. by J. Gregory Morrisett and Simon L. Peyton Jones. ACM, 2006, pp. 168–179. doi: [10.1145/1111037.1111053](https://doi.org/10.1145/1111037.1111053).

- [36] Ankush Das et al. “Resource-Aware Session Types for Digital Contracts”. In: *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 2021, pp. 1–16. doi: [10.1109/CSF51468.2021.00004](https://doi.org/10.1109/CSF51468.2021.00004).
- [37] Jörg Endrullis, Johannes Waldmann, and Hans Zantema. “Matrix Interpretations for Proving Termination of Term Rewriting”. In: *J. Autom. Reason.* 40.2-3 (2008), pp. 195–220. doi: [10.1007/s10817-007-9087-9](https://doi.org/10.1007/s10817-007-9087-9).
- [38] Carsten Fuhs and Cynthia Kop. “Polynomial Interpretations for Higher-Order Rewriting”. In: *23rd International Conference on Rewriting Techniques and Applications (RTA’12), RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*. Ed. by Ashish Tiwari. Vol. 15. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012, pp. 176–192. doi: [10.4230/LIPIcs.RTA.2012.176](https://doi.org/10.4230/LIPIcs.RTA.2012.176).
- [39] Carsten Fuhs and Cynthia Kop. “A Static Higher-Order Dependency Pair Framework”. In: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. Ed. by Luís Caires. Vol. 11423. Lecture Notes in Computer Science. Springer, 2019, pp. 752–782. doi: [10.1007/978-3-030-17184-1_27](https://doi.org/10.1007/978-3-030-17184-1_27).
- [40] Jürgen Giesl et al. “Analyzing Program Termination and Complexity Automatically with AProVE”. In: *J. Autom. Reason.* 58.1 (2017), pp. 3–31. doi: [10.1007/s10817-016-9388-y](https://doi.org/10.1007/s10817-016-9388-y).
- [41] Jürgen Giesl et al. “The Termination and Complexity Competition”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*. Ed. by Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen. Vol. 11429. Lecture Notes in Computer Science. Springer, 2019, pp. 156–166. doi: [10.1007/978-3-030-17502-3_10](https://doi.org/10.1007/978-3-030-17502-3_10).
- [42] Liye Guo and Deivid Vale. “Analyzing Innermost Runtime Complexity Through Tuple Interpretations”. In: *Proceedings 17th International Workshop on Logical and Semantic Frameworks with Applications, LSFA 2022, Belo Horizonte, Brazil (hybrid), 23-24 September 2022*. Ed. by Daniele Nantes-Sobrinho and Pascal Fontaine. Vol. 376. EPTCS. 2022, pp. 34–48. doi: [10.4204/EPTCS.376.5](https://doi.org/10.4204/EPTCS.376.5).
- [43] Raúl Gutiérrez and Salvador Lucas. “mu-term: Verify Termination Properties Automatically (System Description)”. In: *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Vol. 12167. Lecture Notes in Computer Science. Springer, 2020, pp. 436–447. doi: [10.1007/978-3-030-51054-1_28](https://doi.org/10.1007/978-3-030-51054-1_28).
- [44] Emmanuel Hainry, Bruce M. Kapron, Jean-Yves Marion, and Romain Péchoux. “A tier-based typed programming language characterizing Feasible Functionals”. In: *LICS ’20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*. Ed. by Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller. ACM, 2020, pp. 535–549. doi: [10.1145/3373718.3394768](https://doi.org/10.1145/3373718.3394768).
- [45] Emmanuel Hainry, Bruce M. Kapron, Jean-Yves Marion, and Romain Péchoux. “Complete and tractable machine-independent characterizations of second-order polytime”. In: *Foundations of Software Science and Computation Structures - 25th International Conference, FOSSACS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*. Ed. by Patricia Bouyer and Lutz Schröder. Vol. 13242. Lecture Notes

- in Computer Science. Springer, 2022, pp. 368–388. doi: [10.1007/978-3-030-99253-8_19](https://doi.org/10.1007/978-3-030-99253-8_19).
- [46] Emmanuel Hainry and Romain Péchoux. “Theory of higher order interpretations and application to Basic Feasible Functions”. In: *Log. Methods Comput. Sci.* 16.4 (2020). doi: [10.23638/LMCS-16\(4:14\)2020](https://doi.org/10.23638/LMCS-16(4:14)2020).
- [47] Makoto Hamana. “Theory and Practice of Second-Order Rewriting: Foundation, Evolution, and SOL”. In: *Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16, 2020, Proceedings*. Ed. by Keisuke Nakano and Konstantinos Sagonas. Vol. 12073. Lecture Notes in Computer Science. Springer, 2020, pp. 3–9. doi: [10.1007/978-3-030-59025-3_1](https://doi.org/10.1007/978-3-030-59025-3_1).
- [48] Martin A. T. Handley, Niki Vazou, and Graham Hutton. “Liquidate your assets: reasoning about resource usage in liquid Haskell”. In: *Proc. ACM Program. Lang.* 4.POPL (2020), 24:1–24:27. doi: [10.1145/3371092](https://doi.org/10.1145/3371092).
- [49] Juris Hartmanis and Richard Edwin Stearns. “Automata-based computational complexity”. In: *Inf. Sci.* 1.2 (1969), pp. 173–184. doi: [10.1016/0020-0255\(69\)90014-0](https://doi.org/10.1016/0020-0255(69)90014-0).
- [50] Nao Hirokawa and Georg Moser. “Automated Complexity Analysis Based on the Dependency Pair Method”. In: *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*. Ed. by Alessandro Armando, Peter Baumgartner, and Gilles Dowek. Vol. 5195. Lecture Notes in Computer Science. Springer, 2008, pp. 364–379. doi: [10.1007/978-3-540-71070-7_32](https://doi.org/10.1007/978-3-540-71070-7_32).
- [51] Dieter Hofbauer. “Termination Proofs by Multiset Path Orderings Imply Primitive Recursive Derivation Lengths”. In: *Theor. Comput. Sci.* 105.1 (1992), pp. 129–140. doi: [10.1016/0304-3975\(92\)90289-R](https://doi.org/10.1016/0304-3975(92)90289-R).
- [52] Dieter Hofbauer. “Termination Proofs by Context-Dependent Interpretations”. In: *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*. Ed. by Aart Middeldorp. Vol. 2051. Lecture Notes in Computer Science. Springer, 2001, pp. 108–121. doi: [10.1007/3-540-45127-7_10](https://doi.org/10.1007/3-540-45127-7_10).
- [53] Dieter Hofbauer and Clemens Lautemann. “Termination Proofs and the Length of Derivations (Preliminary Version)”. In: *Rewriting Techniques and Applications, 3rd International Conference, RTA-89, Chapel Hill, North Carolina, USA, April 3-5, 1989, Proceedings*. Ed. by Nachum Dershowitz. Vol. 355. Lecture Notes in Computer Science. Springer, 1989, pp. 167–177. doi: [10.1007/3-540-51081-8_107](https://doi.org/10.1007/3-540-51081-8_107).
- [54] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. “Resource Aware ML”. In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 781–786. doi: [10.1007/978-3-642-31424-7_64](https://doi.org/10.1007/978-3-642-31424-7_64).
- [55] Martin Hofmann and Thomas Streicher. “The Groupoid Model Refutes Uniqueness of Identity Proofs”. In: *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS’94), Paris, France, July 4-7, 1994*. IEEE Computer Society, 1994, pp. 208–212. doi: [10.1109/LICS.1994.316071](https://doi.org/10.1109/LICS.1994.316071).
- [56] Robert J. Irwin, James S. Royer, and Bruce M. Kapron. “On characterizations of the basic feasible functionals (Part I)”. In: *J. Funct. Program.* 11.1 (2001), pp. 117–153. doi: [10.1017/s0956796800003841](https://doi.org/10.1017/s0956796800003841).

- [57] Robert J. Irwin, James S. Royer, and Bruce M. Kapron. “On characterizations of the basic feasible functionals (Part I)”. In: *J. Funct. Program.* 11.1 (2001), pp. 117–153. doi: [10.1017/s0956796800003841](https://doi.org/10.1017/s0956796800003841).
- [58] Jean-Pierre Jouannaud and Mitsuhiro Okada. “A Computation Model for Executable Higher-Order Algebraic Specification Languages”. In: *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*. IEEE Computer Society, 1991, pp. 350–361. doi: [10.1109/LICS.1991.151659](https://doi.org/10.1109/LICS.1991.151659).
- [59] David M. Kahn and Jan Hoffmann. “Exponential Automatic Amortized Resource Analysis”. In: *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Ed. by Jean Goubault-Larrecq and Barbara König. Vol. 12077. Lecture Notes in Computer Science. Springer, 2020, pp. 359–380. doi: [10.1007/978-3-030-45231-5_19](https://doi.org/10.1007/978-3-030-45231-5_19).
- [60] Bruce M. Kapron and Stephen A. Cook. “A New Characterization of Type-2 Feasibility”. In: *SIAM J. Comput.* 25.1 (1996), pp. 117–132. doi: [10.1137/S0097539794263452](https://doi.org/10.1137/S0097539794263452).
- [61] Bruce M. Kapron and Florian Steinberg. “Type-two polynomial-time and restricted lookahead”. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. Ed. by Anuj Dawar and Erich Grädel. ACM, 2018, pp. 579–588. doi: [10.1145/3209108.3209124](https://doi.org/10.1145/3209108.3209124).
- [62] Akitoshi Kawamura and Stephen A. Cook. “Complexity Theory for Operators in Analysis”. In: *ACM Trans. Comput. Theory* 4.2 (2012), 5:1–5:24. doi: [10.1145/2189778.2189780](https://doi.org/10.1145/2189778.2189780).
- [63] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. “Combinatory Reduction Systems: Introduction and Survey”. In: *Theor. Comput. Sci.* 121.1&2 (1993), pp. 279–308. doi: [10.1016/0304-3975\(93\)90091-7](https://doi.org/10.1016/0304-3975(93)90091-7).
- [64] Cynthia Kop. “Higher Order Termination: Automatable Techniques for Proving Termination of Higher-Order Term Rewriting Systems”. English. PhD thesis. Vrije Universiteit Amsterdam, 2012. URL: <https://hdl.handle.net/1871/39346>.
- [65] Cynthia Kop. “WANDA - a Higher Order Termination Tool (System Description)”. In: *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*. Ed. by Zena M. Ariola. Vol. 167. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 36:1–36:19. doi: [10.4230/LIPIcs.FSCD.2020.36](https://doi.org/10.4230/LIPIcs.FSCD.2020.36).
- [66] Cynthia Kop. “Cutting a Proof into Bite-Sized Chunks: Incrementally proving termination in higher-order term rewriting (Invited Talk)”. In: *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel*. Ed. by Amy P. Felty. Vol. 228. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 1:1–1:17. doi: [10.4230/LIPIcs.FSCD.2022.1](https://doi.org/10.4230/LIPIcs.FSCD.2022.1).
- [67] Cynthia Kop, Aart Middeldorp, and Thomas Sternagel. “Complexity of Conditional Term Rewriting”. In: *Log. Methods Comput. Sci.* 13.1 (2017). doi: [10.23638/LMCS-13\(1:6\)2017](https://doi.org/10.23638/LMCS-13(1:6)2017).
- [68] Cynthia Kop and Femke van Raamsdonk. “Dynamic Dependency Pairs for Algebraic Functional Systems”. In: *Log. Methods Comput. Sci.* 8.2 (2012). doi: [10.2168/LMCS-8\(2:10\)2012](https://doi.org/10.2168/LMCS-8(2:10)2012).

- [69] Cynthia Kop and Deivid Vale. “Tuple Interpretations for Higher-Order Complexity”. In: *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*. Ed. by Naoki Kobayashi. Vol. 195. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 31:1–31:22. doi: [10.4230/LIPIcs.FSCD.2021.31](https://doi.org/10.4230/LIPIcs.FSCD.2021.31).
- [70] Cynthia Kop and Deivid Vale. “Cost-Size Semantics for Call-By-Value Higher-Order Rewriting”. In: *8th International Conference on Formal Structures for Computation and Deduction, FSCD 2023, July 3-6, 2023, Rome, Italy*. Ed. by Marco Gaboardi and Femke van Raamsdonk. Vol. 260. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 15:1–15:19. doi: [10.4230/LIPIcs.FSCD.2023.15](https://doi.org/10.4230/LIPIcs.FSCD.2023.15).
- [71] Adam Koprowski. “Coq formalization of the higher-order recursive path ordering”. In: *Appl. Algebra Eng. Commun. Comput.* 20.5-6 (2009), pp. 379–425. doi: [10.1007/s00200-009-0105-5](https://doi.org/10.1007/s00200-009-0105-5).
- [72] Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. “Tyrolean Termination Tool 2”. In: *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29 - July 1, 2009, Proceedings*. Ed. by Ralf Treinen. Vol. 5595. Lecture Notes in Computer Science. Springer, 2009, pp. 295–304. doi: [10.1007/978-3-642-02348-4_21](https://doi.org/10.1007/978-3-642-02348-4_21).
- [73] Keiichirou Kusakari. “On Proving Termination of Term Rewriting Systems with Higher-Order Variables”. In: *IPSJ Transactions on Programming* 42.SIG 7 (PRO 11) (2001), pp. 35–45. URL: <http://id.nii.ac.jp/1001/00016864/>.
- [74] Keiichirou Kusakari, Yasuo Isogai, Masahiko Sakai, and Frédéric Blanqui. “Static Dependency Pair Method Based on Strong Computability for Higher-Order Rewrite Systems”. In: *IEICE Trans. Inf. Syst.* 92-D.10 (2009), pp. 2007–2015. doi: [10.1587/transinf.E92.D.2007](https://doi.org/10.1587/transinf.E92.D.2007).
- [75] Keiichirou Kusakari and Masahiko Sakai. “Enhancing dependency pair method using strong computability in simply-typed term rewriting”. In: *Appl. Algebra Eng. Commun. Comput.* 18.5 (2007), pp. 407–431. doi: [10.1007/s00200-007-0046-9](https://doi.org/10.1007/s00200-007-0046-9).
- [76] Ugo Dal Lago and Marco Gaboardi. “Linear Dependent Types and Relative Completeness”. In: vol. 8. 4. 2011. doi: [10.2168/LMCS-8\(4:11\)2012](https://doi.org/10.2168/LMCS-8(4:11)2012).
- [77] Ugo Dal Lago and Simone Martini. “Derivational Complexity Is an Invariant Cost Model”. In: *Foundational and Practical Aspects of Resource Analysis - First International Workshop, FOPARA 2009, Eindhoven, The Netherlands, November 6, 2009, Revised Selected Papers*. Ed. by Marko C. J. D. van Eekelen and Olha Shkaravska. Vol. 6324. Lecture Notes in Computer Science. Springer, 2009, pp. 100–113. doi: [10.1007/978-3-642-15331-0_7](https://doi.org/10.1007/978-3-642-15331-0_7).
- [78] Daniel Leivant. “A Foundational Delineation of Computational Feasibility”. In: *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*. IEEE Computer Society, 1991, pp. 2–11. doi: [10.1109/LICS.1991.151625](https://doi.org/10.1109/LICS.1991.151625).
- [79] Kurt Mehlhorn. “Polynomial and Abstract Subrecursive Classes”. In: *J. Comput. Syst. Sci.* 12.2 (1976), pp. 147–178. doi: [10.1016/S0022-0000\(76\)80035-9](https://doi.org/10.1016/S0022-0000(76)80035-9).
- [80] Robin Milner. “A Theory of Type Polymorphism in Programming”. In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375. doi: [10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4). URL: [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4).

- [81] Fabian Mitterwallner and Aart Middeldorp. “Polynomial Termination Over \mathbb{N} Is Undecidable”. In: *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel*. Ed. by Amy P. Felty. Vol. 228. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 27:1–27:17. doi: [10.4230/LIPIcs.FSCD.2022.27](https://doi.org/10.4230/LIPIcs.FSCD.2022.27).
- [82] Georg Moser. “Derivational Complexity of Knuth-Bendix Orders Revisited”. In: *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*. Ed. by Miki Hermann and Andrei Voronkov. Vol. 4246. Lecture Notes in Computer Science. Springer, 2006, pp. 75–89. doi: [10.1007/11916277_6](https://doi.org/10.1007/11916277_6).
- [83] Georg Moser, Andreas Schnabl, and Johannes Waldmann. “Complexity Analysis of Term Rewriting Based on Matrix and Context Dependent Interpretations”. In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008, December 9-11, 2008, Bangalore, India*. Ed. by Ramesh Hariharan, Madhavan Mukund, and V. Vinay. Vol. 2. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2008, pp. 304–315. doi: [10.4230/LIPIcs.FSTTCS.2008.1762](https://doi.org/10.4230/LIPIcs.FSTTCS.2008.1762).
- [84] Georg Moser, Andreas Schnabl, and Johannes Waldmann. “Complexity Analysis of Term Rewriting Based on Matrix and Context Dependent Interpretations”. In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008, December 9-11, 2008, Bangalore, India*. Ed. by Ramesh Hariharan, Madhavan Mukund, and V. Vinay. Vol. 2. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2008, pp. 304–315. doi: [10.4230/LIPIcs.FSTTCS.2008.1762](https://doi.org/10.4230/LIPIcs.FSTTCS.2008.1762).
- [85] Friedrich Neurauter and Aart Middeldorp. “Revisiting Matrix Interpretations for Proving Termination of Term Rewriting”. In: *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*. Ed. by Manfred Schmidt-Schauß. Vol. 10. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011, pp. 251–266. doi: [10.4230/LIPIcs.RTA.2011.251](https://doi.org/10.4230/LIPIcs.RTA.2011.251).
- [86] Tobias Nipkow. “Higher-Order Critical Pairs”. In: *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*. IEEE Computer Society, 1991, pp. 342–349. doi: [10.1109/LICS.1991.151658](https://doi.org/10.1109/LICS.1991.151658).
- [87] Yue Niu and Jan Hoffmann. “Automatic Space Bound Analysis for Functional Programs with Garbage Collection”. In: *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*. Ed. by Gilles Barthe, Geoff Sutcliffe, and Margus Veanes. Vol. 57. EPiC Series in Computing. EasyChair, 2018, pp. 543–563. doi: [10.29007/xkwx](https://doi.org/10.29007/xkwx).
- [88] Lars Noschinski, Fabian Emmes, and Jürgen Giesl. “A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems”. In: *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wrocław, Poland, July 31 - August 5, 2011. Proceedings*. Ed. by Nikolaj S. Bjørner and Viorica Sofronie-Stokkermans. Vol. 6803. Lecture Notes in Computer Science. Springer, 2011, pp. 422–438. doi: [10.1007/978-3-642-22438-6_32](https://doi.org/10.1007/978-3-642-22438-6_32).
- [89] Lars Noschinski, Fabian Emmes, and Jürgen Giesl. “Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs”. In: *J. Autom. Reason.* 51.1 (2013), pp. 27–56. doi: [10.1007/s10817-013-9277-6](https://doi.org/10.1007/s10817-013-9277-6).

- [90] Isabel Oitavem. “Implicit Characterizations of Pspace”. In: *Proof Theory in Computer Science, International Seminar, PTCS 2001, Dagstuhl Castle, Germany, October 7-12, 2001, Proceedings*. Ed. by Reinhard Kahle, Peter Schroeder-Heister, and Robert F. Stärk. Vol. 2183. Lecture Notes in Computer Science. Springer, 2001, pp. 170–190. doi: [10.1007/3-540-45504-3_11](https://doi.org/10.1007/3-540-45504-3_11).
- [91] C.H. Papadimitriou. *Computational Complexity*. Theoretical computer science. Addison-Wesley, 1994.
- [92] J.C. van de Pol. “Termination of Higher-order Rewrite Systems”. PhD thesis. University of Utrecht, 1996. URL: <https://www.cs.au.dk/~jaco/papers/thesis.pdf>.
- [93] Vineet Rajani, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. “A unifying type-theory for higher-order (amortized) cost analysis”. In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–28. doi: [10.1145/3434308](https://doi.org/10.1145/3434308).
- [94] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. “Liquid types”. In: (2008). Ed. by Rajiv Gupta and Saman P. Amarasinghe, pp. 159–169. doi: [10.1145/1375581.1375602](https://doi.org/10.1145/1375581.1375602).
- [95] José-Luis Ruiz-Reina, José-Antonio Alonso, María-José Hidalgo, and Francisco-Jesús Martín-Mateos. “Formalizing Rewriting in the ACL2 Theorem Prover”. In: *Artificial Intelligence and Symbolic Computation, International Conference AISC 2000 Madrid, Spain, July 17-19, 2000, Revised Papers*. Ed. by John A. Campbell and Eugenio Roanes-Lozano. Vol. 1930. Lecture Notes in Computer Science. Springer, 2000, pp. 92–106. doi: [10.1007/3-540-44990-6_7](https://doi.org/10.1007/3-540-44990-6_7).
- [96] Moritz Sinn, Florian Zuleger, and Helmut Veith. “A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis”. In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 745–761. doi: [10.1007/978-3-319-08867-9_50](https://doi.org/10.1007/978-3-319-08867-9_50).
- [97] Bas Spitters and Eelis van der Weegen. “Type classes for mathematics in type theory”. In: *Math. Struct. Comput. Sci.* 21.4 (2011), pp. 795–825. doi: [10.1017/S0960129511000119](https://doi.org/10.1017/S0960129511000119).
- [98] Joachim Steinbach. “Proving Polynomials Positive”. In: *Foundations of Software Technology and Theoretical Computer Science, 12th Conference, New Delhi, India, December 18-20, 1992, Proceedings*. Ed. by R. K. Shyamasundar. Vol. 652. Lecture Notes in Computer Science. Springer, 1992, pp. 191–202. doi: [10.1007/3-540-56287-7_105](https://doi.org/10.1007/3-540-56287-7_105).
- [99] William W. Tait. “Intensional Interpretations of Functionals of Finite Type I”. In: *J. Symb. Log.* 32.2 (1967), pp. 198–212. doi: [10.2307/2271658](https://doi.org/10.2307/2271658).
- [100] Robert Endre Tarjan. “Amortized Computational Complexity”. In: *SIAM J. Algebraic Discrete Methods* 6.2 (Apr. 1985), pp. 306–318. doi: [10.1137/0606031](https://doi.org/10.1137/0606031).
- [101] René Thiemann and Christian Sternagel. “Certification of Termination Proofs Using CeTA”. In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 452–468. doi: [10.1007/978-3-642-03359-9_31](https://doi.org/10.1007/978-3-642-03359-9_31).

- [102] Yoshihito Toyama. “Counterexamples to Termination for the Direct Sum of Term Rewriting Systems”. In: *Inf. Process. Lett.* 25.3 (1987), pp. 141–143. doi: [10.1016/0020-0190\(87\)90122-0](https://doi.org/10.1016/0020-0190(87)90122-0).
- [103] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. “Abstract Refinement Types”. In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 209–228. doi: [10.1007/978-3-642-37036-6_13](https://doi.org/10.1007/978-3-642-37036-6_13).
- [104] Johannes Waldmann. “Matchbox: A Tool for Match-Bounded String Rewriting”. In: *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*. Ed. by Vincent van Oostrom. Vol. 3091. Lecture Notes in Computer Science. Springer, 2004, pp. 85–94. doi: [10.1007/978-3-540-25979-4_6](https://doi.org/10.1007/978-3-540-25979-4_6).
- [105] Peng Wang, Di Wang, and Adam Chlipala. “TiML: a functional language for practical complexity analysis with invariants”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 79:1–79:26. doi: [10.1145/3133903](https://doi.org/10.1145/3133903).
- [106] Niels van der Weide, Deivid Vale, and Cynthia Kop. “Certifying Higher-Order Polynomial Interpretations”. In: *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*. Ed. by Adam Naumowicz and René Thiemann. Vol. 268. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 30:1–30:20. doi: [10.4230/LIPIcs.ITP.2023.30](https://doi.org/10.4230/LIPIcs.ITP.2023.30).
- [107] Andreas Weiermann. “Termination Proofs for Term Rewriting Systems by Lexicographic Path Orderings Imply Multiply Recursive Derivation Lengths”. In: *Theor. Comput. Sci.* 139.1&2 (1995), pp. 355–362. doi: [10.1016/0304-3975\(94\)00135-6](https://doi.org/10.1016/0304-3975(94)00135-6).
- [108] Akihisa Yamada. “Multi-Dimensional Interpretations for Termination of Term Rewriting”. In: *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*. Ed. by André Platzer and Geoff Sutcliffe. Vol. 12699. Lecture Notes in Computer Science. Springer, 2021, pp. 273–290. doi: [10.1007/978-3-030-79876-5_16](https://doi.org/10.1007/978-3-030-79876-5_16).
- [109] Akihisa Yamada. “Tuple Interpretations for Termination of Term Rewriting”. In: *J. Autom. Reason.* 66.4 (2022), pp. 667–688. doi: [10.1007/s10817-022-09640-4](https://doi.org/10.1007/s10817-022-09640-4).
- [110] Akihisa Yamada, Keiichirou Kusakari, and Toshiaki Sakabe. “Nagoya Termination Tool”. In: *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Ed. by Gilles Dowek. Vol. 8560. Lecture Notes in Computer Science. Springer, 2014, pp. 466–475. doi: [10.1007/978-3-319-08918-8_32](https://doi.org/10.1007/978-3-319-08918-8_32).

Samenvatting

In 1965 stelde Alan Cobham de volgende breed geformuleerde vragen: “is het altijd moeilijker om te vermenigvuldigen dan om op te tellen?” en “waarom?” Het blijkt dat deze ogenschijnlijk onschuldige vragen geformuleerd in rekenkunde een diep verband hebben met algoritmes en hun intrinsieke complexiteit. Een algoritme is als een taartrecept. Het vertelt ons precies wat we moeten doen om een doel te bereiken: de taart. Met intrinsieke complexiteit bedoelen we dat het precieze recept van de taart niet uitmaakt voor de moeilijkheid ervan. Cobham stelt vervolgens een klasse van “eenvoudige” algoritmes vast: zulke algoritmes kunnen we uitvoeren in een klein genoeg aantal stappen. Deze verzameling staat tegenwoordig bekend als de klasse van algoritmes die in polynomiale tijd worden uitgevoerd. De hypothese van Cobham stelt dat we zulke algoritmes in de praktijk efficiënt kunnen uitvoeren.

Sindsdien zijn er talloze berekeningsmodellen voorgesteld en elk daarvan heeft een eigen notie van efficiëntie. Als voorbeeld hebben we termherschrijfsystemen, die gebruikt kunnen worden om polynomiaal berekenbare functies te beschrijven. Dit resultaat is interessant, omdat we algoritmes eenvoudiger kunnen uitdrukken in een declaratieve schrijfstijl dan met Turingmachines. Echter, tot nu toe zijn karakterisering van polynomiale tijd van schrijfsystemen beperkt tot eerste-orde systemen. Dit zijn systemen waarbij functies niet als argumenten mogen worden doorgegeven, maar daarmee zijn functies zoals map en fold uitgesloten.

Het doel van dit proefschrift is om deze tekortkoming op te vullen en om een raamwerk te ontwikkelen waarmee we de hogere-orde complexiteit kunnen analyseren met hogere-orde schrijfsystemen als berekeningsmodel. Om dit te bereiken, introduceren we een nieuwe klasse van hogere-orde algebraïsche semantiek: tupelinterpretaties. Het definiërende kenmerk van tupelinterpretaties is de splitsing van kosten en grootte bij het beschrijven van de complexiteit van hogere-orde systemen. Dit maakt een verfijnde studie van hun looptijd mogelijk door vaak nauwkeurige bovengrenzen te geven, waarbij de kosten en de grootte van de invoer de parameters zijn. Vervolgens passen we deze methode toe op een aantal hogere-orde systemen met verschillende operationele semantiek, d.w.z. volledig herschrijven (dus geen beperking op de evaluatie) en de call-by-value evaluatiestrategie.

Als belangrijkste resultaat van dit proefschrift geven we een karakterisering van de hogere-orde complexiteitsklasse BFF (Basic Feasible Functions) in termen van tweede-orde herschrijfsystemen met een semantische tupelinterpretatie die polynomiaal begrensd is. Tot slot, geven we een certificatie-engine Nijn/ONijn die in staat is om formeel terminatiebewijzen gegeven door hogere-orde terminatiegereedschappen te verifiëren.

Summary

In 1965 Alan Cobham proposed the following broadly formulated questions: “is it always more difficult to multiply than to add?” and “why?”. It turns out that these apparently innocent questions formulated in the context of basic arithmetical operations carry a deep connection with the notions of *algorithm* and their *intrinsic complexity*. An algorithm is like a cake recipe. It tells us precisely how to execute tasks to achieve an end: the cake. By intrinsic complexity, we mean that no matter how we choose to formulate the cake’s recipe there will be an intrinsic difficulty to it. Cobham then proceeds to establish a class of those algorithms that are “simple” in the sense that we can execute them in a small enough number of steps. This collection is today known as the class of algorithms computed in *polynomial time*. Cobham’s thesis states that such algorithms can be efficiently executed in practice.

Since then a myriad of computational models have been proposed and each of them came with their own notion of feasibility. An example of this consists of those term rewriting systems that can be used to capture functions computable in polynomial time. This result is interesting since expressing algorithms in a declarative rewriting style is often easier than explicitly defining them in terms of Turing machines. However, hitherto works in this area have been formulated in terms of first-order rewriting, i.e., systems where functions may not be passed as arguments. As such, one excludes commonly used higher-order functions like map and fold.

The goal of this thesis is to fill in this gap and develop a higher-order complexity analysis framework using higher-order rewriting systems as the underlying computational model. To this end, we introduce a new class of higher-order algebraic semantics: tuple interpretations. The defining characteristic of tuple interpretations is that we can completely split the dual notions of cost and size when measuring the complexity of such systems. This allows for a fine-grained study of their running time by providing often tight upper bounds parametrized by both the cost and the size of the arguments. We then proceed to apply this method in a variety of higher-order systems with different operational semantics, i.e., full rewriting (so no restriction on the evaluation) and the call-by-value evaluation strategy.

As a main result of this thesis, we provide a characterization of the higher-order complexity class BFF (Basic Feasible Functions) in terms of second-order rewriting

systems admitting a semantic tuple interpretation that is polynomially bounded. Finally, we provide a certification engine Nijn/ONijn capable of formally certifying termination proofs given by higher-order termination tools.

Research Data Management

This thesis research has been carried out under the research data management policy of the Institute for Computing and Information Science of Radboud University, The Netherlands.¹ The following research code repositories have been produced during this Ph.D. research:

- Chapter 3. The source code for Hermes is available under MIT license. Version 1.0.0 of Hermes comprises the work done for this thesis. It is permanently “frozen” and available on Zenodo with DOI:[10.5281/zenodo.8317571](https://doi.org/10.5281/zenodo.8317571).
- Chapter 7. This chapter produces two code bases. The first is for Nijn (v1.0.0), which is stored on Zenodo with DOI:[10.5281/zenodo.7913023](https://doi.org/10.5281/zenodo.7913023). The second is the repository for ONijn (v1.0.0), which is also available on Zenodo with DOI:[10.5281/zenodo.7915736](https://doi.org/10.5281/zenodo.7915736).

No data or code has been produced for the other chapters of the thesis.

¹<https://www.ru.nl/icis/research-data-management/>, accessed Wednesday 13th March, 2024.

Titles in the IPA Dissertation Series since 2021

- D. Frumin.** *Concurrent Separation Logics for Safety, Refinement, and Security.* Faculty of Science, Mathematics and Computer Science, RU. 2021-01
- A. Bentkamp.** *Superposition for Higher-Order Logic.* Faculty of Sciences, Department of Computer Science, VU. 2021-02
- P. Derakhshanfar.** *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03
- K. Aslam.** *Deriving Behavioral Specifications of Industrial Software Components.* Faculty of Mathematics and Computer Science, TU/e. 2021-04
- W. Silva Torres.** *Supporting Multi-Domain Model Management.* Faculty of Mathematics and Computer Science, TU/e. 2021-05
- A. Fedotov.** *Verification Techniques for xMAS.* Faculty of Mathematics and Computer Science, TU/e. 2022-01
- M.O. Mahmoud.** *GPU Enabled Automated Reasoning.* Faculty of Mathematics and Computer Science, TU/e. 2022-02
- M. Safari.** *Correct Optimized GPU Programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03
- M. Verano Merino.** *Engineering Language-Parametric End-User Programming Environments for DSLs.* Faculty of Mathematics and Computer Science, TU/e. 2022-04
- G.F.C. Dupont.** *Network Security Monitoring in Environments where Digital and Physical Safety are Critical.* Faculty of Mathematics and Computer Science, TU/e. 2022-05
- T.M. Soethout.** *Banking on Domain Knowledge for Faster Transactions.* Faculty of Mathematics and Computer Science, TU/e. 2022-06
- P. Vukmirović.** *Implementation of Higher-Order Superposition.* Faculty of Sciences, Department of Computer Science, VU. 2022-07
- J. Wagemaker.** *Extensions of (Concurrent) Kleene Algebra.* Faculty of Science, Mathematics and Computer Science, RU. 2022-08
- R. Janssen.** *Refinement and Partiality for Model-Based Testing.* Faculty of Science, Mathematics and Computer Science, RU. 2022-09
- M. Laveaux.** *Accelerated Verification of Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2022-10
- S. Kochanthara.** *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving.* Faculty of Mathematics and Computer Science, TU/e. 2023-01
- L.M. Ochoa Venegas.** *Break the Code? Breaking Changes and Their Impact on Software Evolution.* Faculty of Mathematics and Computer Science, TU/e. 2023-02
- N. Yang.** *Logs and models in engineering complex embedded production software systems.* Faculty of Mathematics and Computer Science, TU/e. 2023-03
- J. Cao.** *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN.*

Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2023-05

J. Smits. *Strategic Language Workbench Improvements*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

A. Arslanagić. *Minimal Structures for Program Analysis and Verification*. Faculty of Science and Engineering, RUG. 2023-07

M.S. Bouwman. *Supporting Railway Standardisation with Formal Verification*. Faculty of Mathematics and Computer Science, TU/e. 2023-08

S.A.M. Lathouwers. *Exploring Annotations for Deductive Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09

J.H. Stoel. *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software*. Faculty of Mathematics and Computer Science, TU/e. 2023-10

D.M. Groenewegen. *WebDSL: Linguistic Abstractions for Web Programming*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11

D.R. do Vale. *On Semantical Methods for Higher-Order Complexity Analysis*. Faculty of Science, Mathematics and Computer Science, RU. 2024-01

Curriculum Vitae



Deivid Rodrigues do Vale was born in 1992 in Unaí — Minas Gerais, Brazil. He graduated from high school in 2010 at the Escola Estadual Delvito Alves da Silva, also in Unaí. In 2011 Deivid completed his technical education *cum laude* in industrial informatics at the Federal Institute of Education, Science and Technology in Unaí. From 2009 to 2013, he worked as a teacher in such institutes teaching computer networks, Linux system administration, database administration, and programming language courses. During this same period, he worked as an information systems technician for a variety of companies in Brazil.

After this period in industry, in 2013, Deivid decided to start his academic life and moved to Brasília. In 2017, he graduated *cum laude* from the Bachelor of Science in Mathematics at the University of Brasília. Deivid then continues to the master's at the University of Brasília, which he graduated from in 2019. In September 2019 he started his Ph.D. at Radboud University in Nijmegen — The Netherlands, advised by Cynthia Kop. As of April 2024, Deivid works as a scientific programmer and post-doctoral researcher at Radboud University.

