Complexité implicite : bilan et perspectives

Romain Péchoux

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Complexité implicite : bilan et perspectives

(Implicit Computational Complexity: past and future)

mémoire présenté et soutenu en ligne le 19 octobre 2020

pour l'obtention d'une

Habilitation à Diriger des Recherches

(mention informatique, Section 27)

par

Romain Péchoux

Composition du jury

Président :	Olivier Bournez	PR, LIX, École Polytechnique
Rapporteurs :	Patrick Baillot	DR CNRS, LIP, ENS Lyon
	Ugo Dal Lago	PR, Università di Bologna
	Simona Ronchi Della Rocca	PR, Università di Torino
Examinateurs :	Emmanuel Jeandel	PR, LORIA, Université de Lorraine
	Delia Kesner	PR, IRIF, Université de Paris
	Jean-Yves Marion	PR, LORIA, Université de Lorraine

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Loria
Laboratoire lorrain de recherche en informatique et ses applications

Mis en page avec la classe thesul.

Remerciements

J'aimerais remercier de nombreuses personnes, membres de ma famille, amis et collègues. J'adresse donc des remerciements les plus chaleureux et sincères:

- à ma famille, pour son soutien, et en particulier, à mon épouse, Clémence, et à mes enfants, Diane et Valère, qui me permettent, par des biais divers, de garder les pieds sur terre (surtout en période de confinement);

- à Patrick Baillot, Olivier Bournez, Ugo Dal Lago, Emmanuel Jeandel, Delia Kesner, Jean-Yves Marion et Simona Ronchi Della Rocca pour avoir accepté de siéger dans ce jury d'habilitation et, en particulier, à Patrick Baillot, Ugo Dal Lago et Simona Ronchi Della Rocca pour avoir accepté de rapporter ce manuscrit ainsi qu'à Emmanuel Jeandel pour avoir accepté d'être mon parrain scientifique ;

- aux différents collègues que j'ai pu cotoyer depuis mon recrutement dans l'équipe Inria Carte puis Mocqua, Frédéric Dupuis, Nazim Fatès, Isabelle Gnaedig, Walid Gomaa, Emmanuel Hainry, Mathieu Hoyrup, Emmanuel Jeandel, Daniel Leivant, Simon Perdrix, Donald Stull et Vladimir Zamdzhiev. Les échanges avec ses différents membres sur des sujets scientifiques et aussi politiques, philosophiques, sportifs, sociétaux et culturels ont été, sont et seront toujours un réel plaisir. Je tiens à remercier tout particulièrement Emmanuel Hainry qui m'a aidé à corriger ce manuscrit à travers une relecture minutieuse ;

- aux anciens étudiants que j'ai encadrés en thèse ou dans le cadre de stages, Hugo Férée, Pierre Mercuriali, Thanh Dinh Ta et Olivier Zeyen. Résumés en deux mots, cela donne sérieux et persévérance ;

- à tous mes collègues du laboratoire Loria et du département 2, méthodes formelles;

- aux autres étudiants croisés dans l'équipe Carte puis dans l'équipe Mocqua, Philippe Beaucamps, Titouan Carette, Alexandre Clément, Henri De Boutray, Matthieu Kaczmarek, Daniel Reynaud-Plantey, Margarita Veshchezerova et Renaud Vilmart qui ont aussi grandement contribué aux échanges de l'équipe décrits ci-dessus ;

- à l'ensemble des membres de la communauté "complexité implicite" pour tous nos échanges fructueux lors de groupes de travail, réunions diverses, workshops et conférences.

  Je remercie Jean-Yves Marion, mon directeur de thèse, pour m'avoir initié à cette thématique.

  Je remercie Roberto Amadio, Patrick Baillot, Neil Jones, Simona Ronchi Della Rocca, contributeurs majeurs de ce domaine, ainsi que Claude Kirchner et Paul Zimmermann, qui étaient tous membres de mon jury de doctorat.

  Je remercie également les collègues du domaine avec lesquels j'ai effectué des travaux scientifiques, administratifs ou d'édition, en particulier, Martin Avanzini, Guillaume Bonfante, Marco Gaboardi, Emmanuel Hainry, Bruce Kapron, Damiano Mazza et Georg Moser.

  J'ai aussi une pensée amicale et chaleureuse pour tous les anciens étudiants passionnés de cette communauté (et de son adhérence mathématique) croisés lors d'événements communautaires et devenus désormais de jeunes actifs (Beniamino Accatoli, Clément Aubert, Flavien Breuvart, Anupam Das, Laure Daviaud, Paulin De Naurois, Romain Demangeon,

Hugo Férée, Stéphane Gimenez, Alexis Ghyselen, Charles Grellois, Giulio Guerrieri, Cynthia Kop, Antoine Madet, Giulio Manzonetto, Jean-Yves Moyen, Colin Riba, Thomas Rubiano, Michael Schaper, Thomas Seiller, Florian Steinberg, Lê Thành Dũng Nguyễn, Paolo Tranquilli, Pierre Vial, Florian Zuleger, ...).

Enfin, j'ai aussi une pensée toute particulière pour la famille et les proches de Martin Hofmann, figure historique de la complexité implicite, dont la disparition tragique a endeuillé notre communauté et bien au delà ;

- à mes coauteurs scientifiques dans le domaine de la théorie des clones et de la théorie de l'agrégation, Miguel Couceiro, Erkko Lehtonen et Abdallah Saffidine.

- à l'ensemble des services (assistantes d'équipe, accueil, services informatiques, cantine, ...) de support de recherche du laboratoire Loria que j'ai côtoyés ;

- à l'ensemble des collègues de ma composante d'enseignement, l'Institut des sciences du Digital: Management & Cognition (IDMC) et aux collègues des différentes composantes d'enseignements avec lesquels j'ai collaboré (IAE Nancy, FST, Télécom Nancy, CLSH, DEG, IGA Rabat).

  Je tiens particulièrement à remercier les collègues avec lesquels j'ai partagé des tâches administratives, Matthieu Barrandon, Isabelle Boni, Armelle Brun, Olivier Bruneau, Marco Dozzi, Pascal Fontaine, Olivier Perrin, Azim Roussanaly, Laurent Thomann et Laurent Vigneron, ainsi que les directeurs successifs de la composante IDMC (ex-UFR MI), Kamel Smaïli, Jeanine Souquières, Odile Thiery et, le directeur actuel, Antoine Tabbone, qui ont toujours su (ré)concilier science, exigence, recherche, enseignement et pédagogie. Je tiens à remercier tout particulièrement Jeanine et Pascal de m'avoir encouragé à écrire ce document.

  Je tiens également à remercier nos services administratifs et techniques, en particulier, Virginie Besse, Marie-Luce Boulet, Marie Granddidier, Pascale Malgras et Bernard Zanga.

  J'ai aussi une pensée particulière pour nos collègues disparus, Jean Malhomme et Hazel Everett.

- à tous mes autres collègues, étudiants et amis qui méritent d'être remerciés.

Contents

Foreword

This document attempts to survey the distinct Implicit Computational Complexity results obtained in the last three decades, focusing not only on the author's results and trying to be as exhaustive as possible by presenting the main advances obtained in the field. For this to be presented in a concise way, some arbitrary choices had been performed and some lines of work could have been put aside or not described in full details with all results.

This document is not introductory to Implicit Computational Complexity (ICC) and it is assumed that the reader is familiar with the following fields of theoretical computer science:

- *Computability theory.* General notions of decidability/undecidability, computability/uncomputability and computational models such as Turing Machines (TM) or Random Access Machines (RAM) will be used throughout the document. An introduction to these notions can be found in [Sav98].

- *Computational complexity.* This document discusses various characterizations of well-known complexity classes such as `Alogtime`, `NC`, (`F`)`P`, `NP`, (`F`)`PSPACE` and `EXPTIME`. [AB09] provides an interesting overview of the subject.

Some more technical and specific knowledge on the following programming languages/computational models and their semantics is required.

- *Function algebra* are used in Chapter 1, Chapter 3, and Chapter 4. Their definition and knowledge on existing characterizations based on such formalism will help the reader.

- *Lambda calculus* is used throughout the document. Basic knowledge about its syntax and semantics is required. Knowledge about the simply typed lambda calculus and System F will also be helpful.

- *Imperative programs* are used in Chapter 1 and Chapter 3. The language under study is an abstract and simple language. Some knowledge on processes and multi-threads will also be required in Chapter 3.

- *Term rewrite systems* are used throughout the document. We provide some very brief introduction to this formalism at the beginning of Section 2.2.

Some extra and specific knowledge about *object oriented programs* and specialized computational models such as the *π-calculus*, *quantum programs*, the *Blum-Shub and Smale model*, and *computable analysis* will be required in Chapter 3 and Chapter 4. For OO programs, basic knowledge in Java programming and about the formal semantics of such languages, *e.g.* FeatherweightJava, is enough to get a full understanding.

For each of these formalisms, we provide the corresponding sections and a complementary reading of interest in Table 1.

Programming language	1.2	1.3	1.4	2.1	2.2	2.3	2.4	3.1	3.2	3.3	4.1	4.2	4.3	4.4	Reference
Function algebra	✓	✓						✓						✓	[Clo99]
Lambda calculus	✓	✓	✓	✓					✓	✓	✓	✓	✓	✓	[B+84]
Imperative programs	✓	✓	✓					✓	✓						[Win93]
Term Rewrite Systems	✓	✓	✓		✓	✓	✓			✓		✓			[BN99, K+01]
Object Oriented programs								✓		✓					[Pie02]
π-calculus									✓						[SW03]
Quantum programs									✓						[Sel04]
Computable analysis												✓		✓	[Wei00]
Blum-Shub and Smale model														✓	[BSS88]

Table 1: Table of prerequisites on programming languages and computational models per section

Technique	1.2	1.3	1.4	2.1	2.2	2.3	2.4	3.1	3.2	3.3	4.1	4.2	4.3	4.4	Reference
Interpretations	✓	✓	✓		✓	✓	✓			✓	✓	✓			[BMM11, Péc13]
Type systems	✓	✓	✓	✓				✓	✓				✓	✓	[Pie02]
Linear logic	✓	✓		✓					✓				✓	✓	[Gir87]
Termination techniques	✓		✓		✓	✓	✓								[BN99, K+01, AG00]
Category theory									✓				✓		[Pie91]
Algebra and coalgebra													✓		[JR97]
Higher-order complexity												✓			[IRK01]
Complexity of real functions												✓		✓	[Ko91]

Table 2: Table of prerequisite on tools and techniques per section

Some specific knowledge on the following tools/techniques is also required.

- *Interpretations*, *type systems*, and *linear logic* are the main tools used throughout the document.
- *Termination* techniques are used in Chapter 1 and Chapter 2. Knowledge on Recursive Path Orders (RPO) and Dependency Pairs (DP) will help the reader in understanding the corresponding sections.
- Some basic on *category theory* will be used in Chapter 3 and Chapter 4. Categorical knowledge on the notions of *algebra and coalgebra* will also be helpful in Chapter 4.
- Some knowledge on higher-order complexity will be helpful in Chapter 4. Some basic notions are provided in Section 4.2.
- Some knowledge on the complexity of real functions, *i.e.* functions over $\mathbb{R}$, will also be required in Chapter 4.

For each of these tools/techniques, we provide the corresponding sections and a complementary reading of interest in Table 2.

Chapter 1

Introduction

Contents

1.1 Computational Complexity

1.1.1 Description

Computational complexity is the discipline of classifying functions depending on their inherent difficulty or cost. In this field, machines, more specifically Turing Machines (TM), are used as standard computational models for estimating the degree of difficulty of a function. Machines are compared by relating the cost (sometimes referred as Blum complexity measure [Blu67]) needed to produce an output *i.e.*, to reach a final state, to the input size (the number of non empty cells on the input tape at the beginning of the execution), for any possible input. Examples of such costs are the time, defined as the number of transitions required for the execution to complete,

or the space, defined as the maximal number of non empty memory cells at any time during the execution.[1]

The comparison between machines can be extended to functions. Given a function $t : \mathbb{N} \to \mathbb{N}$, a function f is computable with cost $t(n)$ if there exists a machine computing f with cost at most $t(n)$ for any input of size n.

Focusing on time cost, `FP` is defined to be the class of functions computable in polynomial time by a deterministic TM. A function f is in `FP` if there exist a deterministic machine M and a polynomial P such that M computes f at cost at most $P(n)$, for all input of size n. The class `FP` is commonly considered to be the class of tractable or feasible first order functions contrarily, for example, to the class of functions computable in exponential time.[2] Indeed, a function computable by an algorithm, whose runtime is equal to 2^n, would take more than 10^{22} years to complete on a computer with clock rate at most 1 GHz on an input of size 128 (*i.e.* only 16 bytes)!

If we restrict our study to decision problems, functions whose codomain is $\{0, 1\}$, then we can define in a similar way `P` and, respectively, `NP` to be the classes of decision problems decidable in polynomial time by a deterministic machine and by a non-deterministic machine, respectively. While `P` is the class of decision problems that can be solved in polynomial time, `NP` is the class of decision problems whose solution can be checked in polynomial time. However, because of non-determinism, the set of such solutions has a cardinality exponential in (a polynomial in) the input and, consequently, it is unclear whether the two classes coincide, thus leading to the well-known open issue $\texttt{P} \stackrel{?}{=} \texttt{NP}$.

Computational complexity has been deeply studied for more than half a century by considering distinct computational models, distinct cost models, distinct functions, and problems and trying to classify and compare them leading to a rich and deep, though uncomplete, understanding of difficulties and limits of computer science [AB09, Pap03, Sip06, Zoo]. The applications of computational complexity are of high interest as understanding and finding the physical limits of computations allows the programmer to avoid them or push them using several methods (approximation algorithms, distributed architectures, heuristics, ...) and highly motivates the researcher to try to develop more efficient technologies for tomorrow (quantum theory, distributed architectures, ...).

1.1.2 Strengths and weaknesses

Although computational complexity also studies alternative computational models (RAM, circuits, ...), its strength lies in the fact that, as in computability theory, TMs remain a robust computational model for a wide variety of complexity classes. Indeed, many classes can be expressed in terms of (variants of) TMs. One can think for example of Alternating Turing Machines (ATM) for characterizing circuit complexity classes [CS76] or Oracle Turing Machines (OTM) that make possible to tame second order polynomial time complexity [KC91].

This strength is emphasized by the *Invariance Thesis* of Van Emde Boas [Boa14] stating that *"there exists a standard class of machine models, including all variants of TM and variants of RAM, where models can simulate each other with a polynomial time bounded overhead and a constant factor space bounded overhead"*.

This robustness strength is also its major weakness. While computational complexity focuses on machines, modern programmers are mostly interested in high level programming languages. It

[1] The input tape is not taken into account for sublinear space classes.

[2] This identification of tractable functions with the complexity class `FP` is also known as Cobham-Edmonds thesis.

is well-known that the complexity of programs is not directly related to computational complexity as standard programmers mostly focus on asymptotic complexity rather than computational complexity. In such a framework, a simple `for` loop program guarded by the integer n will be considered to be linear in n while the corresponding machine simulating this loop using a binary representation will execute in exponential time $2^{|n|}$ in the input size, the size of the binary word representing the integer n being $|n| = \lceil \log_2(n) + 1 \rceil$.

Moreover, the *Invariance Thesis* does not hold for programs based on models like the lambda calculus or term rewriting systems as it is well-known that some reduction strategies can compute an output of exponential size in a linear number of reductions in the input size. Although the *Invariance Thesis* holds on variants of the lambda calculus based on the notion of explicit substitution and the notion of useful reduction [ADL14], it turns out that such models are in need of their own tools to be analyzed correctly.

Worst of all, computational complexity does not give a hint on how to design a program of a given complexity class. This weakness is tied to the explicit nature of the resource bound provided that does not give ways of building/certifying programs. This is particularly problematic for people interested by safety and security applications of computational complexity as it highlights the impossibility to develop automatic methods for certifying the complexity of such machines.

1.2 Implicit Computational Complexity

1.2.1 Description

The development of *Implicit Computational Complexity* (ICC) is concomitant to the will of solving all the aforementioned issues by finding static methods for analyzing automatically the complexity of "real" programming languages without explicitly referring to machines and without explicitly providing a resource bound.

ICC aims at defining criteria such that any program satisfying a given criterion will compute a function of a given complexity class. As the bound on resource consumption is implicit, the analysis is more suited to be applied in a static analysis perspective under the requirement that the criterion is not too hard to check from both a computability perspective and a complexity perspective: the programmer has no prior knowledge on the complexity of the code under analysis and wants to obtain some guarantees on it for a *safe* execution.

The aim of ICC can be summarized as follows. Given a programming language $\mathcal{L}$ and a complexity class $\mathcal{C}$, find a restriction $\mathcal{R} \subseteq \mathcal{L}$ such that the following equality holds:

$$\{[\![\mathtt{p}]\!] \mid \mathtt{p} \in \mathcal{R}\} = \mathcal{C},$$

where $[\![\mathtt{p}]\!]$ is the function computed by the program $\mathtt{p}$.

The above equality is *extensional*, *i.e.* it deals with sets of functions rather than programs and/or machines. The semantics $[\![-]\!]$ is often omitted when it is clear from the context, for example when $\mathtt{p}$ is a program from binary words to binary words.

The inclusion from left to right means that any function computed by a program of $\mathcal{L}$ satisfying the criterion $\mathcal{R}$ is in $\mathcal{C}$. This property is called *Soundness*. Conversely, for any function f in $\mathcal{C}$, there exists a program in $\mathtt{p} \in \mathcal{L}$ such that $f = [\![\mathtt{p}]\!]$ and $\mathtt{p} \in \mathcal{R}$ hold. This property is called (extensional) *Completeness*.

The considered language $\mathcal{L}$, complexity class $\mathcal{C}$, and criterion $\mathcal{R}$ are parameters of ICC characterizations that vary greatly from one work to another. As we will shortly see, the ICC literature consists of a tangle of such results where the language can range over programming languages

from functional to Object Oriented (OO), the complexity class can range from subpolynomial complexity to polynomial complexity and beyond, and where the criterion can, non exhaustively, be a type system, a syntactical restriction or a constraint based method.

1.2.2 Historical background

The paper [Cob65] is commonly accepted as the founding paper of ICC. In this paper, Cobham provides a characterization of polynomial time on function algebra that is formalized in [BC92] as follows.

Theorem 1.2.1 (Cobham's characterization of FP)**.** *The least class of functions containing the constant function $z = 0$, the projection functions $\pi^n_j(x_1, \ldots, x_n) = x_j$, the successor functions $s_i(x) = 2 \times x + i$, $i \in \{0, 1\}$, the smash function $\#(x, y) = 2^{|x|\times|y|}$ and closed under standard composition (COMP) and Bounded Recursion on Notation (BRN):*

$$\begin{aligned}
\mathtt{COMP}(f, \overline{g})(\overline{x}) &= f(\overline{g}(\overline{x})),\ ^{3}\\
\mathtt{BRN}(f, g, h_0, h_1)(0, \overline{x}) &= f(\overline{x}),\\
\mathtt{BRN}(f, g, h_0, h_1)(2y + i, \overline{x}) &= h_i(y, \overline{x}, \mathtt{BRN}(f, g, h_0, h_1)(y, \overline{x})),\ \textit{when } 2y + i \neq 0,\\
&\qquad \textit{provided that } \forall y, \overline{x},\ \mathtt{BRN}(f, g, h_0, h_1)(y, \overline{x}) \leq g(y, \overline{x}),
\end{aligned}$$

is exactly FP*.*

Turning back to previous definition of an ICC characterization, the programming language $\mathcal{L}$ under consideration would be a simple function algebra that can be seen as a first order equational programming language and the complexity class under consideration is FP. A program $p \in \mathcal{L}$ would pass the criterion $\mathcal{R}$, noted $p \in [z, \pi^n_j, s_i, \#; \mathtt{COMP}, \mathtt{BRN}]$, if it can be written from the base functions of Cobham's algebra $\{z, \pi^n_j, s_i, \#\}$ and using the COMP and BRN schemes as building blocks. Hence, Theorem 1.2.1 can be restated in a concise way as

$$\{[\![\mathtt{p}]\!] \mid \mathtt{p} \in [\mathtt{z}, \pi^{\mathtt{n}}_{\mathtt{j}}, \mathtt{s_i}, \#; \mathtt{COMP}, \mathtt{BRN}]\} = \mathtt{FP}.$$

The above characterization of FP proved by Rose in [Ros87] provides a way to build new functions inductively from functions that are already in the class using a combination of the COMP and BRN schemes and is considered to be the first ICC characterization of a complexity class because it is machine independent. However it suffers from requiring an extra resource bound $g(y, \overline{x})$, whose check is very difficult, and is then not purely implicit.

This important drawback has been tackled independently by Bellantoni and Cook [BC92] and Leivant and Marion [LM93, Lei95] who have characterized FP on function algebra using a restricted version of primitive recursion scheme called *safe recursion* and *ramified recurrence* (based on a *tiering* discipline), respectively. These works mostly restrict the way recursion is allowed and provide the first pure ICC characterizations of FP and, hence, can be seen as the seminal works of the ICC community. These works have also been extended to lambda calculus [LM93] and polynomial space [LM94] as well as calculi with higher types [BNS00, Hof00], since then, these lines of works have continued to be of great and growing interest to computer scientists willing to study program complexity.

Before having an overview of the distinct existing analyses, we will try to understand and find the common mechanisms underlying most ICC works. As mentioned above, the class of functions computable in polynomial time (FP) was the main targeted class. This was primarily due to the assumed tractability of its inhabitants by Cobham-Edmonds thesis. As every programmer knows

that an exponential can be encoded simply by iterating data duplication or copy, most of the studies were performed in the spirit of preventing such bad behavior from happening. It turned out that such a simple intuition led to very distinct developments that were mostly influenced by the programming language (or paradigm) on which they were implemented. We can classify the ICC works in the literature in a somewhat arbitrary manner in distinct schools of thought depending mostly of the computational paradigm and the tool used to perform the analysis:

- function algebra and safe recursion,
- lambda calculi and light logics,
- term rewrite systems and interpretations,
- imperative programs and dataflow / control flow methods.

1.2.3 Function algebra and safe recursion

The function algebra approach consists in fixing a finite sequence of initial functions $\mathcal{I}$, in fixing a finite sequence of operators (or recursion schemes) $\mathcal{O}$ mapping functions to functions, and in considering $[\mathcal{I};\mathcal{O}]$, the smallest set of functions containing all the initial functions of $\mathcal{I}$ and closed under the operators of $\mathcal{O}$. Such lines of works, for which a very detailed survey can be found in [Clo99], have provided various characterizations of well-known complexity classes, including, non-exhaustively, subpolynomial classes (`Alogtime` [CK93, Pit98], `NC` [Clo90, All91], `Logspace` [CT95]), polynomial time, `FP` [Cob65, Ros87], polynomial space, `PSPACE` [Clo97], and beyond (Grzegorgczyk's hierarchy [Sch69, Mül74]). Extensions to other complexity classes for counting problems have also been considered (*e.g.* `#P` [VW96]).

Most of the aforementioned characterizations are not *implicit* in the sense that they require the function computed by the closure of the recursion scheme operator to be bounded from above by a function of the class (as in Theorem 1.2.1). As mentioned previously, this problem was solved by Leivant and Marion [LM93] and Bellantoni and Cook [BC92]. We give below the formalism by Bellantoni and Cook where the input of a function is split between *normal* and *safe* data separated using a semicolon. In a function call $f(\overline{x};\overline{y})$, $\overline{x}$ is the normal data and $\overline{y}$ is the safe data.

Theorem 1.2.2 ([BC92])**.** *The least class of functions containing the constant zero-ary function* `0`*, the projection functions* $\pi_j^{n,m}(x_1,\ldots,x_n;x_{n+1},\ldots,x_{n+m}) = x_j$*,* $j \in [1,n+m]$*, the successor functions* $s_i(;x) = 2x+i$*,* $i \in \{0,1\}$*, the predecessor function* $p(;2x+i) = x$*, the conditional function* $C(;x,y,z) = y$*, if* x `mod` $2 = 0$*,* $C(;x,y,z) = z$ *otherwise, and closed under safe composition* (`SCOMP`)[4] *and Predicative Recursion on Notation* (`PRN`)*:*

$$\begin{aligned}
\texttt{SCOMP}(f,\overline{g},\overline{h})(\overline{x};\overline{y}) &= f(\overline{g}(\overline{x};);\overline{h}(\overline{x};\overline{y}))\\
\texttt{PRN}(f,h_0,h_1)(0,\overline{y};\overline{z}) &= f(\overline{y};\overline{z})\\
\texttt{PRN}(f,h_0,h_1)(2x+i,\overline{y};\overline{z}) &= h_i(x,\overline{y};\overline{z},\texttt{PRN}(f,h_0,h_1)(x,\overline{y};\overline{z}))\ \textit{when}\ 2x+i\neq 0
\end{aligned}$$

is exactly `FP`*. In other words,* $\{[\![\mathtt{p}]\!] \mid \mathtt{p} \in [\mathtt{0},\pi_{\mathtt{j}}^{\mathtt{n},\mathtt{m}},\mathtt{s_i},\mathtt{p},\mathtt{C};\mathtt{SCOMP},\mathtt{PRN}]\} = \mathtt{FP}$*.*

In [BC92], the characterization is restricted to functions with only normal data. This restriction is not necessary for our purpose as we have clearly distinguished the syntactical function from the object it computes.

[4] Here provided that $\overline{g} = g_1,\ldots,g_n$, $\overline{g}(\overline{x};\overline{y})$ stands for $g_1(\overline{x};\overline{y}),\ldots,g_n(\overline{x};\overline{y})$.

The data duplication is controlled very cleverly in this framework: any normal data guarding a recursion cannot be used more than linearly (in its size) during this recursion as every call consumes a bit of data. Some copies can be passed to the functions in the context (h_0 and h_1) but the context cannot perform a recursion on the value computed by a recursive call (as such values are safe data), hence preventing iteration of non-linear functions (including the ones duplicating the size of their argument). To illustrate this point, consider the following function:

$$\begin{aligned}\texttt{double}(0;\,) &= 0,\\ \texttt{double}(2x{+}i;\,) &= s_1(;\,s_1(;\,\texttt{double}(x;\,))),\\ \texttt{exp}(2x{+}i;\,) &= \texttt{double}(\texttt{exp}(x;\,);\,).\end{aligned}$$

The function `double` is in $[0, \pi_j^{n,m}, s_i, p, C; \texttt{SCOMP}, \texttt{PRN}]$ and thus computes a polynomial time function (associating $s_1(;\ldots s_1(;0))$, $2n$ times, to any input of size n). However `exp` is (hopefully) not in this class as the use of `double` as contextual function requires the $\texttt{exp}(x;\,)$ recursive call to be normal and this breaks the `PRN` scheme.

Predicative recursion and ramified recurrence have encountered a huge success in the community and have been altered in order to provide purely implicit characterizations of other complexity classes such as the subpolynomial classes `Alogtime` [Blo94, LM00, JdN19], $\texttt{NC}^k$ [BKMO08] and `NC` [Bel95, Lei98, JdN19], and polynomial space [Oit01]. A functional programming language based on safe recursion has also been introduced in [BCR09].

1.2.4 lambda calculus and light logics

Another very popular approach lies at the intersection of the Curry-Howard correspondence, between lambda calculus and intuitionistic logic, and the introduction of Linear Logic (LL) by Girard [Gir87]. Linear logic was introduced as a substructural logic for handling operations of duplication and erasure using new logical connectors ! and ?, called the exponentials, in both classical and intuitionistic settings.

In [GSS92], Girard, Scedrov, and Scott have introduced a variant of Linear logic, named Bounded Linear Logic (BLL), introducing an annotated modality $!_x A$ that can be translated as $1 \otimes A \otimes \ldots \otimes A$, with x occurrences of $\otimes$, ensuring that proof normalization can be performed in polynomial time and that any function in `FP` can be represented. Here variable duplication (also called contraction) is handled in a very natural way by the following principle $!_{x+y}(A \& B) \multimap !_x A \otimes !_y B$. This characterization is a cornerstone result that is not purely implicit since it suffers, as the initial function algebra studies, from the need of providing explicitly the polynomial in the type annotation. This drawback has been solved by two lines of work: Light Linear Logic (LLL) by Girard [Gir98] and Soft Linear Logic (SLL) by Lafont [Laf04].

- LLL and its affine variant, Light Affine Logic (LAL) [Asp98, AR02], provide implicit characterizations of `FP` (A proof of completeness of LAL has been provided in [Rov99]). The affine variant is obtained by adding full weakening: while a linear variable occurs exactly once in a term, an affine variable occurs at most once. The intuition of light logics is as follows: contraction is allowed for ! variables (*i.e.* variables that can be duplicated) but the price to pay for this is an annotation by a new modality §, called neutral. Hence duplication is allowed but cannot be iterated. The natural deduction system for LAL taken from [BT09] is shown in Figure 1.1.

  Following the encoding provided for System F, the Church representation of unary integers can be given the type $Nat = \forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$ in LAL. The unary *double* function

  $$\lambda n.\lambda f.\lambda x.(n\ f\ (n\ f\ x))$$

$$\frac{}{x : A \vdash x : A}\ (\texttt{Var})$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \multimap B}\ (\texttt{I}\multimap) \qquad \frac{\Gamma \vdash M : A \multimap B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash M\ N : B}\ (\texttt{E}\multimap)$$

$$\frac{\Gamma \vdash M : A}{\Gamma, \Delta \vdash M : A}\ (\text{Weak}) \qquad \frac{\Gamma, x : !A, y : !A \vdash M : B}{\Gamma, z : !A \vdash M[z/x, z/y] : B}\ (\text{Cntr})$$

$$\frac{\Gamma, \Delta \vdash M : A}{!\Gamma, \S\Delta \vdash M : \S A}\ (\texttt{I}\S) \qquad \frac{\Gamma \vdash N : \S A \quad \Delta, x : \S A \vdash M : B}{\Gamma, \Delta \vdash M[N/x] : B}\ (\texttt{E}\S)$$

$$\frac{x : B \vdash M : A}{!x : !B \vdash M : !A}\ (\texttt{I}!) \qquad \frac{\Gamma \vdash M : !A \quad \Delta, x : !A \vdash N : B}{\Gamma, \Delta \vdash M[N/x] : B}\ (\texttt{E}!)$$

$$\frac{\Gamma \vdash M : A \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash M : \forall \alpha.A}\ (\texttt{I}\forall) \qquad \frac{\Gamma \vdash M : \forall \alpha.A}{\Gamma \vdash M : A[B/\alpha]}\ (\texttt{E}\forall)$$

where typing contexts are assumed to be disjoint and $\dagger\Gamma$, for $\dagger \in \{!, \S\}$ and $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, stands for $x_1 : \dagger A_1, \ldots, x_n : \dagger A_n$.

Figure 1.1: Typing rules for LAL (version from [BT09])

can be typed by $!Nat \multimap \S Nat$. For any type A, the iterator

$$iter_A = \lambda f.\lambda x.\lambda n.(n\ f\ x)$$

can be given the type $!(A \multimap A) \multimap \S A \multimap Nat \multimap \S A$ and, consequently, *double* cannot be iterated for the reasons explained above.

It was shown in [Gir98] that the reduction of a LLL proof-net π to its normal form can be performed in $O((d+1)|\pi|^{2^{d+1}})$, provided that $|\pi|$ is its size and d its depth (number of nested modalities), using a reduction by levels. Consequently, the reduction can be done in polynomial time for term of fixed depth. This result does not hold for terms as demonstrated in [BT09] but was extended to any reduction strategy in [Ter01, BM10].

Theorem 1.2.3 (From [BT09]). *The class of functions representable by a typable term of LAL on binary words is exactly* FP.[5]

- SLL can be seen as a subsystem of BLL that is complete for polynomial time. The annotations of BLL are replaced by a more careful handling of variable duplication, called multiplexing, as contraction $!A \multimap (!A \otimes !A)$ and digging $!A \multimap !!A$ are not valid in such a system. Multiplexing corresponds to the validity of the following principle $!A \multimap (A \otimes \ldots \otimes A)$,

[5] Each binary word $w \in \{0,1\}^*$ is encoded by a term $\underline{w}$ of type $W = \forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$. A function f is represented by the term M such that $x : W \vdash M : \S^k W$ if the proof of $\vdash M[\underline{w}/x] : \S^k W$ reduces to $\vdash \underline{f(w)} : \S^k W$, for each w.

n times, for any $n \geq 0$. Here ! is a marker for variable duplication. The church numerals are encoded with type $\forall\alpha.!(\alpha \multimap \alpha) \multimap (\alpha \multimap \alpha)$ and the use of duplication is thus restricted as in the case of LLL. Consequently, the degree of the polynomial is equal to the depth (number of nested modalities) of the proof π of the term M plus one (*i.e.* $O(|M|^{d(\pi)+1})$).

Theorem 1.2.4 (Adapted from [Laf04])**.** *The class of predicates representable by a SLL proof on binary lists is exactly* `P`.[6]

The above result can be extended to terms and to the complexity class `FP` (see [BM04]).

LLL and SLL systems have been enriched to complexity classes beyond polynomial time such as elementary time in Elementary Linear Logic (ELL) [Gir98, DJ03] and elementary lambda calculus [BBRDR18] characterizing the elementary recursive functions (functions that can be computed by a TM in time bounded by an exponential tower in the input size). Affine variants such as Elementary Affine Logic (EAL) have also been considered in [BT05]. These systems have also been adapted to the complexity class `Logspace` on a BLL variant [Sch07], the complexity class `NP` for LLL [Mau03] and SLL [GMRDR08b], and the complexity class `PSPACE` for SLL [GMRDR08a].

It is important to mention that the type system for LAL does not enjoy subject reduction and polynomiality is not preserved under β-reduction. Indeed polynomiality is preserved for proof-nets, a circuit-based representation of proofs, but not for terms. This has led to some works on the subject, the most relevant of those being the introduction of Dual Light Affine Logic (DLAL) by Baillot and Terui [BT04] that substitutes the non-linear arrow $\Rightarrow$ to the exponential ! using the standard translation $(A \Rightarrow B)^* = !A^* \multimap B^*$ inspired by Barber and Plotkin's work on Dual Intuitionistic Linear Logic [BP96]. A related type system capturing the functions computable in logarithmic space has been studied in [DLS10, DLS16].

A comparison with Bellantoni and Cook function algebra (BC) has been tackled in [MO04] by Ong and Murawski. The two authors remark that *"safe variables in LAL are not contractible (i.e. not duplicable), though normal variables are"* and manage to embed BC in LAL by restricting BC to a scheme respecting safe variables non-contractibility. This restriction comes at the price of a loss of completeness that is recovered by allowing a restricted form of recursion scheme that can be encoded in LAL on safe variables. This highlights the complementarity of the two approaches as light logics bring polymorphism and higher-order types whereas function algebra provides a more flexible treatment of variable duplication.

There are also related approaches on functional languages. In [Hof99, Hof03], Hofmann develops a type system with linearity properties, called non-size-increasing, ensuring that all definable functions are in `FP`. This system improves upon previous systems based on predicativity or modality restrictions by ensuring that recursive definitions can be arbitrarily nested. The key intuition is to use a resource type that *"cannot be generated out of nothing"* and, consequently, ensuring that programs cannot increase the size of their input. Again the main intuition is that program cannot compute outputs whose size is the double of the size of their input. This system in its own is sound but uncomplete for `FP`: it characterizes the class of functions computable in polynomial time and in linear space. Consequently, the system has to be extended with some form of predicative recursion to recover completeness over `FP`. The expressive power of this system was more deeply studied in [Hof02].

A last interesting and related line of work on the use of constructors for controlling the complexity of functional programs is the one by Jones [Jon01] on the expressive power of functional

[6] Here we have chosen not to specify the precise encoding as it mostly differs depending on the fragment of SLL considered [Laf04, GMRDR08b, Bai08].

languages depending on some of their properties such as the use of constructors (cons-free or non-cons free programs) or the allowed recursive calls (tail-recursion) where it is shown that second order programs are more powerful than first order programs as a decision problem is computable by a cons-free first order functional program if and only if it is in P, whereas it is computable by a cons-free second order program if and only if it is in EXPTIME.

1.2.5 Term rewrite systems and interpretations

A third line of research corresponds to the notion of (polynomial) interpretation introduced to show the termination of Term Rewrite Systems (TRS) [Lan79] and studied in [CL87, Ste92, Gie95]. Interpretation methods basically consist in assigning a weight over a well-founded domain to any expression of a given TRS so that if an expression rewrites to another, then a strict weight decrease is observed. Termination is obtained as a consequence of well-foundedness. In order to ensure such a property, it suffices that the strict weight decrease holds for any rewrite rule of the TRS and that the underlying strict order enjoys suitable properties such as closure by context and closure by substitution.

More specifically, a strictly monotonic function $[b] : \mathbb{N}^n \to \mathbb{N}$ is assigned to any symbol b of arity n of a TRS $\mathcal{R}$ and a fresh variable $[x] \in \mathbb{N}$ is assigned to any variable x of $\mathcal{R}$. The assignment $[-]$ is an interpretation if its canonical extension[7] satisfies that for each rewrite rule $l \to r$ of the TRS $\mathcal{R}$, $[l] > [r]$, where the strict inequality is checked for any possible assignment of its free variables. The interpretation is called polynomial if all the TRS symbols are interpreted as polynomials over the natural numbers.

It was shown in [CL92] that the functions computed by TRS admitting a polynomial interpretation have a polynomially bounded growth rate although the composition of their derivation is doubly exponentially bounded (in the size of the initial term) as demonstrated by [Geu88, Lau88]. This result was extended in [BCMT98] where a first characterization of FP is provided.

Theorem 1.2.5 (Adapted from [BCMT01]). *The set of functions computable by a TRS over unary numbers admitting a polynomial interpretation* $[-]$ *with the subterm property*[8] *and the additivity property*[9] *is exactly* FP.

Characterizations of functions computable in exponential and doubly exponential time were also provided in [BCMT98], depending on whether the successor is interpreted as a linear function of the shape $aX + b, a > 1, b \geq 0$, or polynomial function, respectively.

In this setting variable duplication is allowed but it comes with a price to pay for. Indeed, consider the easiest case where the interpretation ranges over the set of natural numbers $\mathbb{N}$. If we consider a 2-ary symbol f and a polynomial interpretation $[-] \in \mathbb{N}[X, Y]$, by subterm property, $[f](X, Y) \geq X + Y$, and, consequently, a variable duplication in a term is such that $[f(x, x)] \geq 2 \times [x]$. Hence duplication is allowed but its iteration is prevented as the interpretation of a term obtained by duplication is smaller than the interpretation of the initial term, a polynomial in the size of the term by additivity and since polynomials are closed under finite composition. Consequently, no exponential can occur under additivity assumption. To illustrate the discussion,

[7] It consists in an extension to terms defined by $[b(t_1, \ldots, t_n)] = [b]([t_1], \ldots, [t_n])$.

[8] The subterm property holds if for any symbol b of arity $n \geq 1$, $\forall i \leq n, \forall X_i \in \mathbb{N}, [b](\ldots X_i \ldots) \geq X_i$.

[9] *i.e.* the successor has an interpretation of the shape $[suc](X) = X + c, c > 0$. This property is extended to any constructor symbol c of arity strictly greater than 0 by requiring that $[c](X_1, \ldots, X_n) = \Sigma_{i=1}^{n} X_i + c, c > 0$.

consider the following simple TRS computing the unary exponential as a counter-example:

$$\begin{aligned} \texttt{double}(\texttt{0}) &\to \texttt{0}, \\ \texttt{double}(\texttt{suc}(\texttt{x})) &\to \texttt{suc}(\texttt{suc}(\texttt{double}(\texttt{x}))), \\ \texttt{exp}(\texttt{0}) &\to \texttt{suc}(\texttt{0}), \\ \texttt{exp}(\texttt{suc}(\texttt{x})) &\to \texttt{double}(\texttt{exp}(\texttt{x})). \end{aligned}$$

The two rules for `double` admit the following additive and polynomial interpretation $[\texttt{0}] = 0$, $[\texttt{suc}](X) = X + 1$, $[\texttt{double}](X) = 3X + 1$ but it turns out that, for exponential to have an interpretation, the following inequality has to be satisfied $[\texttt{exp}](X + 1) > 3[\texttt{exp}](X) + 1$. This inequality is clearly not satisfiable by a polynomial function.

Though very interesting, this characterization was suffering from a lack of expressive power (in terms of captured programs). We will discuss this point more deeply in Section 1.3. This issue was partially solved by the introduction of the notion of quasi-interpretation [MM00, BMM01] where strictly increasing functions are replaced by increasing functions and where the strict weight decrease is replaced by a non-strict weight decrease. Consequently, contrarily to interpretations, quasi-interpretations no longer ensure a time property: termination. Quasi-interpretations rather ensure a space property under additivity requirements: the size of the computed value (and the size of each intermediate value computed during the evaluation process) is bounded by the interpretation of the initial term.

But it turns out that complexity classes can be recovered if quasi-interpretations are to be intersected with some termination techniques. Let RPO, be the set of TRS that can be shown to terminate using Recursive Path Ordering (RPO, see Subsection 2.3.3 for a definition) and where all equivalent function symbol calls are compared lexicographically (lexicographic status) or argument by argument (product status).

Theorem 1.2.6 (Adapted from [BMM11])**.** *The set of functions computed by TRS admitting an additive and polynomial quasi-interpretation and terminating by RPO, where all function symbols have a product status, is exactly* `FP`*.*

On the practical side, a formal library based on quasi-interpretation and RPO allowing to prove that a given program computes a function in `FP` has been implemented in the Coq proof assistant [FHM+18].

Interestingly this characterization is extended in an elegant manner to the class of functions computable in polynomial space as a lexicographic comparison on a function recursive calls preserves the polynomiality of the space needed for the evaluation of a term.

Theorem 1.2.7 (Adapted from [BMM11])**.** *The set of functions computed by TRS admitting an additive and polynomial quasi-interpretation and terminating by RPO, where all function symbols have either product or lexicographic status, is exactly* `FPSPACE`*.*

A characterization of `Logspace` and linear space based on quasi-interpretations is also provided in [BMM05]. This notion has also been extended in a more refined notion, called sup-interpretation, that also allows us to extend the characterization to subpolynomial complexity classes such as `Alogtime` [BMP06] and $\texttt{NC}^k$ [MP08b]. An exhaustive survey of the complexity class characterizations can be found in [Bon11].

For practical applications, a merge between interpretation techniques and bytecode analysis was presented in [ACGDZJ04] where the authors check properties of pre-compiled first order functional programs on a simple stack machine using quasi-interpretations and RPO. This approach

was also generalized in [ADZ04] to systems of concurrent and interactive first order functional threads.

Another related practical and successful approach is the line of work on amortized resource analysis introduced in [HJ03]. In this work, Hofmann and Jost introduced a type system using a potential-based amortized analysis to infer bounds on first order program heap space consumption. Basically, data types are annotated by potentials and type inference generates a set of linear constraints which are then solved independently by an external tool. While in the seminal papers the analysis was restricted to linear bounds in the size of the input, extensions to polynomial bounds and multi-variate polynomial bounds was designed in [HH10] and [HAH11], respectively. An extension to the heap space analysis of OO programs was performed in [HJ06, HR09, HR13] by counting the memory allocations and deallocations of Java programs and an extension to the time analysis of higher-order functional programs was performed in [JHLH10]. Related approaches using sized types on higher-order functional programs were also developed in [SvKvE07, SvEvK09, SvET13]. The amortized complexity approach is focusing on soundness results rather than on completeness results as the goal is the development of practical tools for program resource analysis as Resource-aware ML (RaML) [HAH12]. This approach is closely linked to that of polynomial interpretations as confluent TRS (or TRS with a fixed deterministic rewrite strategy) can be viewed as first order programs. Moreover, the annotations can be viewed as assignments and the constraints generated are very close to the constraints (inequalities) that can be found in polynomial interpretation methods. Amortized resource analysis has been adapted to TRS in [HM14b, HM15] and has been shown to subsume polynomial interpretations, *i.e.* if a TRS is well-typed then it admits a polynomial interpretation.

1.2.6 Imperative programs and dataflow based methods

The last approach deals with the analysis of imperative programs.

In [Jon99], a characterization of `P` was provided in terms of read-only recursive while loop programs on binary trees.[10] If recursivity is withdrawn then a characterization of `Logspace` is obtained. Such a framework has been extended to a fragment of C with arrays in [KV03] and its study has been extended to the non-deterministic case in [Bon06].

In [KN04], Kristiansen and Niggl use a measure, called μ-measure, ranging over $\mathbb{N}$ and defined on imperative loop and stack programs. This measure was previously defined by Niggl for characterizing complexity classes, including `FP`, on functional languages [Nig00]. On the imperative paradigm, the μ-measure accounts for the number of nested loops by considering only those loops for which the variables have some interdependence, which more or less corresponds to a duplication. Kristian and Niggl show that the set of functions computed by programs of μ-measure n is exactly the class of functions $\mathcal{E}^{n+2}$ in Grzegorczyk's hierarchy [Grz53]. This line of work was extended in [NW06] to a more general and expressive programming language including conditional, loops and more advanced data structures such as lists or trees.

In [JK05, JK09], Jones and Kristiansen present a new approach based on a matrix typing discipline for *loop* and *while* programs.[11] If the program has n variables, the considered matrices are $n \times n$ square matrices whose coefficients are ranging over the finite domain $\{0, m, w, p\}$, with an underlying order $<$ satisfying $0 < m < w < p$. If the matrix M types the command c then $M_{i,j}$ encodes how the j-th variable is overwritten by the i-th variable after the execution of c. The intuition hidden under this 4-elements domain is as follows: a mwp-bound is a function of the shape $max(\overline{X}, P(\overline{Y})) + Q(\overline{Z})$, for some polynomials P and Q. Variables in $\overline{X}$ correspond to a

[10] *i.e.* These programs do not have the ability to build a binary tree.
[11] Loops correspond to linear iteration in the size of the guard value.

maximum flow m, variables in $\overline{y}$ to a *weak-polynomial* flow w, and variables in $\overline{z}$ to a *polynomial* flow p (and variables that do not appear correspond to a 0 flow). Consequently, values in the typing matrix encode the way each variable is related to other variables in the mwp-bound corresponding to the command execution.

To illustrate this with an example, consider the simple loop command `loop` $X_3\{X_1 := X_1 + X_2\}$. Let X_i^0 denote the initial value stored in X_i before execution. At the end of the execution of this command, the three informal equalities $X_3 = X_3^0$, $X_2 = X_2^0$, and $X_1 = X_1^0 + X_2^0 \times X_3^0$ hold. A corresponding matrix is

$$\begin{bmatrix} m & 0 & 0 \\ p & m & 0 \\ p & 0 & m \end{bmatrix}$$

as the inequalities can be reformulated by $X_3 = max(X_3^0)$, $X_2 = max(X_2^0)$, and $X_1 = max(X_1^0) + Q(X_2^0, X_3^0)$, with $Q(X, Y) = X \times Y$.[12]

This type system performs a strict control of variable duplication inside loops and while loops by requiring the closure of the matrix corresponding to the inner command to have nothing but m on its diagonal. The idea is still to prevent iteration of duplication as a command of the shape $X_k := 2 \times X_k$ cannot be typed by a matrix M such that $M_{k,k} = m$. Indeed, as $X_k = max(P(X_k^0))$ with $P(X) = 2 \times X$, $M_{k,k}$ is at least equal to w.[13] Consequently, this command cannot be iterated as the closure will still contain w in the diagonal.

The obtained characterization is that if a command can be typed by a given matrix then the values computed by this command are of polynomially bounded size.

A similar line of work was performed on a low level assembly-like programming language of stack machines in [Moy09]. Here a state is abstracted by its size, each instruction is abstracted by the effect it has on the size of states. The analysis allows to show termination and to study the space consumed during program execution using approaches that are inspired by size change termination principle [LJBA01] and by non-size-increasing principle [Hof02], respectively.

On the practical side, several studies have been performed. Although not always directly related to ICC, these studies mostly target a similar goal (soundness-like): finding worst case analysis for practical programs. Moreover, the techniques used are sometimes based on similar or inspired theoretical approaches. Clearly, in such a practical context, completeness is sacrificed at the price of a better tractability. We will discuss this point more deeply in Section 1.3.

The practical studies can be split in three distinct approaches:

- the SPEED tool [Gul09, GMC09] that implements a multiple counters based approach for computing symbolic bounds on the number of statements a procedure executes depending on (user-defined quantitative functions on) its inputs. The idea is simple but works well: quantitative bounds are generated using invariant generation tools and the the SPEED program has been proved to be efficient and productive on C++ Standard Template Library,

- the COSTA tool [AAG+07a, AAG+07b] that has been introduced as a new symbolic analysis and that can be used to infer termination and complexity properties of Java bytecode; The method tries to infer automatically resource upper bounds on Java bytecode using cost relations. The considered complexity properties are resource usage such as time or space [AGGZ07, AGGZ13],

[12] There is no uniqueness. See next footnote.
[13] p is also a valid option.

- the project CerCo [AAB+13, AARG12] (Certified Complexity) of the European Commission FP7 that was attempting to develop a formally verified complexity preserving compiler from an expressive subset of C to some assembly code for embedded systems; the compiler is also providing certified cost annotations for programs.

Some works have also extended the studies of imperative programs to subpolynomial complexity classes. In [HS10], an imperative programming language with pointers over a graph structure called PURPLE is introduced. This language subsumes Jumping Automata on Graphs of [CR80], a particular kind of automata with a state, a graph structure and, pebbles that can move from one vertex to an adjacent vertex or that can jump directly to a vertex containing another pebble, and is shown to be strictly included in `Logspace` as it cannot encode undirected reachability which is known to be in `Logspace` as a direct consequence of Reingold's Theorem [Rei08]. It was also shown in [HRS13] that PURPLE captures all of `Logspace` on locally ordered graphs.

1.3 Limits

1.3.1 Intensionality vs decidability

Most of the criteria in the literature, including the criteria presented in previous section, are extensionally complete as they consist in characterizations capturing all the functions of a given complexity class $\mathcal{C}$. However most of these criteria are also intensionally incomplete. It means that there exist algorithms computing a function of $\mathcal{C}$ that can be rejected by the criterion. On the one hand, some algorithms are rejected in a highly expected way. For example, if we set $\mathcal{C} = \texttt{FP}$, a naive algorithm computing a function in `FP` using an exponential number of steps will hopefully be rejected. On the other hand, some "good" algorithms will be rejected without compromising the extensional completeness. One may think for example of the sorting function. For a given functional language $\mathcal{F}$, a criterion $\mathcal{K} \subseteq \mathcal{F}$ can accept a naive sorting algorithm running in time $O(n^2)$ and reject the `quicksort` algorithm (which is indeed difficult to capture as its polynomial behavior relies on the ability to show that the input array is semantically divided in two parts with no increase) so that $\{[\![\mathtt{p}]\!] \mid \mathtt{p} \in \mathcal{K}\} = \texttt{FP}$ holds but $\texttt{quicksort} \in \mathcal{K}$ does not. Worst of all, most of the characterizations of `FP` provided in previous section do not capture the natural implementation of the `quicksort` algorithm. Historically, this kind of intensional incompleteness has been highlighted in [Col89] where Colson has shown that a function computing the minimum of two unary integers $\underline{n}$ and $\underline{m}$, encoding the integers n and m, respectively, cannot be implemented by a primitive recursive algorithm on a Call-By-Name (CBN) TRS in time $O(min(n, m))$, whereas this function can easily be implemented within such a time bound if the constraint on the algorithm is relaxed.

There have been attempts to develop intensional criteria $\mathcal{R}$ such that the following intensional property holds:

$$\forall \mathtt{p} \in \mathcal{L},\ \text{if } \Phi(\mathtt{p}) \in \mathcal{C} \text{ then } \mathtt{p} \in \mathcal{R},$$

where Φ is a complexity measure[14] à la Blum [Blu67] and under the assumption that the criterion is extensionally complete (*i.e.* $\{[\![\mathtt{p}]\!] \mid \mathtt{p} \in \mathcal{R}\} = \mathcal{C}$ holds).

For example, if $\mathcal{C} = \texttt{FP}$, we could set $\Phi(\mathtt{p})$ to be the running time of the program $\mathtt{p}$ relating the size of the program input to the number of steps needed to produce the corresponding output.

[14] We make a slight abuse of notation as usually a Blum complexity measure maps an integer, encoding the program data and input, to a Gödel numbering of the partial computable functions by not making the distinction between the Gödel number and the function it represents.

Hence any program whose runtime is a polynomial time computable function would have to satisfy the criterion $\mathcal{R}$.

But clearly, such criteria come at the price of a high undecidability. Indeed, if such a criterion $\mathcal{R}$ can be designed for decision problems of the complexity class `P`. Then one programmer just needs to consider a program `prog` of $\mathcal{L}$ computing a well-known `NP`-complete problem. Depending on whether `prog` $\in \mathcal{R}$ holds or not, the programmer would be able to give a positive (respectively negative) answer to the `P` vs `NP` problem. As an illustrating example, an intensionally complete characterization of PCF programs is given in [DLG11] using a parameterized type system relying on an oracle. The undecidability lies in the oracle. For the particular case of polynomial time, it is well-known for a long time that the problem of providing an intensional characterization is Σ_2^0-complete in the arithmetical hierarchy [Haj79].

1.3.2 Complexity of inference

A given ICC criterion can be judged on two distinct levels:

- its expressive power: has the criterion more expressive power than others?
- its inherent complexity:[15] is the criterion decidable and, if so, what is its complexity?

The expressive power is very difficult to judge in general as most of the methods are incomparable. However several studies on the inherent complexity of the criteria have been performed as we will see below.

Safe recursion

For function algebras, the inherent complexity is the complexity of checking whether a given program is expressed in a safe or ramified recursion scheme. It can be easily implemented in polynomial time as it consists in checking that the program equations can be generated by the underlying function algebra grammar. The tractability of this check is clearly at the root of the most important issue of function algebra: their weak expressive power. The recursion scheme highly restricts the way programs can be written. Moreover, for a given function in `FP`, designing a program computing the function is undecidable in general and can be very hard for some particular functions.

Light logics

The problems of type inference for LAL (EAL, respectively), consisting in checking whether a lambda term can be decorated by a type in LAL (EAL, respectively) has been studied in [Bai02] (in [CM01], respectively) under some slight restrictions (normal forms and simply typed terms)[16] that have been tamed in [Bai04b].[17] Consequently, type inference is decidable.

Moreover, it was shown in [ABT07], that type inference for DLAL can be checked in time polynomial in the size of the lambda term under the assumption that the System F type of the lambda term under consideration is provided.

[15] The inherent complexity is also called *Synthesis problem complexity*, *Derivability problem complexity* or *Type inference complexity* depending on the computational model under consideration.

[16] *i.e.* polymorphism is not allowed.

[17] The normal form hypothesis is also no longer required using a suitable notion of subtyping.

The type inference algorithm for the Soft Type Assignment (STA) of [GRDR07] based on SLL was proposed in [GRDR08] where it is shown to be decidable for simple types. It was extended in [CS12] to ML Soft Type Assignment (MLSTA), an extension to ML-like polymorphism whose type checking and type inferences are decidable. See also [CS16] for a discussion on the undecidability of the family of STA type systems when extended to System F.

These results can be summarized as follows (see [Bai08] for more details. In particular, type inference is shown to be in polynomial time on a restriction of EAL).

Type system	EAL	LAL	DLAL	STA	MLSTA
Restriction	T	T	F	T	F
Complexity	Decidable*	Decidable*	Ptime	Ptime	Decidable

Figure 1.2: Decidability and complexity of the type inference

where T is the restriction to closed simply typed terms and F is the restriction to closed typable terms in System F and where * means that the bound is known to be at least exponential.

Interpretations

For interpretations and quasi-interpretations, the synthesis problem consists in checking if a given TRS has a interpretation and quasi-interpretation, respectively. This problem was first introduced by Amadio in [Ama03, Ama05] and studied for quasi-interpretations on a space of functions including addition, maximum and coefficient over bounded rational numbers or integers: it was shown to be NP-hard on such function space and NP-complete if the coefficients are restricted to the set $\{0, 1\}$. The NP-hardness result was extended to coefficients over positive real numbers $\mathbb{R}^+$ in [BMMP05] and a survey of the distinct results has been provided in [Péc13].

Let $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{Q}^+{}_{\mathsf{d}}, \mathbb{R}^+\}$, $\mathbb{Q}^+{}_{\mathsf{d}}$ being the set of rationals in $\mathbb{Q}^+$ of bounded representation, and define the following sets of polynomials:

- the set $\mathbb{K}[\overline{X}]$ of usual multivariate polynomials whose coefficients are in $\mathbb{K}$ and with n variables $\overline{X} = X_1, \ldots, X_n$ ranging over the field of real numbers,
- the set of $\mathsf{MaxPoly}^{(\mathsf{k},\mathsf{d})}\{\mathbb{K}\}$ polynomials, which consists of functions obtained using constants over $\mathbb{K}$ and arbitrary compositions of the operators $+, \times$ and max of degree bounded by d and a max arity bounded by k,
- and the set of $\mathsf{MaxPlus}^{(\mathsf{k},\mathsf{d})}\{\mathbb{K}\}$ functions, which consists of functions obtained using constants over $\mathbb{K}$ bounded by d and arbitrary compositions of the operators $+$ and max, with a max arity bounded by k.

The obtained results can be summarized by Figure 1.3.

The first and second lines are direct consequences of Hilbert's tenth problem undecidability, whereas the third and fourth lines are a consequence of Tarski's quantifier elimination Theorem over real numbers [Tar51]. One important point to mention here is that the synthesis problem is exponential and not doubly exponential because the synthesis problem is more restricted than general quantifier elimination. In the first column, the symbol "$-$" means that the study of the synthesis problem for polynomial interpretation does not make sense whenever a *max* operator is allowed since *max* is not a strictly monotonic function.

Function space \ tool	Polynomial Interpretation	Polynomial Quasi-Interpretation
$\mathbb{K}[\overline{X}], \mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+\}$	Undecidable	Undecidable
$\mathsf{MaxPoly}\{\mathbb{K}\}, \mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+\}$	–	Undecidable
$\mathbb{R}^+[\overline{X}]$	EXPTIME	EXPTIME
$\mathsf{MaxPoly}^{(\mathsf{k},\mathsf{d})}\{\mathbb{R}^+\}$	–	EXPTIME
$\mathsf{MaxPlus}^{(\mathsf{k},\mathsf{d})}\{\mathbb{K}\}, \mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+{}_{\mathsf{d}}\}$	–	NP-complete
$\mathsf{MaxPlus}^{(\mathsf{k},\mathsf{d})}\{\mathbb{R}^+\}$	–	NP-hard

Figure 1.3: Decidability and complexity of the synthesis problem

A last interesting remark is that we can conclude from the fourth line of Figure 1.3 that the criterion of Theorem 1.2.6, providing a characterization of FP, is decidable in EXPTIME over positive real numbers as the problem of checking whether a TRS terminates by RPO is known to be NP-complete [KN85].

Matrix based type system for imperative programs

For the imperative methods, the question of the complexity of the criterion was less central to the concerns because most of the tools are empirical and cannot be straightforwardly seen as ICC criteria as discussed in Section 1.2.6. In the work of Jones and Kristiansen [JK09], it is shown that the matrix based type system is not deterministic in the sense that several distinct matrices can be assigned to a given command. Jones and Kristiansen introduce an algorithm for solving the derivability problem, *i.e.* "given a command c does there exist a matrix M such that c has type M?", and show that this problem is in NP and conjecture it to be NP-complete.

1.4 Related Work

In this section, we try to relate the works on ICC with other important domains of theoretical computer science such as termination, computability, finite model theory and static analysis.

1.4.1 Termination

The links between ICC and termination are very tight. A program must terminate in order to compute a function of a given (standard) complexity class. In the other direction, a complexity class certificate ensures a termination certificate as the functions of ordinary complexity classes are total. This remark is not that interesting in the sense that complexity class certificates are much harder to infer than termination certificates. However, as termination is usually a first prerequisite for a program to compute a function of a given complexity class, it is not surprising that many ICC criteria are derived or inspired directly from termination techniques.

TRS. It has already been mentioned that polynomial interpretations have been introduced with the primary objective of demonstrating termination of TRSs [Lan79]. However, this tool was severely restricted by its intensionality as the subterm requirements are very strong and reject a lot of desirable programs. One suggestion by Hofbauer was to relax the notion of interpretation and particularly its subterm property by introducing the notion of context dependent-interpretations [Hof01]. Context dependent interpretations take an extra contextual parameter, a strictly positive real number, and are ranging over the real numbers as in the many works extending polynomial interpretation and quasi-interpretation techniques to real numbers [Der87, BMMP05, Luc05, Luc07, MP08c]. Well-foundedness is recovered by assuming that in a rewrite rule, the interpretation induces a strict decrease by at least the parameter value. In [MS08], a subclass of context dependent interpretations inducing a quadratic derivational complexity upper bound is introduced. Another interesting line of work in the use of interpretations for showing program termination are the studies generalizing polynomial interpretations to a more general codomain, the space of square matrices with integer coefficients [HW06, EWZ08]. This method was adapted to infer polynomial derivational complexity upper bounds [NZM10, Wal10, Wal15]. See also [MSW08] for a comparison between a subclass of context dependent interpretations and a subclass of triangular matrix interpretations. An adaptation to context-sensitive rewriting has also been studied in [HM14a].

The RPO based termination techniques (see [Der82, Kam80, Pla78]) were also popular in ICC as they are combined to additive quasi-interpretations to obtain characterizations of `FP` and `FPSPACE` [BMM11], as already presented in Theorem 1.2.6 and Theorem 1.2.7, respectively. The main motivations were that the Lexicographic Path Ordering (LPO), a RPO-like order where the arguments of function calls are compared lexicographically, was already known to yield multiply recursive derivation lengths [Wei95] and that the Multiset Path Ordering (MPO), a RPO-like order where recursive calls arguments are compared using multisets, was already known to yield primitive recursive derivation lengths [Hof92]. In a first attempt, the Light Lexicographic Path Ordering (LLPO) was developed in [CM00] to provide a characterization of `FPSPACE`. The LLPO was adapted to a notion of Polynomial Path Order (PPO*) in [AM08]. The PPO* (and its variant sPOP in [AEM15]) induces polynomial bounds on the maximal number of innermost rewrite steps and its combinations with other termination techniques such as dependency pairs [AM09], discussed in the next paragraph, and semantic labeling [Ava08] have been studied. Moreover a termination tool for automated termination, derivational complexity analysis and runtime analysis of TRS has been developed in [AM13b, AM16, AMS16].

A last very popular technique for showing the termination of TRS is the notion of Dependency Pairs (DP) of [AG00]. It consists in abstracting a given TRS as a dependency graph containing information about the function calls and control flow of the TRS and finding a strict decrease on the graph transitions with an underlying well-founded domain so that any cycle (that may correspond to a recursive call) can only occur a finite number of times. The DP method was used in [HM08, NEG13] to analyze the *innermost computational* complexity of TRS. The innermost complexity of a TRS is a partial function from natural numbers to natural numbers. It associates to a natural number n, the maximal derivation height obtained by deriving any term of the TRS of size smaller than n using an innermost reduction strategy. The partiality comes from the fact that the innermost complexity can be undefined in presence of diverging terms. This technique has been extended to Java programs and other programming languages in a tool called APROVE [GAB+17]. The properties related to the innermost runtime complexity are soundness results and induce that the corresponding computed function is in `FP` as demonstrated in [AM10b, DLM09]. Completeness is here sacrificed at the price of tractability. It is worth noticing that this soundness result has been extended to full rewriting in [AM10a]

which shows that the runtime complexity of a TRS and the runtime complexity of a TM implementation of the TRS, for decision problems and particular class of functions, are polynomially related. This is however not true in general as full rewriting and lambda calculus do not respect in general the *Invariance Thesis* [Boa14]: it means that there exists an exponential time computable function that can be implemented by a TRS with a polynomial derivation length (see also [ADL14] solution for recovering the *Invariance Thesis* in a lambda calculus setting and [ADL18] for the *Invariance Thesis* in TRS with sharing and memoization). It was proved in [MS11] that TRS whose termination can be shown using the DP method under some slight assumptions may induce a multiply recursive derivational complexity and primitive recursive upper bounds have been obtained on DP refinements based on RPO [MS09]. The ICC work presented in [MP08c] provides a sound and complete characterization of `FP` based on a combination of DP and polynomial interpretations. The flexibility provided by the DP method considerably improves the intensionality of this characterization with respect to the pure interpretation-based characterizations.

Lambda calculus and functional languages. For lambda calculi, types in LAL (or EAL) can be viewed as decorations of data types in System F [Gir72, Rey74], that is known to enjoy a strong normalization property. For example, unary integers in System F correspond to $\forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$ and are decorated by $\forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$ in LAL (and by $\forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha)$ in EAL). Other termination techniques such as size based termination [HPS96] have been adapted to functional languages to infer resource upper bounds in practice [ADL17]. In this framework, a type contains an extra information about the size of the terms it represents. A strict decrease is enforced on the sized types of recursive calls, ranging over a well-founded domain, to ensure termination [CK01, BGR08]. This also allows to recover information about the size of the intermediate results computed as in the case of additive interpretations (see [Vas08]). It is worth noticing that this approach is related to the one of [DLG11, DLP14] that combines dependent types and a linear type system to obtain type system relatively complete for PCF. Here *relatively* means that not only the complexity of the computed function is captured but also the complexity of evaluating a term. However, this approach is not a pure ICC study in the sense that, as in BLL, several external resource bounds are required and the target (PCF) is not a complexity class.

A last very popular technique for showing the termination of first order functional programs is the Size Change Principle [LJBA01] (SCP). It basically consists in abstracting a program in a call of sequences and checking that any infinite call of sequences corresponds to an infinite number of decrease on a well-founded domain. It was proved in [BA02] that the class of functions computed by size change terminating programs is the class of multiply recursive functions.

Imperative programs. On the imperative side, the works on studying the complexity of while loop programs are also related to the studies on the termination of such programs, including non exhaustively [BAG13, BAG17, BAGM12, SKvE10]. [HJP10] is also an interesting paper presenting the differences and similarities between the size change principle approach of [LJBA01] and the transition invariant approach of [PR04] that serves as a basis of the software TERMINATOR [CPR06] designed for automatically showing the termination of imperative programs.

1.4.2 Computability

ICC is also highly related to the studies on computational models and computability. Before discussing the complexity of a given function, the first question arising is whether this function

is computable by any reasonable computational model. Hence computability, as termination, is a prerequisite for complexity. After delineating the limits of computational models, researchers have focused on restricted class of computable functions. Historically, the first studies of interest mostly focused on very large classes that are now considered as untractable, *e.g.* the class of recursive functions [Pét67, Kle36]. Other examples of such classes are the class of primitive recursive functions, the class of multiple recursive functions, and classes of the Grzegorgczyk's hierarchy. Primitive recursive functions were introduced by Gödel and Kleene in [Göd31, Kle35]. Kalmar defined in [Kal43] the elementary functions by restricting the primitive recursion scheme to limited sums and products. In [Grz53], Grzegorgczyk defined the hierarchy of classes $\mathcal{E}^k$, obtained by closure of diagonal functions, composition and a restricted scheme of primitive recursion, and showed that $\mathcal{E}^3$ is exactly Kalmar's class, and, in [Rit63], Ritchie demonstrated that $\mathcal{E}^2$ is exactly the class of functions computable by a TM in linear space. The characterizations of these classes were obtained mostly by restricting the recursion scheme and expressive power of the underlying language and can be viewed as the corner stone approaches that led to Cobham's characterization of `FP`.

1.4.3 Finite model theory

The works in ICC are also related to Finite Model Theory (FMT). FMT is a branch of mathematical logic studying the relation between a (logical) language and its interpretation on finite structures. In [Fag74], Fagin showed that the properties expressible by an existential second order formula are exactly the complexity class `NP`. A characterization of `P`, known as the Immerman-Vardi Theorem, was provided (see [Saz80, Imm86] and [Var82], where other characterizations of the main complexity classes are also provided) in terms of first order logic extended by the ability to define new relations by induction expressed as a fixed-point logic over ordered structures captures `P`. An alternative definition was also suggested by Grädel [Grä91] in terms of second order Boolean queries in which the first order part is a universal formula in conjunctive normal form with at most one positive literal per clause. Some characterization of `P` and `Logspace` have also been studied in [Gur83] and extension to elementary recursive functions are studied in [Goe92]. See [Imm12] for a complete description of the descriptive complexity domain.

To some extent FMT can be considered as a subdomain of ICC as it provides characterizations of complexity classes without requiring resource bounds. However this is a rough approximation for two reasons. First, it is well known that the main results of classical model theory fail for finite structures and, in the other direction, the complexity results on finite models cannot be adapted to general models including functions. For example, a characterization of `FP` cannot be provided straightforwardly. Second, the characterizations suffer from being extensional rather than intensional as a Quantified Boolean Formula (QBF) is rather a function than a programming language or computational model.

1.4.4 Static analysis

ICC is also closely related to static program analysis. Static analysis is a subdomain of Formal Methods related to the analysis of program properties performed without actually executing the program. Techniques such as dataflow analysis, model checking, and the use of Hoare logics for correctness proofs belong to static analysis. We have already mentioned in Section 1.2.6 that methods for analyzing the complexity of imperative programs such as [JK05, JK09] were directly inspired by dataflow analysis. One of the most successful techniques in static analysis is the notion of abstract interpretation by Cousot and Cousot [CC77]. Although there is, to our knowledge,

no formal comparison between the notion of abstract interpretation and the ICC tools, it is clear that most of the semantics ICC tools can be viewed as a particular abstractions of program properties. For example, a linear type can be viewed as an abstraction of the term it corresponds to and the polynomial interpretation of an expression can be viewed as an abstraction: an upper bound on its derivation length.

It is worth noticing that most of aforementioned applied complexity analysis of imperative programming languages [Gul09, GMC09, AAG+07a, AAG+07b] are based on the use of static analysis techniques and, mostly, rely on the use of abstract interpretations to compute program invariants.

1.5 Contribution

In the first part of this introduction, we have defined the notion of ICC and described the main families of tools and criteria providing ICC results and complexity classes characterizations. We have also provided a comparison with theoretical works on computability theory, finite model theory, and well-known static analysis tools such as termination tools and abstract interpretations. We have highlighted the two main restrictions of ICC: the complexity of the inference problem and the (lack of) expressive power of the methods.

We now present our main technical and theoretical contributions to the field and will also try to place them in a more general context by not limiting this presentation to our work. The first part of this contribution, Chapter 2, will deal with the works performed on improving the intensionality/expressive power of the ICC methods. The second part, Chapter 3, will deal with the extensions of the techniques to other paradigms. The third part, Chapter 4, will deal with the extensions to infinite (including coinductive) data structures. The last part of this contribution, Chapter 5, will discuss some of the open issues and research perspectives of the domain.

Part of the works presented in this manuscript belongs to the works on ICC I have contributed to from 2004 to 2019. The underlyng research has been supported by Inria Associated Team CRISTAL 2009-2012, ANR COMPLICE 2008-2014 and ANR ELICA 2014-2019 and corresponds to the following list of publications ordered by research topic. They correspond to 5 international journal publications and 14 international peer-reviewed international conferences and do not include my international workshop publications with informal proceedings and some of my publications in computer virology, on data aggregates and optimal representations, and on quantum program semantics.

The following table summarizes for each section of the contributing Chapters, the list of my corresponding publications (indexed below). The sections without publication correspond to works done by other authors.

Section	2.1	2.2	2.3	2.4	3.1	3.2	3.3	4.1	4.2	4.3	4.4
Publications			3	1,2,4,5,6,7	9,10,11,13		8,12	14,15,18	16,19	17	

- Improving the expressive power of ICC tools, Chapter 2:

  1. Guillaume Bonfante and Jean-Yves Marion and Romain Péchoux. A Characterization of Alternating Log Time by First Order Functional Programs. In *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, pages 90–104, 2006.

2. Jean-Yves Marion and Romain Péchoux. Resource Analysis by Sup-interpretation. In *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, pages 163–176, 2006.
3. Guillaume Bonfante and Jean-Yves Marion and Romain Péchoux. Quasi-interpretation Synthesis by Decomposition. In *Theoretical Aspects of Computing - ICTAC 2007, 4th International Colloquium, Macau, China, September 26-28, 2007, Proceedings*, pages 410–424, 2007.
4. Jean-Yves Marion and Romain Péchoux. A Characterization of NC^k. In *Theory and Applications of Models of Computation, 5th International Conference, TAMC 2008, Xi'an, China, April 25-29, 2008. Proceedings*, pages 136–147, 2008.
5. Jean-Yves Marion and Romain Péchoux. Characterizations of polynomial complexity classes with a better intensionality. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15-17, 2008, Valencia, Spain*, pages 79–88, 2008.
6. Jean-Yves Marion and Romain Péchoux. Sup-interpretations, a semantic method for static analysis of program resources. *ACM Trans. Comput. Log.*, 10(4), 31 pages, 2009.
7. Romain Péchoux. Synthesis of sup-interpretations: A survey. *Theoretical Computer Science*, 467:30–52, 2013.

- Adapting ICC tools to other programming paradigms, Chapter 3:

8. Jean-Yves Marion and Romain Péchoux. Analyzing the Implicit Computational Complexity of object-oriented programs. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008, December 9-11, 2008, Bangalore, India*, pages 316–327, 2008.
9. Emmanuel Hainry, Jean-Yves Marion, and Romain Péchoux. Type-based complexity analysis for fork processes. In *the 16th International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 305–320, Lecture Notes in Computer Science, Springer, 2013.
10. Jean-Yves Marion and Romain Péchoux. Complexity information flow in a multi-threaded imperative language. In *the 11th Annual Conference on Theory and Applications of Models of Computation, TAMC 2014, Chennai, India, April 11-13, 2014. Proceedings*, pages 124–140, Lecture Notes in Computer Science, Springer, 2014.
11. Emmanuel Hainry and Romain Péchoux. Objects in Polynomial Time. In *the 13th Asian Symposium on Programming Languages and Systems, APLAS 2015, Pohang, South Korea, November 30- December 2, 2015. Proceedings*, pages 387–404, Lecture Notes in Computer Science, Springer, 2015.
12. Emmanuel Hainry and Romain Péchoux. Higher-order interpretations for higher-order complexity. In *the 21st International Conference on Logic for Programming Artificial Intelligence and Reasoning, LPAR 2017, Maun, Botswana.*
13. Emmanuel Hainry and Romain Péchoux. A Type-Based Complexity Analysis of Object Oriented Programs. *Information and Computation*, 261:78-115, 2018.

- Extending ICC tools to infinite and coinductive data, Chapter 4:

14. Marco Gaboardi and Romain Péchoux. Global and Local Space Properties of Stream Programs. In *Foundational and Practical Aspects of Resource Analysis - First International Workshop, FOPARA 2009, Eindhoven, The Netherlands, November 6, 2009, Revised Selected Papers*, pages 51–66, 2009.

15. Marco Gaboardi and Romain Péchoux. Upper Bounds on Stream I/O Using Semantic Interpretations. In *Computer Science Logic, 23rd international Conference, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, pages 271–286, 2009.

16. Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, Romain Péchoux. Interpretation of Stream Programs: Characterizing Type 2 Polynomial Time Complexity. In *Algorithms and Computation - 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part I, Proceedings*, pages 291–303, Lecture Notes in Computer Science, Springer, 2010.

17. Marco Gaboardi and Romain Péchoux. Algebras and coalgebras in the light affine lambda calculus. In *the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015. Proceedings*, pages 114–126, ACM, 2015.

18. Marco Gaboardi and Romain Péchoux. On Bounding Space Usage of Streams Using Interpretation Analysis. *Science of Computer Programming*, 111(3):395–425, 2015.

19. Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, and Romain Péchoux. Characterizing polynomial time complexity of stream programs using interpretations. *Theoretical Computer Science*, 585:41–54, 2015.

Chapter 2

Towards a better intensionality

Contents

In this chapter, we address the issue of the expressive power of ICC criteria. Let us first restate the notion of an ICC criterion already discussed in Section 1.2.

Definition 2.0.1 (ICC criterion)**.** *Given a programming language $\mathcal{L}$, with semantics $[\![-]\!]$ mapping every program $\mathtt{p} \in \mathcal{L}$ to a partial function $[\![\mathtt{p}]\!] \in \{0,1\}^* \to \{0,1\}^*$, and a complexity class $\mathcal{C}$, an ICC criterion $\mathcal{R}$ is a subset of $\mathcal{L}$ such that the following holds:*

$$\{[\![\mathtt{p}]\!] \mid \mathtt{p} \in \mathcal{R}\} = \mathcal{C}.$$

If we want to mention explicitly the language $\mathcal{L}$ and complexity class $\mathcal{C}$, we write that $\mathcal{R}$ is a $(\mathcal{L},\mathcal{C})$ ICC criterion. For simplicity, we will sometimes write $[\![\mathcal{R}]\!]$ to denote the set of functions computed by programs $\mathtt{p} \in \mathcal{R}$.

The expressive power of an ICC criterion is called its *intensionality* in the literature in opposition to extensionality that refers to functions.[18] Indeed, when a program analyzer is claimed to have a better intensionality than some others, it means that the set of captured algorithms strictly includes the set of algorithms captured by the other analyzers.

Definition 2.0.2 (Intensionality)**.** *For a fixed language $\mathcal{L}$ and a given complexity class $\mathcal{C}$, let $\mathcal{R}$ and $\mathcal{S}$ be two $(\mathcal{L},\mathcal{C})$ ICC criteria. $\mathcal{R}$ has a better intensionality (or more expressive power) than $\mathcal{S}$, if $\mathcal{R} \subseteq \mathcal{S}$. We will write $\mathcal{R} = \mathcal{S}$ in the case where $\mathcal{R} \subseteq \mathcal{S}$ and $\mathcal{S} \subseteq \mathcal{R}$ both hold. In the case where $\mathcal{S} - \mathcal{R} \neq \emptyset$ and $\mathcal{R} - \mathcal{S} \neq \emptyset$ both hold, $\mathcal{R}$ and $\mathcal{S}$ are incomparable.*

Sadly, the intensionalities of most of the ICC criteria that can be found in the literature are very difficult to compare as these criteria mainly target distinct programming paradigms. We can only compare the criteria for a fixed language and a fixed complexity class. Worst of all, even for a fixed language and a fixed complexity class, ICC criteria are most of the time incomparable.

Historically the question of intensionality was introduced due to the difficulty of writing "programs" in the function algebra framework. As discussed in Section 1.3, this issue was highlighted by the work of Colson on the relative difficulty to write functions using primitive recursion [Col89]. Due to the restrictive nature of recursive calls in such a framework, function algebra were not the good candidate for improving the expressive power and the characterizations on lambda calculus, TRS, and imperative programming languages can be thought of as finding a way to improve the intensionality of function algebra.

Whereas only few works have been carried out for imperative programming languages, where completeness was often abandoned for the sake of tractability, this notion has been studied for lambda calculus and logics and studied for TRS and interpretations. We sum up most of the corresponding results in the remainder of this chapter.

In Section 2.1, we discuss the advances obtained in the main linear logic based approaches: soft linear logic and light linear logic. In Section 2.2, we recall some basic notions on TRSs. We introduce the notion of (polynomial) interpretations that was primarily used for showing termination and used later to improve the expressive power of function algebra characterizations. In Section 2.3, we discuss the notion of quasi-interpretation that improves the expressive power of interpretations. The price to pay is the loss of termination properties. We study its combinations with RPOs, a termination technique, and its modularity properties. In Section 2.4, we introduce the last notion: sup-interpretation that allows to improve the expressive power of quasi-interpretation by withdrawing the subterm property requirements. We also study its combination with the dependency pairs method.

2.1 Linear logic based approaches

2.1.1 Light linear logic

For light logics, Dual Light Affine Logic (DLAL) was introduced by Baillot and Terui in [BT04, BT09] in order to solve the main issues of LAL, *i.e.* to ensure subject reduction and to obtain a polynomial time complexity bound on β-reduction. The DLAL system includes a non-linear arrow $\Rightarrow$ to compensate for the absence of the exponential ! and is designed to study complexity properties of terms of the pure lambda calculus

$$M, N ::= x \mid \lambda x.M \mid M\ N,$$

[18] This should not be confused with the non related philosophical concept of intentionality.

$$\frac{}{;x : A \vdash x : A}\ (\texttt{Var})$$

$$\frac{\Gamma; \Delta, x : A \vdash M : B}{\Gamma; \Delta \vdash \lambda x.M : A \multimap B}\ (\texttt{I}\multimap) \qquad \frac{\Gamma; \Delta \vdash M : A \multimap B \quad \Gamma'; \Delta' \vdash N : A}{\Gamma, \Gamma'; \Delta, \Delta' \vdash M\ N : B}\ (\texttt{E}\multimap)$$

$$\frac{\Gamma, x : A; \Delta \vdash M : B}{\Gamma; \Delta \vdash \lambda x.M : A \Rightarrow B}\ (\texttt{I}\Rightarrow) \qquad \frac{\Gamma; \Delta \vdash M : A \Rightarrow B \quad ; z : C \vdash N : A}{\Gamma, z : C; \Delta \vdash M\ N : B}\ (\texttt{E}\Rightarrow)$$

$$\frac{\Gamma; \Delta \vdash M : A}{\Gamma, \Gamma'; \Delta, \Delta' \vdash M : A}\ (\text{Weak}) \qquad \frac{x : A, y : A, \Gamma; \Delta \vdash M : B}{z : A, \Gamma; \Delta \vdash M[z/x, z/y] : B}\ (\text{Cntr})$$

$$\frac{;\Gamma, \Delta \vdash M : A}{\Gamma; \S\Delta \vdash M : \S A}\ (\texttt{I}\S) \qquad \frac{\Gamma; \Delta \vdash N : \S A \quad \Gamma'; x : \S A, \Delta' \vdash M : B}{\Gamma, \Gamma'; \Delta, \Delta' \vdash M[N/x] : B}\ (\texttt{E}\S)$$

$$\frac{\Gamma; \Delta \vdash M : A \quad \Gamma'; \Delta' \vdash N : B}{\Gamma, \Gamma'; \Delta, \Delta' \vdash M \otimes N : A \otimes B}\ (\texttt{I}\otimes) \qquad \frac{\Gamma; \Delta \vdash M : A \otimes B \quad \Gamma'; \Delta', x : A, y : B \vdash N : C}{\Gamma, \Gamma'; \Delta, \Delta' \vdash \text{let } x \otimes y = M \text{ in } N : B}\ (\texttt{E}\otimes)$$

$$\frac{\Gamma; \Delta \vdash M : A \quad \alpha \notin FV(\Gamma) \cup FV(\Delta)}{\Gamma; \Delta \vdash M : \forall \alpha.A}\ (\texttt{I}\forall) \qquad \frac{\Gamma; \Delta \vdash M : \forall \alpha.A}{\Gamma; \Delta \vdash M : A[B/\alpha]}\ (\texttt{E}\forall)$$

Figure 2.1: Typing rules for DLAL

with standard β-reduction $(\lambda x.M)\ N \to_\beta M[N/x]$, where $[N/x]$ is the standard substitution.

The types of DLAL are given by the following grammar

$$A, B ::= \alpha \mid A \multimap B \mid A \otimes B \mid A \Rightarrow B \mid \S A \mid \forall \alpha.A.$$

A judgment in DLAL is of the form $\Gamma; \Delta \vdash M : A$ and means that term M has type A under the disjoint typing environments Γ and Δ. Δ is the affine typing environment for variables, meaning that a variable in Δ occurs at most once in a term, whereas Γ is the non-linear typing environment for variables.

Typing rules for DLAL are given as a natural deduction-like system in Figure 2.1.

Example 2.1.1 (From [BT09]). *Church numerals can be encoded as $\lambda f.\lambda x.f(\ldots(f\ x)\ldots)$ in DLAL and can be given the type $\mathtt{Nat} = \forall\alpha.(\alpha \multimap \alpha) \Rightarrow \S(\alpha \multimap \alpha)$ using the following derivation.*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{;x:\alpha\vdash x:\alpha}\,(\mathtt{Var}) \quad \dfrac{}{;f_1:\alpha\multimap\alpha\vdash f_1:\alpha\multimap\alpha}\,(\mathtt{Var})}{;x:\alpha, f_1:\alpha\multimap\alpha\vdash f_1\ x:\alpha}\,(\mathtt{E}\multimap) \quad \vdots}{\vdots \qquad\qquad \vdots}\,(\mathtt{E}\multimap)}{;x:\alpha,\forall i, f_i:\alpha\multimap\alpha\vdash f_n\ (\ldots(f_1\ x)\ldots):\alpha}\,(\mathtt{E}\multimap)}{;\forall i, f_i:\alpha\multimap\alpha\vdash \lambda x.f_n\ (\ldots(f_1\ x)\ldots):\alpha\multimap\alpha}\,(\mathtt{I}\multimap)}{\forall i, f_i:\alpha\multimap\alpha;\vdash \lambda x.f_n\ (\ldots(f_1\ x)\ldots):\S(\alpha\multimap\alpha)}\,(\mathtt{I}\S)}{\vdots}\,(\mathrm{Cntr})}{f:\alpha\multimap\alpha;\vdash \lambda x.f\ (\ldots(f\ x)\ldots):\S(\alpha\multimap\alpha)}\,(\mathrm{Cntr})}{;\vdash \lambda f.\lambda x.f\ (\ldots(f\ x)\ldots):(\alpha\multimap\alpha)\Rightarrow\S(\alpha\multimap\alpha)}\,(\mathtt{I}\Rightarrow)}{;\vdash \lambda f.\lambda x.f\ (\ldots(f\ x)\ldots):\forall\alpha.(\alpha\multimap\alpha)\Rightarrow\S(\alpha\multimap\alpha)}\,(\mathtt{I}\forall)$$

Using the type Nat*, the following types can be derived for addition and multiplication over Church's numerals.*

$$\begin{aligned}
\mathtt{add} &= \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)\\
\mathtt{add} &: \mathtt{Nat} \multimap \mathtt{Nat} \multimap \mathtt{Nat}\\
\mathtt{mult} &= \lambda m.\lambda n.n\ (\lambda y.\mathtt{add}\ m\ y)\ \lambda f.\lambda x.x\\
\mathtt{mult} &: \mathtt{Nat} \Rightarrow \mathtt{Nat} \multimap \S\mathtt{Nat}
\end{aligned}$$

For a typing derivation π of a term M, define its depth $d(\pi)$ to be the maximal number of premises of § introduction rules (I§) and the number of right-hand-side premises of $\Rightarrow$ elimination rules (E$\Rightarrow$) in each branch of π.

The polynomial upper bound on the number of reduction steps can be formalized as follows.

Theorem 2.1.1 (Polynomial time soundness of DLAL [BT04]). *Given a term M of typing derivation π, for any reduction strategy, M reduces to its normal form in at most $O(|M|^{2^{d(\pi)}})$ reduction steps.*

Let W be the type for binary words defined as $W = \forall\alpha.(\alpha \multimap \alpha) \Rightarrow (\alpha \multimap \alpha) \Rightarrow \S(\alpha \multimap \alpha)$. For a binary word $w \in \{0,1\}^*$, let $\underline{w}$ be a term of type W encoding w. A function f is computed by the term M such that $;\vdash M : \S^k W$ if, $\forall w \in \{0,1\}^*$, $M\underline{w}$ reduces to $\underline{f(w)}$.

Completeness can be stated as follows.

Theorem 2.1.2 (Polynomial time completeness of DLAL [BT04]). *For any function f in* FP*, there exist a term M and a constant k such that $;\vdash M : W \multimap \S^k W$ and M computes f.*

Notice that the presented version of DLAL includes a tensor product that can be trivially withdrawn if we want to compare the DLAL system with the LAL system of Figure 1.1. Moreover, the intensionality of DLAL is strictly smaller than the one of LAL.

Proposition 2.1.1.

$$\text{DLAL} \subsetneq \text{LAL}.$$

The inclusion DLAL $\subseteq$ LAL is proved using the transformation $(A \Rightarrow B)^* :=!(A)^* \multimap (B)^*$ and the strict inclusion follows from the fact that some types of LAL cannot be derived in DLAL.

If one focuses on types rather than algorithms then DLAL can be shown to be equivalent to LAL, as proved in [Bai08] using the translation $(-)^\bullet$ from LAL to DLAL, commuting with all connectives distinct from !, and defined by:

$$(!A)^\bullet := \forall\alpha.((A)^\bullet \Rightarrow \alpha) \multimap \alpha, \text{provided that } \alpha \notin A.$$

In particular, types of the shape $A \multimap !B$ can be removed as discussed in [BT09], where it is stated that *"[This removal] does not cause much loss of expressiveness in practice, since the standard decomposition of intuitionistic logic by linear logic does not use types of the form $A \multimap !B$"*.

The expressive power of these two systems is rather modest but DLAL allows the programmer to encode natural algorithms on lists of type A, encoded as $\forall\alpha.(A \multimap \alpha \multimap \alpha) \Rightarrow \S(\alpha \multimap \alpha)$, such as insertion sort. Consequently, because of its good properties (subject reduction and polynomial time reduction), DLAL is a good candidate in terms of expressive power for light logics.

2.1.2 Soft linear logic

For soft logics, a similar issue was solved in [GRDR07]: the authors introduced a type system, named Soft Type Assignment (STA), ensuring subject-reduction and a polynomial time upper bound on β-reduction, that we present in Figure 2.2. In the ($\multimap$ L) and (cut) rules, Γ and Δ are disjoint and y is fresh. Types σ and linear types A are defined by the following grammar:

$$A, B ::= \alpha \mid \sigma \multimap A \mid \forall\alpha.A,$$
$$\sigma, \tau ::= A \mid !\sigma.$$

Types are defined in such a way to prevent ! from occurring in a positive occurrence of a linear arrow.

For a typing derivation π of a term M, define its depth $d(\pi)$ to be the maximal number of applications of the rule (sp) in each branch of π.

The STA system ensures polynomial time normalization.

Theorem 2.1.3 (Polynomial time soundness of STA [GRDR07]). *Given a term M of typing derivation π, M reduces to its normal form in at most $O(|M|^{d(\pi)+1})$ reduction steps.*

Let B be the encoding $\forall\alpha.(\alpha \multimap \alpha \multimap \alpha)$ for the Boolean numbers and S be the encoding for strings of Boolean numbers.

Theorem 2.1.4 (Polynomial time completeness of STA [GRDR07]). *For any function f in* FP *and any polynomial P of degree n such that f is computable by a TM in time $O(P)$, there exist a term M and $k \leq n+2$ such that $;!^k S \vdash M : B$ and M computes f.*

$$\frac{}{x : A \vdash x : A}\ (\texttt{Var})$$

$$\frac{\Gamma, x : \sigma \vdash M : A}{\Gamma \vdash \lambda x.M : \sigma \multimap A}\ (\multimap\text{R}) \qquad \frac{\Gamma \vdash M : \tau \quad x : A, \Delta \vdash N : \sigma}{\Gamma, y : \tau \multimap A\Delta \vdash N[(y\ M)/x] : \sigma}\ (\multimap\text{L}\ )$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma, x : A \vdash M : \sigma}\ (\text{weak}) \qquad \frac{\Gamma \vdash M : A \quad \Delta, x : A \vdash N : \sigma}{\Gamma, \Delta \vdash N[M/x] : \sigma}\ (\text{cut})$$

$$\frac{\Gamma \vdash M : \sigma}{!\Gamma \vdash M :!\sigma}\ (\text{sp}) \qquad \frac{\Gamma x_1 : \tau, \ldots, x_n : \tau \vdash M : \sigma}{\Gamma, x :!\tau \vdash M[x/x_1, \ldots, x/x_n] : \sigma}\ (\text{mult})$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall\alpha.A}\ (\forall\text{R}) \qquad \frac{\Gamma, x : A[B/\alpha] \vdash M : \sigma}{\Gamma : \forall\alpha.A \vdash M : \sigma}\ (\forall\text{L})$$

Figure 2.2: Typing rules for STA

Again STA is a subsystem of SLL, which means that STA $\subsetneq$ SLL. As in the case of LAL, the inclusion is strict as counter-examples can be derived.

The following result about the incomparability of STA and DLAL also holds.

Proposition 2.1.2 (From comments in [GRDR07])**.** STA *and* DLAL *are incomparable.*

2.2 Interpretations and term rewrite system

In the following sections of this chapter, we will discuss the results obtained on the expressive power of interpretation tools for studying the complexity of TRSs. For that purpose, let us first recall some basic and preliminary notions on TRSs and interpretations.

2.2.1 Term rewrite systems as a computational model

A TRS (Term Rewriting System or Term Rewrite System) is a formal system for manipulating terms over a signature by means of rules (See [BN99] for a more detailed introduction to TRS). Terms are strings of symbols consisting of a countably infinite set of variables $\mathcal{X}$ and a first order signature Σ, a non-empty set of symbols $\mathtt{b}$ of fixed arity $\mathtt{ar}(\mathtt{b})$. $\mathcal{X}$ and Σ are supposed to be disjoint. As usual, the notation $\mathcal{T}(\Sigma, \mathcal{X})$ will be used to denote the set of terms $s, t, \ldots$ of signature Σ and having variables in $\mathcal{X}$. Moreover, let $\mathbb{V}(t)$ denote the set of variables occurring in the term t.

A (one-hole) context $C[\diamond]$ is a term in $\mathcal{T}(\Sigma \cup \{\diamond\}, \mathcal{X})$ with exactly one occurrence of the hole $\diamond$, a symbol of arity 0. Given a term t and context $C[\diamond]$, let $C[t]$ denote the result of replacing the hole $\diamond$ with the term t.

A substitution σ is a mapping from $\mathcal{X}$ to $\mathcal{T}(\Sigma, \mathcal{X})$.

A rewrite rule for a signature Σ is a pair $l \to r$ of terms $l, r \in \mathcal{T}(\Sigma, \mathcal{X})$. A TRS is as a pair $\langle \Sigma, \mathcal{R} \rangle$ of a signature Σ and a set of rewrite rules $\mathcal{R}$. Using the convention of [K+01], for each rewrite rule $l \to r$ of a TRS, all the variables of the right-hand side r are assumed to be included in the variables of l, *i.e.* $\mathbb{V}(r) \subseteq \mathbb{V}(l)$.

A constructor TRS is a TRS $\langle \Sigma, \mathcal{R} \rangle$ in which the signature Σ can be partitioned into the disjoint union $\mathcal{C} \uplus \mathcal{F}$ of a set of function symbols $\mathcal{F}$ and a set of constructors $\mathcal{C}$, such that for every rewrite rule $l \to r \in \mathcal{R}$, $l = \mathtt{f}(p_1, \cdots, p_n)$ where $\mathtt{f} \in \mathcal{F}$ and where $p_1, \cdots, p_n$ are terms in $\mathcal{T}(\mathcal{C}, \mathcal{X})$, called patterns. The constructors are introduced to represent inductive data. They basically consist of a strict subset $\mathcal{C} \subset \Sigma$ of non-defined functions (a function symbol is defined if it is the root of a left-hand side term in a rule).

In what follows, we will consider *orthogonal constructor* TRS that are TRS computing functions. The notion of orthogonality requires that reduction rules of the system are all left-linear, that is each variable occurs only once on the left hand side of each rule, and there is no overlap between patterns. It is a sufficient condition to ensure that the considered TRS is confluent. It implies that an orthogonal TRS computes a fixed (partial) function, mapping input terms to an output term (and not a function mapping input terms to a set of terms in the case of non-confluent systems). This syntactic requirement could have been withdrawn in favor of a semantics restriction that would only consider TRS that compute functions. Notice that, contrarily to the completeness results, the soundness complexity results presented in the remaining sections of this chapter still hold for non-confluent TRS.

Given two terms s and t, we have that $s \to_{\mathcal{R}} t$ if there are a substitution σ, a context $C[\diamond]$ and a rule $l \to r \in \mathcal{R}$ such that $s = C[l\sigma]$ and $t = C[r\sigma]$. Throughout the manuscript, let $\to_{\mathcal{R}}^*$ ($\to_{\mathcal{R}}^+$, respectively) be the reflexive and transitive (transitive, respectively) closure of $\to_{\mathcal{R}}$. Moreover we write $s \to_{\mathcal{R}}^n t$ if n rewrite steps are performed to rewrite s to t. A TRS terminates if there is no infinite reduction through $\to_{\mathcal{R}}$.

A function symbol $\mathtt{f}$ of arity n will define a partial function $[\![\mathtt{f}]\!]$ from constructor terms (sometimes called values) $\mathcal{T}(\mathcal{C})^n$ to $\mathcal{T}(\mathcal{C})$ by:[19]

$$\forall v_1, \cdots, v_n \in \mathcal{T}(\mathcal{C}),\ [\![\mathtt{f}]\!](v_1, \cdots, v_n) = v \text{ if and only if } \mathtt{f}(v_1, \cdots, v_n) \to_{\mathcal{R}}^* v \ \wedge\ v \in \mathcal{T}(\mathcal{C})$$

In this case, we write $[\![\mathtt{f}]\!](v_1, \cdots, v_n) \downarrow$ to mean that the computation ends in a normal form (constructor term). If there is no such a v (because of divergence or because evaluation cannot reach a constructor term), then $[\![\mathtt{f}]\!](v_1, \cdots, v_n) \uparrow$. A TRS computes the function f if it contains a function symbol $\mathtt{f}$ such that $[\![\mathtt{f}]\!] = f$. Finally, we define the notion of size of a term $|e|$ which is equal to the number of symbols in e.

Example 2.2.1. *Consider the following simple orthogonal TRS:*

$$\begin{aligned}
\mathtt{double}(\mathtt{0}) &\to \mathtt{0} \\
\mathtt{double}(\mathtt{x}+\mathtt{1}) &\to ((\mathtt{double}(\mathtt{x})+\mathtt{1})+\mathtt{1}) \\
\mathtt{exp}(\mathtt{0}) &\to \underline{\mathtt{1}} \\
\mathtt{exp}(\mathtt{x}+\mathtt{1}) &\to \mathtt{double}(\mathtt{exp}(\mathtt{x})),
\end{aligned}$$

where $\underline{\mathtt{n}}$ *is a shorthand notation for* $(\ldots(\mathtt{0}+\mathtt{1})\ldots)+\mathtt{1}$, $\mathtt{n}$ *times. For this particular TRS,* $\mathcal{F} = \{\mathtt{double}, \mathtt{exp}\}$, $\mathcal{C} = \{\mathtt{0}, +\mathtt{1}\}$, *and* $\mathtt{x} \in \mathcal{X}$. *The function symbol* `double` *computes the total function* $[\![\mathtt{double}]\!] : \mathcal{T}(\mathcal{C}) \to \mathcal{T}(\mathcal{C})$ *(over unary numbers) defined as* $[\![\mathtt{double}]\!](\underline{\mathtt{n}}) = \underline{\mathtt{2n}}$. *The function symbol* `exp` *computes the total function* $[\![\mathtt{exp}]\!] : \mathcal{T}(\mathcal{C}) \to \mathcal{T}(\mathcal{C})$ *defined as* $[\![\mathtt{exp}]\!](\underline{\mathtt{n}}) = \underline{2^{\mathtt{n}}}$.

2.2.2 Interpretation methods

Interpretation methods were introduced in [MN70, Lan79] and their methodology is as follows. Choose a suitable interpretation domain with some good properties, for example, functions over

[19] where $\mathcal{T}(\mathcal{C}) = \mathcal{T}(\mathcal{C}, \emptyset)$.

natural numbers and well-foundedness. Check a criterion on the program (or TRS) to ensure the required property. For example, a strict decrease (stable by context and substitution) for each reduction and termination, respectively.

The initial goal of interpretation methods was to provide a certificate of termination. Consequently, interpretation methods were mostly studied on well-founded structures such as natural numbers. In the remainder of this Section, let $(\mathbb{K}, \geq_{\mathbb{K}}^{wf})$ be an ordered set such that $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$ and let $\geq_{\mathbb{K}}^{wf}$ be a total order over $\mathbb{K}$ whose corresponding strict order $>_{\mathbb{K}}^{wf}$ is well-founded. Moreover, let $\geq_{\mathbb{K}}$ (and $>_{\mathbb{K}}$) be the standard ordering over $\mathbb{K}$. We will sometimes omit the subscript $\mathbb{K}$ when it is clear from the context.

Definition 2.2.1 (Assignment)**.** *Given a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, an assignment $[-]_{\mathbb{K}}$ over $\mathbb{K}$ maps:*

- *every variable $\mathtt{x} \in \mathbb{V}$ to a variable $[\mathtt{x}]_{\mathbb{K}}$ in $\mathbb{K}$,*
- *every symbol $\mathtt{b} \in \mathcal{C} \uplus \mathcal{F}$ to a total function $[\mathtt{b}]_{\mathbb{K}} : \mathbb{K}^{\mathtt{ar(b)}} \to \mathbb{K}$.*

Definition 2.2.2 (Strict monotonicity)**.** *An assignment $[-]_{\mathbb{K}}$ is strictly monotonic if for every symbol $\mathtt{b}$ of arity $ar(\mathtt{b}) > 0$, $[\mathtt{b}]_{\mathbb{K}}$ is a monotonic function in each of its arguments, i.e. $\forall i \in [1, ar(b)]$,*

$$\forall X, Y \in \mathbb{K}, X >_{\mathbb{K}}^{wf} Y \implies [\mathtt{b}]_{\mathbb{K}}(\ldots, X_{i-1}, X, X_{i+1}, \ldots) >_{\mathbb{K}}^{wf} [\mathtt{b}]_{\mathbb{K}}(\ldots, X_{i-1}, Y, X_{i+1}, \ldots).$$

Definition 2.2.3 (Interpretation)**.** *Given a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, an interpretation over $\mathbb{K}$ is a* strictly *monotonic assignment $[-]_{\mathbb{K}}$ such that*

$$\forall l \to r \in \mathcal{R},\ [l]_{\mathbb{K}} >_{\mathbb{K}}^{wf} [r]_{\mathbb{K}},$$

where the interpretation $[-]_{\mathbb{K}}$ is extended canonically to terms as usual and $[l]_{\mathbb{K}} >_{\mathbb{K}}^{wf} [r]_{\mathbb{K}}$ means that if $\mathbb{V}(l) = \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ then $\forall X_1, \ldots, X_n \in \mathbb{K}$, $[l]_{\mathbb{K}}(X_1, \ldots, X_n) >_{\mathbb{K}}^{wf} [r]_{\mathbb{K}}(X_1, \ldots, X_n)$ where, for a given term t, $[t]_{\mathbb{K}}$ is the function mapping $([\mathtt{x}_1]_{\mathbb{K}}, \ldots, [\mathtt{x}_n]_{\mathbb{K}})$ to $[t]_{\mathbb{K}}$.

As demonstrated in [Lan79], an interpretation defines a reduction ordering (*i.e.* a strict, stable, monotonic, and well-founded ordering).

Theorem 2.2.1. *If a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$ admits an interpretation over $\mathbb{K}$ then it terminates.*

For $\mathbb{K} = \mathbb{N}$, the above definition constitutes the basis of interpretation method as introduced in [MN70, Lan79], where $\geq_{\mathbb{N}}^{wf}$ is taken to be the standard ordering $\geq_{\mathbb{N}}$ on natural numbers.

Example 2.2.2. *The TRS of Example 2.2.1:*

$$\begin{array}{ll} \mathtt{double(0)} \to \mathtt{0} & \mathtt{exp(0)} \to \underline{\mathtt{1}} \\ \mathtt{double(x+1)} \to \mathtt{double(x)+2} & \mathtt{exp(x+1)} \to \mathtt{double(exp(x))} \end{array}$$

admits the following interpretation over $\mathbb{N}$:

$$\begin{array}{rr} [\mathtt{0}]_{\mathbb{N}} = 0, & [\mathtt{+1}]_{\mathbb{N}}(X) = X + 1 \\ [\mathtt{double}]_{\mathbb{N}}(X) = 3 \times X + 1, & [\mathtt{exp}]_{\mathbb{N}}(X) = 3^{2 \times X + 1}. \end{array}$$

Indeed, it is a strictly monotonic assignment and we have a strict inequality for each rule. In particular, for the last rule, we have:

$$\begin{aligned} &[\mathtt{exp(x+1)}]_{\mathbb{N}} = 3^{2 \times [(\mathtt{x+1})]_{\mathbb{N}} + 1} = 3^{2(X+1)+1} = 3^{2X+3} \\ &>_{\mathbb{N}} 3 \times 3^{2X+1} + 1 = [\mathtt{double}]_{\mathbb{N}}(3^{2X+1}) = [\mathtt{double}]_{\mathbb{N}}([\mathtt{exp(x)}]_{\mathbb{N}}) = [\mathtt{double(exp(x))}]_{\mathbb{N}}. \end{aligned}$$

In the case where $\mathbb{K} = \mathbb{R}^+$ (or $\mathbb{Q}^+$), $\geq^{wf}_{\mathbb{R}^+}$ cannot be defined to be the natural ordering $\geq_{\mathbb{R}^+}$ on real numbers as it is well-known that $>$ is not well-founded on such a domain. This issue was solved by Lucas in [Luc07] by defining $\geq^{wf}_{\mathbb{R}^+}$ as follows:

$$\forall X, Y \in \mathbb{R}^+, X \geq^{wf}_{\mathbb{R}^+} Y \text{ if and only if } X \geq_{\mathbb{R}^+} Y + \delta,$$

for some fixed real number $\delta > 0$. Throughout the manuscript, we will sometimes use the notation $\geq_\delta$ or $>_\delta$ to refer explicitly to the order for a given δ. Interpretations over real numbers were initially introduced by Dershowitz in [Der79]. They were required to have a subterm property to compensate for the loss of well-foundedness. A interpretation has the strict subterm property if for any symbol $\mathtt{b}$,

$$\forall i \in [1, ar(\mathtt{b})], \forall X_i \in \mathbb{R}^+,\ [\mathtt{b}]_{\mathbb{R}^+}(X_1, \cdots, X_{ar(\mathtt{b})}) >_{\mathbb{R}^+} X_i.$$

It is worth mentioning that both definitions entail that the considered functions have a derivative strictly greater than 1 as the functions are strictly monotonic. However Lucas has demonstrated in [Luc07] that his notion captures more algorithms than the one in [Der79].

It is natural to restrict the space of considered functions (the interpretation codomain) to polynomials for at least two reasons. First, considering the whole space of functions is too general from a computability perspective. Second, polynomials are admitted to be a relevant set of functions in term of time and space complexity.

In what follows, let $\mathbb{K}[X_1, \ldots, X_n]$ be the set of n-variable polynomials whose coefficients are in $\mathbb{K}$.

Definition 2.2.4 (Polynomial interpretation)**.** *Given a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, an interpretation $[-]_{\mathbb{K}}$ is a polynomial interpretation if for every symbol $\mathtt{b} \in \mathcal{C} \uplus \mathcal{F}$, $[\mathtt{b}]_{\mathbb{K}} \in \mathbb{K}[X_1, \ldots, X_{ar(\mathtt{b})}]$.*

In particular, as mentioned in Section 1.3.2, the synthesis problem is decidable in exponential time for polynomial interpretations over $\mathbb{R}^+$ [BMMP05] whereas Hilbert's tenth problem can be reduced to it, for $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+\}$, as demonstrated in [Péc13].

The synthesis problem for polynomial interpretation has been studied in [CMTU05, Luc07] where algorithms solving the constraints are described. More recently, encoding-based algorithms via SAT or SMT solving have become the state of the art for solving the synthesis problem [FGM+07, BLO+12].

However, as highlighted in [BCMT98, BCMT01], the restriction to polynomial interpretation is not enough if one wants to study polynomial time.

Example 2.2.3. *The following TRS*

$$\mathtt{explode(0)} \to \mathtt{nil} \qquad \mathtt{explode(x+1)} \to \mathtt{c(explode(x), explode(x))}$$

computes a well-balanced binary tree of size 2^n, for any unary input $\underline{n}$, and the corresponding computed function cannot be in FP*, but it admits the following polynomial interpretation over $\mathbb{N}$:*

$$[\mathtt{0}]_{\mathbb{N}} = 1, \qquad [\mathtt{+1}]_{\mathbb{N}}(\mathtt{X}) = 2X + 1, \qquad [\mathtt{nil}]_{\mathbb{N}} = 0,$$
$$[\mathtt{c}]_{\mathbb{N}}(X, Y) = X + Y, \qquad [\mathtt{explode}]_{\mathbb{N}}(X) = X.$$

This has led to the development of another restriction, called additivity, on the interpretation of constructor symbols.

Definition 2.2.5 (Additivity). *An assignment is* additive *if* $\forall \mathtt{c} \in \mathcal{C}$,

$$[\mathtt{c}]_{\mathbb{K}}(X_1, \ldots, X_{ar(\mathtt{c})}) = \begin{cases} \Sigma_{i=1}^{n} X_i + k_{\mathtt{c}}, \ \textit{with } k_{\mathtt{c}} \geq_{\mathbb{K}} 1, \ \textit{if } ar(\mathtt{c}) > 0, \\ 0, \ \textit{otherwise.} \end{cases}$$

An interpretation is additive if it is an additive assignment.

The desirable property of additive interpretations is that the interpretation of a value is in Θ of its size.

Lemma 2.2.1. *Given an additive interpretation* $[-]_{\mathbb{K}}$ *of a TRS, there is a constant* $k \in \mathbb{N} - \{0\}$ *such that, for any* $v \in \mathcal{T}(\mathcal{C})$*:*

$$|v| \leq_{\mathbb{K}} [v]_{\mathbb{K}} \leq_{\mathbb{K}} k \times |v|.$$

Consequently, it allows to bound from above (and also below) the interpretation of a function symbol call on values as $[\mathtt{f}(v)]_{\mathbb{K}} \leq_{\mathbb{K}} [\mathtt{f}]_{\mathbb{K}}(k \times |v|)$, by monotonicity of interpretations.

Let $I_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}^{wf})$ be the set of TRS that admits an additive polynomial interpretation over $\mathbb{K}$.

Theorem 2.2.2. *For* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $[\![I_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}^{wf})]\!] = \mathtt{FP}$.[20]

One important question is what is the best structure, $\mathbb{N}, \mathbb{Q}^+$ or $\mathbb{R}^+$, in term of expressive power to consider to study polynomial interpretations. As $\mathbb{N} \subsetneq \mathbb{Q}^+ \subsetneq \mathbb{R}^+$ and $\geq_{\mathbb{N}} \subsetneq \geq_1$, the counter-examples provided in [Luc07, NM10] provide the following strict inclusions.

Theorem 2.2.3. $I_{add}^{poly}(\mathbb{N}, \geq_{\mathbb{N}}) \subsetneq I_{add}^{poly}(\mathbb{Q}^+, \geq_{\mathbb{Q}^+}^{wf}) \subsetneq I_{add}^{poly}(\mathbb{R}^+, \geq_{\mathbb{R}^+}^{wf})$.

It is worth mentioning that the consideration of non-well-founded domains such as $\mathbb{R}^+$ or $\mathbb{Q}^+$ was motivated by the lack of expressive power of polynomial interpretations as the upper bound obtained on the derivational complexity, *i.e.* the maximal number of reduction steps, of programs is most of the time too rough.

Example 2.2.4. *Consider the function symbol* `double` *of Example 2.2.1. Finding an additive and polynomial interpretation of* `double` *over* $\mathbb{N}$ *consists in finding a function symbol* $f : \mathbb{N} \to \mathbb{N}$ *satisfying the following system of inequalities:*

$$f(0) >_{\mathbb{N}} 0,$$
$$\forall X \in \mathbb{N}, \ f(X + k) >_{\mathbb{N}} f(X) + 2k, \ \textit{with } k \geq_{\mathbb{N}} 1.$$

It can be shown easily that the smallest function in $\mathbb{N}[X]$ *that is a solution of this system is* $f(X) = 3X + 1$*. This bound is rough both from a time and space point of view as the derivational complexity (maximal number of reduction steps with respect to an upper bound on the input size) of this function is linear whereas its space consumption is* $2n$ *on a unary input of size* n*.*

It can be given a tighter interpretation over $\mathbb{Q}^+$*:* $f(X) = (2 + \delta)X + \delta$*, for* $\delta \in \mathbb{Q}^+$ *and such that* $\delta < 1$*.*

Bonfante, Deloup and Henrot [BD10, BDH15] have generalized this result by applying the *Positivstellensatz* Theorem to interpretations over $\mathbb{R}^+[X_1, \ldots, X_n]$ using the standard strict order $>$ over $\mathbb{R}^+$: they have demonstrated that the strict monotonicity with respect to $>_\delta$ is entailed by such interpretations and they recovered a characterization of `FP` using additive and polynomial interpretations over real numbers.

However the extensions to real and rational numbers are not fully satisfactory as they still fail to capture a huge number of programs. One idea was to relax the well-foundedness, only keeping track of size upper bounds: this has led to the development of quasi-interpretations.

[20] The encoding is using TRSs over unary numbers.

2.3 Quasi-interpretation

2.3.1 Motivations, definition, and basic properties

In [MM00, BMM01], the notion of quasi-interpretation is introduced by relaxing the strict decrease in the definition of interpretations. Well-foundedness and, hence, termination are lost and this allows us to consider (non-strictly) increasing functions.

Definition 2.3.1 (Monotonicity). *An assignment $[-]_{\mathbb{K}}$ is monotonic if for every symbol $\mathtt{b}$ of arity $ar(\mathtt{b}) > 0$, $[\mathtt{b}]_{\mathbb{K}}$ is a monotonic function in each of its arguments, i.e. $\forall i \in [1, ar(\mathtt{b})]$,*

$$\forall X, Y \in \mathbb{K},\ X \geq_{\mathbb{K}} Y \implies [\mathtt{b}]_{\mathbb{K}}(\ldots, X_{i-1}, X, X_{i+1}, \ldots) \geq_{\mathbb{K}} [\mathtt{b}]_{\mathbb{K}}(\ldots, X_{i-1}, Y, X_{i+1}, \ldots).$$

Some important space information about the biggest size of a intermediate computed value (any value computed during the evaluation of a term) is still kept, provided that the interpretation enjoys some property, called *subterm property*, a relaxed variation of Dershowitz's subterm property for interpretation over real numbers.

Definition 2.3.2 (Subterm). *An assignment $[-]_{\mathbb{K}}$ is subterm if for every symbol $\mathtt{b}$ of arity $ar(\mathtt{b}) > 0$:*

$$\forall i \in [1, ar(\mathtt{b})],\ \forall X_i \in \mathbb{K},\ [\mathtt{b}]_{\mathbb{K}}(X_1, \ldots, X_n) \geq_{\mathbb{K}} X_i.$$

We are now ready to introduce the notion of quasi-interpretation that is studied more deeply in the survey [BMM11].

Definition 2.3.3 (Quasi-interpretation). *Given a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, a (polynomial or additive, respectively) quasi-interpretation is a monotonic and subterm (polynomial or additive, respectively) assignment $[-]_{\mathbb{K}}$ over $\mathbb{K}$ satisfying:*

$$\forall l \to r \in \mathcal{R},\ [l]_{\mathbb{K}} \geq_{\mathbb{K}} [r]_{\mathbb{K}},$$

where the quasi-interpretation $[-]_{\mathbb{K}}$ is extended canonically to terms.

Contrarily to (polynomial) interpretations, quasi-interpretations can deal with partial functions as they do not imply termination.

Example 2.3.1. *Consider the following TRS:*

$$\mathtt{f(x+2)} \to \mathtt{f(x)+2}, \qquad \mathtt{f(0)} \to \mathtt{f(0)}, \qquad \mathtt{f(1)} \to \mathtt{1}.$$

The function $\mathtt{f}$ computes the identity function on odd numbers whereas it diverges on even numbers. However it admits the trivial polynomial and additive quasi-interpretation $[-]_{\mathbb{K}}$ defined by $[\mathtt{f}]_{\mathbb{K}}(X) = X$, $[+\mathtt{1}]_{\mathbb{K}}(X) = X + 1$, and $[\mathtt{0}]_{\mathbb{K}} = 0$.

Indeed, for the first rule, we check:

$$\begin{aligned}[\mathtt{f(x+2)}]_{\mathbb{K}} &= [\mathtt{f}]_{\mathbb{K}}([(\mathtt{x}+1)+1)]_{\mathbb{K}}) = X + 2\\ &\geq_{\mathbb{K}} [\mathtt{f(x)}]_{\mathbb{K}} + 2 = [(\mathtt{f(x)}+1)+1]_{\mathbb{K}}.\end{aligned}$$

For the second, rule we clearly have $[\mathtt{f(0)}]_{\mathbb{K}} \geq [\mathtt{f(0)}]_{\mathbb{K}}$ and, for the last rule, we have $[\mathtt{f(1)}]_{\mathbb{K}} = [\mathtt{f}]_{\mathbb{K}}([\mathtt{1}]_{\mathbb{K}}) \geq [\mathtt{1}]_{\mathbb{K}}$.

2.3.2 Intensional properties of quasi-interpretations

By monotonicity and subterm properties, quasi-interpretations allow us to bound the interpretation of any intermediate value.

Lemma 2.3.1. *Given a TRS admitting a quasi-interpretation* $[-]_{\mathbb{K}}$ *over* $\mathbb{K}$*, for any term* t*, for any value* $v \in \mathcal{T}(\mathcal{C})$*, and any context* $C[\diamond]$ *such that* $t \to_{\mathcal{R}}^{*} C[v]$*,*

$$[t]_{\mathbb{K}} \geq_{\mathbb{K}} [v]_{\mathbb{K}}.$$

Hence, in the case of a polynomial and additive quasi-interpretation, it turns out that any intermediate value u computed from a term $\mathtt{f}(v)$ has size bounded polynomially in the input size (*i.e.* $|u| \leq_{\mathbb{K}} [\mathtt{f}]_{\mathbb{K}}(k \times |v|)$ for some polynomial $[\mathtt{f}]_{\mathbb{K}}$) by combining the above Lemma with Lemma 2.2.1. The above Lemma was also satisfied by polynomial interpretations over $\mathbb{N}$ as a strictly increasing polynomial has to satisfy the subterm property.

Moreover, because of a relaxed monotonicity, the obtained quasi-interpretations are tighter to the effective space complexity of the studied TRS.

Example 2.3.2. *The program of Example 2.2.2*

$$\begin{aligned} &\mathtt{double}(\mathtt{0}) \to \mathtt{0} \\ &\mathtt{double}(\mathtt{x{+}1}) \to \mathtt{double}(\mathtt{x}){+}\mathtt{2} \end{aligned}$$

admits the following additive and polynomial quasi-interpretation over $\mathbb{N}$*:*

$$\begin{aligned} [\mathtt{0}]_{\mathbb{N}} &= 0, & [+\mathtt{1}]_{\mathbb{N}}(\mathtt{X}) &= X + 1, \\ [\mathtt{double}]_{\mathbb{N}}(X) &= 2 \times X, \end{aligned}$$

whereas we have already shown in Example 2.2.4 that the smallest admissible interpretation over $\mathbb{N}$ *is such that* $[\mathtt{double}]_{\mathbb{N}}(X) = 3 \times X + 1$*.*

Let $QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}})$ be the set of function symbols whose TRS admits an additive and polynomial interpretation over $\mathbb{K}$. We obtain a first intensionality result as the set of programs admitting an additive and polynomial interpretations is strictly included in the set of programs admitting an additive and polynomial quasi-interpretation:[21]

Theorem 2.3.1. *For* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^{+}, \mathbb{R}^{+}\}$*,* $I_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}^{wf}) \subsetneq QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}})$*.*

This result relies on the fact that strictly monotonic polynomials have the subterm property. It is worth mentioning that in the above Theorem the order differs for interpretations and quasi-interpretations on non-well-founded domains. For example, if $\mathbb{K} = \mathbb{R}^{+}$ then the result can be stated as follows $\forall \delta > 0$, $I_{add}^{poly}(\mathbb{R}^{+}, \geq_{\delta}) \subsetneq QI_{add}^{poly}(\mathbb{R}^{+}, \geq_{\mathbb{R}^{+}})$.

As quasi-interpretations do not focus on termination, domains such as $(\mathbb{R}^{+}, \geq_{\mathbb{R}^{+}})$ or $(\mathbb{Q}^{+}, \geq_{\mathbb{Q}^{+}})$ are good candidates to define quasi-interpretations. Moreover, non-strictly monotonic, subterm, and polynomially bounded functions such as the maximum can be added to the interpretation domain in order to increase the expressive power without breaking the soundness. This has led to the definition and study of the class of max-polynomials $\mathsf{MaxPoly}\{\mathbb{K}\}$, consisting in the class of functions containing all polynomials of $\mathbb{K}[X_1, \ldots, X_n]$ and closed under the *max* operation. The synthesis problem remains decidable in exponential time $\mathsf{MaxPoly}\{\mathbb{R}^{+}\}$, for bounded max

[21] This result remains true if additivity and polynomiality properties are withdrawn.

degree (number of operands) k, as any inequality involving the *max* operator can be transformed into at most k^2 inequalities with no max (see [BMMP05, Péc13] for a more detailed treatment).

Let $QI^{maxpoly}_{add}(\mathbb{K}, \geq_{\mathbb{K}})$ be the set of function symbols whose TRS admits an additive max-polynomial and additive interpretation over $\mathbb{K}$.

Theorem 2.3.2. *For* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $QI^{poly}_{add}(\mathbb{K}, \geq_{\mathbb{K}}) \subsetneq QI^{maxpoly}_{add}(\mathbb{K}, \geq_{\mathbb{K}})$.

The interest of considering quasi-interpretations over the reals does not only rely on the decidability of their synthesis contrarily to quasi-interpretations over $\mathbb{N}$ or $\mathbb{Q}^+$. Indeed, we have a result analog to Theorem 2.2.3 over MaxPoly quasi-interpretations. It states that there exist programs that do not have any quasi-interpretation over MaxPoly$\{\mathbb{Q}\}$ and, *a fortiori* MaxPoly$\{\mathbb{N}\}$, but that admit a quasi-interpretation over MaxPoly$\{\mathbb{R}^+\}$. This was demonstrated by contradiction in [Péc13] by introducing a TRS enforcing a multiplicative coefficient a of the interpretation to satisfy the equation $a^2 = 2$.

Theorem 2.3.3. $QI^{maxpoly}_{add}(\mathbb{N}, \geq_{\mathbb{N}}) \subseteq QI^{maxpoly}_{add}(\mathbb{Q}^+, \geq_{\mathbb{Q}^+}) \subsetneq QI^{maxpoly}_{add}(\mathbb{R}^+, \geq_{\mathbb{R}^+})$.

In the above Theorem, it is not known whether the first inclusion is strict or not.

2.3.3 Recursive path orderings

Theorem 2.3.3 is not a characterization. Indeed $QI^{maxpoly}_{add}(\mathbb{K}, \geq_{\mathbb{K}})$ is not a complexity class as it contains non-terminating programs. For capturing a complexity class, quasi-interpretation have to be complemented by some termination ordering in order to recover complexity results. This was done in [MM00, BMM01] and surveyed in [BMM11] by combining quasi-interpretations and recursive path ordering $\prec_{rpo}$ (and its reflexive closure $\preceq_{rpo}$) that we define in Figure 2.3 with respect to a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$. Let $\prec_{\mathcal{F}}$ be a strict ordering on $\mathcal{F}$ and $\approx_{\mathcal{F}}$ a compatible equivalence relation. Define $\preceq_{\mathcal{F}}$ by $\preceq_{\mathcal{F}} = \prec_{\mathcal{F}} \cup \approx_{\mathcal{F}}$. Let st be a mapping from $\mathcal{F}$ to $\{p, l\}$ compatible with $\preceq_{\mathcal{F}}$, *i.e.* $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$ implies that $st(\mathtt{f}) = st(\mathtt{g})$. st provides information on how to compare the arguments of two equivalent function symbols. This comparison can be product p (*i.e.* component by component) or lexicographic l.

A TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$ is oriented by $\prec_{rpo}$ if there is a status st and a precedence $\prec_{\mathcal{F}}$ such that for each rule $t \to_{\mathcal{R}} s \in \langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, $s \prec_{rpo} t$. For $A \subseteq \{p, l\}$, let RPO^A be the set of TRSs that can be oriented by $\prec_{rpo}$ in such a way that all function symbols have a status in A. As rule (Lexi) is implied by rule (Prod) in Figure 2.3, but not the converse, it holds that $RPO^{\{p\}} \subsetneq RPO^{\{l\}}$ and, consequently, $RPO^{\{p,l\}} = RPO^{\{l\}}$.

Example 2.3.3. *Consider the following TRS*

$$\mathtt{concat}(\mathtt{i}(\mathtt{x}), \mathtt{y}) \to \mathtt{i}(\mathtt{concat}(\mathtt{x}, \mathtt{y})), \ \mathtt{i} \in \{\mathtt{0}, \mathtt{1}\},$$
$$\mathtt{concat}(\epsilon, \mathtt{y}) \to \mathtt{y},$$

concatenating two binary words given as input. It can be oriented by $\prec_{rpo}$ *in*

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathtt{x} = \mathtt{x}}{\mathtt{x} \prec_{\mathtt{rpo}} \mathtt{i}(\mathtt{x})}\ (SubI) \quad \dfrac{\mathtt{y} = \mathtt{y}}{\mathtt{y} \preceq_{\mathtt{rpo}} \mathtt{y}}\ (RefI)}{\{\mathtt{x}, \mathtt{y}\} \prec^{\mathtt{p}}_{\mathtt{rpo}} \{\mathtt{i}(\mathtt{x}), \mathtt{y}\}}\ (Prod) \quad \dfrac{\dfrac{\mathtt{x} = \mathtt{x}}{\mathtt{x} \prec_{\mathtt{rpo}} \mathtt{i}(\mathtt{x})}\ (SubI)}{\mathtt{x} \prec_{\mathtt{rpo}} \mathtt{concat}(\mathtt{i}(\mathtt{x}), \mathtt{y})}\ (Sub) \quad \dfrac{\mathtt{y} = \mathtt{y}}{\mathtt{y} \prec_{\mathtt{rpo}} \mathtt{concat}(\mathtt{i}(\mathtt{x}), \mathtt{y})}\ (SubI)}{\mathtt{concat}(\mathtt{x}, \mathtt{y}) \prec_{\mathtt{rpo}} \mathtt{concat}(\mathtt{i}(\mathtt{x}), \mathtt{y})}\ (FEq)}{\mathtt{i}(\mathtt{concat}(\mathtt{x}, \mathtt{y})) \prec_{\mathtt{rpo}} \mathtt{concat}(\mathtt{i}(\mathtt{x}), \mathtt{y})}\ (Cons)$$

$$\frac{s = t \quad \mathtt{b} \in \mathcal{C} \uplus \mathcal{F}}{s \prec_{rpo} \mathtt{b}(\ldots, t, \ldots)}\ (\text{SubI}) \qquad \frac{s \prec_{rpo} t \quad \mathtt{b} \in \mathcal{C} \uplus \mathcal{F}}{s \prec_{rpo} \mathtt{b}(\ldots, t, \ldots)}\ (\text{Sub})$$

$$\frac{s = t}{s \preceq_{rpo} t}\ (\text{RefI}) \qquad \frac{\mathtt{f} \in \mathcal{F} \quad \mathtt{c} \in \mathcal{C} \quad \forall i, s_i \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n)}{\mathtt{c}(s_1, \ldots, s_m) \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n)}\ (\text{Cons})$$

$$\frac{s \prec_{rpo} t}{s \preceq_{rpo} t}\ (\text{Ref}) \qquad \frac{\mathtt{f}, \mathtt{g} \in \mathcal{F} \quad \mathtt{g} \prec_{\mathcal{F}} \mathtt{f} \quad \forall i, s_i \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n)}{\mathtt{g}(s_1, \ldots, s_n) \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n)}\ (\text{FLess})$$

$$\frac{\mathtt{f}, \mathtt{g} \in \mathcal{F} \quad \mathtt{g} \approx_{\mathcal{F}} \mathtt{f} \quad \forall i, s_i \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n) \quad \{s_1, \ldots, s_n\} \prec_{rpo}^{st(\mathtt{f})} \{t_1, \ldots, t_n\}}{\mathtt{g}(s_1, \ldots, s_n) \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n)}\ (\text{FEq})$$

$$\frac{\forall i,\ s_i \preceq_{rpo} t_i \quad \exists j,\ s_j \prec_{rpo} t_j}{\{s_1, \ldots, s_n\} \prec_{rpo}^{p} \{t_1, \ldots, t_n\}}\ (\text{Prod}) \qquad \frac{\exists j,\ \forall i < j,\ s_i \preceq_{rpo} t_i \quad s_j \prec_{rpo} t_j}{\{s_1, \ldots, s_n\} \prec_{rpo}^{l} \{t_1, \ldots, t_n\}}\ (\text{Lexi})$$

Figure 2.3: Recursive path ordering

and in

$$\frac{\mathtt{y} = \mathtt{y}}{\mathtt{y} \prec_{\mathtt{rpo}} \mathtt{concat}(\epsilon, \mathtt{y})}\ \textit{(SubI).}$$

Moreover all function symbols have a product status. Consequently, this TRS is in $RPO^{\{p\}}$.

Example 2.3.4. *Consider the following TRS*

$$\begin{aligned} \mathtt{step}(\mathtt{i}(\mathtt{x}), \mathtt{y}) &\to \mathtt{step}(\mathtt{x}, \mathtt{i}(\mathtt{y})),\ \mathtt{i} \in \{\mathtt{0}, \mathtt{1}\}, \\ \mathtt{step}(\epsilon, \mathtt{y}) &\to \mathtt{y}, \\ \mathtt{reverse}(\mathtt{x}) &\to \mathtt{step}(\mathtt{x}, \epsilon), \end{aligned}$$

computing the reverse binary word of its input. It can be oriented by $\prec_{rpo}$ using the precedence step $\prec_{\mathcal{F}}$ reverse in

$$\frac{\dfrac{\mathtt{x} = \mathtt{x}}{\mathtt{x} \prec_{\mathtt{rpo}} \mathtt{reverse}(\mathtt{x})}\ \textit{(SubI)} \qquad \dfrac{}{\epsilon \prec_{\mathtt{rpo}} \mathtt{reverse}(\mathtt{x})}\ \textit{(Cons)}}{\mathtt{step}(\mathtt{x}, \epsilon) \prec_{\mathtt{rpo}} \mathtt{reverse}(\mathtt{x})}\ \textit{(FLess),} \qquad \frac{\mathtt{y} = \mathtt{y}}{\mathtt{y} \prec_{\mathtt{rpo}} \mathtt{step}(\epsilon, \mathtt{y})}\ \textit{(SubI)}$$

and in

$$\frac{\dfrac{\dfrac{\mathtt{x} = \mathtt{x}}{\mathtt{x} \prec_{\mathtt{rpo}} \mathtt{i}(\mathtt{x})}\ \textit{(SubI)}}{\{\mathtt{x}, \mathtt{i}(\mathtt{y})\} \prec^{\mathtt{l}}_{\mathtt{rpo}} \{\mathtt{i}(\mathtt{x}), \mathtt{y}\}}\ \textit{(Lexi)} \qquad \dfrac{\dfrac{\mathtt{x} = \mathtt{x}}{\mathtt{x} \prec_{\mathtt{rpo}} \mathtt{step}(\mathtt{i}(\mathtt{x}), \mathtt{y})}\ \textit{(SubI)}}{\mathtt{i}(\mathtt{x}) \prec_{\mathtt{rpo}} \mathtt{step}(\mathtt{i}(\mathtt{x}), \mathtt{y})}\ \textit{(Cons)} \qquad \dfrac{\mathtt{y} = \mathtt{y}}{\mathtt{y} \prec_{\mathtt{rpo}} \mathtt{step}(\mathtt{i}(\mathtt{x}), \mathtt{y})}\ \textit{(SubI)}}{\mathtt{step}(\mathtt{x}, \mathtt{i}(\mathtt{y})) \prec_{\mathtt{rpo}} \mathtt{step}(\mathtt{i}(\mathtt{x}), \mathtt{y})}\ \textit{(FEq).}$$

Moreover, the status of step *is enforced to be lexicographic, i.e. to satisfy $st(step) = l$, as it does not hold that* $\mathtt{i}(\mathtt{y}) \prec_{\mathtt{rpo}} \mathtt{y}$. *Consequently, this TRS is in $RPO^{\{l\}}$ but not in $RPO^{\{p\}}$.*

Now we are ready to present the characterizations of FP and FPSPACE presented in [BMM11]. They consist in slight variations, improvements or simplifications of the results of [MM00, BMM01].

Theorem 2.3.4 ([BMM11]). *For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, we have:*

- $[\![QI_{add}^{maxpoly}(\mathbb{K}, \geq_{\mathbb{K}}) \cap RPO^{\{p\}}]\!] = \mathtt{FP}$,
- $[\![QI_{add}^{maxpoly}(\mathbb{K}, \geq_{\mathbb{K}}) \cap RPO^{\{l\}}]\!] = \mathtt{FPSPACE}$.

The characterization of FPSPACE is a direct application of Lemma 2.3.1 in the case of additive and (max-)polynomial interpretations. Indeed, in such a case, any value (and term) computed during the evaluation process has size bounded polynomially by the input size. The characterization of FP is a bit more subtle and needs the application of a memoization technique to show that only a polynomial number (in the input size) of recursive calls can be performed for a given function symbol.

Example 2.3.5. *The TRS of Example 2.3.3 admits the following additive and polynomial quasi-interpretation:*

$$\begin{aligned} [\mathtt{concat}]_{\mathbb{N}}(\mathtt{X}, \mathtt{Y}) &= X + Y, \\ [\mathtt{i}]_{\mathbb{N}}(X) &= X + 1, \ \mathtt{i} \in \{\mathtt{0}, \mathtt{1}\}, \\ [\epsilon]_{\mathbb{N}} &= 0. \end{aligned}$$

As it is in $RPO^{\{p\}}$, we can conclude, using Theorem 2.3.4, that the computed function is in FP.

Example 2.3.6. *The TRS of Example 2.3.4 admits the following additive and polynomial quasi-interpretation:*

$$\begin{aligned} [\mathtt{step}]_{\mathbb{N}}(\mathtt{X}, \mathtt{Y}) &= X + Y, \\ [\mathtt{i}]_{\mathbb{N}}(X) &= X + 1, \ \mathtt{i} \in \{\mathtt{0}, \mathtt{1}\}, \\ [\epsilon]_{\mathbb{N}} &= 0, \\ [\mathtt{reverse}]_{\mathbb{N}}(\mathtt{X}) &= X. \end{aligned}$$

As it is in $RPO^{\{l\}}$, we can conclude, using Theorem 2.3.4, that the computed function is in FPSPACE. *Notice that this result does not provide a good precision on the function complexity. Some other characterizations have been developed in [BMM11]. They manage in particular to show that* $[\![\mathtt{reverse}]\!]$ *belongs to* FP.

2.3.4 Interpretation vs quasi-interpretation

By Theorem 2.3.1, quasi-interpretations have strictly more expressive power than interpretations. However, the characterization of Theorem 2.3.4 does not provide more intensionality than the one of Theorem 2.2.2 as illustrated by the following counter-example.

Example 2.3.7. *The TRS, computing the zero function, and consisting of the two rules*

$$\begin{aligned} \mathtt{f}(\mathtt{x}+\mathtt{1}) &\to \mathtt{f}(\mathtt{f}(\mathtt{x})), \\ \mathtt{f}(\mathtt{0}) &\to \mathtt{0}. \end{aligned}$$

This TRS has the following polynomial and additive interpretation over $\mathbb{N}$

$$[\mathtt{f}]_{\mathbb{N}}(X) = X + 1,$$
$$[+\mathtt{1}]_{\mathbb{N}}(X) = X + 2,$$
$$[\mathtt{0}]_{\mathbb{N}} = 0.$$

It does not terminate by $\prec_{rpo}$ *as, for the first rule, it cannot hold that* $\{\mathtt{f(x)}\} \prec^{\mathtt{p}}_{\mathtt{rpo}} \{\mathtt{x+1}\}$.

The converse also holds, *i.e.* the characterization of Theorem 2.2.2 does not provide more intensionality than the one of Theorem 2.3.4, as illustrated by the following counter-example.

Example 2.3.8. *The TRS, computing the zero function and consisting of the three rules*

$$\mathtt{f(x+1)} \to \mathtt{g(f(x),f(x))},$$
$$\mathtt{g(x,y)} \to \mathtt{x},$$
$$\mathtt{f(0)} \to \mathtt{0},$$

has the following polynomial and additive quasi-interpretation over $\mathbb{N}$

$$[\mathtt{f}]_{\mathbb{N}}(X) = X,$$
$$[\mathtt{g}]_{\mathbb{N}}(X,Y) = \max(X,Y),$$
$$[+\mathtt{1}]_{\mathbb{N}}(X) = X + 1,$$
$$[\mathtt{0}]_{\mathbb{N}} = 0,$$

and can be oriented by $\prec_{rpo}$ *only using product status. However it has no polynomial and additive interpretation as, for the first rule, an interpretation should satisfy the following inequality:*

$$[\mathtt{f}]_{\mathbb{N}}(X + k) > [\mathtt{g(f(x),f(x))}]_{\mathbb{N}} \geq 2 \times [\mathtt{f}]_{\mathbb{N}}(X),$$

which is not satisfiable by any polynomial $[\mathtt{f}]_{\mathbb{N}} \in \mathbb{N}[X]$.

From the two above counter-example, we can deduce the following intensional result.

Theorem 2.3.5. *For* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $I^{poly}_{add}(\mathbb{K}, \geq_{\mathbb{K}})$ *and* $(QI^{poly}_{add}(\mathbb{K}, \geq_{\mathbb{K}}) \cap RPO^{\{p\}})$ *are incomparable.*

RPO is still a powerful technique as Hofbauer has demonstrated in [Hof92] that the set of functions computed by $RPO^{\{p\}}$ programs is the set of primitive recursive functions. However its combination with quasi-interpretations is restrictive and it turns out that the two characterizations of FP using interpretations (Theorem 2.2.2) and using quasi-interpretations with RPO (Theorem 2.3.4) are very close from an intensional point of view as the counter-examples used for showing incomparability mostly rely on the inherent weakness of each technique: the inability to compute a sublinear function such as max on the interpretation side, the inability to nest equivalent function calls on the RPO side.

It is worth mentioning that several works have been developed to improve the expressive power of quasi-interpretation method. In [BDLM12], the blind abstraction of a program on binary lists has been defined as the possibly non-deterministic program obtained by replacing lists with their lengths encoded as unary numbers. A program is blindly polynomial if its blind abstraction terminates in polynomial time. This notion is combined with quasi-interpretation to improve the expressive power of previous characterizations.

Another important point to mention is that we have presented a version of RPO using the Product Path Ordering (PPO). The expressive power of the method can be improved using Multiset Path Ordering (MPO) as suggested in [Bon11]. It differs in the sense that the arguments of recursive calls are compared using a multiset based comparison rather than a product one (*i.e.* component by component), allowing the treatment of rules of the shape $\mathtt{f(x+1,y)} \to \mathtt{f(x,x)}$.

2.3.5 Modularity

The paper [BMP07] has suggested another approach for trying to improve the expressive power of quasi-interpretations: modularity. Modularity is a well know notion for TRSs that has been deeply studied for termination [Gra94, KO92]. Applied to our context, the main idea is to divide a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$ into sub-TRSs $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ and $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ and try to search for quasi-interpretations for each of them. Among the distinct ways a program can be split, there are three main possibilities studied by the rewriting community:

- disjoint union $\uplus$, whenever $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ and $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ do not share any common symbol,
- constructor-sharing union $\sqcup$, whenever only constructor symbols in $\mathcal{C}_1$ and $\mathcal{C}_2$ are shared,
- hierarchical union $\sqsubset$, whenever some constructor symbols of $\mathcal{C}_1$ are function symbols of $\mathcal{F}_2$.

In the case of hierarchical union, constructor symbols can be shared. Consequently, the constructor-sharing union can be seen a restricted (commutative) case of hierarchical union.

Now we can define the standard notion of modularity.

Definition 2.3.4. *A property R (a decision problem mapping each TRS to true or false) is modular for the union in $X \in \{\uplus, \sqcup, \sqsubset\}$, if for all TRSs $\mathcal{p}_1$ and $\mathcal{p}_2$ such that $\mathcal{p}_1 X \mathcal{p}_2$ is defined, if $R(\mathcal{p}_1)$ and $R(\mathcal{p}_2)$ then $R(\mathcal{p}_1 X \mathcal{p}_2)$.*

The following modularity results hold for (quasi-)interpretations.

Proposition 2.3.1. *The property of having a (polynomial) additive quasi-interpretation is:*

- *modular for the disjoint union,*
- *not modular for the constructor-sharing union and the hierarchical union.*

The modularity of the disjoint union is straightforward. We give one example illustrating the non-modularity of additive (quasi-)interpretations for the constructor-sharing union.

Example 2.3.9. *Consider the TRSs $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ and $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ defined by:*

$$\mathcal{R}_1 = \begin{cases} \mathtt{f}(\mathtt{a}(\mathtt{x})) & \to \mathtt{f}(\mathtt{f}(\mathtt{x})) \\ \mathtt{f}(\mathtt{b}(\mathtt{x})) & \to \mathtt{a}(\mathtt{a}(\mathtt{f}(\mathtt{x}))) \end{cases} \qquad \mathcal{R}_2 = \begin{cases} \mathtt{g}(\mathtt{b}(\mathtt{x})) & \to \mathtt{g}(\mathtt{g}(\mathtt{x})) \\ \mathtt{g}(\mathtt{a}(\mathtt{x})) & \to \mathtt{b}(\mathtt{b}(\mathtt{g}(\mathtt{x}))) \end{cases}$$

with $\mathcal{C}_1 = \mathcal{C}_2 = \{\mathtt{a}, \mathtt{b}\}$, $\mathcal{F}_1 = \{\mathtt{f}\}$ and $\mathcal{F}_2 = \{\mathtt{g}\}$.

$\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ admits the polynomial additive quasi-interpretation $[-]^1_{\mathbb{K}}$ defined by $[\mathtt{f}]^1_{\mathbb{K}}(X) = [\mathtt{a}]^1_{\mathbb{K}}(X) = X + 1$ and $[\mathtt{b}]^1_{\mathbb{K}}(X) = X + 2$.

$\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ admits the polynomial additive quasi-interpretation $[-]^2_{\mathbb{K}}$ defined by $[\mathtt{g}]^2_{\mathbb{K}}(X) = [\mathtt{b}]^2_{\mathbb{K}}(X) = X + 1$ and $[\mathtt{a}]^2_{\mathbb{K}}(X) = X + 2$.

However their constructor-sharing union $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle \sqcup \langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ has no additive quasi-interpretation. Indeed suppose that such a quasi-interpretation $[-]_{\mathbb{K}}$ exists. $[\mathtt{f}]_{\mathbb{K}}$ and $[\mathtt{g}]_{\mathbb{K}}$ cannot be greater than a linear function because of the first rules of $\mathcal{R}_1$ and $\mathcal{R}_2$. The second rule of $\mathcal{R}_1$ enforces that $[\mathtt{b}]_{\mathbb{K}}(X) \geq [\mathtt{a}]_{\mathbb{K}}([\mathtt{a}]_{\mathbb{K}}(X))$, whereas the second rule of $\mathcal{R}_2$, enforces that $[\mathtt{a}]_{\mathbb{K}}(X) \geq [\mathtt{b}]_{\mathbb{K}}([\mathtt{b}]_{\mathbb{K}}(X))$. These constraints are not satisfiable by any additive quasi-interpretation.

The example below illustrates the non-modularity hierarchical union for additive polynomial quasi-interpretations.

Example 2.3.10. *Consider the TRSs* $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ *and* $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ *defined by:*

$$\mathcal{R}_1 = \begin{cases} \mathtt{double(0)} & \to \mathtt{0} \\ \mathtt{double(x+1)} & \to \mathtt{double(x)+2} \end{cases} \qquad \mathcal{R}_2 = \begin{cases} \mathtt{exp(0)} & \to \underline{\mathtt{1}} \\ \mathtt{exp(x+1)} & \to \mathtt{double(exp(x))} \end{cases}$$

with $\mathcal{C}_1 = \{\mathtt{0}, \mathtt{+1}\}$, $\mathcal{C}_2 = \{\mathtt{0}, \mathtt{+1}, \mathtt{double}\}$, $\mathcal{F}_1 = \{\mathtt{double}\}$, *and* $\mathcal{F}_2 = \{\mathtt{exp}\}$. $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ *admits the additive polynomial quasi-interpretation* $[-]^1_{\mathbb{K}}$ *defined by* $[\mathtt{double}]^1_{\mathbb{K}}(X) = 2X$, $[\mathtt{+1}]^1_{\mathbb{K}}(X) = X + 1$, *and* $[\mathtt{0}]^1_{\mathbb{K}} = 0$. $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ *admits the additive polynomial quasi-interpretation* $[-]^2_{\mathbb{K}}$ *defined by* $[\mathtt{exp}]^2_{\mathbb{K}}(X) = X$, $[\mathtt{double}]^2_{\mathbb{K}}(X) = [\mathtt{+1}]^2_{\mathbb{K}}(X) = X + 1$, *and* $[\mathtt{0}]^2_{\mathbb{K}} = 0$. *However their hierarchical union* $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle \sqsubset \langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ *has no additive polynomial quasi-interpretation.*

The property of having an additive polynomial interpretation is not modular for constructor-sharing unions but [BMP07] has made the observation that the program obtained by such an union is still computing a polynomial time function. This can be combined with the fact that RPO is modular for constructor-sharing union. Let $\sqcup QI^{poly}_{add}(\mathbb{K}, \geq_{\mathbb{K}})$ be the set of function symbols whose TRS is the constructor-sharing union of two programs admitting an additive polynomial quasi-interpretation over $\mathbb{K}$.

Theorem 2.3.6 ([BMP07])**.** *For* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, *we have:*

- $[\![\sqcup QI^{poly}_{add}(\mathbb{K}, \geq_{\mathbb{K}}) \cap RPO^{\{p\}}]\!] = \mathtt{FP}$,
- $[\![\sqcup QI^{poly}_{add}(\mathbb{K}, \geq_{\mathbb{K}}) \cap RPO^{\{l\}}]\!] = \mathtt{FPSPACE}$.

Moreover, these characterizations are strictly more expressive.

Theorem 2.3.7. *For* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $QI^{poly}_{add}(\mathbb{K}, \geq_{\mathbb{K}}) \subsetneq \sqcup QI^{poly}_{add}(\mathbb{K}, \geq_{\mathbb{K}})$.

Indeed, each TRS can be seen as the constructor sharing union of itself and a program with an empty set of rules. Example 2.3.9 illustrates that the inclusion is strict. Some more examples are provided in [BMP07]. However Theorem 2.3.6 does not hold for hierarchical unions as illustrated by Example 2.3.10 where both TRSs admit an additive polynomial interpretation but their hierarchical union computes an exponential function. The result can be recovered on hierarchical unions by putting some more restrictions. Given a polynomial P and a monomial m, we write $m \in P$ if $P = \alpha m + Q$, for some $\alpha \in \mathbb{N} - \{0\}$ and some polynomial Q such that $m \notin Q$. α is the coefficient of m in P, noted $\mathrm{coeff}(m, P)$.

Definition 2.3.5. *Given the hierarchical union of two TRSs* $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ *and* $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ *having the respective polynomial quasi-interpretations* $[-]^1_{\mathbb{N}}$ *and* $[-]^2_{\mathbb{N}}$, $[-]^1_{\mathbb{N}}$ *and* $[-]^2_{\mathbb{N}}$ *are* kind preserving *if for any symbol* $\mathtt{b} \in \mathcal{F}_1 \cap \mathcal{C}_2$, *the following conditions both hold:*

- *for any monomial* m, $m \in [\mathtt{b}]^1_{\mathbb{N}}$ *if and only if* $m \in [\mathtt{b}]^2_{\mathbb{N}}$,
- *for any monomial* m, *coeff*$(m, [\mathtt{b}]^1_{\mathbb{N}}) = 1$ *if and only if coeff*$(m, [\mathtt{b}]^2_{\mathbb{N}}) = 1$.

Notice that the notion of kind comes from [BCMT98].

Example 2.3.11. *The interpretations of the two TRSs of Example 2.3.10 are not kind preserving. Indeed* $[\mathtt{double}]^1_{\mathbb{K}}(X) = 2X$ *and* $[\mathtt{double}]^2_{\mathbb{K}}(X) = X + 1$. *Consequently, coeff*$(X, [\mathtt{b}]^1_{\mathbb{N}}) = 2$ *and coeff*$(X, [\mathtt{b}]^2_{\mathbb{N}}) = 1$.

For a given TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, we define an equivalence on function symbols $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$ by the reflexive and transitive closure of whether $\mathtt{f}$ called $\mathtt{g}$ in $\mathcal{R}$ and vice versa.

Definition 2.3.6. *The hierarchical union of two programs $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ and $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ is a* stratified union*, noted $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle \ll \langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ if the following conditions hold:*

- *for any rule $\mathtt{f}(p_1, \ldots, p_n) \to r \in \mathcal{R}_2$ and any subterm $\mathtt{g}(e_1, \ldots, e_n)$ of r, if $\mathtt{f} \approx_{\mathcal{F}_2} \mathtt{g}$ then no shared function symbols of $\mathcal{C}_2 \cap \mathcal{F}_1$ occurs in $e_1, \ldots, e_n$,*
- *for any rule $\mathtt{f}(p_1, \ldots, p_n) \to r \in \mathcal{R}_2$ there is no nesting of function symbols in r.*

Let $\ll KPQI^{poly}_{add}(\mathbb{N}, \geq_{\mathbb{N}})$ be the set of function symbols whose TRS is the stratified union of two programs admitting an additive polynomial kind preserving quasi-interpretation over $\mathbb{N}$.[22]

Theorem 2.3.8 ([BMP07])**.** *We have:*

- $[\![\ll KPQI^{poly}_{add}(\mathbb{N}, \geq_{\mathbb{N}}) \cap RPO^{\{p\}}]\!] = \mathtt{FP}$,
- $[\![\ll KPQI^{poly}_{add}(\mathbb{N}, \geq_{\mathbb{N}}) \cap RPO^{\{l\}}]\!] = \mathtt{FPSPACE}$.

An extension to $\mathbb{Q}^+$ and $\mathbb{R}^+$ is considered in [BMP07]. Basically, it just asks for some extra condition on the notion of kind preserving interpretation: all the multiplicative coefficients of the monomials have to be greater than 1. This is always true over $\mathbb{N}$.

Example 2.3.12. *Consider the TRSs $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ and $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ defined by:*

$$\mathcal{R}_1 = \begin{cases} \mathtt{double}(\mathtt{0}) & \to \mathtt{0} \\ \mathtt{double}(\mathtt{x{+}1}) & \to \mathtt{double}(\mathtt{x}){+}\mathtt{2} \\ \mathtt{add}(\mathtt{0}, \mathtt{y}) & \to \mathtt{y} \\ \mathtt{add}(\mathtt{x{+}1}, \mathtt{y}) & \to \mathtt{add}(\mathtt{x}, \mathtt{y}){+}\mathtt{1} \end{cases} \quad \mathcal{R}_2 = \begin{cases} \mathtt{sq}(\mathtt{0}) & \to \mathtt{0} \\ \mathtt{sq}(\mathtt{x{+}1}) & \to \mathtt{add}(\mathtt{sq}(\mathtt{x}), \mathtt{double}(\mathtt{x})) \end{cases}$$

with $\mathcal{C}_1 = \{\mathtt{0}, \mathtt{+1}\}$, $\mathcal{F}_1 = \{\mathtt{double}, \mathtt{add}\}$, $\mathcal{C}_2 = \{\mathtt{0}, \mathtt{+1}, \mathtt{add}, \mathtt{double}\}$, and $\mathcal{F}_1 = \{\mathtt{sq}\}$. The function symbol $\mathtt{sq}$ computes the square of a unary number given as input in the hierarchical union $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle \sqsubset \langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$. Neither $\mathtt{sq} \approx_{\mathcal{F}} \mathtt{double}$, nor $\mathtt{sq} \approx_{\mathcal{F}} \mathtt{add}$ hold. The program $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ is flat as there is no composition of function symbols in its rules. Moreover, there is no shared argument in the recursive call $\mathtt{sq}(\mathtt{x})$. Consequently, $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle \ll \langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ is a stratified union.

Define the following quasi-interpretations $[-]^1_{\mathbb{N}}$ and $[-]^2_{\mathbb{N}}$ by:

$$\begin{aligned} [\mathtt{0}]^1_{\mathbb{N}} &= 0 & [\mathtt{0}]^2_{\mathbb{N}} &= 0, \\ [\mathtt{+1}]^1_{\mathbb{N}}(X) &= X + 1, & [\mathtt{+1}]^2_{\mathbb{N}}(X) &= X + 1, \\ [\mathtt{add}]^1_{\mathbb{N}}(\mathtt{X}, \mathtt{Y}) &= X + Y, & [\mathtt{add}]^2_{\mathbb{N}}(\mathtt{X}, \mathtt{Y}) &= X + Y + 1, \\ [\mathtt{double}]^1_{\mathbb{N}}(\mathtt{X}) &= 3X, & [\mathtt{double}]^2_{\mathbb{N}}(\mathtt{X}) &= 2X, \\ & & [\mathtt{sq}]^2_{\mathbb{N}}(\mathtt{X}) &= 2X^2. \end{aligned}$$

$[-]^1_{\mathbb{N}}$ and $[-]^2_{\mathbb{N}}$ are additive and polynomial kind preserving quasi-interpretations. The two programs can be ordered by RPO with product status and, consequently, the function symbols of the stratified union computes functions in $\mathtt{FP}$.

[22] Here the notion of additivity is slightly modified as we do not require symbols in $\mathcal{F}_1 \cap \mathcal{C}_2$ to be additive.

As a corollary of Theorem 2.3.7, we also have more expressive power for stratified union (constructor-sharing union being a particular case of stratified union):

Corollary 2.3.1. $QI^{poly}_{add}(\mathbb{N}, \geq_{\mathbb{N}}) \subsetneq \ll KPQI^{poly}_{add}(\mathbb{N}, \geq_{\mathbb{N}})$.

Due to the restrictions to kind preserving quasi-interpretations, stratified union does not improve greatly the expressive power of the method compared to constructor-sharing union. However they allow to improve quasi-interpretation synthesis by allowing to apply a divide-and-conquer strategy on TRS rules during inference.

2.4 Sup-interpretation

2.4.1 Motivations, definition, and basic properties

The subterm property drastically limits the quasi-interpretation domain. For example, a function defined by $\mathtt{f}(x, y) \to x$ has a quasi-interpretation at least equal to $[\mathtt{f}]_{\mathbb{K}}(X, Y) = \max_{\mathbb{K}}(X, Y)$, $\max_{\mathbb{K}}$ being the max function over $\mathbb{K}$, whereas one would expect it to be $[\mathtt{f}]_{\mathbb{K}}(X, Y) = X$, since the second parameter is dropped.

The notion of sup-interpretation (SI) was introduced in [MP06, MP09] in order to increase the intensionality of interpretation methods by compensating for such a drawback: sup-interpretations do not need to satisfy the subterm property.

As quasi-interpretations, sup-interpretations do not ensure termination and, consequently, they can be defined either on well-founded or on non well-founded ordered sets $\mathbb{K} \in \{\mathbb{N}, \mathbb{R}^+, \mathbb{Q}^+\}$.

Definition 2.4.1 (Sup-interpretation)**.** *Given a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, a monotonic (additive and polynomial) assignment θ over $\mathbb{K}$ is a (additive and polynomial) sup-interpretation over $\mathbb{K}$ if for any $\mathtt{f} \in \mathcal{F}$ of arity m and for all $v_1, \cdots, v_m \in \mathcal{T}(\mathcal{C})$:*

$$\mathtt{f}(v_1, \cdots, v_m) \downarrow \Longrightarrow [\mathtt{f}(v_1, \cdots, v_m)]_{\mathbb{K}} \geq_{\mathbb{K}} [[\![\mathtt{f}]\!](v_1, \cdots, v_m)]_{\mathbb{K}},$$

where the sup-interpretation $[-]_{\mathbb{K}}$ is extended canonically to terms in a standard way.

As sup-interpretations have no subterm requirements, they allow us to consider non-subterm functions in a tighter way.

Example 2.4.1. *Consider the TRS computing the half of a unary number:*

$$\begin{aligned} \mathtt{half}(\mathtt{0}) &\to \mathtt{0}, \\ \mathtt{half}(\mathtt{1}) &\to \mathtt{0}, \\ \mathtt{half}(\mathtt{x{+}2}) &\to \mathtt{half}(\mathtt{x}){+}\mathtt{1}. \end{aligned}$$

It admits the following additive and polynomial sup-interpretation over $\mathbb{Q}^+$:

$$\begin{aligned} [\mathtt{0}]_{\mathbb{Q}^+} &= 0, \\ [+\mathtt{1}]_{\mathbb{Q}^+}(X) &= X + 1, \\ [\mathtt{half}]_{\mathbb{Q}^+}(x) &= X/2. \end{aligned}$$

Indeed, $[\![\mathtt{half}]\!](\underline{\mathtt{n}}) = \underline{n/2}$, provided that $/2$ is the division over natural numbers and that $\underline{\mathtt{n}}$ is the unary encoding of the natural number n. Consequently, we check that for all $\underline{\mathtt{n}}$,

$$\begin{aligned} [\mathtt{half}(\underline{\mathtt{n}})]_{\mathbb{Q}^+} &= [\mathtt{half}]_{\mathbb{Q}^+}([\underline{\mathtt{n}}]_{\mathbb{Q}^+}) \\ &= [\underline{\mathtt{n}}]_{\mathbb{Q}^+}/2 \\ &\geq_{\mathbb{Q}^+} [\underline{n/2}]_{\mathbb{Q}^+} = [[\![\mathtt{half}]\!](\underline{\mathtt{n}})]_{\mathbb{Q}^+}. \end{aligned}$$

However finding a sup-interpretation of all the function symbols of a given TRS consists in finding a (partial) upper-bound on all their computation and is not feasible. It is shown in [Péc13] that the sup-interpretation verification problem is Π_1^0-complete and that the synthesis problem is in Σ_3^0. Some criteria were developed to overcome this issue.

In [MP09], a criterion was developed to ensure polynomial space computations using sup-interpretations. For that purpose, let $\succeq_{\mathcal{F}}$ be an ordering on function symbols $\mathcal{F}$ of a given TRS defined by $\mathtt{f} \succeq_{\mathcal{F}} \mathtt{g}$ if there are a context $C[\diamond]$ and terms $p_1, \ldots, p_n, t_1, \ldots, t_m$ such that $\mathtt{f}(p_1, \ldots, p_n) \to C[\mathtt{g}(t_1, \ldots, t_m)] \in \mathcal{R}$. $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$ if both $\mathtt{f} \succeq_{\mathcal{F}} \mathtt{g}$ and $\mathtt{g} \succeq_{\mathcal{F}} \mathtt{f}$ hold. $\mathtt{f} \succ_{\mathcal{F}} \mathtt{g}$ if $\mathtt{f} \succeq_{\mathcal{F}} \mathtt{g}$ holds and $\mathtt{g} \succeq_{\mathcal{F}} \mathtt{f}$ does not hold.

Definition 2.4.2. *In a TRS a rule* $\mathtt{f}(p_1, ..., p_n) \to C[\mathtt{g}_1(\overline{e_1}), \ldots, \mathtt{g}_m(\overline{e_m})]$ *is a fraternity if:*

- $\forall i \leq m,\ \mathtt{g}_i \approx_{\mathcal{F}} \mathtt{f}$,
- $\forall \mathtt{h} \in C[\diamond_1, \ldots, \diamond_m],\ \mathtt{f} \succ_{\mathcal{F}} \mathtt{h}$.

Definition 2.4.3. *For* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, *a TRS is quasi-friendly (in* $QF_{\mathbb{K}}$*) if there is a polynomial and additive sup-interpretation* $[-]_{\mathbb{K}}$ *over* $\mathbb{K}$ *and a (partial) polynomial, monotonic, and subterm assignment* ω *over* $\mathbb{K}$, *called weight, such that for each fraternity* $\mathtt{f}(p_1, ..., p_n) \to C[\mathtt{g}_1(\overline{e_1}), \ldots, \mathtt{g}_m(\overline{e_m})]$ *the following holds:*

$$\omega(\mathtt{f})([p_1]_{\mathbb{K}}, \ldots, [p_n]_{\mathbb{K}}) \geq_{\mathbb{K}} [C]_{\mathbb{K}}(\omega(\mathtt{g}_1)([\overline{e_1}]_{\mathbb{K}}), \ldots, \omega(\mathtt{g}_m)([\overline{e_m}]_{\mathbb{K}})),$$

where $[C]_{\mathbb{K}}$ *is the function* $([\diamond_1]_{\mathbb{K}}, \ldots, [\diamond_m]_{\mathbb{K}}) \mapsto [C[\diamond_1, \ldots, \diamond_m]]_{\mathbb{K}}$ *and where the interpretation of a sequence* $\overline{t} = t_1, \cdots, t_k$ *is defined by* $[\overline{t}]_{\mathbb{K}} = [t_1]_{\mathbb{K}}, \ldots, [t_k]_{\mathbb{K}}$.

We have a first result on the space consumption of a TRS in $\mathrm{QF}_{\mathbb{K}}$.

Proposition 2.4.1. *Given a TRS* $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$ *in* $QF_{\mathbb{K}}$, *there is a polynomial* P *over* $\mathbb{K}$ *such that for any* $\mathtt{f} \in \mathcal{F}$ *of arity* n *and any values* $v_1, \cdots, v_n \in \mathcal{T}(\mathcal{C})$, *if* $\mathtt{f}(v_1, \cdots, v_n) \downarrow$ *then* $|[\![\mathtt{f}]\!](v_1, \cdots, v_n)| \leq_{\mathbb{K}} P(|v_1|, \ldots, |v_n|)$.

Example 2.4.2. *The TRS of Example 2.4.1 has only one fraternity*

$$\mathtt{half}(\mathtt{x}{+}2) \to \mathtt{half}(\mathtt{x}){+}1$$

as `half` *is the only function symbol and* $\approx_{\mathcal{F}}$ *is reflexive. Now* $[+1]_{\mathbb{K}}(X) = X + 1$ *defines an additive and polynomial sup-interpretation. For the TRS to be in* $QF_{\mathbb{Q}^+}$, *one has to find a weight (subterm, monotonic, and polynomial assignment)* ω *such that*

$$\omega(\mathtt{half})([\mathtt{x}{+}2]_{\mathbb{K}}) \geq_{\mathbb{Q}^+} [+1]_{\mathbb{K}}(\omega(\mathtt{half})([\mathtt{x}]_{\mathbb{K}})$$

or equivalently

$$\omega(\mathtt{half})(X + 2) \geq_{\mathbb{Q}^+} \omega(\mathtt{half})(X) + 1.$$

It suffices to set $\omega(\mathtt{half})(X) = X$ *and we can conclude that the TRS is in* $QF_{\mathbb{Q}^+}$.

Again, the notion of sup-interpretation has to be combined with some termination argument in order to characterize complexity classes.

Theorem 2.4.1 ([MP09])**.** *For* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, *the following characterizations hold:*

- $[\![QF_{\mathbb{K}} \cap RPO^{\{p\}}]\!] = \mathtt{FP}$,

- $[\![QF_{\mathbb{K}} \cap RPO^{\{l\}}]\!] = \texttt{FPSPACE}$.

Example 2.4.3. *The TRS of Example 2.4.1 is in $QF_{\mathbb{K}}$ and can trivially be oriented by $\prec_{rpo}$. Consequently, the computed function $[\![\texttt{half}]\!]$ is in* FP.

It is worth noticing that the quasi-friendly criterion has also been extended in [MP09] to:

- non-terminating programs over stream data in a criterion called quasi-friendly with bounded recursive calls,
- divide-and-conquer algorithms and, in particular `quicksort`, in a criterion called quasi-friendly modulo projection.

Moreover, combinations with the termination methods such as DPs and SCP have also been considered. We will focus on an adaptation to DPs, a complete method for showing program termination, in the next subsections.

2.4.2 Combination with the dependency pair method

RPO termination techniques have an inherent syntactic restriction on the shape of admissible recursions to avoid the computation of non-primitive recursive functions. To overcome these (intensionality) issues (the use of RPO and the subterm property requirement), the notion of dependency pair (DP), introduced by Arts and Giesl [AG00], was combined with the notion of interpretation in [MP08c, MP09] in order to characterize FP and FPSPACE.

DPs provides a method for showing program termination that is complete with respect to termination and, consequently, captures strictly more programs than RPO, as RPO was shown to be NP-complete in [KN85] and, hence, cannot be complete for termination.

We start by briefly reviewing the DP method.

Definition 2.4.4 (Dependency Pair)**.** *Given a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R}\rangle$, the set of dependency pair symbols $\mathcal{F}^\sharp$ is defined by $\mathcal{F}^\sharp = \mathcal{F} \cup \{\mathtt{f}^\sharp \mid \mathtt{f} \in \mathcal{F}\}$, $\mathtt{f}^\sharp$ being a fresh function symbol of the same arity as $\mathtt{f}$. Given a term $t = \mathtt{f}(t_1, \cdots, t_n)$, let $t^\sharp$ be a notation for $\mathtt{f}^\sharp(t_1, \cdots, t_n)$.*
A dependency pair is a pair $l^\sharp \to u^\sharp$ if $u^\sharp = \mathtt{g}^\sharp(t_1, \cdots, t_n)$, for some $\mathtt{g} \in \mathcal{F}$, and if there is a context $C[\diamond]$ such that $l \to C[u] \in \mathcal{R}$ and u is not a proper subterm of l. Let $DP(\mathcal{R})$ be the set of all dependency pairs in $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R}\rangle$.

Definition 2.4.5 (Dependency Pair Graph)**.** *The dependency pair graph (DPG) of a given TRS is the graph obtained as follows:*

- *the vertices are the dependency pairs,*
- *there is an edge from $t \to s$ to $t' \to s'$, if there is a substitution σ such that $s\sigma \to^*_{\mathcal{R}'} t'\sigma$.*

provided that $\mathcal{R}'$ is the program obtained by extending $\mathcal{R}$ with a new rule for each dependency pair.

A cycle of the DPG is a cycle in the corresponding graph structure.

Definition 2.4.6. *A reduction pair is a couple $(\geq, >)$ such that:*

- *$\geq$ is a (weakly) monotonic quasi-ordering over terms stable by substitution,*
- *$>$ is a well-founded ordering over terms and stable by substitution,*

satisfying $\geq \circ > \subseteq >$ *or* $> \circ \geq \subseteq >$. *If both conditions are satisfied then* $(\geq, >)$ *is called a strong reduction pair.*

Example 2.4.4. $(\geq_{\mathbb{N}}, >_{\mathbb{N}})$ *and* $(\geq_{\mathbb{R}^+}, >^{wf}_{\mathbb{R}^+})$ *are strong reduction pairs but* $(\geq_{\mathbb{Q}^+}, >_{\mathbb{Q}^+})$ *is neither a strong reduction pair, nor a reduction pair, as* $>_{\mathbb{Q}^+}$ *is not well-founded.*

Definition 2.4.7. *Given a a reduction pair* $(\geq, >)$, *let* $DP(\geq, >)$ *be the set of TRSs such that:*

- *for each rule* $l \to r$, $l \geq r$,
- *for each DP* $s \to t$ *in a cycle of the DPG,* $s \geq t$,
- *for each cycle of the DPG, there is a dependency pair* $s \to t$ *such that* $s > t$.

In the particular case where $>$ *is the strict order of* $\geq$, *we simply write* $DP(\geq)$.

The complete characterization of termination can be stated as follows.

Theorem 2.4.2 ([AG00])**.** *A TRS* $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$ *is terminating if and only if there is a well-founded monotonic quasi-ordering* $\geq$ *closed under substitution such that* $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle \in DP(\geq)$.

The implication from right to left is easy to grasp as any infinite reduction sequence will correspond to an infinite descending chain with respect to the quasi-ordering $\geq$. Infinitely many strict decreases have to occur in such a chain and correspond to a path involving infinitely many cycles in the DPG. This contradicts the well-foundedness assumption on the strict order corresponding to $\geq$.

As termination is an undecidable property, it is undecidable to show that a TRS satisfy the DP requirements for some reduction pair. The undecidability of this characterization is hidden behind the difficulty of generating the DPG of a given TRS. Indeed, deciding whether there is a connecting edge between two dependency pairs is an undecidable property. Some simplifications to generate the DPG are considered in [AG00].

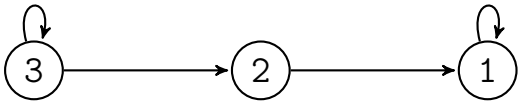
Example 2.4.5. *Consider the following TRS computing the logarithm over unary numbers:*

$$\begin{aligned}
\mathtt{half}(\mathtt{0}) &\to \mathtt{0},\\
\mathtt{half}(\mathtt{1}) &\to \mathtt{0},\\
\mathtt{half}(\mathtt{x{+}2}) &\to \mathtt{half}(\mathtt{x}){+}\mathtt{1},\\
\mathtt{log}(\mathtt{x{+}2}) &\to \mathtt{log}(\mathtt{half}(\mathtt{x{+}2})){+}\mathtt{1},\\
\mathtt{log}(\mathtt{1}) &\to \mathtt{0}.
\end{aligned}$$

It admits the following DPs:

$$\begin{aligned}
1 : \mathtt{half}^\sharp(\mathtt{x{+}2}) &\to \mathtt{half}^\sharp(\mathtt{x}),\\
2 : \mathtt{log}^\sharp(\mathtt{x{+}2}) &\to \mathtt{half}^\sharp(\mathtt{x{+}2}),\\
3 : \mathtt{log}^\sharp(\mathtt{x{+}2}) &\to \mathtt{log}^\sharp(\mathtt{half}(\mathtt{x{+}2})),
\end{aligned}$$

and has the following DPG:



Consequently, showing termination just consists in finding a well-founded quasi-ordering $\geq$ such that for all rule $l \to r$, $l \geq r$ and the following inequalities hold:

$$\begin{aligned} \mathtt{half}^\sharp(\mathtt{x{+}2}) &> \mathtt{half}^\sharp(\mathtt{x}),\\ \mathtt{log}^\sharp(\mathtt{x{+}2}) &\geq \mathtt{half}^\sharp(\mathtt{x{+}2}),\\ \mathtt{log}^\sharp(\mathtt{x{+}2}) &> \mathtt{log}^\sharp(\mathtt{half}(\mathtt{x{+}2})). \end{aligned}$$

The two strict inequalities above correspond to the two cycles in the DPG.

In [MP09], an alternative characterization of FPSPACE was provided using the DP method.

Definition 2.4.8. *For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, a program has strictly bounded recursive calls (is in $SBR_{\mathbb{K}}$) if there are a polynomial and additive sup-interpretation $[-]_{\mathbb{K}}$ over $\mathbb{K}$ and a polynomial and subterm weight ω over $\mathbb{K}$ such that:*

- *it is in $QF_{\mathbb{K}}$ with respect to $[-]_{\mathbb{K}}$ and ω,*
  i.e. for each fraternity $\mathtt{f}(p_1, ..., p_n) \to C[\mathtt{g}_1(\overline{e_1}), \ldots, \mathtt{g}_m(\overline{e_m})]$ the following holds:

$$\omega(\mathtt{f})([p_1]_{\mathbb{K}}, \ldots, [p_n]_{\mathbb{K}}) \geq_{\mathbb{K}} [C]_{\mathbb{K}}(\omega(\mathtt{g}_1)([\overline{e_1}]_{\mathbb{K}}), \ldots, \omega(\mathtt{g}_m)([\overline{e_m}]_{\mathbb{K}})),$$

- *for each DP $\mathtt{f}^\sharp(\overline{p}) \to \mathtt{g}^\sharp(\overline{e})$, we have:*

$$\omega(\mathtt{f})([\overline{p}]_{\mathbb{K}}) \geq_{\mathbb{K}} \omega(\mathtt{g})([\overline{e}]_{\mathbb{K}}),$$

- *for each cycle in the DPG, there is at least one DP $\mathtt{f}^\sharp(\overline{p}) \to \mathtt{g}^\sharp(\overline{e})$ such that:*

$$\omega(\mathtt{f})([\overline{p}]_{\mathbb{K}}) >_{\mathbb{K}}^{wf} \omega(\mathtt{g})([\overline{e}]_{\mathbb{K}}).$$

Example 2.4.6. *The TRS of Example 2.4.5 has already been shown to be in $QF_{\mathbb{Q}^+}$ in Example 2.4.2 for the weight ω defined by $\omega(\mathtt{log})(X) = \omega(\mathtt{half})(X) = X$. Consequently, for this TRS belongs to $SBR_{\mathbb{Q}^+}$, it remains to show:*

$$\begin{aligned} \omega(\mathtt{half})([\mathtt{x{+}2}]_{\mathbb{Q}^+}) &>_{\mathbb{Q}^+}^{wf} \omega(\mathtt{half})([\mathtt{x}]_{\mathbb{Q}^+}),\\ \omega(\mathtt{log})([\mathtt{x{+}2}]_{\mathbb{Q}^+}) &\geq_{\mathbb{Q}^+} \omega(\mathtt{half})([\mathtt{x{+}2}]_{\mathbb{Q}^+}),\\ \omega(\mathtt{log})([\mathtt{x{+}2}]_{\mathbb{Q}^+}) &>_{\mathbb{Q}^+}^{wf} \omega(\mathtt{log})([\mathtt{half}(\mathtt{x{+}2})]_{\mathbb{Q}^+}). \end{aligned}$$

It was shown in Example 2.4.1, that the assignment $[-]_{\mathbb{K}}$ defined by $[+1]_{\mathbb{Q}^+}(X) = X + 1$ and $[\mathtt{half}]_{\mathbb{Q}^+}(X) = X/2$ is a suitable sup-interpertation for $\mathtt{half}$. The above inequalities can be reformulated as follows:

$$\begin{aligned} \omega(\mathtt{half})(X+2) &>_{\mathbb{Q}^+}^{wf} \omega(\mathtt{half})(X),\\ \omega(\mathtt{log})(X+2) &\geq_{\mathbb{Q}^+} \omega(\mathtt{half})(X+2),\\ \omega(\mathtt{log})(X+2) &>_{\mathbb{Q}^+}^{wf} \omega(\mathtt{log})(X/2+1). \end{aligned}$$

They are clearly satisfied for the weight defined above together with the well-founded ordering $>_1$. Consequently, both $[\![\mathtt{log}]\!]$ and $[\![\mathtt{half}]\!]$ are in FPSPACE (we will improve this result soon).

Theorem 2.4.3 ([MP09])**.** *For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $[\![SBR_{\mathbb{K}}]\!] =$ FPSPACE.*

Example 2.4.7. *Consider the following program taken from [BMM11] and computing the QBF problem:*

$$\begin{aligned}
\mathtt{not}(\mathtt{tt}) &\to \mathtt{ff},\\
\mathtt{not}(\mathtt{ff}) &\to \mathtt{tt},\\
\mathtt{or}(\mathtt{tt},\mathtt{x}) &\to \mathtt{tt},\\
\mathtt{or}(\mathtt{ff},\mathtt{x}) &\to \mathtt{x},\\
\mathtt{0} = \mathtt{0} &\to \mathtt{tt},\\
\mathtt{suc}(\mathtt{x}) = \mathtt{0} &\to \mathtt{ff},\\
\mathtt{0} = \mathtt{suc}(\mathtt{y}) &\to \mathtt{ff},\\
\mathtt{suc}(\mathtt{x}) = \mathtt{suc}(\mathtt{y}) &\to \mathtt{x} = \mathtt{y},\\
\mathtt{in}(\mathtt{x},\mathtt{nil}) &\to \mathtt{ff},\\
\mathtt{in}(\mathtt{x},\mathtt{cons}(\mathtt{a},\mathtt{l})) &\to \mathtt{or}(\mathtt{x} = \mathtt{a}, \mathtt{in}(\mathtt{x},\mathtt{l})),\\
\mathtt{verify}(\mathtt{Var}(\mathtt{x}),\mathtt{t}) &\to \mathtt{in}(\mathtt{x},\mathtt{t}),\\
\mathtt{verify}(\mathtt{Not}(\mathtt{u}),\mathtt{t}) &\to \mathtt{not}(\mathtt{verify}(\mathtt{u},\mathtt{t})),\\
\mathtt{verify}(\mathtt{Or}(\mathtt{u},\mathtt{v}),\mathtt{t}) &\to \mathtt{or}(\mathtt{verify}(\mathtt{u},\mathtt{t}),\mathtt{verify}(\mathtt{v},\mathtt{t})),\\
\mathtt{verify}(\mathtt{Exists}(\mathtt{n},\mathtt{u}),\mathtt{t}) &\to \mathtt{or}(\mathtt{verify}(\mathtt{u},\mathtt{cons}(\mathtt{n},\mathtt{t})),\mathtt{verify}(\mathtt{u},\mathtt{t})),\\
\mathtt{qbf}(\mathtt{u}) &\to \mathtt{verify}(\mathtt{u},\mathtt{nil}).
\end{aligned}$$

The following precedence holds $\{\mathtt{not},\mathtt{or},=\} \prec_{\mathcal{F}} \mathtt{in} \prec_{\mathcal{F}} \mathtt{verify} \prec_{\mathcal{F}} \mathtt{qbf}$. *Consider the additive and polynomial sup-interpretation* $[-]_{\mathbb{K}}$ *defined by:*

$$\begin{aligned}
[\mathtt{suc}]_{\mathbb{K}}(X) &= X + 1,\\
[\mathtt{cons}]_{\mathbb{K}}(X,Y) &= X + Y + 1,\\
[\mathtt{Or}]_{\mathbb{K}}(X,Y) &= X + Y + 1,\\
[\mathtt{or}]_{\mathbb{K}}(X,Y) &= 0,\\
[\mathtt{Not}]_{\mathbb{K}}(X) &= X + 1,\\
[\mathtt{not}]_{\mathbb{K}}(X) &= 0,\\
[\mathtt{Exists}]_{\mathbb{K}}(X,Y) &= X + Y + 2,
\end{aligned}$$

and the subterm, monotonic, and polynomial weight ω *defined by:*

$$\begin{aligned}
\omega(=)(X,Y) &= X + Y,\\
\omega(\mathtt{in})(X,Y) &= X + Y,\\
\omega(\mathtt{verify})(X,Y) &= X + Y.
\end{aligned}$$

It is easy to check that the TRS is in $SBR_{\mathbb{K}}$ *and, consequently, the computed function is in* FPSPACE.

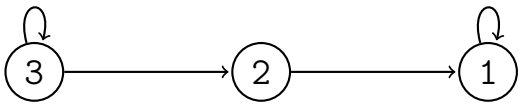
Example 2.4.8. *As a counter-example, consider the TRS of Example 2.2.1:*

$$\begin{aligned}
\mathtt{double}(\mathtt{0}) &\to \mathtt{0},\\
\mathtt{double}(\mathtt{x{+}1}) &\to \mathtt{double}(\mathtt{x}){+}\mathtt{2},\\
\mathtt{exp}(\mathtt{0}) &\to \underline{\mathtt{1}},\\
\mathtt{exp}(\mathtt{x{+}1}) &\to \mathtt{double}(\mathtt{exp}(\mathtt{x})).
\end{aligned}$$

It admits the following DPs:

$$\begin{aligned} 1 : \mathtt{double}^\sharp(\mathtt{x{+}2}) &\to \mathtt{double}^\sharp(\mathtt{x}),\\ 2 : \mathtt{exp}^\sharp(\mathtt{x{+}1}) &\to \mathtt{double}^\sharp(\mathtt{exp}(\mathtt{x})),\\ 3 : \mathtt{exp}^\sharp(\mathtt{x{+}1}) &\to \mathtt{exp}^\sharp(\mathtt{x}), \end{aligned}$$

and has the following DPG:



For this TRS to be in $SBR_{\mathbb{K}}$, it has to be in $QF_{\mathbb{K}}$ and the following property has to be satisfied:

$$\omega(\mathtt{exp})([\mathtt{x{+}1}]_{\mathbb{K}}) \geq [\mathtt{double}]_{\mathbb{K}}(\omega(\mathtt{exp}(\mathtt{x}))),$$

for some polynomial and additive sup-interpretation $[-]_{\mathbb{K}}$ and some monotonic, polynomial, and subterm weight ω. As $[double]_{\mathbb{K}}$ is a sup-interpretation, it has to satisfy $\forall \underline{n}$, $[\mathtt{double}]_{\mathbb{K}}(n) \geq [[\![double]\!]]_{\mathbb{K}}(n) = 2n$. Consequently, the above inequality can be transformed into:

$$\omega(\mathtt{exp})(X+1) \geq [\mathtt{double}]_{\mathbb{K}}(\omega(\mathtt{exp})(X)) \geq 2\omega(\mathtt{exp})(X).$$

Clearly, no polynomial $\omega(\mathtt{exp})$ can satisfy this inequality.

An alternative and intensionally equivalent variant of the result of Theorem 2.4.3 was introduced in [Bon11] using non-subterm assignments rather than sup-interpretations: the symbols $\mathtt{f}^\sharp$ are required to have a subterm assignments. This role is played by the weight $\omega(\mathtt{f})$ in the above setting. Moreover, all the rewrite rules are oriented in [Bon11] whereas this is not needed for sup-interpretations in Definition 2.4.8. The result can be rephrased as follows.

Definition 2.4.9. *For a given assignment $[-]_{\mathbb{K}}$ of a TRS over $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, define $\geq_{\mathbb{K}}^{[-]_{\mathbb{K}}}$ (and $>_{\mathbb{K}}^{[-]_{\mathbb{K}}}$, respectively) by:*

$$\forall s,t \in \mathcal{T}(\Sigma,\mathcal{X}),\ s \geq_{\mathbb{K}}^{[-]_{\mathbb{K}}} t \text{ if and only if } [s]_{\mathbb{K}} \geq_{\mathbb{K}}^{wf} [t]_{\mathbb{K}},$$

(and $s >_{\mathbb{K}}^{[-]_{\mathbb{K}}} t$ if and only if $[s]_{\mathbb{K}} >_{\mathbb{K}}^{wf} [t]_{\mathbb{K}}$, respectively).

Notice that the set $DP(\geq_{\mathbb{K}}^{[-]_{\mathbb{K}}})$ is clearly defined as $>_{\mathbb{K}}^{[-]_{\mathbb{K}}}$ is well-founded, by definition.

Definition 2.4.10. *Let $\mathcal{A}$ be the set of assignments $[-]_{\mathbb{K}}$ that are additive, monotonic, and polynomial. Moreover, let $\mathcal{AS}$ be the subset of assignments in $\mathcal{A}$ such that for each symbol $\mathtt{f} \in \mathcal{F}$ $[\mathtt{f}^\sharp]_{\mathbb{K}}$ has the subterm property.*

We are now ready to reformulate $\mathrm{SBR}_{\mathbb{K}}$ using the notion of DP.

Theorem 2.4.4. *For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $SBR_{\mathbb{K}} = \cup_{[-]_{\mathbb{K}} \in \mathcal{AS}} \{DP(\geq_{\mathbb{K}}^{[-]_{\mathbb{K}}})\}$.*

The proof is easy as it just consists in defining an assignment $[-]'_{\mathbb{K}}$ such that for each symbol $\mathtt{b} \in \mathcal{C} \uplus \mathcal{F}$, $[\mathtt{b}]'_{\mathbb{K}} := [\mathtt{b}]_{\mathbb{K}}$ and for each function symbol $\mathtt{f} \in \mathcal{F}$, $[\mathtt{f}^\sharp]'_{\mathbb{K}} := \omega(\mathtt{f})$.

Given a term $l^\sharp$ let its neighborhood $N(l^\sharp)$ be defined by $t^\sharp \in N(l^\sharp)$ if and only if $(l^\sharp, t^\sharp)$ is involved in at least one cycle of the DPG. Alternative characterizations of FP and FPSPACE were provided in [MP08c] as follows.

Definition 2.4.11. *For a given assignment* $[-]_{\mathbb{K}}$ *over* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$*, a TRS has:*

- *Bounded recursive calls (is in* $BRC([-]_{\mathbb{K}})$*), if for each DP* $\mathtt{f}^\sharp(p_1, \cdots, p_n) \to \mathtt{g}^\sharp(t_1, \cdots, t_m)$ *in a cycle of the DPG, we have:*

$$\Sigma_{i=1}^{n}[p_i]_{\mathbb{K}} \geq_{\mathbb{K}} \Sigma_{j=1}^{m}[t_j]_{\mathbb{K}}.$$

- *Bounded neighborhood (is in* $BN([-]_{\mathbb{K}})$*), if for each neighborhood* $N(t^\sharp) = \{t_1^\sharp, \ldots, t_n^\sharp\}$*, we have:*

$$[l]_{\mathbb{K}} >_{\mathbb{K}}^{wf} \Sigma_{i=1}^{n}[t_i]_{\mathbb{K}}.$$

For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, let $DP(TIME_{\mathbb{K}}) = \cup_{[-]_{\mathbb{K}} \in \mathcal{A}}\{DP(\geq_{\mathbb{K}}^{[-]_{\mathbb{K}}}) \cap BRC([-]_{\mathbb{K}}) \cap BN([-]_{\mathbb{K}})\}$ and $DP(SPACE_{\mathbb{K}}) = \cup_{[-]_{\mathbb{K}} \in \mathcal{A}}\{DP(\geq_{\mathbb{K}}^{[-]_{\mathbb{K}}}) \cap BRC([-]_{\mathbb{K}})\}$. We are now able to give the characterizations of polynomial time and polynomial space.

Theorem 2.4.5 ([MP08c])**.** *For* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$*, the following characterizations hold:*

- $[\![DP(TIME_{\mathbb{K}})]\!] = \mathtt{FP}$,
- $[\![DP(SPACE_{\mathbb{K}})]\!] = \mathtt{FPSPACE}$.

Example 2.4.9. *As a direct application of Theorem 2.4.4, we show that the TRS of Example 2.4.6 is in* $DP(TIME_{\mathbb{Q}^+})$*. Now we have two dependency pairs involved in cycles of the DPG:*

$$\mathtt{half}^\sharp(\mathtt{x{+}2}) \to \mathtt{half}^\sharp(\mathtt{x}),$$
$$\mathtt{log}^\sharp(\mathtt{x{+}2}) \to \mathtt{log}^\sharp(\mathtt{half}(\mathtt{x{+}2})).$$

Hence for this TRS to have bounded recursive calls, we need to check that:

$$[\mathtt{x}+2]_{\mathbb{Q}^+} \geq_{\mathbb{Q}^+} [\mathtt{x}]_{\mathbb{Q}^+},$$
$$[\mathtt{x}+2]_{\mathbb{Q}^+} \geq_{\mathbb{Q}^+} [\mathtt{half}(\mathtt{x{+}2})]_{\mathbb{Q}^+}.$$

These inequalities are satisfied for the additive and polynomial sup-interpretation defined by $[\mathtt{half}]_{\mathbb{Q}^+}(X) = X/2$ *and* $[+1]_{\mathbb{Q}^+}(X) = X + 1$*. Moreover we have* $N(\mathtt{half}^\sharp(\mathtt{x}+2)) = \{\mathtt{half}^\sharp(\mathtt{x})\}$ *and* $N(\mathtt{log}^\sharp(\mathtt{x{+}2})) = \{\mathtt{log}^\sharp(\mathtt{half}(\mathtt{x{+}2}))\}$*. Hence for this TRS to have bounded neighborhoods, we need to check that:*

$$[\mathtt{half}(\mathtt{x{+}2})]_{\mathbb{Q}^+} >_{\mathbb{Q}^+}^{wf} [\mathtt{half}(\mathtt{x})]_{\mathbb{Q}^+},$$
$$[\mathtt{log}(\mathtt{x{+}2})]_{\mathbb{Q}^+} >_{\mathbb{Q}^+}^{wf} [\mathtt{log}(\mathtt{half}(\mathtt{x{+}2}))]_{\mathbb{Q}^+}.$$

These inequalities are satisfied for the above additive and polynomial sup-interpretation extended by $[\mathtt{log}]_{\mathbb{Q}^+}(X) = X$ *with respect to the well-founded ordering* $>_1$*.*

Example 2.4.10. *The TRS of Example 2.4.7 is in* $DP(SPACE_{\mathbb{K}})$*. Indeed, it is in* $SBR_{\mathbb{K}}$*, for some sup-interpretation* $[-]_{\mathbb{K}}$ *and weight* ω*, and, consequently, in* $DP(\geq_{\mathbb{K}}^{[-]'_{\mathbb{K}}})$*, for the assignment* $[-]'_{\mathbb{K}}$ *in* $\mathcal{AS}$ *(and a fortiori in* $\mathcal{A}$*) defined by* $[\mathtt{b}]'_{\mathbb{K}} := [\mathtt{b}]_{\mathbb{K}}$*,* $\forall \mathtt{b} \in \mathcal{C} \uplus \mathcal{F}$*, and* $[\mathtt{f}^\sharp]'_{\mathbb{K}} := \omega(\mathtt{f})$*,* $\forall \mathtt{f} \in \mathcal{F}$*. It is also in* $BRC([-]'_{\mathbb{K}})\}$ *as the functions* $\omega(\mathtt{b})$ *provided in Example 2.4.7 are all linear. Consequently,* $\omega(\mathtt{f})([\overline{p}]'_{\mathbb{K}}) \geq_{\mathbb{K}} \omega(\mathtt{g})([\overline{e}]'_{\mathbb{K}})$ *can be rewritten into:*

$$\Sigma_{i=1}^{n}[p_i]'_{\mathbb{K}} \geq_{\mathbb{K}} \Sigma_{j=1}^{m}[t_j]'_{\mathbb{K}}.$$

However it is (hopefully) not in $DP(TIME_{\mathbb{K}})$. Indeed, consider the rule

$$\mathtt{verify}(\mathtt{Exists}(\mathtt{n},\mathtt{u}),\mathtt{t}) \to \mathtt{or}(\mathtt{verify}(\mathtt{u},\mathtt{cons}(\mathtt{n},\mathtt{t})),\mathtt{verify}(\mathtt{u},\mathtt{t})),$$

the neighborhood $N(\mathtt{verify}^\sharp(\mathtt{Exists}(\mathtt{n},\mathtt{u}),\mathtt{t}))$ is equal to $\{\mathtt{verify}^\sharp(\mathtt{u},\mathtt{cons}(\mathtt{n},\mathtt{t})),\mathtt{verify}^\sharp(\mathtt{u},\mathtt{t})\}$. Consequently, for the program to be in $SN([-]'_{\mathbb{K}})$, one has to check that:

$$[\mathtt{verify}(\mathtt{Exists}(\mathtt{n},\mathtt{u}),\mathtt{t})]'_{\mathbb{K}} >_{\mathbb{K}} [\mathtt{verify}(\mathtt{u},\mathtt{cons}(\mathtt{n},\mathtt{t}))]'_{\mathbb{K}} + [\mathtt{verify}(\mathtt{u},\mathtt{t})]'_{\mathbb{K}}$$

that can be restated as:

$$[\mathtt{verify}]'_{\mathbb{K}}(N+U+1,T) >_{\mathbb{K}} [\mathtt{verify}]'_{\mathbb{K}}(U,N+T+1) + [\mathtt{verify}]'_{\mathbb{K}}(U,T) \geq 2[\mathtt{verify}]'_{\mathbb{K}}(U,T),$$

which is clearly not satisfiable by any polynomial and monotonic assignment.

2.4.3 Sup-interpretation vs (quasi-)interpretation

Let $SI^{poly}_{add}(\mathbb{K},\geq_{\mathbb{K}})$ be the set of function symbols whose TRS admits an additive and polynomial sup-interpretation over $\mathbb{K}$. We obtain an intensionality result, similar to the one of Theorem 2.3.1, as the set of programs admitting an additive and polynomial quasi-interpretations is strictly included in the set of programs admitting an additive and polynomial sup-interpretation.[23]

Theorem 2.4.6 ([MP09][24]). *For $\mathbb{K} \in \{\mathbb{N},\mathbb{Q}^+,\mathbb{R}^+\}$, $QI^{poly}_{add}(\mathbb{K},\geq_{\mathbb{K}}) \subsetneq SI^{poly}_{add}(\mathbb{K},\geq_{\mathbb{K}})$.*

The inclusion is obtained as a direct consequence of Lemma 2.3.1. The strict inclusion is highlighted by the following example.

Example 2.4.11. *Consider the TRS of Example 2.4.5:*

$$\begin{aligned}
\mathtt{half}(0) &\to 0,\\
\mathtt{half}(1) &\to 0,\\
\mathtt{half}(\mathtt{x}{+}2) &\to \mathtt{half}(\mathtt{x}){+}1,\\
\mathtt{log}(\mathtt{x}{+}2) &\to \mathtt{log}(\mathtt{half}(\mathtt{x}{+}2)){+}1,\\
\mathtt{log}(1) &\to 0.
\end{aligned}$$

Notice that because of the fifth rule, the program cannot be oriented by RPO. Moreover, it does not a admit an interpretation or a quasi-interpretation as, by subterm and monotonicity properties, it would imply the following contradiction:

$$[\mathtt{log}^\sharp]_{\mathbb{K}}(X+2k) > [\mathtt{log}^\sharp]_{\mathbb{K}}([\mathtt{half}]_{\mathbb{K}}(X+2k)) \geq [\mathtt{log}^\sharp]_{\mathbb{K}}(X+2k),$$

for $[{+}1]_{\mathbb{K}}(X) = k$.

However it admits the following additive and polynomial sup-interpretation over $\mathbb{Q}^+$:

$$\begin{aligned}
[0]_{\mathbb{K}} &= 0,\\
[{+}1]_{\mathbb{K}}(X) &= X+1,\\
[\mathtt{half}]_{\mathbb{K}}(X) &= X/2,\\
[\mathtt{log}]_{\mathbb{K}}(X) &= X.
\end{aligned}$$

[23] Again, this result remains true if additivity and polynomiality properties are withdrawn.
[24] There is a small distinction with [MP09] where additivity is included in the definition of a sup-interpretation.

Combining Theorem 2.3.1 and Theorem 2.4.6, we obtain the following straightforward corollary.

Corollary 2.4.1. *For* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $I^{poly}_{add}(\mathbb{K}, \geq^{wf}_{\mathbb{K}}) \subsetneq SI^{poly}_{add}(\mathbb{K}, \geq_{\mathbb{K}})$.

In order to obtain intensional results, it is of interest to compare: $QI^{poly}_{add}(\mathbb{K}, \geq_{\mathbb{K}})$ and $\mathrm{QF}_{\mathbb{K}}$. As any quasi-interpretation is a sup-interpretation and as any quasi-interpretation is a weight (see Definition 2.4.3), we obtain the following result.

Theorem 2.4.7 ([MP09])**.** *For* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $QI^{poly}_{add}(\mathbb{K}, \geq_{\mathbb{K}}) \subsetneq QF_{\mathbb{K}}$.

The inclusion is strict as it was demonstrated in Example 2.4.6 that the TRS for `log` belongs to $\mathrm{QF}_{\mathbb{Q}^+}$ and in Example 2.4.11 that it does not belong to $QI^{poly}_{add}(\mathbb{K}, \geq_{\mathbb{K}})$.

As a corollary, we obtain the intensional result that quasi-friendly programs terminating by RPO are at least as expressive as programs admitting an additive and polynomial quasi-interpretation and terminating by RPO.

Theorem 2.4.8 ([MP09])**.** *For* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, *for any* $A \subseteq \{l\}$, $QI^{poly}_{add}(\mathbb{K}, \geq_{\mathbb{K}}) \cap RPO^A \subseteq QF_{\mathbb{K}} \cap RPO^A$.

Notice that it is not known if the above inclusion is strict or not.

As a consequence of Theorem 2.4.8 and Theorem 2.3.5, we obtain the following negative result.

Theorem 2.4.9. *For* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $I^{poly}_{add}(\mathbb{K}, \geq^{wf}_{\mathbb{K}})$ *and* $(QF_{\mathbb{K}} \cap RPO^{\{p\}})$ *are incomparable.*

DP-based characterizations capture natural algorithms that fail to be captured by RPO but it is unclear whether the converse result holds or not.

Theorem 2.4.10. *For* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, *the following statements are true:*

- $DP(TIME_{\mathbb{K}}) - (QF_{\mathbb{K}} \cap RPO^{\{p\}}) \neq \emptyset$,
- $DP(SPACE_{\mathbb{K}}) - (QF_{\mathbb{K}} \cap RPO^{\{l\}}) \neq \emptyset$,
- $SBR_{\mathbb{K}} - (QF_{\mathbb{K}} \cap RPO^{\{l\}}) \neq \emptyset$.

This is due to the fact that natural algorithms such as logarithm or greatest common divisor are captured by the above methods (see [MP08c, MP09] for more examples) whereas they are not captured by RPO due to recursive calls on sublinear computations. However, it is a difficult issue to compare the expressive power of the RPO based characterizations with the DP based ones that differ greatly in essence. Consequently, we conjecture that all these techniques are pairwise incomparable.

2.4.4 DP-interpretations for sup-interpretation synthesis

If the termination requirement is relaxed, it was shown in [MP08c] that DPs can also be used as a technique to infer sup-interpretations.

Definition 2.4.12 (DP-Interpretation)**.** *Given a TRS* $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, *a (additive and polynomial) DP-Interpretation (DPI for short) is a monotonic (additive and polynomial) assignment* $[-]_{\mathbb{K}}$ *over* $\mathbb{K}$ *extended to* $\mathcal{F}^\sharp$ *by* $\forall \mathtt{f}, \in \mathcal{F}$, $[\mathtt{f}^\sharp]_{\mathbb{K}} := [\mathtt{f}]_{\mathbb{K}}$ *and which satisfies:*

1. $\forall l \to r \in \mathcal{R}$, $[l]_{\mathbb{K}} \geq [r]_{\mathbb{K}}$,

2. $\forall l^\sharp \to u^\sharp \in DP(\mathcal{R}),\ [l^\sharp]_\mathbb{K} \geq [u^\sharp]_\mathbb{K}$,

where the DP-interpretation $[-]_\mathbb{K}$ *is extended canonically to terms as usual.*

Example 2.4.12. *Turning back to the TRS of Example 2.4.5, finding a DP-interpretation of the TRS consists in finding a polynomial and additive assignment* $[-]_\mathbb{K}$ *such that the following inequalities are satisfied:*

$$\begin{aligned}
[\mathtt{half(0)}]_\mathbb{K} &\geq [\mathtt{0}]_\mathbb{K},\\
[\mathtt{half(1)}]_\mathbb{K} &\geq [\mathtt{0}]_\mathbb{K},\\
[\mathtt{half(x+2)}]_\mathbb{K} &\geq [\mathtt{half(x)+1}]_\mathbb{K},\\
[\mathtt{log(x+2)}]_\mathbb{K} &\geq [\mathtt{log(half(x+2))+1}]_\mathbb{K},\\
[\mathtt{log(1)}]_\mathbb{K} &\to [\mathtt{0}]_\mathbb{K},\\
[\mathtt{half^\sharp(x+2)}]_\mathbb{K} &\geq [\mathtt{half^\sharp(x)}]_\mathbb{K},\\
[\mathtt{log^\sharp(x+2)}]_\mathbb{K} &\geq [\mathtt{log^\sharp(half(x+2))+1}]_\mathbb{K},\\
[\mathtt{log^\sharp(x+2)}]_\mathbb{K} &\geq [\mathtt{half^\sharp(x+2)}]_\mathbb{K}.
\end{aligned}$$

Taking the additive and polynomial assignment $[\mathtt{0}]_\mathbb{K} = 0$, $[\mathtt{+1}]_\mathbb{K}(X) = X+1$, $[\mathtt{half}]_\mathbb{K}(X) = X/2$, *and* $[\mathtt{log}]_\mathbb{K}(X) = X$, *the above inequalities can be rewritten as follows:*

$$\begin{aligned}
0 &\geq 0,\\
1/2 &\geq 0,\\
X/2+1 &\geq X/2+1,\\
X+2 &\geq X/2+2,\\
1 &\to 0,\\
X/2+1 &\geq X/2,\\
X+2 &\geq X/2+2,\\
X+2 &\geq X/2+1,
\end{aligned}$$

and, as they are all satisfied, $[-]_\mathbb{K}$ *is an additive and polynomial DPI.*

Let $DPI^{poly}_{add}(\mathbb{K}, \geq_\mathbb{K})$ be the set of function symbols whose TRS admits an additive and polynomial DP-interpretation over $\mathbb{K}$.

Notice that the condition on cycles in the DPG has been withdrawn and, consequently, TRS of $DPI^{poly}_{add}(\mathbb{K}, \geq_\mathbb{K})$ can be non-terminating. However we have the following result similar to Theorem 2.3.1.

Theorem 2.4.11 ([MP08c])**.** *For* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$,

$$QI^{poly}_{add}(\mathbb{K}, \geq_\mathbb{K}) \subsetneq DPI^{poly}_{add}(\mathbb{K}, \geq_\mathbb{K}) \subseteq SI^{poly}_{add}(\mathbb{K}, \geq_\mathbb{K}).$$

As expected, the first inclusion is strict because of Example 2.4.12. Knowing whether the second inclusion is strict or not is an open issue.

We have already mentioned that the synthesis problem is undecidable for sup-interpretations in general as finding a sup-interpretation requires some prior knowledge on the termination and growth rate of the function computed by the program under study. However it was shown in [Péc13] that the DPI synthesis problem is equivalent to the quasi-interpretation synthesis problem, whose complexity is provided in Figure 1.3 for distinct function spaces, as DPIs are QIs without subterm property and with some extra inequalities based on the DPG of the TRS under study. Consequently, by Theorem 2.4.11, DPIs seem to be the good candidate at the present time to generate upper-bounds on program space consumption.

2.5 Summary

In this chapter, we have presented several results of the last decade, whose goal was to improve the expressive power of previous complexity class characterizations based on light/soft logics and interpretation methods. Moreover, we have related their expressive power, when possible.

We have studied the intensionality of criteria by comparing the set of captured programs for a fixed language and complexity class. Consequently, two criteria are often incomparable when they rely on distinct techniques.

For example, DLAL and STA are some of the most powerful techniques for light/soft logics and polynomial time but they are incomparable. In the framework of interpretations, we have I $\subsetneq$ QI $\subsetneq$ DPI $\subseteq$ SI and we have presented some intensional (partial) results when such techniques are mixed with termination techniques such as RPO or DP.

Following a remark by Baillot [Bai08], two criteria could also be compared by looking at their inherent complexity but this task is not straightforward. In such a framework DPI would be strictly more efficient than QI as they capture more programs and their synthesis problems are equivalent.

Is is worth mentioning that one consequence of particular interest in the introduction of SI was the relaxation of the subterm property. This has improved the studies of sublinear TRSs and has led to works characterizing subpolynomial complexity classes such as $\texttt{Alogtime}$ [BMP06] or $\texttt{NC}^k$ [MP08b].

Chapter 3

Breaking the paradigm

Contents

In the previous chapter, we have demonstrated that ICC techniques are often difficult to compare. The main reason of this difficulty is that extensional completeness is captured but intensional completeness is sacrificed at the price of tractability. As tractability is always a reasonable requirement for an ICC technique to be effective, we come to an endless problem.

One alternative in finding new criterion with a better expressive power was to consider crossovers by trying to adapt/combine existing ICC criteria to other programming paradigms. We will discuss this interesting issue and the corresponding results obtained in the last decade in this chapter.

We start to show in Section 3.1 that tiering was successfully adapted to imperative programs including multi-threads (Subsection 3.1.2), fork process (Subsection 3.1.3), and Object-Oriented programs (Subsection 3.1.4). We discuss the type inference properties of these type systems and an extension with declassification in Subsection 3.1.5.

In Section 3.2, we focus on the extension of linear logic based approaches to other programming paradigms. We study one extension of LLL to multi-threads in Subsection 3.2.1,

one extension of SLL to a process calculus in Subsection 3.2.2, and one extension of SLL to a lambda calculus with quantum registers in Subsection 3.2.3. We discuss briefly some other extensions based on proof-nets, interaction-nets, categorical models, and realizability models in Subsection 3.2.4.

Section 3.3 presents the main extensions of the interpretation based techniques. In Subsection 3.3.1, we show how interpretations were extended to higher-order rewriting. In Subsection 3.3.2, we introduce their adaptation to a higher-order functional language and, in Subsection 3.3.3, we study their adaptation to a simple Object-Oriented programming language based on Featherweight Java [IPW01]. Several other extensions and applications are discussed in Subsection 3.3.4.

3.1 Extensions of tiering

This section is related to the extension of ramified recursion/safe recursion and, more generally, tier-based typing discipline to the imperative paradigm as initiated by the cornerstone work of Marion [Mar11]. In this section, we will survey the main results that allow the programmer to obtain polynomial time or space upper bounds on terminating and typable programs by extending the tiering technique to imperative programs [Mar11], multi-threaded programs [MP14], fork processes [HMP13], and object-oriented programs [LM13, HP15, HP18].

3.1.1 Imperative programs

In [Mar11], the leading idea is to identify the results of safe recursion [BC92] and tiering [Lei95] as a non-interference result in the context of secure flow analysis (see [SS05] for an overview of the domain). In [VIS96, SV98], Irvine, Smith, and Volpano provide a type system to certify a confidentiality policy on an imperative language and a multi-threaded language, respectively. Types are based on security levels, named High and Low. The type system prevents leak of information from level High to level Low.

In a 2-tiers based approach, the tier **0** (safe) corresponds to the High level and the tier **1** (normal) corresponds to the Low level. The type system of [Mar11] is based on an integrity policy as in [Bib77] with no read down rule rather than on a confidentiality policy as in [BLP76]. By looking at the PRN function scheme of Subsection 1.2.3 and after identifying **0** to be the type of safe data and tier **1** to be the type of normal data, one can check that data (variables) can flow from tier **1** to **0** but not the converse as illustrated below:

$$\mathtt{PRN}(f,h_0,h_1)(\underbrace{2x+i,\overline{y}}_{\mathbf{1}};\underbrace{\overline{z}}_{\mathbf{0}}) = h_i(\underbrace{x,\overline{y}}_{\mathbf{1}};\underbrace{\overline{z},\mathtt{PRN}(f,h_0,h_1)(x,\overline{y};\overline{z})}_{\mathbf{0}}).$$

Indeed tier **1** variables are used in a **0** position in the recursive call $\mathtt{PRN}(f,h_0,h_1)(x,\overline{y};\overline{z})$ and the converse never holds, *i.e.* $\overline{z}$ is never used in a tier **1** position. It can be checked easily that this property holds for any scheme in Bellantoni and Cook's algebra (see Subsection 1.2.3).

A program P enjoys a non-interference property when for any two memory configurations (maps from variables to *values*) $\mathcal{C}_1$ and $\mathcal{C}_2$, if the two configurations coincide on tier **1** (*i.e.* the data is the same for each variable of type **1**), P evaluates to memory configuration $\mathcal{C}'_i$ when fed with memory configuration $\mathcal{C}_i$ as input, for $i \in \{1,2\}$, then $\mathcal{C}'_1$ and $\mathcal{C}'_2$ coincide on tier **1**. In other words, data of tier **0** do not have control over data of tier **1**. We demonstrate a similar non-interference result in an imperative setting which states that values stored in tier **1** variables are independent from tier **0** variables.

$$\frac{}{(\mathcal{C}, \mathtt{x}) \to \mathcal{C}(\mathtt{x})}\ (\text{Var}) \qquad \frac{(\mathcal{C}, \mathtt{e}_1) \to w_1 \quad \dots \quad (\mathcal{C}, \mathtt{e}_n) \to w_n}{(\mathcal{C}, \mathtt{op}(\mathtt{e}_1, \dots, \mathtt{e}_n)) \to [\![\mathtt{op}]\!](w_1, \dots, w_n)}\ (\text{Op})$$

$$\frac{}{(\mathcal{C}, \mathtt{skip}) \to \mathcal{C}}\ (\text{Skip}) \qquad \frac{(\mathcal{C}, \mathtt{e}) \to w}{(\mathcal{C}, \mathtt{x} := \mathtt{e}) \to \mathcal{C}[\mathtt{x} \leftarrow w]}\ (\text{Asg})$$

$$\frac{(\mathcal{C}, \mathtt{c}_1) \to \mathcal{C}_1 \quad (\mathcal{C}_1, \mathtt{c}_2) \to \mathcal{C}_2}{(\mathcal{C}, \mathtt{c}_1 \ ; \ \mathtt{c}_2) \to \mathcal{C}_2}\ (\text{Seq}) \qquad \frac{(\mathcal{C}, \mathtt{e}) \to w \quad (\mathcal{C}, \mathtt{c}_w) \to \mathcal{C}' \quad w \in \{1, 0\}}{(\mathcal{C}, \mathtt{if(e)\{c_1\}\ else\ \{c_0\}}) \to \mathcal{C}'}\ (\text{Cond})$$

$$\frac{(\mathcal{C}, \mathtt{e}) \to 0}{(\mathcal{C}, \mathtt{while(e)\{c\}}) \to \mathcal{C}}\ (Wh_0) \qquad \frac{(\mathcal{C}, \mathtt{e}) \to 1 \quad (\mathcal{C}, \mathtt{c; while(e)\{c\}}) \to \mathcal{C}'}{(\mathcal{C}, \mathtt{while(e)\{c\}}) \to \mathcal{C}'}\ (Wh_1)$$

Figure 3.1: Big step operational semantics of imperative programs

For that purpose, we consider a simple imperative programming language computing on words of a fixed alphabet Σ, with $\{0, 1\} \subseteq \Sigma$, defined by the following grammar:

Expressions	$\mathtt{e}, \mathtt{e}_1, \dots, \mathtt{e}_n$	::=	$\mathtt{x} \mid \mathtt{op}(\mathtt{e}_1, \dots, \mathtt{e}_{ar(\mathtt{op})})$
Commands	$\mathtt{c}, \mathtt{c}_1, \mathtt{c}_2$	::=	$\mathtt{skip} \mid \mathtt{x} := \mathtt{e} \mid \mathtt{c}_1 \ ; \ \mathtt{c}_2$
			$\mid \mathtt{if(e)\{c_1\}\ else\ \{c_2\}} \mid \mathtt{while(e)\{c\}}$
Programs	$\mathtt{p}(\mathtt{x}_1, \dots, \mathtt{x}_n)$	::=	$\mathtt{c\ return\ x}$,

where $\mathtt{x}, \mathtt{x}_1, \dots, \mathtt{x}_n$ are variables of the countably infinite set $\mathbb{V}$ and $\mathtt{op}$ is a prefix, postfix, or infix operator of arity $ar(\mathtt{op})$ of the countably infinite set $\mathbb{O}$. A memory configuration $\mathcal{C}$ is a partial mapping from variables in $\mathbb{V}$ to words in $\mathbb{W} = \Sigma^*$. Given a symbol a in Σ and a word w in $\mathbb{W}$, let $a.w$ denote the word obtained by concatenating a and w.

The semantics of the language maps a pair $(\mathcal{C}, \mathtt{c})$ consisting in a memory configuration $\mathcal{C}$ and a command $\mathtt{c}$ to a memory configuration $\mathcal{C}'$ and is described in Figure 3.1, where $[\![\mathtt{op}]\!]$ is a total function over words associated to the operator $\mathtt{op} \in \mathbb{O}$ and $\mathcal{C}[\mathtt{x} \leftarrow w]$ is the memory configuration $\mathcal{C}'$ equal to $\mathcal{C}$ but on $\mathtt{x}$ where $\mathcal{C}'(\mathtt{x}) = w$.

A program $\mathtt{p}(\mathtt{x}_1, \dots, \mathtt{x}_n) = \mathtt{c\ return\ x}$ computes the function $f : \mathbb{W}^n \to \mathbb{W}$ if for any words $w, w_1, \dots, w_n \in \mathbb{W}$:

$$(\mathcal{C}[\mathtt{x}_1 \leftarrow w_1, \dots, \mathtt{x}_n \leftarrow w_n], \mathtt{c}) \to \mathcal{C}'[\mathtt{x} \leftarrow w] \text{ if and only if } f(w_1, \dots, w_n) = w.$$

Example 3.1.1. *Consider the program* $add(\mathtt{x}, \mathtt{y}) = \mathtt{c\ return\ y}$ *where the command* $\mathtt{c}$ *is equal to*

```
while(x > 0){
      x := x−1 ;
      y := y+1
}.
```

Given a unary word w, the operator > 0 tests whether it is empty or not, the operator -1 computes the predecessor, and the operator $+1$ computes the successor. These three operators can be defined formally as follows:

$$[\![> 0]\!](v) = \begin{cases} 1 & \text{if } \exists w \in \mathbb{W},\ v = 1.w \\ 0 & \text{otherwise} \end{cases} \qquad [\![-1]\!](v) = \begin{cases} \epsilon & \text{if } v = \epsilon \\ u & \text{if } v = a.u,\ a \in \Sigma \end{cases}$$

$$\frac{\Gamma(\mathtt{x}) = \alpha}{\Gamma, \Delta \vdash \mathtt{x} : \alpha}\ (\mathrm{V})$$

$$\frac{\forall i,\ 1 \leq i \leq n,\ \Gamma, \Delta \vdash \mathtt{e}_i : \alpha_i \qquad \alpha_1 \to \ldots \to \alpha_n \to \alpha \in \Delta(\mathtt{op})}{\Gamma, \Delta \vdash \mathtt{op}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \alpha}\ (\mathrm{OP})$$

$$\frac{}{\Gamma, \Delta \vdash \mathtt{skip} : \alpha}\ (\mathrm{SK}) \qquad \frac{\Gamma, \Delta \vdash \mathtt{x} : \alpha \qquad \Gamma, \Delta \vdash \mathtt{e} : \beta \qquad \alpha \preceq \beta}{\Gamma, \Delta \vdash \mathtt{x} := \mathtt{e} : \alpha}\ (\mathrm{A})$$

$$\frac{\Gamma, \Delta \vdash \mathtt{c} : \mathbf{0}}{\Gamma, \Delta \vdash \mathtt{c} : \mathbf{1}}\ (\mathrm{SUB}) \qquad \frac{\Gamma, \Delta \vdash \mathtt{e} : \alpha \qquad \Gamma, \Delta \vdash \mathtt{c} : \alpha \qquad \Gamma, \Delta \vdash \mathtt{c}' : \alpha}{\Gamma, \Delta \vdash \mathtt{if(e)\{c\}\ else\ \{c'\}} : \alpha}\ (\mathrm{C})$$

$$\frac{\Gamma, \Delta \vdash \mathtt{c} : \alpha \qquad \Gamma, \Delta \vdash \mathtt{c}' : \alpha}{\Gamma, \Delta \vdash \mathtt{c}\ ;\ \mathtt{c}' : \alpha}\ (\mathrm{S}) \qquad \frac{\Gamma, \Delta \vdash \mathtt{e} : \mathbf{1} \qquad \Gamma, \Delta \vdash \mathtt{c} : \alpha}{\Gamma, \Delta \vdash \mathtt{while(e)\{c\}} : \mathbf{1}}\ (\mathrm{W})$$

Figure 3.2: Tier-based imperative type system

$$[\![+1]\!](v) = 1.v.$$

This program computes the unary addition. Indeed, for $n \in \mathbb{N}$, let 1^n be the unary word where the symbol 1 occurs n times. It holds that $(\mathcal{C}[\mathtt{x} \leftarrow 1^n, \mathtt{y} \leftarrow 1^m], \mathtt{c}) \to \mathcal{C}[\mathtt{x} \leftarrow \epsilon, \mathtt{y} \leftarrow 1^{m+n}]$.

We suppose given a precedence $\preceq$ on tiers defined to be the reflexive closure of the relation $\prec$ satisfying $\mathbf{0} \prec \mathbf{1}$. The considered tier-based type system is described in Figure 3.2. It is a subsystem of the type system of [MP14] and a simplified version of the type system of [Mar11], as it does not involve a complex lattice structure. Judgments are of the shape $\Gamma, \Delta \vdash b : \alpha$, with b an expression or command, α a tier in $\{\mathbf{0}, \mathbf{1}\}$, Γ a variable typing environment mapping each variable $\mathtt{x}$ to a tier in $\{\mathbf{0}, \mathbf{1}\}$, and Δ an operator typing environment mapping each operator $\mathtt{op}$ to a set $\Delta(\mathtt{op})$ of types of the shape $\alpha_1 \to \ldots \alpha_{ar(\mathtt{op})} \to \alpha$, with $\alpha_1, \ldots, \alpha_{ar(\mathtt{op})}, \alpha \in \{\mathbf{0}, \mathbf{1}\}$.

Tiers $\mathbf{0}$ and $\mathbf{1}$ have the following intuitive meaning:

- tier $\mathbf{1}$ variables will be used as guards of while loops; they should not be allowed to take more than a polynomial number of distinct values and cannot increase,
- tier $\mathbf{0}$ variables may increase and cannot be used as while loop guards.

Before stating the main non-interference and complexity results, we need to restrict the operator typing environments under consideration. Notice that considering non-restricted operator typing environments would break both results.

Definition 3.1.1. *A polynomial time computable operator* $\mathtt{op}$ *is*

- neutral *if:*
  1. *either* $[\![\mathtt{op}]\!] : \mathbb{W}^{ar(\mathtt{op})} \to \{0, 1\}$ *is a predicate;*
  2. *or* $\forall w_1, \ldots, w_{ar(\mathtt{op})} \in \mathbb{W},\ \exists i \in \{1, \ldots, ar(\mathtt{op})\},\ [\![\mathtt{op}]\!](w_1, \ldots, w_{ar(\mathtt{op})}) \trianglelefteq w_i$.

- positive *if there is a constant* $c_{\mathtt{op}} \in \mathbb{N}$ *such that:*

$$\forall w_1, \ldots, w_{ar(\mathtt{op})} \in \mathbb{W},\ |[\![\mathtt{op}]\!](w_1, \ldots, w_{ar(\mathtt{op})})| \leq \max_i |w_i| + c_{\mathtt{op}},$$

where $\trianglelefteq$ *is the subword relation over* $\mathbb{W}$ *and the size of a word* $|w|$ *is equal to its number of symbols.*

A neutral operator is always a positive operator but the converse is not true. In the remainder, we name positive operators those operators that are positive but not neutral.

Example 3.1.2. *The operators* > 0 *and* -1 *of Example 3.1.1 defined by*

$$[\![> 0]\!](v) = \begin{cases} 1 & \text{if } \exists w \in \mathbb{W},\ v = 1.w \\ 0 & \text{otherwise} \end{cases} \qquad [\![-1]\!](v) = \begin{cases} \epsilon & \text{if } v = \epsilon \\ u & \text{if } v = a.u,\ a \in \Sigma \end{cases}$$

are neutral. Indeed, > 0 *computes a polynomial time computable predicate and* -1 *computes a subword of its input.*

The $+1$ *operator defined by*

$$[\![+1]\!](v) = 1.v$$

is positive as it is not neutral and $|[\![+1]\!](v)| = |1.v| = |v| + 1$.

Definition 3.1.2. *An operator typing environment* Δ *is* safe *if for each* $\mathtt{op} \in dom(\Delta)$, $\mathtt{op}$ *is neutral or positive and* $\forall \alpha_1 \to \ldots \to \alpha_n \to \alpha \in \Delta(\mathtt{op})$, *we have:*

- $\alpha \preceq \wedge_{i=1,\ldots,ar(\mathtt{op})} \alpha_i$,
- *if the operator* $\mathtt{op}$ *is positive then* $\alpha = \mathbf{0}$.

Definition 3.1.3 (Safe program)**.** *Given a variable typing environment* Γ *and an operator typing environment* Δ, *the program* `c return x` *is a* safe program *if there is a tier* α *such that* $\Gamma, \Delta \vdash \mathtt{c} : \alpha$ *and* Δ *is safe.*

Example 3.1.3. *Consider the program* $add(\mathtt{x}, \mathtt{y})$ *of Example 3.1.1. As* > 0 *and* -1 *are neutral and* $+1$ *is positive, the operator typing environment defined by* $\Delta(\mathtt{op}) = \{\mathbf{1} \to \mathbf{1}, \mathbf{0} \to \mathbf{0}, \mathbf{1} \to \mathbf{0}\}$, *for* $\mathtt{op} \in \{> 0, -1\}$, *and* $\Delta(+1) = \{\mathbf{0} \to \mathbf{0}\}$ *is safe.*

The program can be typed as follows (we omit the environments in the judgments and one premise in the (OP) rule in order to lighten the notation).

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma(\mathtt{x}) = \mathbf{1}}{\vdash \mathtt{x} : \mathbf{1}}\,(V)}{\vdash \mathtt{x} > 0 : \mathbf{1}}\,(OP) \qquad \dfrac{\dfrac{\dfrac{\Gamma(\mathtt{x}) = \mathbf{1}}{\vdash \mathtt{x} : \mathbf{1}}\,(V) \qquad \dfrac{\Gamma(\mathtt{x}) = \mathbf{1}}{\vdash \mathtt{x} : \mathbf{1}}\,(V)}{\vdash \mathtt{x-1} : \mathbf{1}}\,(OP)}{\vdash \mathtt{x} := \mathtt{x-1} : \mathbf{1}}\,(A) \qquad \dfrac{\dfrac{\dfrac{\Gamma(\mathtt{y}) = \mathbf{0}}{\vdash \mathtt{y} : \mathbf{0}}\,(V) \qquad \dfrac{\Gamma(\mathtt{y}) = \mathbf{0}}{\vdash \mathtt{y} : \mathbf{0}}\,(V)}{\vdash \mathtt{y+1} : \mathbf{0}}\,(OP)}{\vdash \mathtt{y} := \mathtt{y+1} : \mathbf{0}}\,(A)}{\vdash \mathtt{y} := \mathtt{y+1} : \mathbf{1}}\,(SUB)}{\vdash \mathtt{x} := \mathtt{x-1}\ ;\ \mathtt{y} := \mathtt{y+1} : \mathbf{1}}\,(S)}{\vdash \mathtt{while}(\mathtt{x} > 0)\{\mathtt{x} := \mathtt{x-1}\ ;\ \mathtt{y} := \mathtt{y+1}\} : \mathbf{1}}\,(W)$$

For a given variable typing environment Γ and a given tier α, let $\approx_\alpha^\Gamma$ be an equivalence relation on memory configurations defined by $\mathcal{C} \approx_\alpha^\Gamma \mathcal{C}'$ if $\forall \mathtt{x} \in dom(\Gamma), \alpha \preceq \Gamma(\mathtt{x}) \implies \mathcal{C}(\mathtt{x}) = \mathcal{C}'(\mathtt{x})$. We are now ready to state the main non-interference and complexity results.

Theorem 3.1.1 (Non-interference)**.** *Given a safe program* `c return x` *with respect to the typing environments* Γ, Δ*. For any stores* $\mathcal{C}_1$ *and* $\mathcal{C}_2$*, if* $\mathcal{C}_1 \approx_{\mathbf{1}}^{\Gamma} \mathcal{C}_2$*,* $(\mathcal{C}_1, \mathtt{c}) \to \mathcal{C}_1'$*, and* $(\mathcal{C}_2, \mathtt{c}) \to \mathcal{C}_2'$*, then* $\mathcal{C}_1' \approx_{\mathbf{1}}^{\Gamma} \mathcal{C}_2'$*.*

Example 3.1.4. *We have shown in Example 3.1.3 that the program of Example 3.1.1 can be typed with respect to a variable typing environment* Γ *such that* $\Gamma(\mathtt{x}) = \mathbf{1}$ *and* $\Gamma(\mathtt{y}) = \mathbf{0}$*. For* $n, m, l \in \mathbb{N}$*, we have* $(\mathcal{C}[\mathtt{x} \leftarrow 1^n, \mathtt{y} \leftarrow 1^m], \mathtt{c}) \to \mathcal{C}[\mathtt{x} \leftarrow \epsilon, \mathtt{y} \leftarrow 1^{m+n}]$ *and* $(\mathcal{C}[\mathtt{x} \leftarrow 1^n, \mathtt{y} \leftarrow 1^l], \mathtt{c}) \to \mathcal{C}[\mathtt{x} \leftarrow \epsilon, \mathtt{y} \leftarrow 1^{n+l}]$*. Moreover* $\mathcal{C}[\mathtt{x} \leftarrow 1^n, \mathtt{y} \leftarrow 1^m] \approx_{\mathbf{1}}^{\Gamma} \mathcal{C}[\mathtt{x} \leftarrow 1^n, \mathtt{y} \leftarrow 1^l]$*. Consequently, by Theorem 3.1.1,* $\mathcal{C}[\mathtt{x} \leftarrow \epsilon, \mathtt{y} \leftarrow 1^{n+m}] \approx_{\mathbf{1}}^{\Gamma} \mathcal{C}[\mathtt{x} \leftarrow \epsilon, \mathtt{y} \leftarrow 1^{n+l}]$*, which is obviously true.*

Theorem 3.1.2 ([Mar11, MP14])**.** *The set of functions over words computed by safe and terminating programs is exactly* `FP`*.*

In the above Theorem, soundness is a consequence of non-interference together with the fact that the cardinality of the set of tier $\mathbf{1}$ configurations is polynomially bounded in the input size (*i.e.* the cardinal of the space of distinct values that can be obtained as outputs of neutral operators is bounded polynomially in the size of the initial values). Consequently, if a program terminates then only a polynomial number of memory configurations distinct on tier $\mathbf{1}$ data can be encountered. Completeness is demonstrated as usual by simulating polynomials and a standard TM.

Example 3.1.5. *The program of Example 3.1.1 is safe and terminating. Consequently, it computes a function in* `FP`*.*

In what follows, let $\mathtt{e}^{\alpha}$, respectively $\mathtt{c} : \alpha$, be a notation meaning that the expression `e`, respectively command `c`, is of tier α under the considered typing environments.

Example 3.1.6. *Consider the following program that computes the exponential as a counter-example.*

```
while(x^1 > 0){
    z^? := y^0 : ? ;
    while(z^? > 0){
        y^0 := y^0 + 1 : 0 ;
        z^? := z^? − 1 : ?
    } ; : 1
    x^1 := x^1 − 1 : 1
}
return y
```

It is not typable in our formalism. Indeed, suppose that it is typable. The command $\mathtt{y} := \mathtt{y} + 1$ *enforces* `y` *to be of tier* $\mathbf{0}$ *since* $+1$ *is positive. Consequently, the command* $\mathtt{z} := \mathtt{y}$ *enforces* `z` *to be of tier* $\mathbf{0}$ *because of typing discipline for assignments. However, the innermost while loop enforces* $\mathtt{z} > 0$ *to be of tier* $\mathbf{1}$*, so that* `z` *has to be of tier* $\mathbf{1}$ *(because* $\mathbf{0} \to \mathbf{1}$ *is not permitted for a safe operator typing environment) and we obtain a contradiction.*

3.1.2 Multi-threaded programs

An extension to a multi-threaded language has been considered in [MP14]. A multi-threaded program $M(\mathtt{x}_1, \ldots, \mathtt{x}_n)$, M for short, is a map from a finite set of threads identifier in $dom(M)$ to commands. In this setting, thread creation is prohibited. For a multi-threaded program M, the store $\mathcal{C}$ plays the role of a global shared memory and is the only way for threads to communicate.

$$\frac{}{(\mathcal{C}, \mathtt{x}) \to_{\mathtt{e}} \mathcal{C}(\mathtt{x})}\ (\text{Var}) \qquad \frac{(\mathcal{C}, \mathtt{e}_1) \to_{\mathtt{e}} w_1 \quad \dots \quad (\mathcal{C}, \mathtt{e}_n) \to_{\mathtt{e}} w_n}{\mathcal{C}, \mathtt{op}(\mathtt{e}_1, \dots, \mathtt{e}_n) \to_{\mathtt{e}} [\![\mathtt{op}]\!](w_1, \dots, w_n)}\ (\text{Op})$$

$$\frac{}{(\mathcal{C}, \mathtt{skip}\ ;\ \mathtt{c}) \to_{\mathtt{c}} (\mathcal{C}, \mathtt{c})}\ (\text{Skp}) \qquad \frac{(\mathcal{C}, \mathtt{e}) \to_{\mathtt{e}} w}{(\mathcal{C}, \mathtt{x} := \mathtt{e}) \to_{\mathtt{c}} (\mathcal{C}[\mathtt{x} \leftarrow w], \mathtt{skip})}\ (\text{Asg})$$

$$\frac{(\mathcal{C}, \mathtt{c}_1) \to_{\mathtt{c}} (\mathcal{C}_1, \mathtt{c}_1')}{(\mathcal{C}, \mathtt{c}_1\ ;\ \mathtt{c}_2) \to_{\mathtt{c}} (\mathcal{C}_1, \mathtt{c}_1'; \mathtt{c}_2)}\ (\text{Seq}) \qquad \frac{(\mathcal{C}, \mathtt{e}) \to_{\mathtt{c}} w,\ \ w \in \{0, 1\}}{(\mathcal{C}, \mathtt{if(e)\{c_1\}\ else\ \{c_0\}}) \to_{\mathtt{c}} (\mathcal{C}, \mathtt{c}_w)}\ (\text{Cond})$$

$$\frac{(\mathcal{C}, \mathtt{e}) \to_{\mathtt{c}} 0}{(\mathcal{C}, \mathtt{while(e)\{c\}}) \to_{\mathtt{c}} (\mathcal{C}, \mathtt{skip})}\ (\text{W}_0) \qquad \frac{(\mathcal{C}, \mathtt{e}) \to_{\mathtt{c}} 1}{(\mathcal{C}, \mathtt{while(e)\{c\}}) \to_{\mathtt{c}} (\mathcal{C}, \mathtt{c}; \mathtt{while(e)\{c\}})}\ (\text{W}_1)$$

$$\frac{M(x) = \mathtt{skip}}{(\mathcal{C}, M) \to_{\mathtt{nd}} (\mathcal{C}, M - x)}\ (\text{NDStop}) \qquad \frac{M(x) = \mathtt{c} \quad (\mathcal{C}, \mathtt{c}) \to_{\mathtt{c}} (\mathcal{C}_1, \mathtt{c}_1)}{(\mathcal{C}, M) \to_{\mathtt{nd}} (\mathcal{C}_1, M[x := \mathtt{c}_1])}\ (\text{NDStep})$$

Figure 3.3: Small step operational semantics of multi-threads

Non-deterministic scheduling

The small step operational semantics of multi-threads corresponds to the global relation $\to_{\mathtt{nd}}$ in Figure 3.3. Let ${\to_{\mathtt{nd}}}^n$ be its n-fold composition. In Figure 3.3, $M - x$ is the restriction of M to $dom(M) - \{x\}$ and $M[x := \mathtt{c}]$ is a notation for the multi-threaded program M where the command assigned to x is updated to $\mathtt{c}$. At each step, a thread x is chosen using a fixed non-deterministic scheduling policy. Then, one step of x is performed and the control returns to the upper level. Note that the rule *(Stop)* halts the computation of a thread.

Let $\emptyset$ be a notation for the empty multi-threaded program where all threads have terminated. A multi-threaded program $M(\mathtt{x}_1, \dots, \mathtt{x}_n)$ strongly terminates if for any store $\mathcal{C}$ and any derivation $(\mathcal{C}, M) {\to_{\mathtt{nd}}}^n (\mathcal{C}', M')$ there exist a store $\mathcal{C}''$ and $m \in \mathbb{N}$ such that $(\mathcal{C}', M') {\to_{\mathtt{nd}}}^m (\mathcal{C}'', \emptyset)$.

The running time of a multi-threaded program $M(\mathtt{x}_1, \dots, \mathtt{x}_n)$ on inputs $w_1, \dots, w_n \in \mathbb{W}$, noted $time_M$, is a partial function defined by:

$$time_M(w_1, \dots, w_n) = max\{n \mid \exists \mathcal{C}',\ (\mathcal{C}[\mathtt{x}_1 \leftarrow w_1, \dots, \mathtt{x}_n \leftarrow w_n], M) {\to_{\mathtt{nd}}}^n (\mathcal{C}', \emptyset)\}.$$

In the special case where M strongly terminates, $time_M$ is a total function.

The type system of Figure 3.2 can be extended to multi-threaded programs by the rule of Figure 3.4. The judgment $\Gamma, \Delta \vdash M : \diamond$ means that the multi-thread M is well-typed under the variable typing environment Γ and the operator typing environment Δ.

The notion of safe programs can be extended to safe multi-threaded programs as follows:

Definition 3.1.4. *Given Γ a variable typing environment and Δ an operator typing environment, the multi-threaded program M is* safe *if $\Gamma, \Delta \vdash M : \diamond$ and Δ is safe.*

$$\frac{\forall x \in dom(M),\ \exists \alpha \in \{\mathbf{0},\mathbf{1}\},\ \Gamma,\Delta \vdash M(x) : \alpha}{\Gamma,\Delta \vdash M : \diamond}\ (\mathrm{W})$$

Figure 3.4: Tier-based multi-threads typing rule

Definition 3.1.5. *Let Γ be a variable typing environment and Δ be an operator typing environment.*

- *The relation $\approx_{\Gamma,\Delta}$ on commands is defined by:*
  1. *If $\mathtt{c}_1 = \mathtt{c}_2$ then $\mathtt{c}_1 \approx_{\Gamma,\Delta} \mathtt{c}_2$,*
  2. *If $\Gamma,\Delta \vdash \mathtt{c}_1 : \mathbf{0}$ and $\Gamma,\Delta \vdash \mathtt{c}_2 : \mathbf{0}$ then $\mathtt{c}_1 \approx_{\Gamma,\Delta} \mathtt{c}_2$,*
  3. *If $\mathtt{c}_1 \approx_{\Gamma,\Delta} \mathtt{c}_2$ and $\mathtt{c}_3 \approx_{\Gamma,\Delta} \mathtt{c}_4$ then $\mathtt{c}_1;\mathtt{c}_3 \approx_{\Gamma,\Delta} \mathtt{c}_2;\mathtt{c}_4$.*
- *It is extended to configurations as follows*
  If $\mathtt{c}_1 \approx_{\Gamma,\Delta} \mathtt{c}_2$ and $\mathcal{C} \approx_{\mathbf{1}}^{\Gamma} \mathcal{C}'$ then $(\mathcal{C},\mathtt{c}_1) \approx_{\Gamma,\Delta} (\mathcal{C}',\mathtt{c}_2)$.
- *Finally, it is extended to multi-threads and multi-thread configurations as follows:*
  - *If $\forall x \in dom(M) = dom(M'),\ M(x) \approx_{\Gamma,\Delta} M'(x)$ then $M \approx_{\Gamma} M'$,*
  - *If $M \approx_{\Gamma,\Delta} M'$ and $\mathcal{C} \approx_{\mathbf{1}}^{\Gamma} \mathcal{C}'$ then $(\mathcal{C},M) \approx_{\Gamma,\Delta} (\mathcal{C}',M')$.*

We obtain a concurrent non-interference property.

Theorem 3.1.3 (Concurrent non-interference)**.** *Let M_1 and M_2 be two safe multi-threaded programs with respect to the variable typing environment Γ and the operator typing environment Δ and let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two memory configurations such that $(\mathcal{C}_1,M_1) \approx_{\Gamma,\Delta} (\mathcal{C}_2,M_2)$.*

If $(\mathcal{C}_1,M_1) \to_{\mathtt{nd}} (\mathcal{C}_1',M_1')$ then there are $\mathcal{C}_2'$, M_2', and $n \in \mathbb{N}$ such that $(\mathcal{C}_2,M_2) \to_{\mathtt{nd}}{}^{n} (\mathcal{C}_2',M_2')$ and $(\mathcal{C}_1',M_1') \approx_{\Gamma,\Delta} (\mathcal{C}_2',M_2)$.

It is worth noticing that the above result also holds in a sequential context when only the relation $\to_{\mathtt{c}}$ on commands is considered. Moreover, temporal variants relating the cost of evaluating `while` constructs with the rule (W_1) are also considered in [MP14].

An upper-bound on the derivation length can also be obtained in the case of strongly terminating multi-threads.

Theorem 3.1.4 ([MP14])**.** *Assume that $M(\mathtt{x}_1,\ldots,\mathtt{x}_n)$ is a safe multi-threaded program that strongly terminates. There is a polynomial $Q \in \mathbb{N}[X]$ such that*

$$\forall w_1,\ldots,w_n \in \mathbb{W},\ time_M(w_1,\ldots,w_n) \leq Q(\max_{i=1}^{n}(|w_i|)).$$

The soundness of Theorem 3.1.2 can be viewed as a direct application of Theorem 3.1.4 in the particular case of a multi-thread consisting in one single thread.

Example 3.1.7 ([MP14])**.** *Consider the following multi-thread M composed of two threads x and y computing on unary numbers.*

$x:$ `while`$(\mathtt{x}^{\mathbf{1}} > 0)^{\mathbf{1}}\{$
$\quad \mathtt{z}^{\mathbf{0}} := \mathtt{z}^{\mathbf{0}} + 1 : \mathbf{0}$;
$\quad \mathtt{x}^{\mathbf{1}} := \mathtt{x}^{\mathbf{1}} - 1 : \mathbf{1}$
$\}:\mathbf{1}$

$y:$ `while`$(\mathtt{y}^{\mathbf{1}} > 0)^{\mathbf{1}}\{$
$\quad \mathtt{z}^{\mathbf{0}} := 0 : \mathbf{0}$;
$\quad \mathtt{y}^{\mathbf{1}} := \mathtt{y}^{\mathbf{1}} - 1 : \mathbf{1}$
$\}:\mathbf{1}$

This program is strongly terminating. Moreover, given a store $\mathcal{C}$ such that $\mathcal{C}(\mathtt{x}) = n$ and $\mathcal{C}(\mathtt{z}) = 0$, if $(\mathcal{C}, M) \to_{nd}^{k} (\mathcal{C}', \emptyset)$ then $\mathcal{C}'(\mathtt{z}) \in [0, n]$. M is safe using an operator typing environment Δ such that $\Delta(-1) = \Delta(> 0) = \{\mathbf{1} \to \mathbf{1}\}$ and $\Delta(+1) = \{\mathbf{0} \to \mathbf{0}\}$ and M strongly terminates. Consequently, by Theorem 3.1.4, it terminates in polynomial time.

Example 3.1.8 ([MP14])**.** *Consider the following multi-thread M that shuffles two strings given as inputs.*

$x:$ `while`$(!(\mathtt{x}^{\mathbf{1}} == \epsilon))^{\mathbf{1}}\{$
$\quad \mathtt{z}^{\mathbf{0}} := \mathtt{cons}(\mathtt{head}(\mathtt{x}^{\mathbf{1}}), \mathtt{z}^{\mathbf{0}}) : \mathbf{0}$;
$\quad \mathtt{x}^{\mathbf{1}} := \mathtt{x}^{\mathbf{1}} - 1 : \mathbf{1}$
$\}:\mathbf{1}$

$y:$ `while`$(!(\mathtt{y}^{\mathbf{1}} == \epsilon))^{\mathbf{1}}\{$
$\quad \mathtt{z}^{\mathbf{0}} := \mathtt{cons}(\mathtt{head}(\mathtt{y}^{\mathbf{1}}), \mathtt{z}^{\mathbf{0}}) : \mathbf{0}$;
$\quad \mathtt{y}^{\mathbf{1}} := \mathtt{y}^{\mathbf{1}} - 1 : \mathbf{1}$
$\}:\mathbf{1}$

where `cons` *is an operator defined by $[\![\mathtt{cons}]\!](\epsilon, w) = w$ and $[\![\mathtt{cons}]\!](a.v, w) = a.w$, that performs the concatenation of the symbol given in its first argument with its second argument. The operators ! and $==\epsilon$ are unary predicates and consequently can be typed by $\mathbf{1} \to \mathbf{1}$. The operator head returns the first symbol of a string given as input and can be typed by $\mathbf{1} \to \mathbf{0}$ since it is neutral. The -1 operator can typed by $\mathbf{1} \to \mathbf{1}$ since its computation is a subterm of the input. Finally, the* `cons` *operator can be typed by $\mathbf{0} \to \mathbf{0} \to \mathbf{0}$ since $|[\![\mathtt{cons}]\!](u, v)| = |v| + 1$. This program is safe and strongly terminating consequently it also terminates in polynomial time.*

Example 3.1.9 ([MP14])**.** *Consider the following multi-thread M.*

$x:$ `while`$(\mathtt{x}^{\mathbf{1}} > 0)^{\mathbf{1}}\{$
$\quad \mathtt{y}^{\mathbf{1}} := \mathtt{x}^{\mathbf{1}};:\mathbf{1}$
$\quad \mathtt{x}^{\mathbf{1}} := \mathtt{x}^{\mathbf{1}} - 1;:\mathbf{1}$
$\}:\mathbf{1}$

$y:$ `while`$(\mathtt{y}^{\mathbf{1}} > 0)^{\mathbf{1}}\{$
$\quad \mathtt{z}^{\mathbf{0}} := \mathtt{z}^{\mathbf{0}} + 1;:\mathbf{0}$
$\quad \mathtt{y}^{\mathbf{1}} := \mathtt{y}^{\mathbf{1}} - 1;:\mathbf{1}$
$\}:\mathbf{1}$

Observe that, contrarily to previous examples, the guard of `y` *depends on information flowing from x to y. Given a store $\mathcal{C}$ such that $\mathcal{C}(\mathtt{x}) = n$, $\mathcal{C}(\mathtt{y}) = \mathcal{C}(\mathtt{z}) = 0$, if $(\mathcal{C}, M) \to_{\mathtt{nd}}^{k} (\mathcal{C}', \emptyset)$ then $\mathcal{C}'(\mathtt{z}) \in [0, n \times (n+1)/2]$. This multi-thread is safe with respect to a safe typing operator environment Δ such that $\Delta(-1) = \Delta(> 0) = \{\mathbf{1} \to \mathbf{1}\}$ and $\Delta(+1) = \{\mathbf{0} \to \mathbf{0}\}$. Moreover it strongly terminates. Consequently, it also terminates in polynomial time.*

Deterministic scheduling

We extend previous results to a class of deterministic schedulers. Define $\mathcal{C}{\downarrow}\mathbf{1}$ as the restriction of the store $\mathcal{C}$ to tier $\mathbf{1}$ variables. A deterministic scheduler $\mathcal{S}$ is *quiet* if the scheduling policy depends only on the current state of the multi-threaded program M and on $\mathcal{C}{\downarrow}\mathbf{1}$. For example, a deterministic scheduler whose policy just depends on running threads, is quiet whereas a deterministic scheduler depending on $\mathbf{0}$ data is not quiet.

Next, we replace the two non-deterministic global transitions (NDStop) and (NDStep) of Figure 3.3 by the rules of Figure 3.5. As in previous section, the running time of a multi-threaded program $M(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ on inputs $w_1, \ldots, w_n \in \mathbb{W}$ under the quiet scheduler $\mathcal{S}$, noted $time_M^{\mathcal{S}}$, is a partial function defined by:

$$time_M^{\mathcal{S}}(w_1, \ldots, w_n) = n \mid \exists \mathcal{C}',\ (\mathcal{C}[\mathtt{x}_1 \leftarrow w_1, \ldots, \mathtt{x}_n \leftarrow w_n], M) \to_{\mathtt{d}}^{n} (\mathcal{C}', \emptyset).$$

In the special case where M terminates under the strategy $\mathcal{S}$, $time_M^{\mathcal{S}}$ is a total function.

$$\frac{\mathcal{S}(M,\mathcal{C}\downarrow\mathbf{1}) = x \qquad M(x) = \texttt{skip}}{(\mathcal{C},M) \to_{\mathtt{d}} (\mathcal{C},M-x)}\ (\text{DStop}) \qquad \frac{\mathcal{S}(M,\mathcal{C}\downarrow\mathbf{1}) = x \qquad (\mathcal{C},M(x))\to_{\mathtt{c}}(\mathcal{C}',\mathtt{c}')}{(\mathcal{C},M) \to_{\mathtt{d}} (\mathcal{C}',M[x := \mathtt{c}'])}\ (\text{DStep})$$

Figure 3.5: Small step operational semantics with deterministic scheduling

Theorem 3.1.5 ([MP14]). *Assume that M is a safe multi-threaded program that terminates with respect to the deterministic and quiet scheduler $\mathcal{S}$. There is a polynomial Q such that:*

$$\forall w_1,\ldots,w_n \in \mathbb{W},\ time^{\mathcal{S}}_M(w_1,\ldots,w_n) \leq Q(\max_{i=1,n}(|w_i|)).$$

3.1.3 Fork processes

We now consider the extension of tiering to fork processes presented in [HMP13]. This extension requires the addition of a new tier $-\mathbf{1}$ for data communication between processes and allows us to characterize the class of polynomial space computable functions `FPSPACE`.

Processes consist in programs of Section 3.1.1 extended with `fork` and `wait` constructs as follows.

`Expressions`	$\mathtt{e},\mathtt{e}_1,\ldots,\mathtt{e}_n$	::=	$\mathtt{x} \mid \mathtt{op}(\mathtt{e}_1,\ldots,\mathtt{e}_{ar(\mathtt{op})})$
`Commands`	$\mathtt{c},\mathtt{c}_1,\mathtt{c}_2$	::=	$\mathtt{fork}() \mid \mathtt{wait}(\mathtt{e})$
			$\mid \mathtt{skip} \mid \mathtt{x} := \mathtt{e} \mid \mathtt{c}_1 \,;\, \mathtt{c}_2$
			$\mid \mathtt{if}(\mathtt{e})\{\mathtt{c}_1\}\ \mathtt{else}\ \{\mathtt{c}_2\} \mid \mathtt{while}(\mathtt{e})\{\mathtt{c}\}$
`Processes`	P	::=	$\mathtt{c} \,;\, \mathrm{P} \mid \mathtt{return}\ \mathtt{x}$

Each `fork` call creates a new child process with a distinct identifier (id) and duplicates the execution context, *i.e.* the store, including the program counter. The parent process keeps track of the ids of its children but not the converse. The communications between children and their parent are performed through the use of a `wait` instruction that allows a returning child to pass a value to its parent process. This programming language is simple but it is a natural fragment of a real-life programming language like `C` [KP84].

Given a store $\mathcal{C}$ and a process P, the triplet $c = (\mathrm{P},\mathcal{C})_\rho$, where ρ is an element of $\mathcal{P}(\mathbb{N})$, is called a *configuration*. In a configuration $c = (\mathrm{P},\mathcal{C})_\rho$, the set ρ will contain the indexes of the children of the process P created during an evaluation. Let $\perp$ be a special symbol for erased configurations.

An *environment* $\mathcal{E}$ is a partial function from $\mathbb{N}$ to configurations. The domain of $\mathcal{E}$ is denoted $dom(\mathcal{E})$ and we denote $\sharp\mathcal{E}$ its cardinal when it is finite. We abbreviate $\mathcal{E}(n)$ by $\mathcal{E}_n$. The size of an environment $|\mathcal{E}|$ is defined by $|\mathcal{E}| = \sum_{i\in dom(\mathcal{E})} |\mathcal{E}_i|$. The notation $\mathcal{E}[i := c]$ is the environment $\mathcal{E}'$ defined by $\mathcal{E}'(j) = \mathcal{E}(j)$ for all $j \neq i \in dom(\mathcal{E})$ and $\mathcal{E}'(i) = c$. As usual $\mathcal{E}[i_1 := c_1,\ldots,i_k := c_k]$ is a shortcut for $\mathcal{E}[i_1 := c_1]\ldots[i_k := c_k]$. The *initial* environment, noted $\mathcal{E}_{init}[\mathrm{P},\mathcal{C}]$, consists in the main process with no child. That is $\mathcal{E}_{init}[\mathrm{P},\mathcal{C}](1) = (\mathrm{P},\mathcal{C})_\emptyset$ and $dom(\mathcal{E}_{init}[\mathrm{P},\mathcal{C}]) = \{1\}$. An environment $\mathcal{E}$ is *terminal* if the root process satisfies $\mathcal{E}_1 = (\mathtt{return}\ \ \mathtt{x},\mathcal{C})_\rho$, *i.e.* the main process is returning.

The small step semantics of process configurations and environments is presented in Figure 3.6. It consists in an extension of the sequential fragment $\to_{\mathtt{c}}$ of the small step semantics of

$$(\mathtt{c}\ ;\ \mathrm{P}, \mathcal{C})_\rho \to (\mathtt{c}'\ ;\ \mathrm{P}, \mathcal{C}')_\rho \quad \text{if } (\mathtt{c}, \mathcal{C}) \to_{\mathtt{c}} (\mathtt{c}', \mathcal{C}')$$

$$\mathcal{E}[i := c] \to \mathcal{E}[i := c'] \quad \text{if } c \to c' \qquad \text{(Conf)}$$

$$\mathcal{E}[i := (\mathtt{x} := \mathtt{fork}()\ ;\ \mathrm{P}, \mathcal{C})_\rho] \to \mathcal{E}[i := (\mathrm{P}, \mathcal{C}\{\mathtt{x} \leftarrow \underline{n}\})_{\rho \cup \{n\}}, n := (\mathrm{P}, \mathcal{C}\{\mathtt{x} \leftarrow \underline{0}\})_\emptyset] \qquad \text{(Fork)}$$
with $n = \sharp\mathcal{E} + 1$

$$\mathcal{E}[i := (\mathtt{x} := \mathtt{wait}(\mathtt{e})\ ;\ \mathrm{P}, \mathcal{C})_\rho] \to \mathcal{E}[i := (\mathrm{P}, \mathcal{C}\{\mathtt{x} \leftarrow \mathcal{C}'(\mathtt{y})\})_\rho, n := \bot] \qquad \text{(Wait)}$$
if $(\mathtt{e}, \mathcal{C}) \to_{\mathtt{e}} \underline{n}$, $n \in \rho$ and $\mathcal{E}_n = (\mathtt{return\ \ y}, \mathcal{C}')_{\rho'}$

Figure 3.6: Small step operational semantics of environments

multi-threads presented in Figure 3.3 to configurations together with the definition of the small step semantics $\to$ for environments.

All standard sequential commands are evaluated using rule (Conf).

The rule (Fork) creates a new configuration, a new child process, and a new store, and adds them to the environment by extending the environment domain. The parent process keeps track of the new configuration by recording its id $\underline{n}$ inside variable $\mathtt{x}$. The child id is initialized to the value $\sharp\mathcal{E} + 1$ not to conflict with other ids. Moreover, the child set of the parent configuration is updated to $\rho \cup \{n\}$.

The rule (Wait) evaluates the expression $\mathtt{e}$ to some binary numeral $\underline{n}$. If $\underline{n}$ is equal to the id of a *terminating configuration* $\mathcal{E}_n$, with $n \in \rho$, (*i.e.* a configuration of the shape $(\mathtt{return\ \ y}, \mathcal{C}')_{\rho'}$) then the output value $\mathcal{C}'(\mathtt{y})$ is transmitted and stored in variable $\mathtt{x}$. The returning process n is deleted by $n := \bot$.

A process P is *strongly normalizing* if there is no infinite reduction starting from the initial environment $\mathcal{E}_{init}[\mathrm{P}, \mathcal{C}]$ through the relation $\to$, for any store $\mathcal{C}$.

Given an initial environment $\mathcal{E}_{init}[\mathrm{P}, \mathcal{C}]$, for some strongly normalizing process P and some store $\mathcal{C}$, if $\mathcal{E}_{init}[\mathrm{P}, \mathcal{C}] \to^* \mathcal{E}'$, for some environment $\mathcal{E}'$ such that there is no environment $\mathcal{E}''$, $\mathcal{E}' \to \mathcal{E}''$, then either $\mathcal{E}'$ is a terminal configuration, *i.e.* $\mathcal{E}'_1 = (\mathtt{return\ \ x}, \mathcal{C}')_\rho$ (It means that the main process is returning), or $\mathcal{E}'_1 = (\mathtt{x} := \mathtt{wait}(\mathtt{e})\ ;\ \mathtt{c}', \mathcal{C}')_\rho$ (We say that the environment $\mathcal{E}'$ is locked). A process $\mathrm{P} = \mathtt{c}\ ;\ \mathtt{return\ \ x}$ is *lock-free* if for any initial environment $\mathcal{E}_{init}[\mathrm{P}, \mathcal{C}]$, there is no locked environment $\mathcal{E}'$ such that $\mathcal{E}_{init}[\mathrm{P}, \mathcal{C}] \to_* \mathcal{E}'$.

A process P is *confluent* if for each initial environment $\mathcal{E}_{init}[\mathrm{P}, \mathcal{C}]$ and any two reductions $\mathcal{E}_{init}[\mathrm{P}, \mathcal{C}] \to^* \mathcal{E}'$ and $\mathcal{E}_{init}[\mathrm{P}, \mathcal{C}] \to^* \mathcal{E}''$ there exists an environment $\mathcal{E}^3$ such that $\mathcal{E}' \to^* \mathcal{E}^3$ and $\mathcal{E}'' \to^* \mathcal{E}^3$.

A strongly normalizing, lock free, and confluent process P computes a total function $f : \mathbb{W}^n \to \mathbb{W}$ defined by:

$$\forall w_1, \ldots, w_n \in \mathbb{W},\ f(w_1, \ldots, w_n) = w$$

if $\mathcal{E}_{init}[\mathrm{P}, \mathcal{C}[\mathtt{x}_i \leftarrow w_i]] \to^* \mathcal{E}$, for some terminal environment $\mathcal{E}$ with $\mathcal{E}_1 = (\mathtt{return\ \ x}, \mathcal{C}')_\rho$ and $\mathcal{C}'(\mathtt{x}) = w$.

The tier based analysis can now be adapted to this programming language. In this particular setting, we need a three tier based lattice $(\{-\mathbf{1}, \mathbf{0}, \mathbf{1}\}, \vee, \wedge)$. The induced order is such that $-\mathbf{1} \preceq \mathbf{0} \preceq \mathbf{1}$. Typing environments are defined in a standard way.

The new tier $-\mathbf{1}$ has the following intuitive meaning: tier $-\mathbf{1}$ variables will store values returned by child processes and cannot increase. They play the role of a shared memory that

$$\frac{\Gamma(\mathtt{x}) = \alpha}{\Gamma, \Delta \vdash \mathtt{x} : \alpha}\ (\mathrm{V})$$

$$\frac{\forall i,\ 1 \leq i \leq n,\ \Gamma, \Delta \vdash \mathtt{e}_i : \alpha_i \qquad \alpha_1 \to \ldots \to \alpha_n \to \alpha \in \Delta(\mathtt{op})}{\Gamma, \Delta \vdash \mathtt{op}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \alpha}\ (\mathrm{OP})$$

$$\frac{}{\Gamma, \Delta \vdash \mathtt{skip} : \alpha}\ (\mathrm{SK}) \qquad \frac{\Gamma, \Delta \vdash \mathtt{x} : \alpha \qquad \Gamma, \Delta \vdash \mathtt{e} : \beta \qquad \alpha \preceq \beta}{\Gamma, \Delta \vdash \mathtt{x} := \mathtt{e} : \alpha}\ (\mathrm{A})$$

$$\frac{\Gamma, \Delta \vdash \mathtt{c} : \mathbf{0}}{\Gamma, \Delta \vdash \mathtt{c} : \mathbf{1}}\ (\mathrm{SUB}) \qquad \frac{\Gamma, \Delta \vdash \mathtt{e} : \alpha \qquad \Gamma, \Delta \vdash \mathtt{c} : \alpha \qquad \Gamma, \Delta \vdash \mathtt{c}' : \alpha}{\Gamma, \Delta \vdash \mathtt{if(e)\{c\}\ else\ \{c'\}} : \alpha}\ (\mathrm{C})$$

$$\frac{\Gamma, \Delta \vdash \mathtt{c} : \alpha \qquad \Gamma, \Delta \vdash \mathtt{c}' : \alpha}{\Gamma, \Delta \vdash \mathtt{c}\ ;\ \mathtt{c}' : \alpha}\ (\mathrm{S}) \qquad \frac{\Gamma, \Delta \vdash \mathtt{e} : \mathbf{1} \qquad \Gamma, \Delta \vdash \mathtt{c} : \alpha}{\Gamma, \Delta \vdash \mathtt{while(e)\{c\}} : \mathbf{1}}\ (\mathrm{WH})$$

$$\frac{\Gamma, \Delta \vdash \mathtt{x} : \mathbf{0}}{\Gamma, \Delta \vdash \mathtt{x} := \mathtt{fork}() : \mathbf{0}}\ (\mathrm{F}) \qquad \frac{\Gamma, \Delta \vdash \mathtt{e} : \mathbf{0} \qquad \Gamma, \Delta \vdash \mathtt{x} : -\mathbf{1}}{\Gamma, \Delta \vdash \mathtt{x} := \mathtt{wait(e)} : -\mathbf{1}}\ (\mathrm{W})$$

$$\frac{\Gamma, \Delta \vdash \mathtt{c} : \alpha \qquad \alpha \preceq \beta}{\Gamma, \Delta \vdash \mathtt{c} : \beta}\ (\mathrm{SUB}) \qquad \frac{\Gamma, \Delta \vdash \mathtt{x} : \alpha}{\Gamma, \Delta \vdash \mathtt{return\ \ x} : \alpha}\ (\mathrm{RET})$$

Figure 3.7: Tier-based processes type system

cannot accumulate data.

Typing rules for processes consist in the rules of Figure 3.7. One can check that the 8 first rules are similar to the rules of Figure 3.2. Consequently, the type system of Figure 3.7 is a strict extension of the type system of Figure 3.2.

As in previous type systems, the typing discipline precludes values from flowing from tier α to tier β, whenever $\alpha \preceq \beta$. The (F) typing rule enforces the tier of the variable storing the process id to be of tier $\mathbf{0}$ since the value stored will increase dynamically during the process execution. Finally, in rule (W), the tier of the variable storing the result returned by a child process has to be of tier $-\mathbf{1}$, which means that no information may flow from a variable of a child process to tier $\mathbf{0}$ and tier $\mathbf{1}$ variables of its parent process.

Again, we need to restrict the considered operators in the operator typing environment Δ. We extend the notions of neutral and positive operators of Definition 3.1.1 with the notions of max and polynomial operators.

Definition 3.1.6. *A polynomial time computable operator* `op` *is*

1. neutral *if:*

(a) *either* $[\![\mathtt{op}]\!] : \mathbb{W}^{ar(\mathtt{op})} \to \{0,1\}$ *is a predicate,*

(b) *or* $\forall w_1, \ldots, w_{ar(\mathtt{op})} \in \mathbb{W},\ \exists i \in \{1, \ldots, ar(\mathtt{op})\},\ [\![\mathtt{op}]\!](w_1, \ldots, w_{ar(\mathtt{op})}) \trianglelefteq w_i$.

2. max *if for all* $w_1, \ldots, w_{ar(\mathtt{op})},\ |[\![\mathtt{op}]\!](w_1, \ldots, w_{ar(\mathtt{op})})| \leq \max_{i \in [1, ar(\mathtt{op})]} |w_i|$.

3. positive *if there is a constant* $c_{\mathtt{op}}$ *such that:*

$$\forall w_1, \ldots, w_{ar(\mathtt{op})} \in \mathbb{W},\ |[\![\mathtt{op}]\!](w_1, \ldots, w_{ar(\mathtt{op})})| \leq \max_{i \in [1, ar(\mathtt{op})]} |w_i| + c_{\mathtt{op}}.$$

4. polynomial *if there is a polynomial* Q *such that for all* $w_1, \ldots, w_{ar(\mathtt{op})}$,

$$|[\![\mathtt{op}]\!](w_1, \ldots, w_{ar(\mathtt{op})})| \leq Q(\max_{i \in [1, ar(\mathtt{op})]} |w_i|).$$

Again a neutral operator is a max operator, a max operator is a positive operator, and a positive operator is a polynomial operator.

Example 3.1.10. *We illustrate this classification by the operators* `or`, `dis`, *and* `calloc` *(from the malloc family) computing respectively the Boolean disjunction, the disjunction on binary words, and a word of size* $n \times m$ *on inputs of size* n *and* m.

(Neutral)	$[\![\mathtt{or}]\!](u, w)$	$= 1$	*if* $u = 1$ *or* $w = 1$
		$= 0$	*otherwise*
(Max)	$[\![\mathtt{dis}]\!](u_1, u_2)$	$= [\![\mathtt{or}]\!](a_1, a_2).[\![\mathtt{dis}]\!](w_1, w_2)$	*if* $u_i = a_i.w_i,\ a_i \in \Sigma$
		$= u_{(i+1)\%2}$	*if* $u_i = \varepsilon, i \in \{1, 2\}$
(Polynomial)	$[\![\mathtt{calloc}]\!](u, w)$	$= \underbrace{w.\cdots.w}_{\lvert u\rvert \text{ times}}$	*if* $u, w \in \mathbb{W}$

The operator `or` *is neutral because it computes a Boolean predicate in* `P`. *(see Definition 3.1.1). The operator* `dis` *is max since it satisfies Item 2 of Definition 3.1.6. Finally,* `calloc` *is neither neutral, nor positive using the same reasoning and is polynomial by setting* $P_{\mathtt{calloc}}(n) = n^2$ *in Item 4 of Definition 3.1.6 since* $\forall w \in \mathbb{W},\ |[\![\mathtt{calloc}]\!](w)| = |w| \times |w| = P_{\mathtt{calloc}}(|w|)$.

We extend the notion of safe operator typing environment of Definition 3.1.2 to these new classes of operators.

Definition 3.1.7. *An operator typing environment* Δ *is* safe *if there exist four disjoint classes of operators* $Ntr, Max, Pos,$ *and* Pol *such that for any operator* `op` *and* $\forall \alpha_1 \to \ldots \to \alpha_n \to \alpha \in \Delta(\mathtt{op})$, *the following conditions hold:*

- $\alpha \preceq \wedge_{i=1,n} \alpha_i$,
- *if* $\mathtt{op} \in Ntr$ *then* `op` *is a neutral operator,*
- *if* $\mathtt{op} \in Max$ *then* `op` *is a max operator and* $\alpha \neq \mathbf{1}$,
- *if* $\mathtt{op} \in Pos$ *then* `op` *is a positive operator and* $\alpha = \mathbf{0}$,
- *if* $\mathtt{op} \in Pol$ *then* `op` *is a polynomial operator,* $\alpha = \mathbf{0}$, *and* $\forall i,\ \alpha_i = \mathbf{1}$.

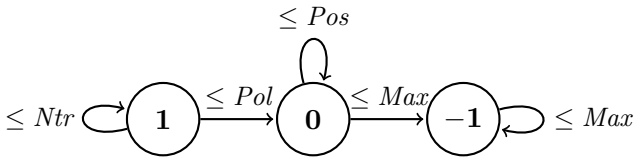


Figure 3.8: Admissible types for unary operators

We define an order relation $\leq$ on classes of operators $Ntr \leq Max \leq Pos \leq Pol$. For $X \in \{Ntr, Max, Pos, Pol\}$, let $\leq X$ denote the set $\{Y \mid Y \leq X\}$. The constraints on operator types are summed up in Figure 3.8 for operators of arity 1.

The key point is that the type system guarantees that information flow goes from tier $\mathbf{1}$ to tier $-\mathbf{1}$. Let us briefly explain the intuitions that lie behind such a definition. Operator types are all non-increasing wrt $\preceq$. Neutral operators can be iterated. A unary neutral operator can be typed by $\mathbf{1} \to \mathbf{1}$ and, consequently, can be used in the guard of a while-loop. Max operators cannot be iterated since the number of words bounded by a max function is exponential in the size. Positive operators cannot be iterated because they may increase their argument, thus leading to divergence. However they can be composed, which means that they can be typed by $\mathbf{0} \to \mathbf{0}$ and can be used in a while-loop command with no restriction. Finally, polynomial operators cannot be iterated and cannot be composed. They have to be typed by $\mathbf{1} \to \mathbf{0}$ in order to prevent composition (and, a fortiori, iteration).

Example 3.1.11. *The operators* $==$ *and* -1 *are neutral. Consequently,* -1 *can have the following types* $\{-\mathbf{1} \to -\mathbf{1}, \mathbf{0} \to -\mathbf{1}, \mathbf{0} \to \mathbf{0}, \mathbf{1} \to \mathbf{1}, \mathbf{1} \to \mathbf{0}, \mathbf{1} \to -\mathbf{1}\}$ *and* $==$ *will have type in* $\{\alpha \to \beta \to \gamma \mid \gamma \preceq \alpha \wedge \beta\}$. `dis` *is a max operator since the disjunction of two words has size bounded by the maximal input size. Notice that* `dis` $\notin Ntr$ *since* `dis` *does not compute a subword, so it admits any type in* $\{\alpha \to \beta \to \gamma \mid \gamma \preceq \alpha \wedge \beta\} - \{\mathbf{1} \to \mathbf{1} \to \mathbf{1}\}$. *The operator* $+1$ *is positive and, consequently, we may have* $\Delta(+1) = \{\mathbf{0} \to \mathbf{0}\}$ *if* $+1 \in Pos$ *or* $\Delta(+1) = \{\mathbf{1} \to \mathbf{0}\}$ *if* $+1 \in Pol$. *Finally, we have* $\Delta($`calloc`$) = \{\mathbf{1} \to \mathbf{1} \to \mathbf{0}\}$ *since* `calloc` *is polynomial.*

Definition 3.1.8 (Safe process)**.** *A process* P *is a* safe *if there are a variable typing environment* Γ*, a safe operator typing environment* Δ*, and a tier* β *such that* $\Gamma, \Delta \vdash \mathrm{P} : \beta$.

The algorithm of Example 3.1.12, searching for a character in a given string and returning a Boolean number, illustrates the notion of safe process. In this example, all operators are in *Ntr* apart from length which is in *Pos*. Note that, for the operator $-$ to be neutral, we need to consider unary numbers.

Example 3.1.12 ([HMP13])**.** *The following process splits its input string* `str` *in two parts, and performs the search of the character* `*` *using two forks and the operators* $==$ *and* $!=$ *over characters or strings,* $-$*, and* > 0 *over numbers, and Boolean disjunction or; the operator* `getchar` *such that the evaluation of* `getchar(str,i)` *computes the* `i`*-th character of the string* `str`*; the operator* `tail(str)` *which returns the string* `str` *minus its first symbol; and the operator* `length(str)` *which computes the length of the string* `str`.

$\mathtt{found}^{-\mathbf{1}} := \mathtt{0}^{-\mathbf{1}} : -\mathbf{1}$;
$\mathtt{s}^{\mathbf{1}} := \mathtt{str}^{\mathbf{1}} : \mathbf{1}$;
$\mathtt{l}^{\mathbf{0}} := \mathtt{length}(\mathtt{s}^{\mathbf{1}}) : \mathbf{0}$;
$\mathtt{n}^{\mathbf{0}} := \mathtt{l}^{\mathbf{0}} : \mathbf{0}$;
$\mathtt{x}^{\mathbf{0}} := \mathtt{fork}()^{\mathbf{0}} : \mathbf{0}$;

```
while (s != " ")^1{
    if(x > 0^0){
        c^0 := getchar(str^1, l^1)^1 : 0
    } else {
        c^0 := getchar(str^1, n − l^1)^1 : 0
    }
    if(c == '*')^0{
        found^{-1} := 1^{-1} : 0
    } else {skip} : 0 ;
    l^0 := l − 1 : 0 ;
    s^1 := tail(tail(s))^1 : 1
} : 1
if(x > 0)^0{
    sonf^{-1} := wait(x^0)^{-1} : 0 ;
    found^{-1} := or(found, sonf)^{-1} : 0
} else {skip} : 0 ;
return found^{-1}
```

This process is safe as it can be typed with respect to the type annotations provided above under a variable typing environment Γ *such that* $\Gamma(\mathtt{n}) = \Gamma(\mathtt{l}) = \Gamma(\mathtt{str}) = \mathbf{1}$, $\Gamma(\mathtt{x}) = \Gamma(\mathtt{c}) = \mathbf{0}$, *and* $\Gamma(\mathtt{found}) = \Gamma(\mathtt{sonf}) = -\mathbf{1}$ *and under a safe operator typing environment* Δ *s.t.* $\Delta(\mathtt{tail}) = \{\mathbf{1} \to \mathbf{1}\}$, $\Delta(-) = \Delta(\mathtt{getchar}) = \{\mathbf{1} \to \mathbf{1} \to \mathbf{1}\}$, $\Delta(==) = \Delta(!=) = \{\mathbf{1} \to \mathbf{1} \to \mathbf{1}, \mathbf{0} \to \mathbf{0} \to \mathbf{0}\}$, $\Delta(\mathtt{or}) = \{-\mathbf{1} \to -\mathbf{1} \to -\mathbf{1}\}$, $\Delta(> 0) = \{\mathbf{0} \to \mathbf{0}, \mathbf{1} \to \mathbf{1}\}$ *and* $\Delta(\mathtt{length}) = \{\mathbf{1} \to \mathbf{0}\}$.

The `wait(x)` *instruction enforces the type of the variable* `found` *to be* $-\mathbf{1}$ *using rule (W). Commands of tier* $-\mathbf{1}$ *can be used in a branching statement of tier* $\mathbf{0}$ *thanks to the subtyping rule (SUB). The* `return` *statement disrupts the information flow and may be considered as a declassification mechanism in the type security paradigm [VIS96].*

Example 3.1.13 ([HMP13]). *As another example of a fork process, let us program the counting of a character* `*` *in a string, which is the canonical example of distributed algorithms using MapReduce.*

```
s^1 := str^1 : 1 ;
r^0 := 1 : 0 ;
b^{-1} := ul(str^1) : −1 ;
f^{-1} := 0 : −1 ;
flag^0 := tt : −1 ;
while(s^1 != " ")^1{
    if(flag^0){
        pidl^0 := fork() : 0
        if(pidl > 0)^0{
            r^0 := 2 × r + 1 : 0 ;
            pidr^0 := fork() : 0
        } else {
            r^0 := 2 × r : 0
        }
        if(or((pidl == 0), (pidr == 0))^0{
            if(getchar(str, r) == '*')^0{
                f^{-1} := addmod(f^{-1}, 1, b^{-1}) : 0
            } else {skip}
        } else {
            flag^0 := ff : 0 ;
            xl^{-1} := wait(pidl) : 0 ;
            xr^{-1} := wait(pidr) : 0 ;
```

```
                f^-1 := addmod(f^-1, xl^-1, b^-1) : 0 ;
                f^-1 := addmod(f^-1, xr^-1, b^-1) : 0
            }
        }
        s^1 := half(s)^1 : 1
    } ;
    return f
```

`or`, $!=$, $==$, *and* `getchar(str, i)` *are the neutral operators described in Example 3.1.12.* `half` *is returning the first half of a string. It is also neutral.* `ul(str)` *stands for unary length, it is the binary number containing as many 1 as there are characters in* `str`*. This operator is positive.* $2\times$ *and* $+1$ *are the binary numerical operators. They are positive.* `addmod(x,y,b)` *computes the addition of* `x` *and* `y` *modulo* `b`*. It is in Max. It is straightforward to verify that this process is safe with respect to the type annotations provided above.*

Now we are ready to state the main result, a characterization of polynomial space computable functions.

Theorem 3.1.6 ([HMP13])**.** *The set of functions computed by strongly normalizing, lock-free, confluent, and safe processes is exactly* `FPSPACE`*.*

Soundness is achieved as follows: the type discipline still precludes flows from lower levels to higher level. Data of tier $\mathbf{1}$ does not increase and control while loops. Data of tier $\mathbf{0}$ can increase at most polynomially in each process. Because of the (Fork) typing rule, ids are of tier $\mathbf{0}$ and, consequently, there cannot be more than a polynomial number of created processes in depth (where depth is the childhood relation). However the total number of processes can be exponential. The polynomial upper bound on computations is recovered by the type restriction on communication data: they are of tier $-\mathbf{1}$ and only max operators can be applied. Hence no data accumulation is allowed between processes.

Completeness is shown by simulating a `PSPACE`-complete problem: QBF by a strongly normalizing and safe process. Consequently, we can compute the i-th output bit of each `FPSPACE` function as illustrated by Example 3.1.14.

Example 3.1.14 ([HMP13])**.** *The following strongly normalizing and safe process computes QBF on a formula* `phi` *given as input. It generates* 2^n *forks,* n *being the number of variables in* `phi`*, each of these processes is responsible for computing* `phi` *on a fixed Boolean assignment. This process will use the following operators for which we provide a safe operator typing environment* Δ*:*

- `fst`, `snd`, `tail` *and* `rmd` *respectively giving the first element, the second element, the word without its first character, and the word without its two first characters:*
  $\Delta(\texttt{fst}) = \Delta(\texttt{snd}) = \Delta(\texttt{tail}) = \Delta(\texttt{rmd}) = \{\mathbf{0} \to \mathbf{0}, \mathbf{1} \to \mathbf{1}\}$.
- `cons` *a positive operator adding a head character to a word:* $\Delta(\texttt{cons}) = \{\mathbf{1} \to \mathbf{0} \to \mathbf{0}\}$.
- *the neutral Boolean operators* `or` *and* `and` *and the neutral operator* `evaluate` *which evaluates a Boolean formula with respect to an evaluation encoded as an array of Boolean numbers.*
- `calloc` *which builds an array;* `read` *and* `write` *for accessing this array.*

  `tab := calloc(Y, Z)` *allocates a table of size* $|Y| \times |Z|$ *(for storing* $|Y|$ *elements of size at most* $|Z|$*) in which we can read with* `v := read(tab, i, Z)` *which queries the word* `tab[ i|Z| : (i+1)|Z| ]` *and we can write using* `status := write(tab, v, i, Z)` *which*

writes word v *in* tab *between positions* i|Z| *and* (i+1)|Z|. $\Delta(\texttt{calloc}) = \{\mathbf{1} \to \mathbf{1} \to \mathbf{0}\}$. $\Delta(\texttt{read}) = \{\mathbf{0} \to \mathbf{0} \to \mathbf{1} \to \mathbf{0}\}$, $\Delta(\texttt{write}) = \{\mathbf{0} \to \mathbf{0} \to \mathbf{0} \to \mathbf{1} \to \mathbf{0}\}$.

The operators > 0, $==$, fst, snd, tail, rmd, or, and, read, write, *and* evaluate *are neutral. Consequently, we can set* $-\mathbf{1} \to -\mathbf{1} \to -\mathbf{1} \in \Delta(\texttt{or}) \cap \Delta(\texttt{and})$. *The operators* cons, -1, *and* $+1$ *are positive and the operator* calloc *is polynomial.*

$\texttt{psi}^{\mathbf{1}} := \texttt{phi}^{\mathbf{1}} : \mathbf{1}$;
$\texttt{q}^{\mathbf{1}} := \texttt{fst}(\texttt{phi}^{\mathbf{1}}) : \mathbf{1}$;
$\texttt{pidtab}^{\mathbf{0}} := \texttt{calloc}(\texttt{phi}^{\mathbf{1}}, \texttt{phi}^{\mathbf{1}})^{\mathbf{0}} : \mathbf{0}$;
$\texttt{vartab}^{\mathbf{0}} := \texttt{calloc}(\texttt{phi}^{\mathbf{1}}, \texttt{tt}^{\mathbf{1}})^{\mathbf{0}} : \mathbf{0}$;
$\texttt{i}^{\mathbf{0}} := 0^{\mathbf{0}} : \mathbf{0}$;
$\texttt{v}^{\mathbf{1}} := \texttt{snd}(\texttt{phi}^{\mathbf{1}})^{\mathbf{1}} : \mathbf{1}$;
$\texttt{while}(\texttt{or}(\texttt{q}^{\mathbf{1}} == '\exists'), (\texttt{q}^{\mathbf{1}} == '\forall'))^{\mathbf{1}}\{$
  $\texttt{phi}^{\mathbf{1}} := \texttt{rmd}(\texttt{phi}^{\mathbf{1}}) : \mathbf{1}$;
  $\texttt{pid}^{\mathbf{0}} := \texttt{fork}() : \mathbf{0}$;
  $\texttt{if}(\texttt{pid}^{\mathbf{0}} > 0)\{$
    $\texttt{status}^{\mathbf{0}} := \texttt{write}(\texttt{pidtab}, \texttt{pid}, \texttt{i}, \texttt{phi}^{\mathbf{1}}) : \mathbf{0}$;
    $\texttt{status}^{\mathbf{0}} := \texttt{write}(\texttt{vartab}, \texttt{tt}^{\mathbf{0}}, \texttt{v}, \texttt{tt}) : \mathbf{0}$;
    $\texttt{i}^{\mathbf{0}} := \texttt{i} + 1^{\mathbf{0}} : \mathbf{0}$
  } else {
    $\texttt{opstack}^{\mathbf{0}} := \varepsilon^{\mathbf{0}} : \mathbf{0}$;
    $\texttt{status}^{\mathbf{0}} := \texttt{write}(\texttt{vartab}, \texttt{ff}, \texttt{v}, \texttt{tt}) : \mathbf{0}$;
    $\texttt{i}^{\mathbf{0}} := 0^{\mathbf{0}} : \mathbf{0}$
  }
  $\texttt{if}(q^{\mathbf{1}} == '\exists'^{\mathbf{1}})\{$
    $\texttt{opstack}^{\mathbf{0}} := \texttt{cons}(' \vee ', \texttt{opstack}^{\mathbf{0}}) : \mathbf{1}$
  } else {
    $\texttt{opstack}^{\mathbf{0}} := \texttt{cons}(' \wedge ', \texttt{opstack}^{\mathbf{0}}) : \mathbf{1}$;
  }
  $\texttt{q}^{\mathbf{1}} := \texttt{fst}(\texttt{phi}^{\mathbf{1}}) : \mathbf{1}$;
  $\texttt{v}^{\mathbf{1}} := \texttt{snd}(\texttt{phi}^{\mathbf{1}}) : \mathbf{1}$
}
$\texttt{res}^{-\mathbf{1}} := \texttt{evaluate}(\texttt{phi}^{\mathbf{1}}, \texttt{vartab})^{\mathbf{0}} : \mathbf{0}$;
$\texttt{q}^{\mathbf{1}} := \texttt{fst}(\texttt{psi}^{\mathbf{1}}) : \mathbf{1}$;
$\texttt{while}(\texttt{or}((\texttt{q}^{\mathbf{1}} == '\exists'), (\texttt{q}^{\mathbf{1}} == '\forall'))^{\mathbf{1}}\{$
  $\texttt{phi}^{\mathbf{1}} := \texttt{rmd}(\texttt{phi}^{\mathbf{1}}) : \mathbf{1}$;
  $\texttt{q}^{\mathbf{1}} := \texttt{fst}(\texttt{psi}^{\mathbf{1}}) : \mathbf{1}$;
  $\texttt{i}^{\mathbf{0}} := \texttt{i} - 1^{\mathbf{0}} : \mathbf{0}$;
  $\texttt{pid}^{\mathbf{0}} := \texttt{read}(\texttt{pidtab}^{\mathbf{0}}, \texttt{i}, \texttt{phi}^{\mathbf{1}}) : \mathbf{0}$;
  $\texttt{op}^{\mathbf{0}} := \texttt{fst}(\texttt{opstack}^{\mathbf{0}}) : \mathbf{0}$;
  $\texttt{opstack}^{\mathbf{0}} := \texttt{tail}(\texttt{opstack}^{\mathbf{0}}) : \mathbf{0}$;
  $\texttt{res}_{\texttt{son}}^{-\mathbf{1}} := \texttt{wait}(\texttt{pid}^{\mathbf{0}})^{-\mathbf{1}} : -\mathbf{1}$
  $\texttt{if}(\texttt{op}^{\mathbf{0}} == ' \vee ')\{$
    $\texttt{res}^{-\mathbf{1}} := \texttt{or}(\texttt{res}, \texttt{res}_{\texttt{son}})^{-\mathbf{1}} : \mathbf{0}$;
  } else {
    $\texttt{res}^{-\mathbf{1}} := \texttt{and}(\texttt{res}, \texttt{res}_{\texttt{son}})^{-\mathbf{1}} : \mathbf{0}$;
  }
} ;
return $\texttt{res}^{-\mathbf{1}}$

This process satisfies the lock-freedom and confluence properties. Consequently, we can compute the i-th output bit of each polynomial space computable function since we can reduce any polynomial space decision problem to QBF using a process without forks. Now it remains to

show that all these results can be combined in order to compute the whole function. The decision problem $\{(x, y) \mid f(x) = y \text{ and } f \in \texttt{FPSPACE}\}$ *is in* PSPACE*. So, in order to compute* $f(x)$*, we first generate all possible words* y *of the right size and then we test whether or not* (x, y) *is in the graph of* f *using a fork process for each case.*

3.1.4 Object oriented programs

In [LM13], ramification and tiering were extended to a programming language over a dynamical graph structure, a structure were edges and vertices can be created, deleted, and updated. This paper provides a characterization of polynomial time on such programs. It has been extended in [HP15, HP18] to the study of Object-Oriented Java-like programs including recursive methods. [HP18] provides a more general treatment of recursive methods than [HP15] as they can compute increasing data. This improvement is handled by restricting the contexts under which such methods can be called.

The syntax of considered programs is defined by the following grammar:

$$
\begin{array}{rcl}
\text{Expressions} \ni \texttt{e} & ::= & \texttt{x} \mid \texttt{cst}_\tau \mid \texttt{null} \mid \texttt{this} \mid \texttt{op}(\overline{\texttt{e}}) \mid \texttt{new C}(\overline{\texttt{e}}) \mid \texttt{e.m}(\overline{\texttt{e}}) \\
\text{Instructions} \ni \texttt{I} & ::= & \texttt{;} \mid [\tau]\ \texttt{x:=e;} \mid \texttt{I}_1\ \texttt{I}_2 \mid \texttt{while(e)\{I\}} \mid \texttt{if(e)\{I}_1\texttt{\}else\{I}_2\texttt{\}} \mid \texttt{e.m}(\overline{\texttt{e}})\texttt{;} \\
\text{Methods} \ni \texttt{m}_\texttt{C} & ::= & \tau\ \texttt{m}(\overline{\tau\ \texttt{x}})\texttt{\{I[return x;]\}} \\
\text{Constructors} \ni \texttt{k}_\texttt{C} & ::= & \texttt{C}(\overline{\tau\ \texttt{y}})\texttt{\{I\}} \\
\text{Classes} \ni \texttt{C} & ::= & \texttt{C}\ [\texttt{extends D}]\ \{\overline{\tau\ \texttt{x;}}\ \overline{\texttt{k}_\texttt{C}}\ \overline{\texttt{m}_\texttt{C}}\},
\end{array}
$$

where $\texttt{x} \in \mathbb{V}$, $\texttt{op} \in \mathbb{O}$, $\texttt{m} \in \mathbb{M}$, and $\texttt{C} \in \mathbb{C}$. The four disjoint sets $\mathbb{V}$, $\mathbb{O}$, $\mathbb{M}$, and $\mathbb{C}$ represent the set of variables, the set of operators, the set of method names, and the set of class names, respectively. In the above grammar, $[e]$ denotes some optional syntactic element e and $\overline{e}$ denotes a sequence of syntactic elements $e_1, \ldots, e_n$. $\mathbb{C}$ is the set of class names C and $\mathbb{T}$ is the set of primitive data types τ, including $\{\texttt{void}, \texttt{boolean}, \texttt{int}, \texttt{char}\}$. The metavariable $\texttt{cst}_\tau$ represents a primitive type constant of type $\tau \in \mathbb{T}$. Each primitive operator $\texttt{op} \in \mathbb{O}$ has a fixed arity n and comes equipped with a signature of the shape $\texttt{op} :: \tau_1 \times \cdots \times \tau_n \to \tau_{n+1}$ fixed by the language implementation. Given a method $\tau\ \texttt{m}(\tau_1\ \texttt{x}_1, \ldots, \tau_n\ \texttt{x}_n)\texttt{\{I [return x;]\}}$ of C, its signature is $\tau\ \texttt{m}^\texttt{C}(\tau_1, \ldots, \tau_n)$, the notation $\texttt{m}^\texttt{C}$ denoting that m is declared in C. The signature of a constructor $\texttt{k}_\texttt{C}$ is $\texttt{C}(\overline{\tau})$.

Variables are split into local variables, fields and parameters. In what follows, let $\texttt{C}.\mathcal{F}$ be the set of field names of the class C.

The fields of a class instance can only be accessed through the use of getters, *i.e.* cannot be accessed directly using the "." operator. Consequently, all fields are implicitly `private`. On the opposite, methods and classes are all implicitly `public`.

Inheritance (and override) is allowed through the use of the `extends` construct. If `C extends D`, the constructors $\texttt{C}(\overline{\tau\ \texttt{y}})\texttt{\{I\}}$ are constructors initializing both the fields of C and the fields of D. Inheritance defines a partial order on classes denoted by $\texttt{C} \trianglelefteq \texttt{D}$.

A *program* is a collection of classes including exactly one class $\texttt{Exe}\{\texttt{void main()}\{\texttt{Init Comp}\}\}$, with $\texttt{Init}, \texttt{Comp} \in$ Instructions. The method `main` of the class `Exe` is intended to be the entry point of the program. The instruction `Init` is called the *initialization instruction*. Its purpose is to compute the program input, which is strongly needed in order to define the complexity of a given program since, without input, all terminating programs would be considered to be constant time programs. The instruction `Comp` is called the *computational instruction*. The type system presented below will analyze the complexity of this latter instruction.

We restrict the considered programs to well-formed programs satisfying the following conditions. There is only one class per class name. A local variable `x` is declared and initialized exactly once by a τ `x` := `e`; instruction. A method output type is `void` when it has no `return` statement. Each signature is unique.

The operational semantics of this programming language is fully described in [HP18]. It relates pairs of memory configuration and an instruction. In this particular setting, a memory configuration is both a heap, describing the state of global variables, and a stack, keeping information on method calls and parameter values.

A *tiered type* is a pair $\tau(\alpha)$ consisting of a type $\tau \in \mathbb{C} \cup \mathbb{T}$ together with a tier $\alpha \in \{\mathbf{0}, \mathbf{1}\}$. Given a sequence of types $\overline{\tau} = \tau_1, \ldots, \tau_n$, a sequence of tiers $\overline{\alpha} = \alpha_1, \ldots, \alpha_n$, and a tier α, let $\overline{\tau}(\overline{\alpha})$ denote $\tau_1(\alpha_1), \ldots, \tau_n(\alpha_n)$, $\overline{\tau}(\alpha)$ denote $\tau_1(\alpha), \ldots, \tau_n(\alpha)$, and $\langle\overline{\tau}\rangle$ ($\langle\overline{\tau}(\overline{\alpha})\rangle$, respectively) denote the Cartesian product of types (tiered types, respectively).

Before introducing the considered type system, we introduce four other kinds of environments:

- *operator typing environments* Δ defined as usual,
- *method typing environments* δ associating a tiered type to each program variable $v \in \mathbb{V}$,
- *typing environments* Ω associating a method typing environment δ to each method signature τ $\mathtt{m}^{\mathtt{C}}(\overline{\tau})$, *i.e.* $\Omega(\tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau})) = \delta$,
- *contextual typing environments* $\Gamma = (s, \Omega)$, a pair consisting of a method signature and a typing environment; the method (or constructor) signature s indicates under which context (typing environment) the fields are typed.

For each $\mathtt{x} \in \mathbb{V}$, define $\Gamma(\mathtt{x}) = \Omega(s)(\mathtt{x})$. Also define $\Gamma\{\mathtt{x} \leftarrow \tau(\alpha)\}$ to be the contextual typing environment Γ' such that $\forall \mathtt{y} \neq \mathtt{x}$, $\Gamma'(\mathtt{y}) = \Gamma(\mathtt{y})$ and $\Gamma'(\mathtt{x}) = \tau(\alpha)$. Let $\Gamma\{s'\}$ be a notation for the contextual typing environment that is equal to (s', Ω), *i.e.* the signature s' has been substituted to s in Γ.

The type system is presented in Figure 3.9 where, given a sequence $\overline{\mathtt{e}} = \mathtt{e}_1, \ldots, \mathtt{e}_n$ of expressions, a sequence of types $\overline{\tau} = \tau_1, \ldots, \tau_n$, and two sequences of tiers, $\overline{\alpha} = \alpha_1, \ldots, \alpha_n$ and $\overline{\beta} = \beta_1, \ldots, \beta_n$, the notation $\Gamma, \Delta \vdash_{\overline{\beta}} \overline{\mathtt{e}} : \overline{\tau}(\overline{\alpha})$ means that $\Gamma, \Delta \vdash_{\beta_i} \mathtt{e}_i : \tau_i(\alpha_i)$ holds, for all $i \in [1, n]$.

(*) The rule (Ass) enforces the following side conditions:

- if `x` is a field then $\alpha = \mathbf{0}$,
- if $\tau \in \mathbb{T}$ then $\alpha' \leq \alpha$,
- if $\tau \in \mathbb{C}$ then $\alpha' = \alpha$.

(**) The rule (Body) requires that τ $\mathtt{m}(\overline{\tau\ \mathtt{x}})\{\mathtt{I}\ [\mathtt{return\ x;}]\} \in \mathtt{C}$.

There are four kinds of typing judgments:

- $\Gamma, \Delta \vdash_\beta \mathtt{e} : \tau(\alpha)$ for expressions, meaning that the expression `e` is of tiered type $\tau(\alpha)$ under the contextual typing environment Γ and operator typing environment Δ and can only be assigned to in an instruction of tier at least β,
- $\Gamma, \Delta \vdash \mathtt{I} : \mathtt{void}(\alpha)$ for instructions, meaning that the instruction `I` is of tiered type `void`(α) under the contextual typing environment Γ and operator typing environment Δ,

$$\frac{}{\Gamma, \Delta \vdash_\beta \mathtt{cst}_\tau : \tau(\alpha)}\ (\text{Cst}) \qquad \frac{}{\Gamma, \Delta \vdash_\beta \mathtt{null} : \mathtt{C}(\alpha)}\ (\text{Null})$$

$$\frac{\Gamma(\mathtt{x}) = \tau(\alpha)}{\Gamma, \Delta \vdash_\beta \mathtt{x} : \tau(\alpha)}\ (\text{Var}) \qquad \frac{\Gamma, \Delta \vdash_{\overline{\beta}} \overline{\mathtt{e}} : \overline{\tau}(\alpha) \qquad \langle \overline{\tau}(\alpha) \rangle \to \tau(\alpha) \in \Delta(\mathtt{op})}{\Gamma, \Delta \vdash_{\vee\overline{\beta}} \mathtt{op}(\overline{\mathtt{e}}) : \tau(\alpha)}\ (\text{Op})$$

$$\frac{\forall \mathtt{x} \in \mathtt{C}.\mathcal{F}, \exists \tau, \Gamma(\mathtt{x}) = \tau(\alpha)}{\Gamma, \Delta \vdash_\beta \mathtt{this} : \mathtt{C}(\alpha)}\ (\text{Self}) \qquad \frac{\Gamma, \Delta \vdash_\beta \mathtt{e} : \mathtt{D}(\alpha) \quad \mathtt{D} \trianglelefteq \mathtt{C}}{\Gamma, \Delta \vdash_\beta \mathtt{e} : \mathtt{C}(\alpha)}\ (\text{Pol})$$

$$\frac{\Gamma, \Delta \vdash_{\overline{\beta}} \overline{\mathtt{e}} : \overline{\tau}(\mathbf{0}) \quad \Gamma\{\mathtt{C}(\overline{\tau})\}, \Delta \vdash \mathtt{C}(\overline{\tau}) : \langle \overline{\tau}(\mathbf{0}) \rangle \to \mathtt{C}(\mathbf{0})}{\Gamma, \Delta \vdash_{\vee\overline{\beta}} \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) : \mathtt{C}(\mathbf{0})}\ (\text{New})$$

$$\frac{\Gamma, \Delta \vdash_\beta \mathtt{e} : \mathtt{C}(\alpha') \qquad \Gamma, \Delta \vdash_{\overline{\beta}} \overline{\mathtt{e}} : \overline{\tau}(\overline{\alpha}) \qquad \Gamma\{\tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau})\}, \Delta \vdash_\beta \tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau}) : \mathtt{C}(\beta) \times \langle \overline{\tau}(\overline{\alpha}) \rangle \to \tau(\alpha)}{\Gamma, \Delta \vdash_\beta \mathtt{e} : \mathtt{C}(\alpha)}\ (\text{Call})$$

$$\frac{\forall i,\ \Gamma, \Delta \vdash \mathtt{I}_i : \mathtt{void}(\alpha_i)}{\Gamma, \Delta \vdash \mathtt{I}_1\ \mathtt{I}_2 : \mathtt{void}(\alpha_1 \vee \alpha_2)}\ (\text{Seq}) \qquad \frac{\Gamma, \Delta \vdash \mathtt{I} : \mathtt{void}(\mathbf{0})}{\Gamma, \Delta \vdash \mathtt{I} : \mathtt{void}(\mathbf{1})}\ (\text{Sub})$$

$$\frac{\Gamma, \Delta \vdash_\alpha \mathtt{e} : \mathtt{boolean}(\alpha) \qquad \forall i,\ \Gamma, \Delta \vdash \mathtt{I}_i : \mathtt{void}(\alpha)}{\Gamma, \Delta \vdash \mathtt{if(e)\{I_1\}else\{I_2\}} : \mathtt{void}(\alpha)}\ (\text{If})$$

$$\frac{\Gamma, \Delta \vdash_{\mathbf{1}} \mathtt{e} : \mathtt{boolean}(\mathbf{1}) \qquad \Gamma, \Delta \vdash \mathtt{I} : \mathtt{void}(\mathbf{1})}{\Gamma, \Delta \vdash \mathtt{while(e)\{I\}} : \mathtt{void}(\mathbf{1})}\ (\text{Wh})$$

$$\frac{[\Gamma, \Delta \vdash_{\mathbf{0}} \mathtt{x} : \tau(\alpha')] \quad \Gamma, \Delta \vdash_\beta \mathtt{e} : \tau(\alpha)}{\Gamma, \Delta \vdash [[\tau]\ \mathtt{x{:=}]e;} : \mathtt{void}(\alpha \vee \beta)}\ (\text{Ass*})$$

$$\frac{\Gamma\{\overline{\mathtt{x}} \leftarrow \overline{\tau}(\mathbf{0})\}, \Delta \vdash I : \mathtt{void}(\mathbf{0}) \qquad \mathtt{C}(\overline{\tau\ \mathtt{x}})\mathtt{\{I\}} \in \mathtt{C}}{\Gamma, \Delta \vdash \mathtt{C}(\overline{\tau}) : \langle \overline{\tau}(\mathbf{0}) \rangle \to \mathtt{C}(\mathbf{0})}\ (\text{Cons})$$

$$\frac{\mathtt{C} \trianglelefteq \mathtt{D} \qquad \Gamma, \Delta \vdash_\gamma \tau\ \mathtt{m}^{\mathtt{D}}(\overline{\tau}) : \mathtt{D}(\alpha') \times \langle \overline{\tau}(\overline{\alpha}) \rangle \to \tau(\alpha) \qquad \tau\ \mathtt{m}(\overline{\tau\ \mathtt{x}})\mathtt{\{I\ [return\ x;]\}} \in \mathtt{D}}{\Gamma, \Delta \vdash_\gamma \tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau}) : \mathtt{C}(\alpha') \times \langle \overline{\tau}(\overline{\alpha}) \rangle \to \tau(\alpha)}\ (\text{OR})$$

$$\frac{\Gamma\{\mathtt{this} \leftarrow \mathtt{C}(\beta), \overline{\mathtt{x}} \leftarrow \overline{\tau}(\overline{\alpha}), [\mathtt{x} \leftarrow \tau(\alpha)]\}, \Delta \vdash I : \mathtt{void}(\gamma) \qquad \Gamma, \Delta \vdash_\gamma \mathtt{this} : \mathtt{C}(\beta)}{\Gamma, \Delta \vdash_\gamma \tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau}) : \mathtt{C}(\beta) \times \langle \overline{\tau}(\overline{\alpha}) \rangle \to \tau(\alpha)}\ (\text{Body**})$$

Figure 3.9: Tier-based OO type system

- $\Gamma, \Delta \vdash_\beta s : \mathtt{C}(\beta) \times \langle \overline{\tau}(\overline{\alpha}) \rangle \to \tau(\alpha)$ for method signatures, meaning that the method `m` of signature s belongs to the class `C` ($\mathtt{C}(\beta)$ is the tiered type of the current object `this`), has parameters of type $\langle \overline{\tau}(\overline{\alpha}) \rangle$, has a return variable of type $\tau(\alpha)$, with $\tau = \mathtt{void}$ in the particular case where there is no return statement, and can only be called in instructions of tier at least β,

- $\Gamma, \Delta \vdash \mathtt{C}(\overline{\tau}) : \langle \overline{\tau}(\mathbf{0}) \rangle \to \mathtt{C}(\mathbf{0})$ for constructor signatures, meaning that the constructor `C` has parameters of type $\langle \overline{\tau}(\mathbf{0}) \rangle$ and a return variable of type $\mathtt{C}(\mathbf{0})$, matching the class type `C`.

A program of executable `Exe{void main(){Init Comp}}` is well-typed if there are a typing environment Ω and an operator typing environment Δ such that $(\mathtt{void}\ \mathtt{main}^{\mathtt{Exe}}(), \Omega), \Delta \vdash \mathtt{Comp} : \mathtt{void}(\mathbf{1})$ can be derived.

Example 3.1.15 ([HP18]). *Consider the class of linked lists of Boolean numbers* `BList`.

```
BList{
    boolean value ;
    BList queue ;

    BList(boolean v, BList q){
        value := v ;
        queue := q ;
    }

    BList getQueue(){return queue ; }

    void setQueue(BList q){
        queue := q ;
    }

    boolean getValue(){return value ; }

    BList clone(){
        BList n ;
        if(value != null){
            n := new BList(value, queue.clone()) ;
        } else {
            n := new BList(null, null) ;
        }
        return n ;
    }

    void decrement(){
        if(value == true){
            value := false ;
        } else {
            if(queue != null){
                value := true ;
                queue.decrement() ;
            } else {
                value := false ;
            }
        }
    }
```

```
        int length(){
            int res := 1 ;
            if(queue ! = null){
                res := queue.length() ;
                res := res + 1 ;
            } else { ; }
            return res ;
        }
    }
```

We are trying to type each each method and constructor of this class using the type system of Figure 3.9.

```
        BList(boolean v,BList q){
            value := v ;
            queue := q ;
        }
```

The constructor `BList` *can be typed by* $\texttt{boolean}(\mathbf{0}) \times \texttt{Blist}(\mathbf{0}) \to \texttt{BList}(\mathbf{0})$*, using rules (Cons), (Seq), and twice rule (Ass). In the two applications of rule (Ass), the tiers of the fields* `value` *and* `queue` *are enforced to be* $\mathbf{0}$ *because of the side condition (*). As a consequence, it is not possible to create objects of tiered type* $\texttt{BList}(\mathbf{1})$ *in a computational instruction.*

```
        BList getQueue(){return queue ; }
```

`getQueue` *can be typed by* $\texttt{BList}(\mathbf{1}) \to \texttt{BList}(\mathbf{1})$ *or* $\texttt{BList}(\mathbf{0}) \to \texttt{BList}(\mathbf{0})$*, by rules (Body) and (Self). The types* $\texttt{BList}(\alpha) \to \texttt{BList}(\beta)$*,* $\alpha \neq \beta$*, are prohibited because of rules (Body) and (Self) since the tier of the current object has to match the tier of its fields. In the same manner, the method* `getValue` *can be given the types* $\texttt{BList}(\alpha) \to \texttt{boolean}(\alpha)$*,* $\alpha \in \{\mathbf{0}, \mathbf{1}\}$*.*

The method `setQueue`

```
        void setQueue(BList q){
            queue := q ;
        }
```

can only be given the types $\texttt{BList}(\mathbf{0}) \times \texttt{BList}(\mathbf{0}) \to \texttt{void}(\beta)$*. Indeed, by rule (Ass),* `queue` *and* `q` *are enforced to be of tier* $\mathbf{0}$*. Consequently,* `this` *is of tier* $\mathbf{0}$ *by rules (Body) and (Self). The tier of the body can be* $\mathbf{0}$ *or* $\mathbf{1}$*, using subtyping rule (Sub).*

The method `decrement` *can be typed by the following annotations:*

```
        void decrement(){
            if(value^0 == true){
                value := false ; : 0
            } else {
                if(queue ! = null){
                    value^0 := true ;
                    queue^0.decrement() ; : 0
                } else {
                    value^0 := false ; : 0
                }
            }
        }.
```

Because of the rules (Ass) and (Body), it can be given the type $\texttt{BList}(\mathbf{0}) \to \texttt{void}(\mathbf{0})$*. As we shall see later, this method will be rejected by the safety condition as it might lead to exponential length derivation whenever it is called in a while loop.*

The method `length`

```
int length(){
    int res := 1 ; : 0
    if(queue^1 ! = null){
        res := queue.length() ; : 1
        res := res + 1 ; : 0
    }
     else { ; }
    return res ;
}
```

can be typed by $\Gamma, \Omega \vdash_{\mathbf{1}} \texttt{int length}^{\texttt{BList}}() : \texttt{BList}(\mathbf{1}) \to \texttt{int}(\mathbf{0})$ *with respect to the annotations provided above.*

Let us now give some intuitions on the type system of Figure 3.9. First, as in the imperative case, data may only flow from higher tier to lower tier. However, in the rule (Ass) the side condition enforces equality of tiers whenever the assignment is on objects. Consequently, a flow from **1** to **0** can only occur on primitive data. While this could look restrictive at first glance, this restriction is natural. As object data are passed by reference (and not by value), a flow from **1** to **0** could allow tier **0** variable to access and change tier **1** data, hence breaking the non-interference property, as illustrated by the following example.

Example 3.1.16 ([HP18])**.** *Consider the following method*

```
BList extend(BList l){
    BList x := l ;
    while(l ! = null){
        x := new BList(true, x) ;
        l := l.getQueue() ;
    }
    return x ;
}.
```

In the rule (Ass) of Figure 3.9, for the assignment `x := l` *to be typed, the tiers of* `x` *and* `l` *are required to be equal as the corresponding type is object (BList). Consequently, the above code cannot be typed as* `l` *is enforced to be of tier* **1** *in the while loop guard and* `x` *is enforced to be of tier* **0** *by rules (Ass) and (New) applied to subcommand* `l := l.getQueue() ;`*. This typing discipline is restrictive but prevents a tier* **0** *variable* `x` *from pointing to tier* **1** *data. Indeed this would allow to change tier* **1** *data structure by iterating on this tier* **0** *variable and, consequently, the non-interference would be broken. Notice however that the following variant with cloning can be typed*

```
BList extend(BList l){
    BList x := l.clone() ;
    while(l ! = null){
        x := new BList(true, x) ;
        l := l.getQueue() ;
    }
    return x ;
},
```

provided that the method `clone` *can be given the type* $BList(\mathbf{1}) \to BList(\mathbf{0})$*. This is possible as we will shortly see in Example 3.1.18. Here the clone method deep copies the tier* **1** *data. Hence accessing this value inside a tier* **0** *variable does not break the non-interference.*

The type system of Figure 3.9 is by nature not restrictive enough for handling recursive calls in polynomial time. Indeed two recursive calls can be combined or accumulated in the body of a recursive method, allowing to compute, for example, exponential time functions such as the Fibonnacci sequence. Moreover, the nesting of a while loop in the body of a recursive method can lead to exponential behavior by allowing dynamically nested while loops. Worst of all, non-interference does not hold for such methods. Indeed, a recursive method call assigned to a tier **0** variable could be controlled by a tier **0** expression.

Now we put some aside restrictions on recursive methods to ensure that their computations remain polynomially bounded by preventing the above issues.

Definition 3.1.9 (Safe OO program)**.** *A well-typed program with respect to a typing environment* Ω *and operator typing environment* Δ *is* safe *if for each recursive method* $\tau\ \mathtt{m}(\overline{\tau\ \mathtt{x}})\{\mathtt{I}\ [\mathtt{return\ x;}]\}$*:*

1. *there is exactly one call to the recursive method (or any mutually recursive method) in the instruction* `I`*,*
2. *there is no while loop inside* `I`*, and*
3. *only judgments of the shape* $(s, \Omega), \Delta \vdash_{\mathbf{1}} \tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau\ \mathtt{x}}) : \mathtt{C}(\mathbf{1}) \times \langle\overline{\tau}(\mathbf{1})\rangle \to \tau(\alpha)$ *can be derived using rules* $(Body)$ *and* (OR)*.*

Example 3.1.17. *Consider a well-typed program whose computational instruction calls the methods* `getQueue`*,* `getValue`*,* `setQueue`*, or* `length` *of Example 3.1.15. This program is safe. Indeed, the only recursive method is* `length`*. As illustrated in Example 3.1.15, it can be typed by* $\mathtt{BList}(\mathbf{1}) \to \mathtt{int}(\mathbf{0})$*, it does not contain any while loop, and it has only one recursive call in its body.*

A program whose computational instruction uses the method `decrement` *cannot be safe as this method can only be given the type* $BList(\mathbf{0}) \to \mathtt{void}(\mathbf{0})$*. Though it entails a lack of expressive power, changing the type system by allowing* `decrement` *to apply to tier* **1** *objects would allow codes like*

```
while(!o.isEqual(l)){
     o.decrement() ;
},
```

where `isEqual` *is a method testing the equality of two lists, and* `l` *is a* `BList` *of Boolean numbers equal to* 0*. Clearly, such a loop can be executed exponentially in the size of the list* `o`*. This behavior is highly undesirable.*

An important point to mention is that safety allows an easy way to perform expression subtyping for objects through the use of cloning. In other words, it is possible to bring a reference type expression from tier **1** to tier **0** based on the premise that it has been cloned in memory. This was already true for primitive data by rule (Ass). Thus a form of subtyping that does not break the non-interference properties is admissible as illustrated by the following example.

Example 3.1.18. *The method* `clone`

```
BList clone(){
     BList res⁰ := null ;
     int v⁰ := value¹ ;
     if(queue¹ == null){
          res⁰ := new BList(v⁰, null)⁰ ;
```

```
        } else {
            res^0 := new BList(v^0, queue.clone()^0) ;
        }
        return res ;
    }
```

can be typed by $\texttt{BList}(\mathbf{1}) \to \texttt{BList}(\mathbf{0})$. *Moreover, the method is safe, as there is only one recursive call and no while loop.*

We are now ready to state the main characterization of polynomial time functions in terms of safe and terminating programs.

Theorem 3.1.7 ([HP18])**.** *The set of functions computed by safe and terminating programs is exactly* FP.

3.1.5 Type inference and declassification

The type systems of Figure 3.2, Figure 3.7, and Figure 3.9 have a decidable type inference.

Proposition 3.1.1 (Type inference)**.** *Given a safe operator typing environment* Δ *and a program, deciding if there exists a variable typing environment* Γ *such that the program is safe using typing rules of Figure 3.2, Figure 3.7, or Figure 3.9 can be done in polynomial time in the size of the program.*

The proof of decidability of type system is provided in [HMP13] for processes and in [HP18] for OO programs. They rely on a reduction to 2-SAT which is known to be decidable in polynomial time. The result follows for the type system of Figure 3.2 as it is a strict subsystem of the system of Figure 3.7.

Observe that the decidability of type inference does not imply that the provided criteria (characterizing complexity classes) are decidable as they sill require some extra hypothesis on program termination.

In [Mar11], the type system is based on a tier lattice allowing the type to perform some declassification mechanisms. Hence allowing programs of the shape

```
while(x > 0){
      x := x−1 ;
      y := y+1
} ;
while(y > 0){
      y := y−1 ;
      z := z+1
} ;
```

to type. However the above program cannot be typed using the type system of Figure 3.2. Indeed, the first loop enforces the y variable to be of tier **0**, as the operator +1 is positive. The second loop enforces the y variable to be of tier **1** and, consequently, we obtain a contradiction.

However such kind of programs can still be analyzed in two ways.

- As suggested in [HMP13], the static analysis can be allowed to split programs in a constant number of subprograms and to perform the analysis on these subprograms. As a constant number of polynomial compositions remains polynomial, this would allow to analyze the above counter-example.

- Alternatively, declassification can be handled by allowing more than two tiers. In such a framework, the operator types would depend on the tier of their context as suggested by the program annotation below.

```
while(x^2 > 0){
      x^2 := x−1 ;
      y^1 := y+1 : 1
} ;
while(y^1 > 0){
      y^1 := y−1 ;
      z :^0  := z+1 : 0
} ;
```

  Here the operator $+1$ would be restricted to have an output of tier strictly smaller than the tier of the outermost while loop. Hence it could be given the type $\mathbf{1} \to \mathbf{1}$ in the first while loop and would be enforced to be of type $\mathbf{0} \to \mathbf{0}$ in the second while loop.

3.2 Extensions of light/soft logics

On the light/soft logics side, a lot of work has been carried out on developing and studying the notion of proofs [BM10] in systems such as L^3 and L^4. As already mentioned, proof-nets turn out to be the natural notion for studying reductions in the lambda calculus.

Extensions of this typing discipline have also been considered by several authors to capture new computational paradigms such as multi-threaded programs [MA11, Mad12], process calculi [DLMS10, DLMS16], and quantum programs [DLMZ10]. We will briefly describe them and discuss them in this section. This description is non-exhaustive but covers some of the most important aspects: extension to a concurrent programming language; handling of imperative features, threads, and side effects; and extension to a quantum programming language and characterizations of quantum polynomial time complexity classes.

3.2.1 Light logic and multi-threaded programs

Extension of light logics to programs with multi-threads and side effects have been considered by Amadio and Madet to capture elementary time and polynomial time in [MA11] and [Mad12], respectively.

Syntax and semantics of $\lambda^{!,\S,||}$

The language $\lambda^{!,\S,||}$ of [Mad12] guarantees termination in polynomial time, covering any scheduling of threads, and is defined by the following grammar:

$$
\begin{array}{lllll}
\texttt{Terms} & M & ::= & \texttt{x} \mid r \mid * \mid \lambda \texttt{x}.M \mid M\ M \mid !M \mid \S M \\
 & & & \texttt{let}\ !x = M\ \texttt{in}\ M \mid \texttt{let}\ \S x = M\ \texttt{in}\ M \mid \\
 & & & \texttt{get}(r) \mid \texttt{set}(r, M) \mid (M||M) \\
\texttt{Stores} & \mathcal{S} & ::= & r \Leftarrow M \mid (\mathcal{S}||\mathcal{S}) \\
\texttt{Program} & \mathcal{P} & ::= & r \Leftarrow M \mid \mathcal{S} \mid (\mathcal{P}||\mathcal{P}),
\end{array}
$$

where r belongs to a fixed set of regions (memory locations), $*$ is the unit, the operator $\texttt{get}(\text{r})$ reads the region r, the operator $\texttt{set}(\text{r, M})$ assigns M to the region r, and $||$ is a parallel composition operator. A store is a parallel composition of assignments of a term M to a region r, $r \Leftarrow M$. A program is a parallel composition of terms and stores. Programs are always considered up to the

following structural equivalence rules $(\mathcal{P}_1||\mathcal{P}_2) \equiv (\mathcal{P}_2||\mathcal{P}_1)$ and $(\mathcal{P}_1||(\mathcal{P}_2||\mathcal{P}_3)) \equiv ((\mathcal{P}_1||\mathcal{P}_2)||\mathcal{P}_3)$. The depth $d(\mathcal{P})$ of a program $\mathcal{P}$ is the maximal number of nested modalities and its size $|\mathcal{P}|$ is the number of symbols in $\mathcal{P}$.

Reduction contexts are defined by the following grammar:

$$\mathcal{E} ::= [] \mid \mathcal{E}\ M \mid V\ \mathcal{E} \mid \S\mathcal{E} \mid \texttt{let}\ !\texttt{x} = \mathcal{E}\ \texttt{in}\ M \mid \texttt{let}\ \S\texttt{x} = \mathcal{E}\ \texttt{in}\ M \mid \texttt{set}(r, \mathcal{E}) \mid (\mathcal{E}||\mathcal{P}) \mid (\mathcal{P}||\mathcal{E}),$$

where V is a *value* defined by $V ::= x \mid r \mid * \mid \lambda\texttt{x}.M \mid \S V \mid !V$.

A Call-By-Value (CBV) evaluation strategy can then be defined:

- $\mathcal{E}[(\lambda\texttt{x}.M)\ V] \to_v \mathcal{E}[M\{V/\texttt{x}\}]$,
- $\mathcal{E}[\texttt{let}\ !\texttt{x} = !V\ \texttt{in}\ M] \to_v \mathcal{E}[M\{V/\texttt{x}\}]$,
- $\mathcal{E}[\texttt{let}\ \S\texttt{x} = \S V\ \texttt{in}\ M] \to_v \mathcal{E}[M\{V/\texttt{x}\}]$,
- $\mathcal{E}[\texttt{get}(r)]||r \Leftarrow V \to_v \mathcal{E}[V]$,
- $\mathcal{E}[\texttt{set}(r, V)] \to_v \mathcal{E}[*]||r \Leftarrow V$,
- $\mathcal{E}[*||M] \to_v \mathcal{E}[M]$.

Notice that, by definition of reduction contexts, no reduction occurs under a !.

Light type system for $\lambda^{!,\S,||}$

Before introducing the type system, define a *region context* R to be a mapping from some finite set of regions $dom(R)$ to a natural number, noted $R = r_1 : n_1, \ldots, r_k : n_k$, for $n_i \in \mathbb{N}$ and $dom(R) = \{r_1, \ldots, r_k\}$. Define a variable context Γ to be a mapping from some finite set of variables to a usage in $\{!, \S, \lambda\}$, noted $\Gamma = \texttt{x}_1 : u_1, \ldots, \texttt{x}_k : u_k$, for $u_i \in \{!, \S, \lambda\}$ and $dom(\Gamma) = \{\texttt{x}_1, \ldots, \texttt{x}_k\}$. The resource usage indicates a constraint on the variable bound. Let Γ_u, $u \in \{!, \S, \lambda\}$, be a shorthand notation for the variable context Γ where all variable in $dom(\Gamma)$ have usage u.

The light linear depth type system is presented in Figure 3.10. This type system works in a standard way through a stratification mechanism by depth level. It is worth mentioning that the type system of Figure 3.10 does not prevent programs from going wrong (deadlocks, ...). An improved version preventing such behaviors has also been presented in [Mad12].

Let us consider a small example.

Example 3.2.1. *Let* `add` *be the standard encoding of addition over Church numerals of type* $Nat \multimap Nat \multimap Nat$ *with* $Nat = \forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$. *The term*

$$\texttt{inc} = \lambda\texttt{x}.\lambda\texttt{z}.(\S(\texttt{set}(\texttt{x}, \texttt{let}\ !\texttt{y} = \texttt{get}(\texttt{x})\ \texttt{in}\ !\texttt{add}(\underline{1}, \texttt{y})))||\texttt{z})$$

increases the Church numeral stored at some location. Its well-formedness can be derived as

$$\frac{\mathtt{x} : \lambda \in \Gamma}{R; \Gamma \vdash^n \mathtt{x}}\ (\text{Var}) \qquad \frac{}{R; \Gamma \vdash^n *}\ (\text{Unit}) \qquad \frac{}{R; \Gamma \vdash^n r}\ (\text{Reg})$$

$$\frac{FO(\mathtt{x}, M) = 1 \qquad R; \Gamma, \mathtt{x} : \lambda \vdash^n M}{R; \Gamma \vdash^n \lambda \mathtt{x}.M}\ (\text{Abs}) \qquad \frac{R; \Gamma \vdash^n M \qquad R; \Gamma \vdash^n N}{R; \Gamma \vdash^n M\ N}\ (\text{App})$$

$$\frac{FO(M) \leq 1 \qquad R; \Gamma_\lambda \vdash^{n+1} M}{R; \Gamma_!, \Delta_\S, \Omega_\lambda \vdash^n !M}\ (\text{Pr}_!) \qquad \frac{FO(\mathtt{x}, N) \geq 1 \qquad R; \Gamma \vdash^n M \qquad R; \Gamma, \mathtt{x} :! \vdash^n N}{R; \Gamma \vdash^n \mathtt{let}\ !\mathtt{x} = M\ \mathtt{in}\ N}\ (\text{E}_!)$$

$$\frac{FO(M) \leq 1 \qquad R; \Gamma_\lambda, \Delta_\lambda \vdash^{n+1} M}{R; \Gamma_!, \Delta_\S, \Omega_\lambda \vdash^n \S M}\ (\text{Pr}_\S) \qquad \frac{FO(\mathtt{x}, N) = 1 \qquad R; \Gamma \vdash^n M \qquad R; \Gamma, \mathtt{x} : \S \vdash^n N}{R; \Gamma \vdash^n \mathtt{let}\ \S\mathtt{x} = M\ \mathtt{in}\ N}\ (\text{E}_\S)$$

$$\frac{r : n \in R}{R; \Gamma \vdash^n \mathtt{get}(r)}\ (\text{Get}) \qquad \frac{r : n \in R \qquad R; \Gamma \vdash^n M}{R; \Gamma \vdash^n \mathtt{set}(r, M)}\ (\text{Set})$$

$$\frac{r : n \in R \qquad R; \Gamma \vdash^n M}{R; \Gamma \vdash^0 r \Leftarrow M}\ (\text{Str}) \qquad \frac{R; \Gamma \vdash^n \mathcal{P}_1 \qquad R; \Gamma \vdash^n \mathcal{P}_2}{R; \Gamma \vdash^n \mathcal{P}_1 || \mathcal{P}_2}\ (\text{Par})$$

Figure 3.10: LLL-based well-formedness rules for $\lambda^{!,\S,||}$

follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\mathtt{x}:1;\mathtt{y}:\lambda\vdash^1 \mathtt{add}(\underline{1},\mathtt{y})}\,(Pr_!)}{\mathtt{x}:1;\vdash^1 \mathtt{get}(\mathtt{x})}\,(Get) \quad \mathtt{x}:1;\mathtt{y}:!\vdash^1 !\mathtt{add}(\underline{1},\mathtt{y})}\,(E_!)}{\mathtt{x}:1;\vdash^1 \mathtt{let}\ !\mathtt{y}=\mathtt{get}(\mathtt{x})\ \mathtt{in}\ !\mathtt{add}(\underline{1},\mathtt{y})}\,(Set)}{\mathtt{x}:1;\vdash^1 \mathtt{set}(\mathtt{x},\mathtt{let}\ !\mathtt{y}=\mathtt{get}(\mathtt{x})\ \mathtt{in}\ !\mathtt{add}(\mathtt{y},\mathtt{y}))}}{\mathtt{x}:1;\mathtt{x}:\lambda,\mathtt{z}:\lambda\vdash^0 \S\mathtt{set}(\mathtt{x},\mathtt{let}\ !\mathtt{y}=\mathtt{get}(\mathtt{x})\ \mathtt{in}\ !\mathtt{add}(\underline{1},\mathtt{y}))}\,(Pr_\S) \quad \dfrac{}{\mathtt{x}:1;\mathtt{x}:\lambda,\mathtt{z}:\lambda\vdash^0 z}\,(Var)}{\mathtt{x}:1;\mathtt{x}:\lambda,\mathtt{z}:\lambda\vdash^0 \S(\mathtt{set}(\mathtt{x},\mathtt{let}\ !\mathtt{y}=\mathtt{get}(\mathtt{x})\ \mathtt{in}\ !\mathtt{add}(\underline{1},\mathtt{y})))||z}\,(Par)$$

$$\dfrac{\dfrac{\mathtt{x}:1;\mathtt{x}:\lambda,\mathtt{z}:\lambda\vdash^0 \S(\mathtt{set}(\mathtt{x},\mathtt{let}\ !\mathtt{y}=\mathtt{get}(\mathtt{x})\ \mathtt{in}\ !\mathtt{add}(\underline{1},\mathtt{y})))||z}{\mathtt{x}:1;\mathtt{x}:\lambda\vdash^0 \lambda\mathtt{z}.(\S(\mathtt{set}(\mathtt{x},\mathtt{let}\ !\mathtt{y}=\mathtt{get}(\mathtt{x})\ \mathtt{in}\ !\mathtt{add}(\underline{1},\mathtt{y})))||z)}\,(Abs)}{\mathtt{x}:1;\vdash^0 \mathtt{inc}}\,(Abs)$$

The last two rules can be applied as the number of free occurrences of `x` *and* `z` *in the term is exactly* 1*. The promotion rules* $Pr_\S$ *and* $Pr_!$ *can be applied for the same reason.*

We can now state the main result, a polynomial time soundness result on such programs.

Theorem 3.2.1 ([Mad12])**.** *Given a program* $\mathcal{P}$*, if there are* Γ*,* R*, and* n *such that* $R;\Gamma\vdash^n \mathcal{P}$ *then the CBV evaluation of* $\mathcal{P}$ *can be computed in time* $O(|\mathcal{P}|^{2^{d(\mathcal{P})}})$*.*

A polynomial completeness result also holds for `FP` as the type system extends LAL. An extension taking into account thread generation using higher-order references is also presented in [Mad12].

3.2.2 Soft logic and process calculi

An extension of SLL in the context of a higher-order process calculus $HO\pi$ has been studied by Dal Lago, Martini, and Sangiorgi in [DLMS10, DLMS16], where a soundness result is shown: typable processes terminate in polynomial time.

Syntax and semantics of $LHO\pi$

Considered processes are defined by the following grammar.

$$\begin{array}{lrcl} \texttt{Processes} & P & ::= & 0 \mid (P|P) \mid a(\mathtt{x}).P \mid \overline{a}\langle V\rangle.P \mid (\nu a)P \mid V\ V \\ \texttt{Values} & V & ::= & * \mid \mathtt{x} \mid \lambda\mathtt{x}.P \end{array}$$

Before introducing a linear type system, [DLMS16] extends standard processes into a language of linear processes LHOπ defined by the following grammar and reduction.

$$\begin{array}{lrcl} \texttt{Linear Processes} & LP & ::= & 0 \mid (LP|LP) \mid a(\mathtt{x}).LP \mid \overline{a}\langle LV\rangle.LP \mid (\nu a)LP \\ & & & \mid LV\ LV \mid a(!\mathtt{x}).LP \\ \texttt{Linear Values} & LV & ::= & * \mid \mathtt{x} \mid \lambda\mathtt{x}.LP \mid \lambda!\mathtt{x}.LP \mid !LV \end{array}$$

Processes in HOπ can be transformed into LHOπ processes using the following embedding $(-)^\bullet$.

$$\begin{array}{ll} (\overline{a}\langle V\rangle.P)^\bullet = \overline{a}\langle !(V)^\bullet\rangle.(P)^\bullet & (\overline{a}(\mathtt{x}).P)^\bullet = a(!\mathtt{x}).(P)^\bullet \\ (\lambda\mathtt{x}.P) = \lambda!\mathtt{x}.(P)^\bullet & (V\ W)^\bullet = (V)^\bullet!(W)^\bullet \\ (P|Q)^\bullet = (P)^\bullet|(Q)^\bullet & ((\nu a)P)^\bullet = (\nu a)(P)^\bullet \\ (k)^\bullet = k,\ k\in\{0,*,\mathtt{x}\}. & \end{array}$$

The reduction relation $\rightarrow$ on closed linear processes is standard and defined in Figure 3.11.

$$\frac{}{a(\mathtt{x}).LP|\overline{a}\langle LV\rangle.LQ \rightarrow LP\{V/\mathtt{x}\}|LQ} \qquad \frac{}{\lambda\mathtt{x}.LP\ LV \rightarrow LP\{V/\mathtt{x}\}}$$

$$\frac{}{a(!\mathtt{x}).LP|\overline{a}\langle !LV\rangle.LQ \rightarrow P\{LV/\mathtt{x}\}|LQ} \qquad \frac{}{\lambda!\mathtt{x}.LP\ !V \rightarrow LP\{LV/\mathtt{x}\}}$$

$$\frac{LP \rightarrow LQ}{LR|LP \rightarrow LR|LQ} \qquad \frac{LP \rightarrow QL}{(\nu a)LP \rightarrow (\nu a)LQ} \qquad \frac{LR \equiv LP \quad LP \rightarrow LQ \quad LQ \equiv LS}{LR \rightarrow LS}$$

Figure 3.11: Operational semantics of LHOπ

$\equiv$ is defined to be the smallest congruence closed under the rules:

$$LP \equiv LQ, \text{ if } LP =_\alpha LQ, \qquad (LP|(LQ|LR)) \equiv ((LP|LQ)|LR),$$
$$(LP|LQ) \equiv (LQ|LP) \qquad (\nu a)((\nu b)LP) \equiv (\nu b)((\nu a)LP),$$
$$((\nu a)LP|LQ) \equiv (\nu a)(LP|LQ), \text{ provided } a \notin FV(LQ).$$

The standard rules of the π-calculus $(LP|0) \equiv LP$ and $(\nu a)0 \equiv 0$ are withdrawn from the congruence as they are not well-suited for a resource sensitive analysis.

Soft type system for $LHO\pi$

The soft type system of [DLMS10] is then introduced in Figure 3.12. As in previous section, a

$$\frac{k \in \{*, 0\}}{\#\Gamma \vdash_{SP} k} \qquad \frac{\Gamma, \#\Omega \vdash_{SP} LP \quad \Delta, \#\Omega \vdash_{SP} LQ}{\Gamma,\ \Delta, \#\Omega \vdash_{SP} LP|LQ} \qquad \frac{\Gamma, \mathtt{x}:\ \lambda \vdash_{SP} LP}{\Gamma \vdash_{SP} \lambda\mathtt{x}.LP} \qquad \frac{\Gamma \vdash_{SP} LP}{\Gamma \vdash_{SP} (\nu a)LP}$$

$$\frac{\Gamma, \mathtt{x} : \lambda \vdash_{SP} LP}{\Gamma \vdash_{SP} a(\mathtt{x}).LP} \qquad \frac{\Gamma, \mathtt{x}:\ u \vdash_{SP} LP \quad u \in \{!, \#\}}{\Gamma \vdash_{SP} a(!\mathtt{x}).LP}$$

$$\frac{\Gamma, \mathtt{x}:\ u \vdash_{SP} LP \quad u \in \{!, \#\}}{\Gamma \vdash_{SP} \lambda!\mathtt{x}.LP} \qquad \frac{u \in \{\lambda, \#\}}{\#\Gamma, \mathtt{x} : u \vdash_{SP} \mathtt{x}}$$

$$\frac{\Gamma, \#\Omega \vdash_{SP} LV \quad \Delta, \#\Omega \vdash_{SP} LW}{\Gamma, \Delta, \#\Omega \vdash_{SP} LV\ LW} \qquad \frac{\Gamma \vdash_{SP} LV}{!\Gamma, \#\Omega \vdash_{SP} !LV} \qquad \frac{\Gamma, \#\Omega \vdash_{SP} LV \quad \Delta, \#\Omega \vdash_{SP} LP}{\Gamma, \Delta, \#\Omega \vdash_{SP} \overline{a}\langle LV\rangle.LP}$$

Figure 3.12: SLL-based process type system

variable environment Γ is of the shape $\mathtt{x}_1 : u_1, \ldots, \mathtt{x}_n : u_n$, where each variable $\mathtt{x}_i$ comes with a unique usage $u_i \in \{\lambda, !, \#\}$, where $\#$ is a mark for contraction or weakening. For a given usage $u \in \{\lambda, !, \#\}$, $u\Gamma$ stands for an environment where all variables have usage u. In a context of the shape $\Gamma, u\Delta$, all variables in Γ are assumed to be of usage distinct from u.

As usual, the depth $d(P)$ of a process P in LHOπ is the maximal number of nested modalities and the size of a $|P|$ process P is its number of symbols.

Again, this system has a inherent stratification property ensuring that linear variables (of usage λ) occur exactly once at depth 0, non-linear variables of usage ! occur exactly once at depth 1, and non-linear variables of usage # occur at depth 0. This latter variables may occur several times.

Theorem 3.2.2 ([DLMS10]). *There are polynomials $(P_n)_{n\in\mathbb{N}}$ such that for each process P, if $\vdash_{SP} P$ and $P \to^m Q$ then* $\max(m, |Q|) \leq P_{d(P)}(|P|)$.

The property of Theorem 3.2.2 can be seen as restrictive for processes as, by essence, it forbids any non-terminating process from being analyzed. To overcome this issue, an extension to non-terminating processes has been studied in [DLMS16], where a polynomial bound is ensured on the number of internal computation steps performed by a process between any two interactions with its environment.

3.2.3 Soft linear logic and quantum programs

In [DLMZ10], the methodology of SLL is applied to a restricted quantum programming language by Dal Lago, Masini, and Zorzi. Measurement (see Chapter 5 for a discussion) is not handled by the language itself and only occurs at the end of a computation. [DLMZ10] introduces a quantum programming language SQ with a soft linear logic based type system that is shown to be sound and complete with respect to polynomial time Quantum Turing Machines (QTM) (see [Deu85] for a definition), hence providing characterizations of the three quantum complexity classes `EQP`, `BQP`, and `ZQP` (see [BV97] for a definition).

Syntax and semantics of SQ

We will not present the full quantum computation paradigm in this subsection and we invite the interested reader to read either [Sel04] or [SV06] for an interesting presentation of an imperative quantum language and a functional quantum language, respectively.

Terms of SQ are defined by the following grammar:

$$\begin{array}{llll} \texttt{Patterns} & \phi & ::= & x \mid !x \mid \langle x_1, \ldots, x_n \rangle \\ \texttt{Terms} & \texttt{M} & ::= & 0 \mid 1 \mid x \mid r \mid !\texttt{M} \mid U \mid \texttt{new}(\texttt{M}) \mid \langle \texttt{M}, \ldots, \texttt{M} \rangle \mid \texttt{M}\ \texttt{M} \mid \lambda\phi.\texttt{M}, \end{array}$$

where $x, x_1, \ldots, x_n$ are variables for classical data, r is a variable for quantum data, and U is a unitary operator on the 2^n dimensional Hilbert space $\mathcal{H}(\{0,1\}^n)$. Recall that a unitary operator is a bounded linear operator $U : \mathcal{H} \to \mathcal{H}$ on a Hilbert space $\mathcal{H}$ that satisfies $U^\dagger U = UU^\dagger = I$, where $U^\dagger$ is the conjugate transpose of U and I is the identity operator on $\mathcal{H}$, *e.g.* the Hadamard gate is an example of such an operator.

Given a term `M`, let $Q(\texttt{M})$ be the set of quantum variables occurring in `M`. A preconfiguration is a triplet $[\mathcal{Q}, \mathcal{QV}, \texttt{M}]$ consisting in a set of quantum variables $\mathcal{QV}$ such that $Q(\texttt{M}) \subseteq \mathcal{QV}$, a state $\mathcal{Q}$ of the Hilbert space $\mathcal{H}(\{0,1\}^{card(\mathcal{QV})})$, and a term `M`. A configuration is just an equivalence class of preconfigurations obtained from a preconfiguration $[\mathcal{Q}, \mathcal{QV}, \texttt{M}]$ and whose term and set of quantum variables can be obtained by a bijective renaming of quantum variables in $\mathcal{QV}$. The reduction rules consist in standard reduction rules on configuration described in Figure 3.13 together with commutation rules and contextual closure rules that we avoid to present here (the full rules are presented in [DLMZ10]). Let $\to^*$ be the reflexive and transitive closure of the commutative and contextual closure of $\to$.

$$
\begin{aligned}
[\mathcal{Q}, \mathcal{QV}, (\lambda x.\mathtt{M})\ \mathtt{N}] &\to [\mathcal{Q}, \mathcal{QV}, \mathtt{M}\{\mathtt{N}/x\}] \\
[\mathcal{Q}, \mathcal{QV}, (\lambda\langle x_1, \ldots, x_n\rangle.\mathtt{M})\ \langle r_1, \ldots, r_n\rangle] &\to [\mathcal{Q}, \mathcal{QV}, \mathtt{M}\{r_1/x_1, \ldots, r_n/x_n\}] \\
[\mathcal{Q}, \mathcal{QV}, (\lambda !x.\mathtt{M})\ !\mathtt{N}] &\to [\mathcal{Q}, \mathcal{QV}, \mathtt{M}\{\mathtt{N}/x\}] \\
[\mathcal{Q}, \mathcal{QV}, U\ \langle r_1, \ldots, r_n\rangle] &\to [U_{r_1,\ldots,r_n}\mathcal{Q}, \mathcal{QV}, \langle r_1, \ldots, r_n\rangle] \\
[\mathcal{Q}, \mathcal{QV}, \mathtt{new}(i)] &\to [\mathcal{Q} \otimes |r \mapsto i\rangle, \mathcal{QV} \cup \{r\}, r] \quad i \in \{0, 1\}
\end{aligned}
$$

Figure 3.13: Standard reduction rules of SQ

In the last rule of Figure 3.13, $\otimes$ is the standard tensor product of Hilbert spaces and r is a fresh quantum variable. In the penultimate rule, $U_{r_1,\ldots,r_n}$ stands for the unitary operator transformation applied on qubits referenced by $r_1, \ldots, r_n$ in the state $\mathcal{Q}$. We do not present the full details here to avoid technicalities.

Soft type system for SQ

As in previous Subsection, a typing environment Γ comes with type annotations on its variables. The typing environments $!\Gamma$ and $\#\Gamma$ are obtained from Γ by annotating all variables with ! and #, respectively. The type system of SQ is presented in Figure 3.14 where it is implicitly assumed that each classical or quantum variable occurs at most once in each environment of a typing judgment. The aim of the type system is again to enforce some kind of stratification discipline ensuring that a free variable x with no annotation occurs at most once in a typable term (and never under the ! modality), that a free variable x annotated by # occurs at least once in a typable term (and never under a ! modality), and that a free variable x annotated by ! occurs at most once in a typable term (it might occur under a !).

$$
\frac{k \in \{0,1\}}{!\Gamma \vdash k} \qquad \frac{}{!\Gamma, r \vdash r} \qquad \frac{\dagger \in \{\emptyset,\ !, \#\}}{!\Gamma, \dagger x \vdash x} \qquad \frac{;\forall i \leq n,\ \Gamma_i, \#\Delta_i \vdash \mathtt{M}_i}{\cup_{i=1}^n \Gamma_i, \# \cup_{i=1}^n \Delta_i \vdash \langle \mathtt{M}_1, \ldots, \mathtt{M}_n\rangle}
$$

$$
\frac{\Gamma, \#\Delta \vdash \mathtt{M} \quad \Gamma', \#\Delta' \vdash \mathtt{N}}{\Gamma, \Gamma', \#\Delta, \#\Delta' \vdash \mathtt{M}\ \mathtt{N}} \qquad \frac{\Gamma \vdash \mathtt{M}}{!\Gamma, !\Delta \vdash !\mathtt{M}} \qquad \frac{\Gamma \vdash \mathtt{M}}{\Gamma \vdash \mathtt{new}(\mathtt{M})}
$$

$$
\frac{\Gamma, x_1, \ldots, x_n \vdash \mathtt{M}}{\Gamma \vdash \lambda\langle x_1, \ldots, x_n\rangle.\mathtt{M}} \qquad \frac{\Gamma, x \vdash \mathtt{M}}{\Gamma \vdash \lambda x.\mathtt{M}} \qquad \frac{\Gamma, \dagger x \vdash \mathtt{M} \quad \dagger \in \{\#, !\}}{\Gamma \vdash \lambda !x.\mathtt{M}}
$$

Figure 3.14: SQ type system

Before stating the main characterizations of [DLMZ10], we first show how to encode decision problems in the language.

For that purpose, we introduce some preliminary notations corresponding to data structures encoding presented in [DLMZ10]. Given some terms $\mathtt{M}_1, \ldots, \mathtt{M}_n$, let $[\mathtt{M}_1, \ldots, \mathtt{M}_n]$ be an encoding of the sequence of terms $\mathtt{M}_1, \ldots, \mathtt{M}_n$. For a binary string t, let $\tilde{t}$ be an encoding the binary string t as a typable term of SQ. The notation $!^n\mathtt{M}$ stands for $!\ldots!\mathtt{M}$, n times.

A term M outputs the binary string $s \in \{0,1\}^*$ with probability p on input N if there is a constant $m \geq |s|$ such that $[1, \emptyset, \mathtt{M}\ \mathtt{N}] \to^* [Q, \{r_1, \ldots, r_n\}, [r_1, \ldots, r_n]]$ and the probability of observing s when *projecting* $\mathcal{Q}$ into $\mathcal{H}(\{0,1\}^{m-|s|})$ is exactly p.

Definition 3.2.1. *Given $n \in \mathbb{N}$, two binary strings $s, r \in \{0,1\}^*$, and $p \in [0,1]$, a typable term* M *of SQ and a language $L \subseteq \{0,1\}^*$.*

- M *(n, s, r, p)-decides L if and only if for every binary string t the following two conditions hold:*
  - M *outputs s with probability at least p on input $!^n\tilde{t}$, if $t \in L$,*
  - M *outputs r with probability at least p on input $!^n\tilde{t}$, if $t \notin L$.*
- M *is (n, s, r)-error-free if and only for every binary string t, the following two conditions hold on input $!^n\tilde{t}$:*
  - *if* M *outputs s with positive probability then* M *outputs r with null probability,*
  - *if* M *outputs r with positive probability then* M *outputs s with null probability.*

Definition 3.2.2. *The class of language* ESQ*,* BSQ *and* ZSQ *are defined as follows.*

- ESQ *is the class of languages that are $(n, s, r, 1)$-decided by a term of SQ.*
- BSQ *is the class of languages that are (n, s, r, p)-decided by a term of SQ, with $p > 1/2$.*
- ZSQ *is the class of languages that are (n, s, r, p)-decided by a (n, s, r)-error-free term of SQ, with $p > 1/2$.*

We recall briefly the definition of the quantum complexity classes EQP, BQP, and ZQP of [BV97].

Definition 3.2.3. *The complexity classes* EQP*,* BQP*, and* ZQP *are defined as follows.*

- EQP *is the class of decision problems computed by QTMs that output the correct answer with probability 1 and run in polynomial time.*
- BQP *is the class of decision problems computed by QTMs that output the correct answer with probability $p > 1/2$ and run in polynomial time.*
- ZQP *is the class of decision problems computed by QTMs that output the answer with probability $p > 1/2$ and error of probability 0, and run in polynomial time.*

Now we are ready to state the characterization of these complexity classes.

Theorem 3.2.3 ([DLMZ10])**.** *$\forall$*x *$\in \{$*E*,* B*,* Z*$\}$,* xSQ *=* xQP*.*

3.2.4 Miscellaneous

In this subsection, we briefly discuss some other extensions or alternative uses of soft/light logics approaches in the framework of proof-nets, interaction-nets, and model theory.

Proof-nets and interaction nets

The paper [Per18] introduces a decidable syntactic proof-net-based subsystem of linear logic, called SDNLL, that is sound and complete for polynomial time and that strictly embeds LLL. Although not directly related to light logics, the complexity of functional programs has been studied in [GM16] using an interaction-net computational model, a generalization of linear logic proof-nets. This model has been combined with sized types to certify time and space complexity bounds for both sequential and parallel proof-net reductions.

Categorical and realizability models

Categorical models for ELL and SLL have been provided in [LTdF06]. The paper [Red07] has provided an axiomatization of categorical-based SLL models. Categorical models for ELL and LLL have been studied in [Bai04a]. Realizability models have been adapted to the ICC setting in [DLH05, DLH11] for soundness proofs of EAL and Soft Affine Logic (SAL). They have also been adapted to show the soundness of a fragment of second order linear logic with type fixpoints (with respect to FP) in [BT10] and of LAL in [BM12].

3.3 Extensions of interpretations

As presented in Section 2.2 and Section 2.3, the interpretation method and the quasi-interpretation method were originally designed to study termination properties and complexity properties, respectively, of first order TRSs. In this section, we discuss their extensions to other programming paradigms including higher-order TRSs [BMP07, BDL12, BDL16], first order and higher-order functional programs [GP09a, GP09b, GP15b, HP17], Object Oriented programs [MP08a], Java bytecode [ACGDZJ04], cooperative threads [ADZ04], and polygraphs [BG08, BG07].

3.3.1 Higher-order rewrite systems

In this subsection, we focus on extensions of interpretation methods to higher-order rewriting. The considered computational model will be Simply Typed TRSs (STTRSs) studied by Yamada [Yam01]. They consist in a simple extension of TRS with higher-order functions. Consider a fixed set of basic datatypes $\mathcal{B}$. The set of types is defined by:

$$A ::= b \mid A_1 \times \ldots \times A_n \to A,$$

with $b \in \mathcal{B}$.

The set of terms is defined by:

$$t^A ::= \mathtt{x}^A \mid \mathtt{c}^{b_1 \times \ldots \times b_n \to b} \mid \mathtt{fun}^A \mid (t^{A_1 \times \ldots \times A_n \to A}\ t_1^{A_1}\ \ldots\ t_n^{A_n})^A,$$

where $\mathtt{x}$ is a variable in $\mathcal{X}$, $\mathtt{c}$ is a constructor symbol in $\mathcal{C}$, and $\mathtt{fun}$ is a function symbol in $\mathcal{F}$. Constructor symbol types are restricted to functionals on basic types. As usual, for a given term t, $\mathbb{V}(t)$ denotes the set of variables in t. Consequently, $\mathbb{V}(t) \subseteq \mathcal{X}$. A pattern p is a term with no occurrence of a function symbol.

A STTRS consists in a set of non-overlapping rules $l^A \to r^A$, satisfying:

- $\mathbb{V}(r) \subseteq \mathbb{V}(l)$ and variables occur once in l,
- $l = \mathtt{fun}^{A_1 \times \ldots \times A_n \to A}\ p_1^{A_1}\ \ldots\ p_n^{A_n}$, for some patterns $p_1, \ldots, p_n$.

The underlying CBV rewrite relation is defined in a standard way by verifying that types match in a substitution. A substitution σ maps variables to values of the same type that are defined inductively by

$$v ::= \mathtt{c}^{b_1\times\ldots\times b_n\to b}\ v_1\ \ldots v_n \mid \mathtt{fun}^{A_1\times\ldots\times A_n\to A}\ v_1\ \ldots\ v_k,$$

with $k < n$. Contexts are defined in a standard manner. We write $t \to_{\mathcal{R}} s$ if there are a context $\mathtt{C}[\diamond^A]$, a rule $l^A \to r^A$, and a substitution σ such that $t = \mathtt{C}[l\sigma]$ and $s = \mathtt{C}[r\sigma]$.

Example 3.3.1. *Consider the following STTRS as an illustrating example:*

$$\begin{aligned}
(\mathtt{fold\ f\ z\ nil}) &\to \mathtt{z},\\
(\mathtt{fold\ f\ z\ (c\ hd\ tl)}) &\to (\mathtt{f\ hd\ (fold\ f\ z\ tl)}),\\
(\mathtt{add\ 0\ y}) &\to \mathtt{y},\\
(\mathtt{add\ (x{+}1)\ y}) &\to (\mathtt{add\ x\ y}){+}1,\\
(\mathtt{sum\ l}) &\to (\mathtt{fold\ add\ 0\ l}),
\end{aligned}$$

with $\mathcal{X} = \{\mathtt{x}^{Nat}, \mathtt{y}^{Nat}, \mathtt{z}^{Nat}, \mathtt{f}^{Nat\times Nat\to Nat}, \mathtt{hd}^{Nat}, \mathtt{tl}^{L(Nat)}, \mathtt{l}^{L(Nat)}\}$, $\mathcal{C} = \{\mathtt{0}^{Nat}, {+}\mathtt{1}^{Nat\to Nat}, \mathtt{nil}^{L(Nat)}, \mathtt{c}^{Nat\times L(Nat)\to L(Nat)}\}$, *and* $\mathcal{F} = \{\mathtt{fold}^{(Nat\times Nat\to Nat)\times Nat\times L(Nat)\to Nat}, \mathtt{add}^{Nat\times Nat\to Nat}, \mathtt{sum}^{L(Nat)\to Nat}\}$, *Nat being the basic type of unary numbers and* $L(Nat)$ *being the basic type of lists of unary numbers.*

Defunctionalization

STTRSs are handled in [BMP07] using defunctionalization techniques [Rey72, CD96]. See [Dan06] for an introduction to defunctionalization and Continuation-Passing Style (CPS). To avoid technical details, we just illustrate this transformation through the following example.

Example 3.3.2. *The following TRS is the CPS defunctionalization of the STTRS of Example 3.3.1:*

$$\begin{aligned}
\overline{\mathtt{fold}}(\mathtt{z},\mathtt{nil}) &\to \mathtt{z},\\
\overline{\mathtt{fold}}(\mathtt{z},\mathtt{c}(\mathtt{hd},\mathtt{tl})) &\to \mathtt{app}(\mathtt{c}_0(\mathtt{hd}),\overline{\mathtt{fold}}(\mathtt{z},\mathtt{tl})),\\
\mathtt{add}(\mathtt{0},\mathtt{y}) &\to \mathtt{y},\\
\mathtt{add}((\mathtt{x}{+}1),\mathtt{y}) &\to \mathtt{add}(\mathtt{x},\mathtt{y}){+}1,\\
\mathtt{sum}(\mathtt{l}) &\to \overline{\mathtt{fold}}(\mathtt{0},\mathtt{l}),\\
\mathtt{app}(\mathtt{c}_0(\mathtt{x}_1),\mathtt{x}_2) &\to \mathtt{add}(\mathtt{x}_1,\mathtt{x}_2).
\end{aligned}$$

Let $DF(S)$ be the set of function symbols of a STTRS of type $b_1 \times \ldots \times b_n \to b$, for some basic types $b_1, \ldots, b_n, b$, and whose CPS defunctionalization is in S, for some set of TRSs S. We obtain the following straightforward characterizations.

Theorem 3.3.1 ([BMP07])**.** *For* $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$*, we have:*

- $[\![DF(QI^{poly}_{add}(\mathbb{K}, \geq_{\mathbb{K}}) \cap RPO^{\{p\}})]\!] = \mathtt{FP}$,
- $[\![DF(QI^{poly}_{add}(\mathbb{K}, \geq_{\mathbb{K}}) \cap RPO^{\{l\}})]\!] = \mathtt{FPSPACE}$.

Example 3.3.3. *The TRS of Example 3.3.2 admits the following interpretation:*

$$\begin{aligned}
[\mathtt{0}]_{\mathbb{K}} &= [\mathtt{nil}]_{\mathbb{K}} = 0,\\
[\mathtt{c_0}]_{\mathbb{K}}(X) &= [+\mathtt{1}]_{\mathbb{K}}(X) = X + 1,\\
[\mathtt{c}]_{\mathbb{K}}(X,Y) &= X + Y + 1,\\
[\overline{\mathtt{fold}}]_{\mathbb{K}}(X,Y) &= [\mathtt{add}]_{\mathbb{K}}(X,Y) = [\mathtt{app}]_{\mathbb{K}}(X,Y) = X + Y,\\
[\overline{\mathtt{sum}}]_{\mathbb{K}}(X) &= X.
\end{aligned}$$

Moreover it can be oriented by RPO with status p. Consequently, the function symbol sum *of the STTRS of Example 3.3.1 computes a function in* FP.

Some modular extensions using the notion of stratified union of Section 2.3.5 are also studied in [BMP07].

Related applied works studying the complexity of higher-order functionals can be found in [ADLM15]. The programs are analyzed automatically by applying program transformations to a defunctionalization and using standard complexity analysis techniques for first order TRSs. Similar clock based transformations were also studied in [DLR15]. More generally, techniques for reasoning formally about execution costs of higher-order programs were studied in [San90, San91].

Higher-order interpretations

The direct study of STTRS complexity is tackled in [BDL12, BDL16] by extending interpretations to higher-order polynomials defined as the β-normal forms of terms generated by the following grammar:

$$M ::= \mathtt{x}^A \mid \mathtt{op} \mid (M^{A\to B}\ N^A)^B \mid (\lambda \mathtt{x}^A.M^B)^{A\to B},$$

with $\mathtt{op} \in \{+ : Nat \times Nat \to Nat, * : Nat \times Nat \to Nat\} \cup \{\underline{n} : Nat \mid \forall n \in Nat\}$ and

$$A, B ::= Nat \mid A \times B \mid A \to B.$$

Higher-Order Polynomials (HOP) are restricted to strictly increasing total functions relatively to a pointwise order such that $f >_{A\to B} g$ if and only if $\forall a \in A,\ f(a) >_B g(a)$.

STTRS types are interpreted inductively by $[Nat]_{\mathbb{N}} = \mathbb{N}$ and $[A_1 \times \ldots \times A_n \to A]_{\mathbb{N}} = [A_1]_{\mathbb{N}} \times \ldots \times [A_n]_{\mathbb{N}} \to [A]_{\mathbb{N}}$. An assignment of a STTRS is a map from symbols in $\mathcal{C} \cup \mathcal{F}$ to HOPs: each symbol $\mathtt{b}^A \in \mathcal{C} \cup \mathcal{F}$ is mapped to a HOP $[\mathtt{b}]_{\mathbb{N}}$ of type $[A]_{\mathbb{N}}$. Assignments are extended to terms in a standard way. Each variable $\mathtt{x}^A$ of $\mathcal{X}$ is interpreted by a variable $[\mathtt{x}]_{\mathbb{N}}$ of type $[A]_{\mathbb{N}}$. Each term of the shape $(\mathtt{b}^{A_1\times\ldots\times A_n\to A}\ t_1^{A_1} \ldots\ t_n^{A_n})^A$ is mapped to $([\mathtt{b}]_{\mathbb{N}}^{[A_1]_{\mathbb{N}}\times\ldots\times[A_n]_{\mathbb{N}}\to[A]_{\mathbb{N}}}\ [t_1]_{\mathbb{N}}^{[A_1]_{\mathbb{N}}} \ldots\ [t_n]_{\mathbb{N}}^{[A_n]_{\mathbb{N}}})^{[A]_{\mathbb{N}}} \equiv_\beta M^{[A]_{\mathbb{N}}}$, for some HOP M.[25]

Definition 3.3.1. *An assignment $[-]_{\mathbb{N}}$ is a higher-order polynomial interpretation of a STTRS if for each rule $l^A \to r^A$, $[l]_{\mathbb{N}}^{[A]_{\mathbb{N}}} >_{[A]_{\mathbb{N}}} [r]_{\mathbb{N}}^{[A]_{\mathbb{N}}}$. If the interpretation of each symbol in $\mathcal{C}$ is additive (see Definition 2.2.5) then $[-]_{\mathbb{N}}$ is an additive higher-order polynomial interpretation.*

Let $HOPI_{add}^{poly}$ be the set of TRSs that admit an additive higher-order polynomial interpretation over $\mathbb{N}$. We obtain a characterization of FP extending the characterization of Theorem 2.2.2 to STTRS.

[25] Some β reduction steps are needed to compute the β normal form M. Hence we consider HOPs up to β equivalence, $\equiv_\beta$.

Theorem 3.3.2 ([BDL12]). $[\![HOPI^{poly}_{add}]\!] = \mathtt{FP}$.[26]

Example 3.3.4 ([BDL12]). *Consider the following STTRS:*

$$(\mathtt{map\ f\ nil}) \to \mathtt{nil},$$
$$(\mathtt{map\ f\ (c\ hd\ tl)}) \to (\mathtt{c\ (f\ hd)\ (map\ f\ tl)}).$$

It admits the following additive higher-order polynomial interpretation:

$$[\mathtt{nil}]^{\mathbb{N}}_{\mathbb{N}} = 2,$$
$$[\mathtt{c}]^{\mathbb{N}\times\mathbb{N}\to\mathbb{N}}_{\mathbb{N}} = \lambda n.\lambda m.m + n + 1,$$
$$[\mathtt{map}]^{(\mathbb{N}\to\mathbb{N})\to\mathbb{N}\to\mathbb{N}}_{\mathbb{N}} = \lambda\phi.\lambda n.n \times \phi(n).$$

For the first rule, we check that:

$$\begin{aligned}[\mathtt{map\ f\ nil}]_{\mathbb{N}} &= \lambda\phi.\lambda n.n \times \phi(n)\ [\mathtt{f}]_{\mathbb{N}}\ [\mathtt{nil}]_{\mathbb{N}}\\ &= \lambda\phi.\lambda n.n \times \phi(n)[\mathtt{f}]_{\mathbb{N}}\ 2\\ &\equiv_\beta 2 \times [\mathtt{f}]_{\mathbb{N}}(2)\\ &> 2\\ &= [\mathtt{nil}]_{\mathbb{N}}.\end{aligned}$$

The strict inequality holds as HOP are restricted to strictly increasing total functions and, consequently, $\forall n \in \mathbb{N},\ [\mathtt{f}]_{\mathbb{N}}(n) \geq n$. *This explains the reason why the interpretation of* `nil` *has been set to* 2.

We let the reader check that the strict inequality also holds for the second rule. For any first order function symbol `fun` *(of the right type) admitting a polynomial interpretation, the function symbol* $\mathtt{map}_{\mathtt{fun}}$ *defined by the rule* $(\mathtt{map}_{\mathtt{fun}}\ \mathtt{l}) = (\mathtt{map\ fun\ l})$ *admits an additive higher-order polynomial interpretation defined by:* $[\mathtt{map}_{\mathtt{fun}}]_{\mathbb{N}} = \lambda l.([\mathtt{map}]_{\mathbb{N}}\ [\mathtt{fun}]_{\mathbb{N}}\ l) + 1$, *and, consequently, computes a function in* `FP`, *by Theorem 3.3.2.*

As polynomial interpretations, the notion of higher-order polynomial interpretation suffers from a weak expressive power. It has been extended in [BDL16] to obtain a notion of higher-order quasi-interpretation. This latter notion has been combined to a syntactic termination criterion using a linearity restriction to provide a more expressive characterization of `FP` in [BDL16].

3.3.2 Functional programs

The issue of knowing whether interpretation methods and quasi-interpretation methods can be adapted to functional programs was open for a while: the difficulty was hidden in the basic nature of interpretations that provides a well-founded decreasing order on program reductions. While the decrease is explicit in TRSs as data (constructor symbols) are explicitly consumed by the rules in recursive calls, this phenomenon is less obvious for general functional programs where data may be consumed using destructors or pattern matching. This issue was tackled in two steps with an extension of interpretations to first order programs in [GP09a, GP09b, GP15b] and to higher-order programs in [HP17]. We will present this latter approach in this subsection.

Consider the following higher-order programming language:

$$\mathtt{M, N} ::= \mathtt{x} \mid \mathtt{c} \mid \mathtt{op} \mid \mathtt{M\ N} \mid \lambda\mathtt{x.M} \mid \mathtt{letRec\ f = M} \mid \mathtt{case\ M\ of\ c_1\ x_1 : M_1 | ... | c_n\ x_n : M_n},$$

[26] This results holds for type 1 function symbols over basic types, *e.g.* unary numbers or binary words.

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma;\Delta \vdash \mathtt{x} : \mathtt{T}}\ (\mathtt{Var}) \qquad \frac{\Delta(\mathtt{c}) = \mathtt{T}}{\Gamma;\Delta \vdash \mathtt{c} : \mathtt{T}}\ (\mathtt{Cons}) \qquad \frac{\Delta(\mathtt{op}) = \mathtt{T}}{\Gamma;\Delta \vdash \mathtt{op} : \mathtt{T}}\ (\mathtt{Op})$$

$$\frac{\Gamma;\Delta \vdash \mathtt{M} : \mathtt{T}_1 \longrightarrow \mathtt{T}_2 \quad \Gamma;\Delta \vdash \mathtt{N} : \mathtt{T}_1}{\Gamma;\Delta \vdash \mathtt{M\ N} : \mathtt{T}_2}\ (\mathtt{App}) \qquad \frac{\Gamma, \mathtt{x} : \mathtt{T}_1;\Delta \vdash \mathtt{M} : \mathtt{T}_2}{\Gamma;\Delta \vdash \lambda\mathtt{x.M} : \mathtt{T}_1 \to \mathtt{T}_2}\ (\mathtt{Abs})$$

$$\frac{\Gamma;\Delta \vdash \mathtt{M} : \mathtt{b} \quad \Gamma;\Delta \vdash \mathtt{c}_i : \mathtt{b}_i \longrightarrow \mathtt{b} \quad \forall i \leq m,\ \Gamma, \mathtt{x}_i : \mathtt{b}_i;\Delta \vdash \mathtt{M}_i : \mathtt{T}}{\Gamma;\Delta \vdash \mathtt{case\ M\ of\ c}_1\ \mathtt{x}_1 : \mathtt{M}_1 | ... | \mathtt{c}_n\ \mathtt{x}_n : \mathtt{M}_n : \mathtt{T}}\ (\mathtt{Case})$$

$$\frac{\Gamma, \mathtt{f} :: \mathtt{T};\Delta \vdash \mathtt{M} :: \mathtt{T}}{\Gamma;\Delta \vdash \mathtt{letRec\ f} = \mathtt{M}\ :: \mathtt{T}}\ (\mathtt{Let})$$

Figure 3.15: Type system for the higher-order functional program

where $\mathtt{c}, \mathtt{c}_1, \cdots, \mathtt{c}_n$ are constructor symbols of fixed arity and $\mathtt{op}$ is an operator of fixed arity. In the $\mathtt{case}$ construct, $\mathtt{x}_1, \ldots, \mathtt{x}_n$ can be (empty) sequences of variables. A program is a closed term of the language.

Each operator is associated with a total function $[\![\mathtt{op}]\!]$ of the same arity mapping programs to programs. We define the following reductions:

- $\lambda\mathtt{x.M\ N} \to_\beta \mathtt{M}\{\mathtt{N/x}\}$,
- $\mathtt{case\ c}_j\ \mathtt{N}_j\ \mathtt{of}\ \ldots \mid \mathtt{c}_j\ \mathtt{x}_j : \mathtt{M}_j \mid \ldots \to_{\mathtt{case}} \mathtt{M}_j\{\mathtt{N}_j/\mathtt{x}_j\}$,
- $\mathtt{letRec\ f} = \mathtt{M} \to_{\mathtt{letRec}} \mathtt{M}\{\mathtt{letRec\ f} = \mathtt{M/f}\}$,
- $\mathtt{op\ M}_1 \ldots \mathtt{M}_n \to_{\mathtt{op}} [\![\mathtt{op}]\!](\mathtt{M}_1, \ldots, \mathtt{M}_n)$.

Let $\Rightarrow$ be the left-most outermost evaluation strategy associated to $\to_\beta \cup \to_{\mathtt{case}} \cup \to_{\mathtt{letRec}} \cup \to_{\mathtt{op}}$.

The set of simple types is defined by:

$$\mathtt{T} ::= \mathtt{b} \mid \mathtt{T} \longrightarrow \mathtt{T}, \quad \text{with } \mathtt{b} \in \mathbf{B}.$$

where the basic type $\mathtt{b}$ belongs to a fixed set $\mathbf{B}$ of basic inductive types $\mathtt{b}$ described by their constructor set $\mathcal{C}_\mathtt{b}$. For example, the type of (unary) natural numbers Nat is described by $\mathcal{C}_{\mathrm{Nat}} = \{\mathtt{0}, \mathtt{+1}\}$. As usual Γ and Δ will denote a variable typing environment and an operator (and constructor) typing environment, respectively.

A well-typed term will consist in a term $\mathtt{M}$ such that $\emptyset;\Delta \vdash \mathtt{M} : \mathtt{T}$ using the rules of Figure 3.15. Consequently, it is mandatory for a term to be closed in order to be well-typed.

Example 3.3.5. *Consider the following term* $\mathtt{M}$ *that maps a function to a list given as inputs:*

$$\begin{aligned}\mathtt{letRec\ f} = \lambda\mathtt{g}.\lambda\mathtt{x}.\mathtt{case\ x\ of\ c\ y\ z} : \mathtt{c\ (g\ y)\ (f\ g\ z)},\\ \mid \mathtt{nil} : \mathtt{nil}.\end{aligned}$$

Suppose that $L(Nat)$ *is the base type for lists of natural numbers of constructor set* $\mathcal{C}_{L(Nat)} = \{\mathtt{nil}, \mathtt{c}\}$. *The term* $\mathtt{M}$ *can be typed by* $\mathtt{T} = (Nat \longrightarrow Nat) \longrightarrow L(Nat) \longrightarrow L(Nat)$ *for some context* Δ *such that* $\Delta(\mathtt{c}) = Nat \longrightarrow L(Nat) \longrightarrow L(Nat)$ *and* $\Delta(\mathtt{nil}) = L(Nat)$.

Let $(\mathbb{N}, \leq, \sqcup, \sqcap)$ be the set of natural numbers equipped with the usual ordering $\leq$, a max operator $\sqcup$ and min operator $\sqcap$ and let $\overline{\mathbb{N}}$ be $\mathbb{N} \cup \{\top\}$, where $\forall n \in \mathbb{N},\ n \leq \top$, $n \sqcup \top = \top \sqcup n = \top$ and $n \sqcap \top = \top \sqcap n = n$. The strict order relation over natural numbers $<$ will also be used in the sequel and is extended in a somewhat unusual manner, by $\top < \top$.

The (higher-order) *interpretation* of a type is defined inductively by:

$$[\mathtt{b}] = \overline{\mathbb{N}},\ \text{if } \mathtt{b} \text{ is a basic type},$$
$$[\mathtt{T} \longrightarrow \mathtt{T}'] = [\mathtt{T}] \longrightarrow^{\uparrow} [\mathtt{T}'],\ \text{otherwise},$$

where $[\mathtt{T}] \longrightarrow^{\uparrow} [\mathtt{T}']$ denotes the set of total strictly monotonic functions from $[\mathtt{T}]$ to $[\mathtt{T}']$. As in previous section, a function F from the set A to the set B is strictly monotonic if and only if for each $X, Y \in A$, $X <_A Y$ implies $F(X) <_B F(Y)$, where $<_A$ is the usual ordering induced by $<$ and defined inductively by:

$$n <_{\overline{\mathbb{N}}} m, \text{ if and only if } n < m,$$
$$F <_{A \longrightarrow^{\uparrow} B} G, \text{ if and only if } \forall X \in A,\ F(X) <_B G(X).$$

Example 3.3.6. *The type* $\mathtt{T} = (Nat \longrightarrow Nat) \longrightarrow L(Nat) \longrightarrow L(Nat)$ *of the term* `letRec f = M` *in Example 3.3.5 is interpreted by:*

$$[\mathtt{T}] = (\overline{\mathbb{N}} \longrightarrow^{\uparrow} \overline{\mathbb{N}}) \longrightarrow^{\uparrow} (\overline{\mathbb{N}} \longrightarrow^{\uparrow} \overline{\mathbb{N}}).$$

Each closed term of type $\mathtt{T}$ will be interpreted by a functional in $[\mathtt{T}]$. The application is denoted as usual whereas we use the notation Λ for abstraction on this function space in order to avoid confusion between terms of our calculus and objects of the interpretation domain. Variables of the interpretation domain will be denoted using upper case letters. Moreover, we will sometimes use Church typing discipline in order to highlight the type of the bound variable in a lambda abstraction.

An important distinction between the terms of our language and the objects of the interpretation domain lies in the fact that beta-reduction is considered as an equivalence relation on (closed terms of) the interpretation domain, *i.e.* $(\Lambda X.F)G = F\{G/X\}$ underlying that $(\Lambda X.F)\ G$ and $F\{G/X\}$ are distinct notations that represent the same higher-order function. The same property holds for η-reduction, *i.e.* $\Lambda X.(FX)$ and F denote the same function.

Since we are interested in complete lattices, we need to complete each type $[\mathtt{T}]$ by a lower bound $\bot_{[\mathtt{T}]}$ and an upper bound $\top_{[\mathtt{T}]}$ inductively as follows:

$$\bot_{\overline{\mathbb{N}}} = 0, \qquad \top_{\overline{\mathbb{N}}} = \top,$$
$$\bot_{[\mathtt{T} \longrightarrow \mathtt{T}']} = \Lambda X^{[\mathtt{T}]}.\bot_{[\mathtt{T}']}, \qquad \top_{[\mathtt{T} \longrightarrow \mathtt{T}']} = \Lambda X^{[\mathtt{T}]}.\top_{[\mathtt{T}']}.$$

Now we need to define a unit (or constant) cost function for any interpretation of type $\mathtt{T}$ in order to take the cost of recursive calls into account. For that purpose, let $+$ denote natural number addition extended to $\overline{\mathbb{N}}$ by $\forall n,\ \top + n = n + \top = \top$. For each type $[\mathtt{T}]$, we define inductively a dyadic sum function $\oplus_{[\mathtt{T}]}$ by:

$$X^{\overline{\mathbb{N}}} \oplus_{\overline{\mathbb{N}}} Y^{\overline{\mathbb{N}}} = X + Y,$$
$$F \oplus_{[\mathtt{T} \longrightarrow \mathtt{T}']} G = \Lambda X^{[\mathtt{T}]}.(F(X) \oplus_{[\mathtt{T}']} G(X)).$$

Let us also define the constant function $n_{[\mathtt{T}]}$, for each type $\mathtt{T}$ and each integer $n \geq 1$, by:

$$n_{\overline{\mathbb{N}}} = n,$$
$$n_{[\mathtt{T} \longrightarrow \mathtt{T}']} = \Lambda X^{[\mathtt{T}]}.n_{[\mathtt{T}']}.$$

- $[\mathtt{f}]_\rho = \rho(\mathtt{f}), \quad \text{if } \mathtt{f} \in \mathbb{V},$
- $[\mathtt{c}]_\rho = 1 \oplus (\Lambda X_1 . \ldots . \Lambda X_n . \sum_{i=1}^{n} X_i), \quad \text{if } ar(\mathtt{c}) = n,$
- $[\mathtt{MN}]_\rho = [\mathtt{M}]_\rho [\mathtt{N}]_\rho,$
- $[\lambda \mathtt{x}.\mathtt{M}]_\rho = 1 \oplus (\Lambda [\mathtt{x}]_\rho . [\mathtt{M}]_\rho),$
- $[\mathtt{case\ M\ of\ c}_1\ \mathtt{x}_1 : \mathtt{M}_1 | \ldots | \mathtt{c}_n\ \mathtt{x}_n : \mathtt{M}_n]_\rho = 1 \oplus \sqcup_{1 \leq i \leq m} \{[\mathtt{M}_i]_\rho \{R_i / [\mathtt{x}_i]_\rho\} \mid \forall R_i \text{ s.t. } [\mathtt{M}]_\rho \geq [\mathtt{c}_i]_\rho R_i\},$
- $[\mathtt{letRec\ f} = \mathtt{M}]_\rho = \sqcap \{F \in [\mathtt{T}] \mid F \geq \Lambda [\mathtt{f}]_\rho . [\mathtt{M}]_\rho (1 \oplus F)\}.$

Figure 3.16: Higher-order interpretation of a term

Once again, we will omit the type when it is unambiguous using the notation $n\oplus$ to denote the function $n_{[\mathtt{T}]}\oplus_{[\mathtt{T}]}$ when $[\mathtt{T}]$ is clear from the typing context.

In the same spirit, we extend inductively the *max* and min operators $\sqcup$ and $\sqcap$ over $\overline{\mathbb{N}}$ to arbitrary higher-order functions F, G of type $[\mathtt{T}] \longrightarrow^{\uparrow} [\mathtt{T}']$ by:

$$\sqcup^{[\mathtt{T}] \longrightarrow^{\uparrow} [\mathtt{T}']}(F, G) = \Lambda X^{[\mathtt{T}]} . \sqcup^{[\mathtt{T}']} (F(X), G(X)),$$
$$\sqcap^{[\mathtt{T}] \longrightarrow^{\uparrow} [\mathtt{T}']}(F, G) = \Lambda X^{[\mathtt{T}]} . \sqcap^{[\mathtt{T}']} (F(X), G(X)).$$

In the following, we use the notations $\bot$, $\top$, $\leq$, $<$, $\sqcup$ and $\sqcap$ instead of $\bot_{[\mathtt{T}]}$, $\top_{[\mathtt{T}]}$, $\leq_{[\mathtt{T}]}$, $<_{[\mathtt{T}]}$, $\sqcup^{[\mathtt{T}]}$ and $\sqcap^{[\mathtt{T}]}$, respectively, when $[\mathtt{T}]$ is clear from the typing context.

Definition 3.3.2. *A* variable assignment*, denoted ρ, is a map associating to each $\mathtt{f} \in \mathbb{V}$ of type $\mathtt{T}$, a variable F of type $[\mathtt{T}]$.*

Definition 3.3.3 (Interpretation)**.** *Given a variable assignment ρ, an* interpretation *is the extension of ρ to well-typed terms, mapping each term of type $\mathtt{T}$ to an object in $[\mathtt{T}]$ and defined in Figure 3.16 and extended to operators in such a way that $[\mathtt{op}]_\rho$ is a sup-interpretation, i.e. a total function such that:*

$$\forall \mathtt{M}_1, \ldots, \forall \mathtt{M}_n, \ [\mathtt{op\ M}_1\ \ldots\ \mathtt{M}_n]_\rho \geq [[\![\mathtt{op}]\!](\mathtt{M}_1, \ldots, \mathtt{M}_n)]_\rho.$$

As operator sup-interpretations are fixed, an interpretation should also be indexed by some mapping m assigning a sup-interpretation to each operator of the language. To simplify the formalism, we have omitted this mapping.

Let the size $|\mathtt{M}|$ of a term $\mathtt{M}$ be defined by $|\mathtt{x}| = 0$, $|\mathtt{c}| = 1$, $|\mathtt{M\ N}| = |\mathtt{M}| + |\mathtt{N}|$, $|\lambda \mathtt{x}.\mathtt{M}| = 1 + |\mathtt{M}|$, $|\mathtt{letRec\ f} = \mathtt{M}| = 1 + |\mathtt{M}|$, and $|\mathtt{case\ M\ of\ c}_1\ \mathtt{x}_1 : \mathtt{M}_1 | \ldots | \mathtt{c}_n\ \mathtt{x}_n : \mathtt{M}_n| = 1 + |\mathtt{M}| + \sum_{i=1}^{n} |\mathtt{M}_i|$. First order values V are defined as in Subsection 2.2.1.

The developed notion of interpretation has the following properties.

Lemma 3.3.1 ([HP17])**.** *For each value V, $[V]_\rho = |V|$.*

Lemma 3.3.1 corresponds to Lemma 2.2.1 of Subsection 2.2.2. The equality is obtained as the additive constant of the interpretation of constructors is enforced to be 1 in Figure 3.16.

Lemma 3.3.2 ([HP17]). *For any program* $\mathtt{M}$*, if* $\mathtt{M} \Rightarrow \mathtt{N}$ *then* $[\mathtt{M}]_\rho \geq [\mathtt{N}]_\rho$*. If* $\mathtt{M} \rightarrow_\alpha \mathtt{N}$*,* $\alpha \in \{\beta, \mathtt{letRec}, \mathtt{case}\}$*, then* $[\mathtt{M}]_\rho > [\mathtt{N}]_\rho$*.*

Lemma 3.3.2 is the higher-order counter-part of Lemma 2.3.1 of Subsection 2.3.2. Indeed, if M reduces to some value V then by transitivity, we obtain $[\mathtt{M}]_\rho \geq [V]_\rho = |V|$, by Lemma 3.3.1.

Corollary 3.3.1 ([HP17]). *For any programs,* $\mathtt{M}$*,* $\mathtt{N}$*, such that* $\emptyset; \Delta \vdash \mathtt{M}\ \mathtt{N} :: \mathtt{b}$*, with* $\mathtt{b} \in \boldsymbol{B}$*, if* $[\mathtt{M}\ \mathtt{N}]_\rho \neq \top$ *then* $\mathtt{M}\ \mathtt{N}$ *terminates in a number of reduction steps in* $O([\mathtt{M}\ \mathtt{N}]_\rho)$*.*

Corollary 3.3.1 is the higher-order counter-part of Theorem 2.2.1 of Subsection 2.2.2. In the particular case where for any program $\mathtt{N}$ of the right type, $[\mathtt{M}\ \mathtt{N}]_\rho \neq \top$, the higher-order program $\mathtt{M}$ terminates on any input (hence, computes a total function).

We end the presentation of these interpretations by exhibiting the interpretation of the program of Example 3.3.5. We apply some relaxations on the interpretation (upper bounds) to simplify the presentation.

Example 3.3.7 ([HP17]). *Consider the program* $\mathtt{M}$ *of Example 3.3.5:*

$$\begin{aligned}\mathtt{letRec\ f} = \lambda\mathtt{g}.\lambda\mathtt{x}.\mathtt{case\ x\ of\ c\ y\ z} : \mathtt{c\ (g\ y)\ (f\ g\ z)},\\ \mid \mathtt{nil} : \mathtt{nil}.\end{aligned}$$

By applying the rules of Figure 3.16, the following inequalities can be derived.

$$\begin{aligned}
&[\mathtt{M}]_\rho\\
&= \sqcap\{F \in [\mathtt{T}] \mid F \geq \Lambda[\mathtt{f}]_\rho.[\lambda\mathtt{g}.\lambda\mathtt{x}.\mathtt{case\ x\ of\ c\ y\ z} : \mathtt{c\ (g\ y)\ (f\ g\ z)} \mid \mathtt{nil} : \mathtt{nil}]_\rho(1 \oplus F)\}\\
&= \sqcap\{F \in [\mathtt{T}] \mid F \geq 2 \oplus (\Lambda[\mathtt{f}]_\rho.\Lambda[\mathtt{g}]_\rho.\Lambda[\mathtt{x}]_\rho.[\mathtt{case\ x\ of\ c\ y\ z} : \mathtt{c\ (g\ y)\ (f\ g\ z)} \mid \mathtt{nil} : \mathtt{nil}]_\rho(1 \oplus F))\}\\
&= \sqcap\{F \in [\mathtt{T}] \mid F \geq 3 \oplus (\Lambda[\mathtt{f}]_\rho.\Lambda[\mathtt{g}]_\rho.\Lambda[\mathtt{x}]_\rho.\sqcup([\mathtt{nil}]_\rho, \sqcup_{[\mathtt{x}]_\rho \geq [\mathtt{c(y,z)}]_\rho}([\mathtt{c(g\ y)(f\ g\ z)}]_\rho))(1 \oplus F))\}\\
&= \sqcap\{F \in [\mathtt{T}] \mid F \geq 5 \oplus (\Lambda[\mathtt{g}]_\rho.\Lambda[\mathtt{x}]_\rho.\sqcup_{[\mathtt{x}]_\rho \geq 1 \oplus [\mathtt{y}]_\rho + [\mathtt{z}]_\rho}(([\mathtt{g}]_\rho\ [\mathtt{y}]_\rho) + (F\ [\mathtt{g}]_\rho\ [\mathtt{z}]_\rho)))\}\\
&\leq \sqcap\{F \in [\mathtt{T}] \mid F \geq 5 \oplus (\Lambda[\mathtt{g}]_\rho.\Lambda[\mathtt{x}]_\rho.(([\mathtt{g}]_\rho\ ([\mathtt{x}]_\rho - 1)) + (F\ [\mathtt{g}]_\rho\ ([\mathtt{x}]_\rho - 1))))\}\\
&\leq \Lambda[\mathtt{g}]_\rho.\Lambda[\mathtt{x}]_\rho.(5 \oplus ([\mathtt{g}]_\rho\ [\mathtt{x}]_\rho)) \times [\mathtt{x}]_\rho
\end{aligned}$$

In the penultimate line, we obtain an upper-bound on the interpretation by approximating the case interpretation, substituting $[x]_\rho - 1$ *to both* $[y]_\rho$ *and* $[z]_\rho$*. In the last line, we obtain an upper-bound on the interpretation by approximating the* `letRec` *interpretation, just checking that the function* $\Lambda[\mathtt{g}]_\rho.\Lambda[\mathtt{x}]_\rho.(5 \oplus ([\mathtt{g}]_\rho\ [\mathtt{x}]_\rho)) \times [\mathtt{x}]_\rho$*, where* $\times$ *is the usual multiplication symbol over natural numbers, satisfies the inequality* $F \geq 5 \oplus (\Lambda[\mathtt{g}]_\rho.\Lambda[\mathtt{x}]_\rho.(([\mathtt{g}]_\rho\ ([\mathtt{x}]_\rho - 1)) + (F\ [\mathtt{g}]_\rho\ ([\mathtt{x}]_\rho - 1))))$*.*

Consequently, the program $\mathtt{M}$ *admits an interpretation bounded by* $\Lambda[\mathtt{g}]_\rho.\Lambda[\mathtt{x}]_\rho.(5 \oplus ([\mathtt{g}]_\rho\ [\mathtt{x}]_\rho)) \times [\mathtt{x}]_\rho$*.*

The properties of higher-order interpretations can be used to characterize `FP` and also higher-order polynomial time complexity classes. For that purpose, we need to restrict the space of interpretations to higher-order polynomials. These results will be presented in Section 4.2.

3.3.3 Object oriented programs

Sup-interpretations have also been extended to the case of OO programming in [MP08a].

The syntax of considered OO programs is very close to the syntax of programs in Subsection 3.1.4 and is defined by:

$$\begin{array}{rcl}
\text{Expressions} \ni \mathtt{e} & ::= & \mathtt{x} \mid \mathtt{cst}_\tau \mid \mathtt{null} \mid \mathtt{this} \mid \mathtt{op}(\overline{\mathtt{e}}) \mid \mathtt{new\ C}(\overline{\mathtt{e}}) \mid \mathtt{x.m}(\overline{\mathtt{e}}) \\
\text{Instructions} \ni \mathtt{I} & ::= & \mathtt{skip;} \mid \mathtt{x{:=}e;} \mid \mathtt{I_1\ I_2} \mid \mathtt{while(e)\{I\}} \mid \mathtt{loop(x)\{I\}} \mid \mathtt{if(e)\{I_1\}else\{I_2\}} \\
\text{Methods} \ni \mathtt{m_C} & ::= & \mathtt{m}(\mathtt{x}_1,\ldots,\mathtt{x}_n)\{\mathtt{I\ return\ x;}\} \\
\text{Constructors} \ni \mathtt{k_C} & ::= & \mathtt{C}(\mathtt{x}_1,\ldots,\mathtt{x}_n)\{\mathtt{X}_1{:=}\mathtt{x}_1,\ldots,\mathtt{X}_n{:=}\mathtt{x}_n\} \\
\text{Classes} \ni \mathtt{C} & ::= & \mathtt{class\ C}\ \{\overline{\mathtt{var\ X;}}\ \overline{\mathtt{k_C}}\ \overline{\mathtt{m_C}}\}.
\end{array}$$

The distinctions are that methods always return, that method calls are performed on variables, and that an extra-loop instruction is included in the language. As a consequence, method calls are considered to be expressions. The variable guarding a loop cannot be assigned to within the loop body. In the constructor $\mathtt{C}$, the variable $\mathtt{X}_1,\ldots,\mathtt{X}_n$ have to match exactly the attributes $\overline{\mathtt{var\ X;}}$ in the declaration $\mathtt{C}\ \{\overline{\mathtt{var\ X;}}\ \overline{\mathtt{k_C}}\ \overline{\mathtt{m_C}}\}$ of the corresponding class. Moreover, to simplify the discussion, local variables, overload, override, and inheritance and more generally name clashes are prohibited. We also assume that methods are non-recursive.

A main class is a special class called main with no constructor and no method (definitions).

A program is composed by a set of classes with exactly one main class. All the programs have to enjoy some well-formedness conditions that are similar to the well-formedness conditions of Subsection 3.1.4.

Example 3.3.8. *Consider the linked list class* `LL` *defined below.* X *and* Y *represent the head and tail attributes.* W *and* Z *are extra-attributes storing intermediate computations. These latter variables are required to compensate for the absence of local variables.*

```
class LL {
    var X; var Y; var W; var Z;

    LL(x,y,w,z){X := x; Y := y; W := w; Z := z;}

    getHead(){skip; return X; }

    getTail(){skip; return Y; }

    setTail(y){Y := y; return X; }

    reverse(){
        Z := new LL(X,null,null,nul);
        W := Y;
        loop(Y){
            Z := new LL(W.getHead(),Z,null,null);
            W := W.getTail();
        }
       return Z;
    }
}
```

```
class main {
    var T; var U; var V;
    V := new LL(U, null, null, null);
    loop(T){U := V.setTail(V);}
}
```

The semantics however greatly differs as it is a non-reference based semantics. For that purpose, we consider objects as first order values defined by

$$o ::= \texttt{null} \mid \texttt{C}(o_1, \ldots, o_n).$$

The size of an object is defined as usual. A store σ is a partial map from attributes and parameters to objects. Let $\texttt{I}^n$ be a shorthand notation for $\texttt{I} \ \ldots \ \texttt{I}$, n times, The big step operational semantics of a program is defined in Figure 3.17, provided that the truth values 1

$$\frac{}{(\texttt{x}, \sigma) \downarrow (\texttt{x}\sigma, \sigma)} \qquad \frac{}{(\texttt{null}, \sigma) \downarrow (\texttt{null}, \sigma)} \qquad \frac{(\texttt{e}_i, \sigma) \downarrow (o_i, \sigma)}{(\texttt{new C}(\texttt{e}_1, \ldots, \texttt{e}_n), \sigma) \downarrow (\texttt{new C}(o_1, \ldots, o_n), \sigma)}$$

$$\frac{(\texttt{x}, \sigma) \downarrow (\texttt{new C}(o_1, \ldots, o_m), \sigma) \qquad \forall j,\ (\texttt{e}_j, \sigma) \downarrow (o'_j, \sigma)}{\texttt{C}\ \{\ldots \texttt{var X}_i; \ldots \texttt{m}(\tau_1\ \texttt{x}_1, \ldots, \tau_n\ \texttt{x}_n)\{\texttt{I return y;}\}\ldots\} \qquad (\texttt{I}, \sigma\{\texttt{X}_i \leftarrow o_i, \texttt{x}_j \leftarrow o'_j\}) \downarrow \sigma'}$$
$$(\texttt{x.m}(\texttt{e}_1, \ldots, \texttt{e}_n), \sigma) \downarrow (\texttt{y}\sigma', \sigma'\{\texttt{x} \leftarrow \texttt{new C}(\texttt{X}_1\sigma', \ldots, \texttt{X}_m\sigma')\})$$

$$\frac{}{(\texttt{skip;}, \sigma) \downarrow \sigma} \qquad \frac{(\texttt{e}, \sigma) \downarrow (o, \sigma')}{(\texttt{x:=e;}, \sigma) \downarrow \sigma'\{\texttt{x} \leftarrow o\}} \qquad \frac{(\texttt{e}, \sigma) \downarrow w \in \{0, 1\} \qquad (\texttt{I}_w, \sigma) \downarrow \sigma'}{(\texttt{if(e)\{I}_1\texttt{\}else\{I}_0\texttt{\}}, \sigma) \downarrow \sigma'}$$

$$\frac{(\texttt{I}_1, \sigma) \downarrow \sigma' \qquad (\texttt{I}_2, \sigma') \downarrow \sigma''}{(\texttt{I}_1\ \texttt{I}_2, \sigma) \downarrow \sigma''} \qquad \frac{(\texttt{e}, \sigma) \downarrow 1 \qquad (\texttt{I while(e)\{I\}}, \sigma) \downarrow \sigma'}{(\texttt{while(e)\{I\}}, \sigma) \downarrow \sigma'}$$

$$\frac{(\texttt{e}, \sigma) \downarrow 0}{(\texttt{while(e)\{I\}}, \sigma) \downarrow \sigma} \qquad \frac{(\texttt{x}, \sigma) \downarrow (o, \sigma) \qquad (\texttt{I}^{|o|}, \sigma) \downarrow \sigma'}{(\texttt{loop(x)\{I\}}, \sigma) \downarrow \sigma'}$$

Figure 3.17: Big step operational semantics of OO programs

and 0 are encoded by the object `null` and any object distinct from `null`, respectively.

Example 3.3.9 ([MP08a]). *Consider the program of Example 3.3.8. For any store σ such that $\texttt{U}\sigma = o$ and $\texttt{T}\sigma = o'$, we have:*

$$\begin{aligned}(\texttt{V} := \texttt{new LL}(\texttt{U}, \overline{\texttt{null}});, \sigma) \downarrow \sigma\{\texttt{V} \leftarrow \texttt{new LL}(\texttt{o}, \overline{\texttt{null}})\} = \sigma',\\ (\texttt{U} := \texttt{V.setTail}(\texttt{V});, \sigma') \downarrow \sigma\{\texttt{V} \leftarrow \texttt{new LL}(\texttt{o}, \texttt{new LL}(\texttt{o}, \overline{\texttt{null}}), \overline{\texttt{null}})\},\\ (\texttt{loop}(\texttt{T})\{\texttt{U} := \texttt{V.setTail}(\texttt{V});\}, \sigma') \downarrow \sigma\{\texttt{V} \leftarrow \texttt{f}^{|o'|+1}(\texttt{null})\},\end{aligned}$$

where $\texttt{f} = \texttt{x} \mapsto \texttt{new LL}(\texttt{o}, \texttt{x}, \texttt{null}, \texttt{null})$ and $\texttt{f}^n$ is defined in a standard manner.

We can now adapt the notion of assignment in the context of OO programs.

Definition 3.3.4 (Assignment). *Given an OO program, an assignment $[-]_{\mathbb{R}^+}$ maps:*

- *every variable* $\mathtt{x}$ *to a variable* $[\mathtt{x}]_{\mathbb{R}^+}$ *in* $\mathbb{R}^+$,
- *every constructor symbol* $\mathtt{C}$ *corresponding to a class of* n *attributes to a total monotonic function* $[\mathtt{C}]_{\mathbb{R}^+} : (\mathbb{R}^+)^n \to \mathbb{R}^+$,
- *every method symbol* $\mathtt{m}$ *with* n *parameters to a total monotonic function* $[\mathtt{m}]_{\mathbb{R}^+} : (\mathbb{R}^+)^{n+1} \to \mathbb{R}^+$.

Assignments are extended canonically to expressions by:

$$[\mathtt{new\ C}(\mathtt{e}_1, \ldots, \mathtt{e}_n)]_{\mathbb{R}^+} = [\mathtt{C}]_{\mathbb{R}^+}([\mathtt{e}_1]_{\mathbb{R}^+}, \ldots, [\mathtt{e}_n]_{\mathbb{R}^+})$$
$$[\mathtt{x.m}(\mathtt{e}_1, \ldots, \mathtt{e}_n)]_{\mathbb{R}^+} = [\mathtt{m}]_{\mathbb{R}^+}([\mathtt{e}_1]_{\mathbb{R}^+}, \ldots, [\mathtt{e}_n]_{\mathbb{R}^+}, [\mathtt{x}]_{\mathbb{R}^+})$$

The notion of additive and polynomial assignment can be defined as in Subsection 2.2.2.

Definition 3.3.5 (Sup-interpretation)**.** *Given an OO program, an assignment* $[-]_{\mathbb{R}^+}$ *is a sup-interpretation if for each method* $\mathtt{m}$ *of arity* n *and each store* σ *such that* $(\mathtt{x.m}(\mathtt{x}_1, \ldots, \mathtt{x}_n), \sigma) \downarrow (o, \sigma')$*, we have:*

$$[\mathtt{m}]_{\mathbb{R}^+}([\mathtt{x}_1\sigma]_{\mathbb{R}^+}, \ldots, [\mathtt{x}_n\sigma]_{\mathbb{R}^+}, [\mathtt{x}\sigma]_{\mathbb{R}^+}) \geq \max([o]_{\mathbb{R}^+}, [\mathtt{x}\sigma']_{\mathbb{R}^+}).$$

The above definition is very similar to Definition 2.4.1 of Section 2.4. The main distinction is that, not only the return value sup-interpretation $[o]_{\mathbb{R}^+}$ is bounded from above, but also the side-effect $[\mathtt{x}\sigma']_{\mathbb{R}^+}$.

The absence of references makes the sup-interpretation to roughly approximate complex data structures such as cyclic data structure. This is not a serious drawback as our goal is to approximate the number of instances, which is preserved by the representation of objects by terms.

Example 3.3.10 ([MP08a])**.** *Consider the method* `setTail` *of Example 3.3.8 and define*

$$[\mathtt{null}]_{\mathbb{R}^+} = 0, \qquad [\mathtt{LL}]_{\mathbb{R}^+}(X, Y, W, Z) = X + Y + W + Z + 1,$$
$$[\mathtt{setTail}]_{\mathbb{R}^+}(Y, Z) = Y + Z.$$

Consider a store σ *such that* $\mathtt{x}\sigma = \mathtt{new\ LL}(\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d})$ *and* $\mathtt{y}\sigma = \mathtt{l}$*.* $(\mathtt{x.setTail}(\mathtt{y}), \sigma) \downarrow (\mathtt{a}, \sigma\{\mathtt{x} \leftarrow \mathtt{new\ LL}(\mathtt{a}, \mathtt{l}, \mathtt{c}, \mathtt{d})\})$ *holds and we check that:*

$$[\mathtt{setTail}]_{\mathbb{R}^+}([\mathtt{l}]_{\mathbb{R}^+}, [\mathtt{new\ LL}(\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d})]_{\mathbb{R}^+}) = [\mathtt{l}]_{\mathbb{R}^+} + [\mathtt{new\ LL}(\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d})]_{\mathbb{R}^+},$$
$$\geq \max([\mathtt{a}]_{\mathbb{R}^+}, [\mathtt{new\ LL}(\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d})]_{\mathbb{R}^+}.$$

Consequently, it defines a (partial, as we do not provide the full assignment) sup-interpretation.

Now we can develop a criterion on OO programs. For that purpose, we need to introduce a notion of weight.

Definition 3.3.6. *Given a program having a main class with* n *attributes* $\mathtt{X}_1, \ldots, \mathtt{X}_n$*, a* loop command *is a loop that does not contain any while loop. A* while command *is either a while loop or a loop that contains at least one while loop. A weight* ω *associates to each loop command* $\mathtt{I}$*, a total, monotonic, and subterm function* $\omega(\mathtt{I})$*. A weight is max-polynomial if for every loop command* $\mathtt{I}$*,* $\omega(\mathtt{I})$ *is a max-polynomial.*

Following [MP08a], we now define a criterion called brotherly ensuring that the size of the objects held by the attributes remains polynomially bounded within loops and while loops.

Definition 3.3.7. *A program having a main class with n attributes $\mathtt{X}_1, \ldots, \mathtt{X}_n$ is* brotherly *if there are a total, polynomial, and additive sup-interpretation $[-]_{\mathbb{R}^+}$ and a (max-)polynomial weight ω such that*

- *for each loop command* `I` *of the main class:*
  - *for every method call $a = \mathtt{X}_j.\mathtt{m}(\mathtt{e}_1, \ldots, e_m)$ in* `I`*:*

$$\omega(\mathtt{I})(T+1, [\mathtt{X}_1]_{\mathbb{R}^+}, \ldots, [\mathtt{X}_n]_{\mathbb{R}^+}) \geq \omega(\mathtt{I})(T, [\mathtt{X}_1]_{\mathbb{R}^+}, \ldots, [\mathtt{X}_{j-1}]_{\mathbb{R}^+}, [a]_{\mathbb{R}^+}, [\mathtt{X}_{j+1}]_{\mathbb{R}^+}, [\mathtt{X}_n]_{\mathbb{R}^+}),$$

    with T a fresh variable.
  - *for every assignment $\mathtt{X}_j := \mathtt{e}$; in* `I`*:*

$$\omega(\mathtt{I})(T+1, [\mathtt{X}_1]_{\mathbb{R}^+}, \ldots, [\mathtt{X}_n]_{\mathbb{R}^+}) \geq \omega(\mathtt{I})(T, [\mathtt{X}_1]_{\mathbb{R}^+}, \ldots, [\mathtt{X}_{j-1}]_{\mathbb{R}^+}, [\mathtt{e}]_{\mathbb{R}^+}, [\mathtt{X}_{j+1}]_{\mathbb{R}^+}, [\mathtt{X}_n]_{\mathbb{R}^+}),$$

    with T a fresh variable.
- *for each while command* `I` *of the main class and each variable assignment $\mathtt{X}_j := \mathtt{e}$; in* `I`*,*

$$max([\mathtt{X}_1]_{\mathbb{R}^+}, \ldots, [\mathtt{X}_n]_{\mathbb{R}^+}) \geq [\mathtt{e}]_{\mathbb{R}^+}.$$

We can now state soundness result on brotherly programs, stating that any object computed by such a program has size polynomially bounded in the input size.

Theorem 3.3.3 ([MP08a])**.** *Consider a brotherly program, whose main class is of the shape* `class main {Var` $\mathtt{X}_1$`; ...Var` $\mathtt{X}_n$`; I}`*, there exists a polynomial P such that for any store σ, if $(\mathtt{I}, \sigma) \downarrow \sigma'$ then*

$$P(|\mathtt{X}_1\sigma|, \ldots, |\mathtt{X}_n\sigma|) \geq \max_{i=1}^{n} |\mathtt{X}_i\sigma'|$$

.

Example 3.3.11 ([MP08a])**.** *Consider the following program that makes use of the class of Example 3.3.8.*

```
     class main {
          Var X; Var Y; Var Z;
I :       loop(Z){
               X := Y.reverse();
          }
          Y := X.setTail(Z);
     }
```

`I` *is the only loop command and there is no while command. Consequently, for the program to be brotherly, we have to find a polynomial weight ω and a polynomial and additive sup-interpretation $[-]_{\mathbb{R}^+}$ such that:*

$$\omega(\mathtt{I})(T+1, [\mathtt{X}]_{\mathbb{R}^+}, [\mathtt{Y}]_{\mathbb{R}^+}, [\mathtt{Z}]_{\mathbb{R}^+}) \geq \omega(\mathtt{I})(T, [\mathtt{X}]_{\mathbb{R}^+}, [\mathtt{Y.reverse()}]_{\mathbb{R}^+}, [\mathtt{Z}]_{\mathbb{R}^+}),$$
$$\omega(\mathtt{I})(T+1, [\mathtt{X}]_{\mathbb{R}^+}, [\mathtt{Y}]_{\mathbb{R}^+}, [\mathtt{Z}]_{\mathbb{R}^+}) \geq \omega(\mathtt{I})(T, [\mathtt{Y.reverse()}]_{\mathbb{R}^+}, [\mathtt{Y}]_{\mathbb{R}^+}, [\mathtt{Z}]_{\mathbb{R}^+}).$$

We let the reader check that the assignment $[-]_{\mathbb{R}^+}$ defined by $[\mathtt{reverse}]_{\mathbb{R}^+}(Y) = Y$ and extended with the assignment of Example 3.3.10 defines a total (i.e. defined for every method symbol), polynomial, and additive sup-interpretation. Moreover, setting $\omega(\mathtt{I})(T, X, Y, Z) = Y \times T + X + Y + Z$, we obtain that this program is brotherly as the above inequalities are satisfied. Consequently, by Theorem 3.3.3, all the computed objects are polynomially bounded in the input size.

This subsection has demonstrated that interpretation methods can be gently adapted to OO programs. However the extension has a weak expressive power with respect to the tier based methods of Section 3.1 due to the fact that while loops are treated as non-size increasing computations (see [Hof99, Hof02]) in the definition of the brotherly criterion. This explains partly the reason why most of the advances on OO programs have been obtained using the tier-based techniques of Section 3.1.

3.3.4 Miscellaneous

In this subsection, we discuss other extensions of interpretations to bytecode, concurrent systems, and polygraphs.

Bytecode

The paper [ACGDZJ04] has studied a setting where functional programs in pre-compiled form are exchanged by untrusted parties. It provides code annotations to guarantee type, size, and termination properties at the bytecode level on a simple stack machine. The size upper-bounds are obtained through the use of (polynomial) quasi-interpretations. They basically provide upper-bounds of the stack frame size of the interpreted bytecode.

Concurrent systems

The notion of quasi-interpretation has also been extended in [ADZ04] to statically infer upper bounds on the resources needed for the execution of synchronous cooperative threads. It is combined with termination techniques to guarantee that each instant terminates and to infer a polynomial bound on the resources needed for the execution of the system during an instant in the size of the parameters at the beginning of the instant. This work has been extended in [AD06], obtaining a polynomial bound on the size of the parameters of the program for arbitrarily many instants. It has been extended to ensure feasible reactivity in the framework of the synchronous pi-calculus in [AD06].

Polygraphs

In [BG08, BG07], Bonfante and Guiraud have adapted the interpretation methodology to the context of 3-polygraphs, a family of rewriting systems on circuits with sequential and parallel compositions that can be viewed as a fragment of graph rewriting. They define a notion of *polygraphic program* as a particular subset of 3-polygraphs whose circuit gates (called 2-cells) can be decomposed into function symbols, constructor symbols, and some structural rules and whose rewriting rules (called 3-cells) can be decomposed into computations and structural rules. The structural rules are fully described in [Bon11].

The notion of functorial interpretation is defined on circuits by assigning a non-empty subset of the positive natural numbers to any wire, a weakly monotonic function over the interpretation of wires to each gate of a circuit (sequential composition is interpreted as functional composition and parallel composition as Cartesian product), and by requiring for each rewrite rule that the interpretation of the left-hand side (circuit) is greater or equal to the interpretation of the right-hand side (circuit).

As the inequality is non-strict and also for structural reasons, some further requirements are added, leading to the notion of *Cartesian, conservative and 3-cells-compatible, polygraphic interpretation*. The following alternative characterization of `FP` is obtained.

Theorem 3.3.4 ([Bon11]). *The set of functions computed by confluent and complete polygraph programs having an additive, polynomial, Cartesian, conservative, and 3-cells-compatible polygraphic interpretation is exactly* `FP`.

This result is interesting from a theoretical perspective as it shows that interpretations can be adapted fruitfully to computational models based on graph rewriting.

3.4 Extensions of other techniques

In this chapter, we have discussed and studied extensions of the main ICC tools (tiering, soft/light logics, and interpretations) to several programming paradigms (imperative, concurrent, OO, ...). We now briefly discuss extensions of other techniques.

An attempt to extend the matrix based typing analysis of imperative programs à la Jones and Kristiansen to an imperative programming language with higher-order functions was developed in [AKM09]. However the expected results are left as open conjectures.

The framework of automatic amortized resource analysis initiated by [HJ03] for first order functional programs and briefly discussed in Section 1.2.5 has been extended to parallel first order functional programs [HS15], lazy functional languages with coinductive data [VJFH15], C programs [CHS15], and probabilistic programs [NCH18]. Though not directly related, the work of [BHMS04] has also provided an assertion format for automatic certification of heap consumption in a low level language.

Several extensions of the COSTA tool discussed in Subsection 1.2.6 have also been considered, non exhaustively, to non cumulative programming languages with garbage collection [AFRD15, AGGZ13], OO programs [AAG+12], real-time Java [KSvG+12], and concurrent and distributed programs [ACJ+18].

Chapter 4

Extensions to infinite data types

Contents

Another line of research in ICC was to try to extend the language with new features and constructs. An extension of interest is the ability to add coinductive data such as infinite lists or infinite trees. The introduction of possibly infinite data raises the difficulty to adapt the corresponding ICC tools that are most of the time based on well-founded properties. Consequently, quasi-interpretations can be viewed a natural candidate to overcome this problem.

Coinductive properties of programming languages and, more generally coinduction, have been deeply studied from a categorical point of view [JR97]. In general, in order to generate and manipulate coinductive data, one typically uses a programming language based on corecursive functions in conjunction with a lazy evaluation strategy. Streams are one kind of coinductive data of interest that allows to represent infinite lists. A main property of streams is the notion of *productivity* [Dij80]. A stream program is productive if it can be evaluated continuously in such a

way that a uniquely determined stream is obtained as the limit. This property is undecidable and several works have been carried out to enforce stream productivity, *e.g.* [Sij89, Coq94, HPS96, EGH+10, AM13a, CBGB15, Sev17].

In this section, we are interested in complexity properties of streams and, hence, we will assume to be in a productive setting. The complexity properties we will focus on are quantitative properties such as upper bounds on the number of reduction steps needed to produce the n-th stream element, upper bounds on the size of the output stream elements, or upper bounds on the number of stream inputs needed to produce one stream output element. Section 4.1 will be devoted to prove such properties on first order programs on streams using interpretation methodology as in [GP09a, GP09b, GP15b]. After introducing a first order language on streams in Subsection 4.1.1, we adapt the notion of quasi-interpretation to this setting in Subsection 4.1.2. Criteria to infer upper bounds and bounded input/output properties are studied in Subsection 4.1.3 and Subsection 4.1.4, respectively.

As a stream of integers can be seen as a sequence of integers, which is a basic way to encode real numbers, *e.g.* Cauchy sequences, an extension of ICC tools to streams could also been adapted to study the complexity programs computing over real numbers. Moreover, as a sequence of integers can also be viewed as a function mapping a natural number, the index, to an integer, these extensions could also naturally be considered to study higher-order complexity classes. These two points will be discussed in Section 4.2. After a brief reminder on type 2 complexity in Subsection 4.2.1, Subsection 4.2.2 provides a stream based characterization of the complexity class $\mathtt{BFF}_2$. Subsection 4.2.3 provides a stream based characterization of $\mathtt{FP}(\mathbb{R})$, the class of functions computable polynomial time over the real numbers. Finally, Subsection 4.2.4 presents an extension of these results which characterizes (discrete) polynomial time at any order.

Section 4.3 will be devoted to ensure polynomial time normalization properties on a lambda calculus augmented with inductive and coinductive datatypes as in [GP15a]. In other words, we show how to extend the expressive power of polynomial time programming languages based on LAL to coinductive data, without breaking soundness. Some reminders on algebra and coalgebra in System F are presented in Subsection 4.3.2. Algebra in LAL are studied in Subsection 4.3.3 and coalgebra in LAL are studied in Subsection 4.3.4.

Finally, in Section 4.4, we discuss other ICC results obtained on stream languages and on real numbers complexity classes. We study the result obtained by Mazza on a parsimonious stream language characterizing the complexity classes `P/poly` and `L/poly` using an affine based type system in Subsection 4.4.1. In Subsection 4.4.2, we study a function algebra based characterization of $\mathtt{FP}(\mathbb{R})$. Finally, we discuss in Subsection 4.4.3, a function algebra based characterization of the class $\mathtt{FP}_{\mathbb{R}}$, an alternative notion of polynomial time over the reals based on the Blum, Shub, and Smale (BSS) computational model [BSS88].

4.1 Streams and quasi-interpretations

The papers [GP09a, GP09b, GP15b] consider a simple first order lazy language and study two classes of *space properties* of programs working on streams by means of interpretation methods.

- *Stream Upper Bounds*: these are properties about the size of each stream element produced by a program.
- *Bounded Input/Output Properties*: these are properties about the number of stream elements produced by a program.

These properties analyze two *dimensions* of programs working on streams. The combination of these properties allows one to study a reasonable class of programs and to obtain the information needed in order to improve the memory management process (in terms of bufferization) of programs working on streams.

4.1.1 A first order lazy stream programming language

The language under consideration is a first order lazy variant of the higher-order functional languages described in Subsection 3.3.2:

$$\mathtt{M},\mathtt{N} ::= \mathtt{x} \mid \mathtt{c} \mid \mathtt{M}\ \mathtt{N} \mid \mathtt{letRec}\ \mathtt{f} = \lambda \mathtt{x}_1.\cdots\lambda \mathtt{x}_{ar(\mathtt{f})}.\mathtt{M} \mid \mathtt{case}\ \mathtt{M}\ \mathtt{of}\ \mathtt{c}_1\ \mathtt{x}_1 : \mathtt{M}_1 | ... | \mathtt{c}_n\ \mathtt{x}_n : \mathtt{M}_n.$$

More precisely, the use of lambda abstraction is restricted to function definition of the shape $\mathtt{letRec}\ \mathtt{f} = \lambda \mathtt{x}_1.....\lambda \mathtt{x}_{ar(\mathtt{f})}.\mathtt{M}$, each function variable $\mathtt{f}$ declared through a $\mathtt{letRec}$ being equipped with a fixed arity $ar(\mathtt{f})$. Moreover, there is no partial application, *i.e.* each function application is of the shape $(\mathtt{letRec}\ \mathtt{f} = \mathtt{M})\ \mathtt{M}_1\ \cdots\ \mathtt{M}_{ar(\mathtt{f})}$. In what follows, let $\mathtt{p}$ denote a pattern, *i.e.* $\mathtt{p} = \mathtt{c}\ \mathtt{x}_1\ \ldots\ \mathtt{x}_{ar(\mathtt{c})}$, for some constructor symbol $\mathtt{c}$ and some variables $\mathtt{x}_1, \ldots, \mathtt{x}_{ar(\mathtt{c})}$. The language is also equipped with a type system, that can be found in [GP15b], to ensure that programs do not go wrong. Let $\mathtt{Nat}$ be the type of unary numbers, α be a type variable, and $[\alpha]$ be the type of streams whose elements are of type α. As usual, $\underline{\mathtt{n}} : \mathtt{Nat}$ represents the unary number n. Streams correspond to both finite and infinite lists and are encoded using the constructor symbols $\mathtt{nil} : [\alpha]$ and $\mathtt{cons} : \alpha \to [\alpha] \to [\alpha]$.

Let also $\bot$ be a special constructor symbol of zero arity representing errors.

We will write $\mathtt{f}\ \mathtt{x}_1\ \cdots\ \mathtt{x}_n = \mathtt{M}$ for $\mathtt{letRec}\ \mathtt{f} = \lambda \mathtt{x}_1.\cdots\lambda \mathtt{x}_n.\mathtt{M}$. We define a lazy operational semantics in Figure 4.1 for the language based on the notion of lazy values. A lazy value is just a term whose top most symbol is a constructor symbol

$$\mathtt{lv} := \mathtt{c}\ \mathtt{M}_1\ \cdots\ \mathtt{M}_{ar(\mathtt{c})}.$$

As in previous sections, a (strict) value $\mathtt{v}$ is a term that only contains constructor symbols. A term $\mathtt{M}$ evaluates to the lazy value $\mathtt{lv}$ whenever $\mathtt{M} \Downarrow \mathtt{lv}$. If $\mathtt{M} \Downarrow \mathtt{lv}$ using k pattern matching rules over lists (*i.e.* k times a rule (pm) with a conclusion of the shape $\mathtt{case}\ \mathtt{M}\ \mathtt{of}\ \mathtt{nil} : \mathtt{M}_1 | \mathtt{cons}\ \mathtt{x}\ \mathtt{y} : \mathtt{M}_2$) then we write $\mathtt{M} \Downarrow^k \mathtt{lv}$. A term $\mathtt{M}$ evaluates to the value $\mathtt{v}$, noted $\mathtt{M} \Downarrow_v \mathtt{v}$, if the strict evaluation of the term $\mathtt{M}$ is equal to the value $\mathtt{v}$. In other words, $\mathtt{eval}\ \mathtt{M} \Downarrow \mathtt{v}$ where $\mathtt{eval}$ is defined for any type α by:

$$\mathtt{eval} : \alpha \to \alpha,$$
$$\mathtt{eval}\ \ (\mathtt{c}\ \mathtt{x}_1 \cdots\ \mathtt{x}_n) = \hat{\mathtt{C}}\ (\mathtt{eval}\ \mathtt{x}_1)\ \cdots\ (\mathtt{eval}\ \mathtt{x}_n),$$

where $\hat{\mathtt{C}}$ is a function symbol representing the *strict* version of the primitive constructor $\mathtt{c}$. For instance in the case where $\mathtt{c}$ is $+1$ we can define $\hat{\mathtt{C}}$ to be the function $\mathtt{succ} :: \mathtt{Nat} \to \mathtt{Nat}$ defined as:

$$\begin{array}{lcl} \mathtt{succ}\quad \mathtt{0} & = & \mathtt{0{+}1}, \\ \mathtt{succ}\ \ (\mathtt{x{+}1}) & = & (\mathtt{x{+}1){+}1}. \end{array}$$

We also write $\mathtt{M} \Downarrow_v^k \mathtt{v}$ whenever $\mathtt{eval}\ \mathtt{M} \Downarrow^k \mathtt{v}$.

Throughout this section, we will use the notation

$$\begin{array}{l} \mathtt{f}\ \mathtt{p}_1^1\ \cdots\ \mathtt{p}_n^1 = \mathtt{M}_1 \\ \vdots \\ \mathtt{f}\ \mathtt{p}_1^k\ \cdots\ \mathtt{p}_n^k = \mathtt{M}_k \end{array}$$

$$\frac{\mathtt{c} \in \mathcal{C}}{\mathtt{c}(\mathtt{M}_1, \ldots, \mathtt{M}_n) \Downarrow \mathtt{c}(\mathtt{M}_1, \ldots, \mathtt{M}_n)}\ (\text{val}) \qquad \frac{\mathtt{M}\{(\mathtt{f}\ \mathtt{x}_1\ \cdots\ \mathtt{x}_n = \mathtt{M})/\mathtt{f}, \mathtt{M}_1/\mathtt{x}_1, \ldots, \mathtt{M}_n/\mathtt{x}_n\} \Downarrow \mathtt{lv}}{(\mathtt{f}\ \mathtt{x}_1\ \cdots\ \mathtt{x}_n = \mathtt{M})\ \mathtt{M}_1 \cdots \mathtt{M}_n \Downarrow \mathtt{lv}}\ (\text{fun})$$

$$\frac{\mathtt{M} \Downarrow \mathtt{c}(\mathtt{M}'_1, \ldots, \mathtt{M}'_m) \qquad \mathtt{p}_i = \mathtt{c}(\mathtt{x}_1, \ldots, \mathtt{x}_m) \qquad \mathtt{M}_i\{\mathtt{M}'_1/\mathtt{x}_1, \cdots, \mathtt{M}'_m/\mathtt{x}_m\} \Downarrow \mathtt{lv}}{\mathtt{case}\ \mathtt{M}\ \mathtt{of}\ \mathtt{p}_1 : \mathtt{M}_1 | \ldots | \mathtt{p}_n : \mathtt{M}_n \Downarrow \mathtt{lv}}\ (\text{pm})$$

Figure 4.1: First order lazy operational semantics

as syntactic sugar for a term of the shape

$$\begin{aligned} \mathtt{letRec}\ \mathtt{f} = \lambda \mathtt{x}_1. \cdots \lambda \mathtt{x}_n. \mathtt{case}\ \mathtt{x}_1\ \mathtt{of}\ \mathtt{p}_1^1 : \cdots\ \mathtt{case}\ \mathtt{x}_n\ \mathtt{of}\ \mathtt{p}_n^1 : \mathtt{M}_1 \\ \vdots \\ | \mathtt{p}_1^k : \cdots\ \mathtt{case}\ \mathtt{x}_n\ \mathtt{of}\ \mathtt{p}_n^k : \mathtt{M}_k. \end{aligned}$$

We will call each $\mathtt{f}\ \mathtt{p}_1^i\ \cdots\ \mathtt{p}_n^i = \mathtt{M}_i$ a *definition*. Similarly to the case of TRSs, we will call any variable $\mathtt{f}$ used in such a function definition a *function symbol*. In what follows, let $\mathcal{C}$ be the set of constructor symbols distinct from `cons`, let $\mathcal{F}$ be the set of function symbols, and let $\mathbb{V}$ be the set of variables that are not function symbols.

We define some basic functions over streams. !! is the usual indexing function (used in prefix notation) returning the n-th element of a list.

$$\begin{aligned} &!! : [\alpha] \to \mathtt{Nat} \to \alpha \\ &!!\ \mathtt{nil}\ \mathtt{y} = \bot \\ &!!\ (\mathtt{cons}\ \mathtt{x}\ \mathtt{xs})\ \mathtt{0} = \mathtt{x} \\ &!!\ (\mathtt{cons}\ \mathtt{x}\ \mathtt{xs})\ (\mathtt{y{+}1}) = !!\ \mathtt{xs}\ \mathtt{y} \end{aligned}$$

`take` is the usual function which returns the first n elements of a list.

$$\begin{aligned} &\mathtt{take} : \mathtt{Nat} \to [\alpha] \to [\alpha] \\ &\mathtt{take}\ 0\ \mathtt{s} = \mathtt{nil} \\ &\mathtt{take}\ (\mathtt{x{+}1})\ \mathtt{nil} = \mathtt{nil} \\ &\mathtt{take}\ (\mathtt{x{+}1})\ (\mathtt{cons}\ \mathtt{y}\ \mathtt{ys}) = \mathtt{cons}\ \mathtt{y}\ (\mathtt{take}\ \mathtt{x}\ \mathtt{ys}) \end{aligned}$$

Finally, `lg` denotes the function that returns the number of elements in a finite partial list.

$$\begin{aligned} &\mathtt{lg} : [\alpha] \to \mathtt{Nat} \\ &\mathtt{lg}\ \ \mathtt{nil} = \mathtt{0} \\ &\mathtt{lg}\ \ \bot = \mathtt{0} \\ &\mathtt{lg}\ \ (\mathtt{cons}\ \mathtt{x}\ \mathtt{xs}) = (\mathtt{lg}\ \mathtt{xs}){+}\mathtt{1} \end{aligned}$$

4.1.2 Parameterized quasi-interpretations

We are now ready to adapt the notion of quasi-interpretation described in Section 2.3 to first order programs on streams. For that purpose, we need to introduce a notion of parameterized quasi-interpretation to be able to deal with local properties of streams such as local upper bounds, properties relating the size of the stream output in a term with respect to its index. Here the parameter is used to take the index into account. Notice that it could be seen as a quasi-interpretation variant and a first order restriction of the higher-order interpretations of Subsection 3.3.2 extended by a parameter.

Definition 4.1.1 (Parameterized assignment)**.** *Given a term* $\mathtt{M}$*, a parameterized assignment* $[-]^l_{\mathbb{R}^+}$ *over* $\mathbb{R}^+$ *maps:*

- *every variable* $\mathtt{x} \in \mathbb{V}$ *to a variable* $[\mathtt{x}]^l_{\mathbb{R}^+}$ *in* $\mathbb{R}^+$*,*
- *every symbol* $\mathtt{b} \in \mathcal{C} \uplus \mathcal{F}$ *to a total function* $[\mathtt{b}]^{\mathtt{l}}_{\mathbb{R}^+} : \mathbb{R}^{+\mathtt{ar(b)}} \to \mathbb{R}^+$*.*

and is extended to the language constructs as follow:

- $[\mathtt{M}\ \mathtt{N}]^l_{\mathbb{R}^+} := [\mathtt{M}]^l_{\mathbb{R}^+}\ [\mathtt{N}]^l_{\mathbb{R}^+}$, $\mathtt{M} \neq \mathtt{cons}$,
- $[\mathtt{cons}\ \mathtt{M}_1\ \mathtt{M}_2]^l_{\mathbb{R}^+} := [\mathtt{cons}]^l_{\mathbb{R}^+}\ [\mathtt{M}_1]^l_{\mathbb{R}^+}\ [\mathtt{M}_2]^{l-1}_{\mathbb{R}^+}$,
- $[\mathtt{letRec}\ \mathtt{f} = \lambda \mathtt{x}_1.\cdots\lambda \mathtt{x}_{ar(\mathtt{f})}.\mathtt{M}]_{\mathbb{R}^+} := [\mathtt{f}]^l_{\mathbb{R}^+}$,
- $[\mathtt{case}\ \mathtt{M}\ \mathtt{of}\ \mathtt{p}_1 : \mathtt{M}_1 | \ldots | \mathtt{p}_m : \mathtt{M}_m]^l_{\mathbb{R}^+} := \max_{1\leq i\leq m}\{[\mathtt{M}_i]^l_{\mathbb{R}^+} \mid [\mathtt{M}]^l_{\mathbb{R}^+} \geq [\mathtt{p}_i]^l_{\mathbb{R}^+}\}$.

A parameterized assignment is monotonic if it is monotonic in each argument and in the parameter. The notions of additive and polynomial parameterized assignment are defined in a standard way as in Subsection 2.2.2. We set all the additive constant to 1 throughout the following Subsection. A parameterized assignment is *almost-additive* if it is additive for any constructor symbol distinct from the stream constructor symbol $\mathtt{cons}$.

Definition 4.1.2 (Parameterized quasi-interpretation)**.** *A term* $\mathtt{M}$ *admits a* parameterized quasi-interpretation $[\mathtt{M}]^l_{\mathbb{R}^+}$ *if the parameterized assignment* $[-]^l_{\mathbb{R}^+}$ *is monotonic and for each function definition* $\mathtt{f}\ \mathtt{p}_1\ \cdots\ \mathtt{p}_n = \mathtt{N}$ *in* $\mathtt{M}$*, the following holds:*

$$[\mathtt{f}\ \mathtt{p}_1\ \cdots\ \mathtt{p}_n]^l_{\mathbb{R}^+} \geq [\mathtt{N}]^l_{\mathbb{R}^+}.$$

In the particular case where $[-]^l_{\mathbb{R}^+}$ *does not depend on the parameter then it is called a standard quasi-interpretation, noted* $[-]_{\mathbb{R}^+}$*.*

A result similar to Lemma 3.3.2, holds.

Lemma 4.1.1 ([GP15b])**.** *Given a term* $\mathtt{M}$ *admitting the parameterized interpretation* $[\mathtt{M}]^l_{\mathbb{R}^+}$*, if* $\mathtt{M} \Downarrow \mathtt{lv}$ *then* $[\mathtt{M}]^l_{\mathbb{R}^+} \geq [\mathtt{lv}]^l_{\mathbb{R}^+}$*. Moreover if* $\mathtt{M} \Downarrow^k \mathtt{lv}$ *then* $[\mathtt{M}]^l_{\mathbb{R}^+} \geq [\mathtt{lv}]^{l-k}_{\mathbb{R}^+}$*.*

4.1.3 Stream upper bounds

In order to process stream data in a memory-efficient way it is useful to obtain an estimate of the memory needed to store the elements produced by a stream program.

In some situations, an estimate can be obtained by considering in a global way the greatest size of the elements produced by the program as outputs. In other situations, however, there is no such maximal element with respect to the size measure. As a consequence, only an estimate considering the local position of the element in the stream can be given.

Example 4.1.1. *Consider the following stream definitions.*

```
ones : [Nat]                    nats : Nat → [Nat]
ones = cons 1 ones              nats x = cons x (nats (x+1))
```

Indeed, in the stream definition of `ones` *all the elements have the same size, while in the definition of* `nats` *every element has a size depending on its position in the stream.* `ones` *has a* global upper bound *whereas for any* $\underline{\mathtt{n}}$, `nats` $\underline{\mathtt{n}}$ *has a* local upper bound.

We provide a formal definition of those properties in the following definition.

Definition 4.1.3 (Local and Global Upper Bounds)**.** *A stream program* $\mathtt{M} : [\alpha]$ *has a* local upper bound *if there is a function* $F \in \mathbb{R}^+ \to \mathbb{R}^+$ *such that*

$$\forall \underline{\mathtt{n}},\ \text{if } !!\ \mathtt{M}\ \underline{\mathtt{n}} \Downarrow_v \mathtt{v} \text{ then } F(|\underline{\mathtt{n}}|) \geq |\mathtt{v}|$$

If the function F *is constant, then* $\mathtt{M} : [\alpha]$ *has also a* global upper bound.

Definition 4.1.4 (Linear program)**.** *Let* $\downarrow^k$ *and* $\downarrow^k_v$ *be the relations defined by* $\mathtt{M} \downarrow^k \mathtt{lv}$ *if there exists* $k' \leq k$ *such that* $\mathtt{M} \Downarrow^{k'} \mathtt{lv}$ *and* $\mathtt{M} \downarrow^k_v \mathtt{v}$ *if there exists* $k' \leq k$ *such that* $\mathtt{M} \Downarrow^{k'}_v \mathtt{v}$*, respectively. A program* $\mathtt{M} : [\alpha]$ *is linear if there is a* $k \geq 1$ *such that for all* $\underline{\mathtt{n}} \in \mathtt{Nat}$, $!!\ \mathtt{M}\ \underline{\mathtt{n}} \downarrow_v^{k\times(|\underline{\mathtt{n}}|+1)} \mathtt{v}$ *holds.*

Now we are ready to define two criteria for ensuring the two upper bounds.

Definition 4.1.5 (LUB)**.** *A program* $\mathtt{M} : [\alpha]$ *is LUB if it is linear and it admits a parameterized quasi-interpretation* $[-]^l_{\mathbb{R}^+}$ *that is almost-additive and such that*

$$[\mathtt{cons}]^l_{\mathbb{R}^+}(X, Y) = \max(X, Y).$$

Definition 4.1.6 (GUB)**.** *A program* $\mathtt{M} : [\alpha]$ *is GUB if it admits a standard quasi-interpretation* $[-]_{\mathbb{R}^+}$ *that is almost-additive and such that*

$$[\mathtt{cons}]_{\mathbb{R}^+}(X, Y) = \max(X, Y).$$

Theorem 4.1.1 ([GP15b])**.** *If a program is* LUB *(*GUB*) then it admits a local (global) upper bound.*

Example 4.1.2. *Consider the stream definition of* `nats` *of Example 4.1.1. We want to show that* `nats` *is LUB. First, notice that the program* `nats` *is linear with linearity constant* $k = 1$*. Indeed, the definition of* `nats` *does not involve pattern matching on stream data so the only possible pattern matching corresponds to the* $!!$ *definition where one read is needed to produce one output. Now, consider the parameterized quasi-interpretation* $[-]^l_{\mathbb{R}^+}$ *defined by:* $[\mathtt{nats}]^l_{\mathbb{R}^+}(X) = X + l$*,* $[+1]^l_{\mathbb{R}^+}(X) = X + 1$*,* $[0]^l_{\mathbb{R}^+} = 0$*, and* $[\mathtt{cons}]^l_{\mathbb{R}^+}(X, Y) = \max(X, Y)$*. We check that* $\forall l \in \mathbb{R}$*:*

$$\begin{aligned}
[\mathtt{nats\ x}]^l_{\mathbb{R}^+} &= [\mathtt{nats}]^l_{\mathbb{R}^+}\ [\mathtt{x}]^l_{\mathbb{R}^+} = [\mathtt{x}]^l_{\mathbb{R}^+} + l \\
&\geq \max([\mathtt{x}]^l_{\mathbb{R}^+}, ([\mathtt{x}]^l_{\mathbb{R}^+} + 1) + (l - 1)) \\
&\geq \max([\mathtt{x}]^l_{\mathbb{R}^+}, [\mathtt{nats(x+1)}]^{l-1}_{\mathbb{R}^+}) \\
&\geq [\mathtt{cons\ x\ (nats\ (x+1))}]^l_{\mathbb{R}^+}.
\end{aligned}$$

The quasi-interpretation $[-]^l_{\mathbb{R}^+}$ *clearly respects the required criterion for* `nats` *to be LUB: it is monotonic and almost-additive as* $[\mathtt{cons}]^l_{\mathbb{R}^+}(X, Y) = \max(X, Y)$*.*
So, `nats` *admits a local upper bound. We obtain the required bound by setting* $F(X) = [\mathtt{nats}(\underline{\mathtt{m}})]^X_{\mathbb{R}^+} = X + [\underline{\mathtt{m}}]^X_{\mathbb{R}^+} = X + |\underline{\mathtt{m}}|$*. For each unary numbers* $\underline{\mathtt{m}}$*,* $\underline{\mathtt{n}} \in \mathtt{Nat}$ *such that* $\mathtt{eval}((\mathtt{nats}\ \underline{\mathtt{m}})\ !!\ \underline{\mathtt{n}}) \Downarrow_v \mathtt{v}_{\underline{\mathtt{n}}}$*, the following holds* $F(|\underline{\mathtt{n}}|) \geq |\underline{\mathtt{n}}| + |\underline{\mathtt{m}}| \geq |\mathtt{v}_{\underline{\mathtt{n}}}|$*. Indeed for all* $\underline{\mathtt{n}}$*,* $\mathtt{v}_{\underline{\mathtt{n}}} = \underline{\mathtt{m} + \mathtt{n}}$ *and* $|\underline{\mathtt{m} + \mathtt{n}}| = |\underline{\mathtt{n}}| + |\underline{\mathtt{m}}|$*.*

Example 4.1.3. *Consider expressions built using the following function definitions.*

```
repeat : Nat → [Nat]
repeat x  = cons x (repeat x)
zip : [α] → [α] → [α]
zip (cons x xs) ys  =  cons x (zip ys xs)
square : [Nat] → [Nat]
square (cons x xs) =  cons (mul x x) (square xs)
mul : Nat → Nat → Nat
mul (x+1) y =  add y (mul x y)
mul 0 y =  0
```

Each program will only produce output stream elements whose size is bounded by some constant k. For instance, the program

$$\texttt{square (zip (repeat } \underline{\texttt{5}}\texttt{) (square (zip (repeat } \underline{\texttt{7}}\texttt{) (repeat } \underline{\texttt{4}}\texttt{))))}$$

computes stream elements whose size is bounded by $k = 2401 = 7^4$. So, its global upper bound is given by the constant $k = 2401$. Now, consider the following generalization of the above program

$$\texttt{square (zip (repeat } \underline{\texttt{5}}\texttt{) (square (zip x (repeat } \underline{\texttt{4}}\texttt{)))).}$$

It is easy to verify that when `x` *is substituted by a stream* `s` *having element sizes bounded by a constant k, then the output stream will have only elements whose sizes are bounded either by 4^4 or by k^4. So this expression has a global upper bound given by the function F defined by $F(X) = \max(256, X^4)$.*

We can show that it is GUB with respect to the quasi-interpretation $[-]_{\mathbb{R}^+}$ defined by:

$$\begin{aligned}
[\texttt{0}]_{\mathbb{R}^+} &= 0,\\
[\texttt{+1}]_{\mathbb{R}^+}(X) &= X + 1,\\
[\texttt{add}]_{\mathbb{R}^+}(X, Y) &= X + Y,\\
[\texttt{zip}]_{\mathbb{R}^+}(X, Y) &= [\texttt{cons}]_{\mathbb{R}^+}(X, Y) = \max(X, Y),\\
[\texttt{square}]_{\mathbb{R}^+}(X) &= X^2,\\
[\texttt{mul}]_{\mathbb{R}^+}(X, Y) &= X \times Y,\\
[\texttt{repeat}]_{\mathbb{R}^+}(X) &= X.
\end{aligned}$$

Indeed, we can check the inequalities of the quasi-interpretation are satisfied for every definition and for every variable assignment.

4.1.4 Bounded input/output properties

Another information of interest in order to improve memory-efficiency is an estimate of the number of elements produced by a stream program when fed with only a restricted portion of the input stream.

In some situations, such an estimate can be obtained by considering only the *length* of the portion of the input stream. In other situations, however, this is not sufficient and so in order to obtain the estimate one needs to consider also the *size* of the elements in the portion.

Example 4.1.4. *Consider the following definitions*

$\texttt{merge} : [\alpha] \to [\alpha] \to [a \times a]$
$\texttt{merge}\ (\texttt{cons x xs})\ (\texttt{cons y ys}) = \texttt{cons}\ (\texttt{x}, \texttt{y})\ (\texttt{merge ys xs})$

$\texttt{dup} : [\alpha] \to [\alpha]$
$\texttt{dup}\ (\texttt{cons x xs}) = \texttt{cons x}\ (\texttt{cons x}\ (\texttt{dup xs}))$

Each stream expression built using only merge *and* dup *will only generate a number of output elements that depends on the number of input read elements. For example, the expression* dup (merge (dup s) (dup s)) *produces for each read element of the input stream* s *four elements of the type* $a \times a$*. In what follows, we will study a* Length Based I/O Upper Bound *criterion ensuring that the number* m *of output stream elements is bounded by a function in the number* n *of stream elements read on the input.*

There are stream functions that generate a number of output elements also depends on the size of the input read elements.

Example 4.1.5. *Consider the following definitions:*

$\texttt{app} : [\alpha] \to [\alpha] \to [\alpha]$
$\texttt{app}\ (\texttt{cons x xs})\ \texttt{ys} = \texttt{cons x}\ (\texttt{app xs ys})$
$\texttt{app nil ys} = \texttt{ys}$

$\texttt{upto} : \texttt{Nat} \to [\texttt{Nat}]$
$\texttt{upto 0} = \texttt{nil}$
$\texttt{upto}\ (\texttt{x+1}) = \texttt{cons}\ (\texttt{x+1})\ (\texttt{upto x})$

$\texttt{extendupto} : [\texttt{Nat}] \to [\texttt{Nat}]$
$\texttt{extendupto}\ (\texttt{cons x xs}) = \texttt{app}\ (\texttt{upto x})\ (\texttt{extendupto xs})$

Every stream expression built using only upto *and* extendupto *will generate a number of output elements that is related to both the number and the size of input read elements. For example, the expression* extendupto (extendupto s) *outputs* $\sum_{i=1}^{|\underline{\texttt{n}}|} i$ *elements, for each natural number* $\underline{\texttt{n}}$ *in the input stream* s*. In what follows, we will also study a* Size-Based I/O Upper Bound *criterion ensuring that the number* m *of output stream elements is bounded by a function in the number and the size of the stream elements in input.*

A stream function is a term M from stream to stream with one free variable. By abuse of notation, we denote such a term by $\lambda\texttt{x.M}$ to name explicitly the free variable x in M. Indeed, the use of lambda abstractions is restricted in our first order language.

Definition 4.1.7 (Length and size based I/O upper bounds)**.**

- *A stream function* $\lambda\texttt{x.M} : [\alpha] \to [\beta]$ *has a* length based I/O upper bound *if there is a function* $F : \mathbb{R}^+ \to \mathbb{R}^+$ *such that for every expression* $\texttt{s} : [\alpha]$*:*

$$\forall\ \underline{\texttt{n}} \in \texttt{Nat},\ \textit{if}\ \texttt{take}\ \underline{\texttt{n}}\ \texttt{s} \Downarrow_v \texttt{v}\ \ \textit{implies}\ \texttt{lg}\ \texttt{M}\{\texttt{v/x}\} \Downarrow_v \underline{\texttt{m}}\ \textit{then}\ F(|\underline{\texttt{n}}|) \geq |\underline{\texttt{m}}|.$$

- *A stream function* $\lambda\texttt{x.M} : [\alpha] \to [\beta]$ *has a* size based I/O upper bound *if there is a function* $F : \mathbb{R}^+ \to \mathbb{R}^+$ *such that for every expression* $\texttt{s} : [\alpha]$*:*

$$\forall\ \underline{\texttt{n}} \in \texttt{Nat},\ \textit{if}\ \texttt{take}\ \underline{\texttt{n}}\ \texttt{s} \Downarrow_v \texttt{v}\ \ \textit{implies}\ \texttt{lg}\ \texttt{M}\{\texttt{v/x}\} \Downarrow_v \underline{\texttt{m}}\ \textit{then}\ F(|\texttt{v}|) \geq |\underline{\texttt{m}}|).$$

We are now ready to define two restrictions on quasi-interpretations to ensure the above properties.

Definition 4.1.8 (LBUB and SBUB)**.** *A stream function is LBUB (respectively SBUB) if it admits an almost-additive (respectively additive) standard quasi-interpretation* $[-]_{\mathbb{R}^+}$ *such that* $[+1]_{\mathbb{R}^+}(X) = X + 1$ *and* $[\mathtt{cons}]_{\mathbb{R}^+}(X, Y) = Y + 1$ *(respectively* $[\mathtt{cons}]_{\mathbb{R}^+}(X, Y) = X + Y + 1$*).*

Theorem 4.1.2 ([GP15b])**.** *If a stream function* $\lambda \mathtt{x}.\mathtt{M}$ *is LBUB (SBUB, respectively) then it has a length (respectively size) based I/O upper bound.*

Example 4.1.6. *Consider again the stream definitions presented in Example 4.1.4:*

```
merge : [α] → [α] → [α]
merge (cons x xs) (cons y ys)  =  cons (x, y) (merge ys xs)

dup : [α] → [α]
dup (cons x xs) =  cons x (cons x (dup xs))
```

To verify that every expression built using these definitions has a length based I/O upper bound it is sufficient to verify that it is LBUB with respect to the following standard quasi-interpretation:

$$[\mathtt{merge}]_{\mathbb{R}^+}(X, Y) = \max(X, Y),\ [\mathtt{dup}]_{\mathbb{R}^+}(X) = 2X,\ \textit{and}\ [\mathtt{cons}]_{\mathbb{R}^+}(X, Y) = Y + 1.$$

As an example, for each `s` *we have:*

$$[\mathtt{dup}\ (\mathtt{merge}\ (\mathtt{dup}\ \mathtt{s})\ (\mathtt{dup}\ \mathtt{s}))]_{\mathbb{R}^+} = 4[\mathtt{s}]_{\mathbb{R}^+}$$

and this gives a length based I/O upper bound.

Example 4.1.7. *Consider again the stream definitions of Example 4.1.5:*

```
app : [α] → [α] → [α]
app (cons x xs) ys = cons x (app xs ys)
app nil ys = ys

upto : Nat → [Nat]
upto 0 = nil
upto (x+1) = cons (x+1) (upto x)

extendupto : [Nat] → [Nat]
extendupto (cons x xs) = app (upto x) (extendupto xs)
```

To show that every expression built using these definitions has a size based I/O upper bound it is enough to show that the program is SBUB with respect to the following standard quasi-interpretation:

$$[\mathtt{nil}]_{\mathbb{R}^+} = [\mathtt{0}]_{\mathbb{R}^+} = 0, \quad [+1]_{\mathbb{R}^+}(X) = X + 1, \quad [\mathtt{cons}]_{\mathbb{R}^+}(X, Y) = X + Y + 1,$$

$$[\mathtt{app}]_{\mathbb{R}^+}(X, Y) = X + Y, \quad [\mathtt{upto}]_{\mathbb{R}^+}(X) = [\mathtt{extendupto}]_{\mathbb{R}^+}(X) = 2 \times X^2.$$

In particular, by taking $F(X) = [\mathtt{extendupto}]_{\mathbb{R}^+}([\mathtt{extendupto}]_{\mathbb{R}^+}(X)) = 8 \times X^4$ *we obtain a size based I/O upper bound for the expression* `extendupto (extendupto s)`*. The function* F *also gives a bound also on the size of produced elements.*

4.2 Complexity class characterizations

In [FHHP10, FHHP15], second order interpretations were combined to a first order language on streams to characterize the class of type-2 Feasible Functionals, also known as *Basic Feasible*

Functionals $\mathtt{BFF}_2$, that is the type-2 counterpart of the class of polynomial time computable functions $\mathtt{FP}$.[27] In this Section, after recalling some basic background on type-2 feasibility in Subsection 4.2.1, we present the characterization of [FHHP10, FHHP15] in Subsection and extensions characterizing the polynomial time over the reals and at higher-orders in Subsection 4.2.3 and Subsection 4.2.4, respectively.

4.2.1 Type-2 feasible functionals

The notion of feasibility for type-2 functionals was first studied by Constable [Con73] and Mehlhorn [Meh76] using an explicit bound on the size of computations. Later, Cook and Kapron [CK90], Cook [Coo92], Cook and Urquhart [CU93] have provided alternative (non ICC) characterizations of this notion of type-2 polynomial time complexity based on programming languages with explicit bounds, machines, and recursion scheme, respectively.

Other (non ICC) characterizations of $\mathtt{BFF}_2$ have been provided in the literature. The characterizations of [IRK01] are based on a simple imperative programming language that enforces an explicit external bound on the size of oracle outputs within loops. Function algebra characterizations were developed in [KS19]. Several characterizations using type-2 polynomials [KC91, KC96] and type-1 polynomials with *bounded lookahead revisions* [KS18] were also developed using Oracle Turing Machines (OTMs).

In this section, we recall Kapron and Cook characterization of basic polynomial time functionals ($\mathtt{BFF}$) [KC96] based on he notion of OTM.

Definition 4.2.1 (Oracle Turing Machine)**.** *An Oracle Turing Machine $\mathcal{M}$ with k oracles (where oracles are functions from $\mathbb{N}$ to $\mathbb{N}$) and l input tapes is a TM with, for each oracle, a state, one query tape and one answer tape.*

If $\mathcal{M}$ is used with oracles $F_1, \dots F_k : \mathbb{N} \to \mathbb{N}$, then on the oracle state $i \in \{1, \dots, k\}$, $F_i(x)$ is written on the corresponding answer tape, whenever x is the content of the corresponding query tape.

Definition 4.2.2 (Size of a function)**.** *The size $|F| : \mathbb{N} \to \mathbb{N}$ of a function $F : \mathbb{N} \to \mathbb{N}$ is defined by:*

$$|F|(n) = \max_{k \leq n} |F(k)|$$

where $|F(k)|$ represents the size of the binary representation of $F(k)$.

Definition 4.2.3 (Second order polynomial)**.** *A second order polynomial is a polynomial generated by the following grammar:*

$$P := c \mid X \mid P + P \mid P \times P \mid Y(P),$$

where X represents a first order variable, Y a second order variable, and c a constant in $\mathbb{N}$.

The following example shows the implicit meaning of a substitution of a second order variable.

Example 4.2.1. *Given $P(Y, X) = Y(X^2 + 1)$ and $f : \mathbb{N} \to \mathbb{N}$ define by $f(X) = X^2$, we have $P(f, X) = f(X^2 + 1) = (X^2 + 1)^2$.*

In the following, $P(Y_1, \dots, Y_k, X_1, \dots, X_l)$ will denote a second order polynomial where each Y_i, $1 \leq i \leq k$, represents a second order variable, and each X_j, $1 \leq j \leq l$, a first order variable.

Definition 4.2.4 (Polynomial running time)**.** *The cost of a transition is:*

[27] This class is also called $\mathtt{BFF}$ for short when the type-2 setting is clear from the context.

- $|F|(|x|)$, *if the machine is in a query state of the oracle* F *on input query* x,
- 1 *otherwise.*

An OTM $\mathcal{M}$ *operates in time* $T : (\mathbb{N} \to \mathbb{N})^k \to \mathbb{N}^l \to \mathbb{N}$ *if for all inputs* $x_1, \dots x_l : \mathbb{N}$ *and* $F_1, \dots F_k : \mathbb{N} \to \mathbb{N}$, *the sum of the transition costs before* $\mathcal{M}$ *halts on these inputs is less than* $T(|F_1|, \dots, |F_k|, |x_1|, \dots, |x_l|)$.

A function $G : (\mathbb{N} \to \mathbb{N})^k \to \mathbb{N}^l \to \mathbb{N}$ *is OTM computable in polynomial time if there exists a second order polynomial* P *such that* $G(F_1, \dots, F_k, x_1, \dots x_l)$ *is computed by an OTM in time* $P(|F_1|, \dots, |F_k|, |F_1|, \dots, |x_l|)$ *on inputs* $x_1, \dots, x_l$ *and oracles* $F_1, \dots, F_k$.

Theorem 4.2.1 ([KC96])**.** *The set of polynomial time OTM computable functions is exactly the class of type-2 (Basic) Feasible Functionals,* $\mathtt{BFF}_2$.

4.2.2 A stream based characterization of type-2 feasible functionals

The syntax of the first order language on stream studied in [FHHP10, FHHP15] is very similar to the language of Section 4.1.1.

The main distinction is that the semantics is only lazy on stream data (and strict on other datatypes) that are encoded as in Section 4.1.1 using the constructor symbols `nil` and `cons`. This amounts to considering a reduction $\to$ that corresponds to $\Downarrow$ on streams of type $[\alpha]$ and to $\Downarrow_v$ on other types (see Section 4.1.1). Consequently, in this context, lazy values are of the shape:

$$\mathtt{lv} := \mathtt{cons}\ \mathtt{M}_1\ \mathtt{M}_2,$$

for some terms $\mathtt{M}_1$ and $\mathtt{M}_2$.

Using an idea similar to the encoding of a function over natural numbers as a stream of integers, the interpretation of a stream will be a type-1 function from natural numbers to natural numbers. Here we avoid the use of parameterized interpretations to simplify the discussion.

In the following, let positive functionals denote functions in $((\mathbb{N} \to \mathbb{N})^k \times \mathbb{N}^l) \to T$ with $k, l \in \mathbb{N}$ and $T \in \{\mathbb{N}, \mathbb{N} \to \mathbb{N}\}$. Given a positive functional $F : ((\mathbb{N} \to \mathbb{N})^k \times \mathbb{N}^l) \to T$, the arity of F is $k + l$.

Let $>$ denote the usual ordering on $\mathbb{N}$ and $\mathbb{N} \to \mathbb{N}$, *i.e.* given $F, G : \mathbb{N} \to \mathbb{N}$, $F > G$ if $\forall X \in \mathbb{N} \setminus \{0\}, F(X) > G(X)$. We extend this ordering to positive functionals of arity l by: $F > G$ if $\forall X_1, \dots, \forall X_l \in \{\mathbb{N} \setminus \{0\}, \mathbb{N} \to^{\uparrow} \mathbb{N}\}$, $F(X_1, \dots, X_l) > G(X_1, \dots, X_l)$, where $\mathbb{N} \to^{\uparrow} \mathbb{N}$ is the set of increasing functions on positive integers. A positive functional F of arity n is monotonic if $\forall i \in \{1, n\}, \forall X_i > X_i',\ F(\dots, X_i, \dots) > F(\dots, X_i', \dots)$, where $X_i, X_i' \in \{\mathbb{N} \setminus \{0\}, \mathbb{N} \to^{\uparrow} \mathbb{N}\}$.

Definition 4.2.5. *An assignment of a program is a total mapping of the function and constructor symbols to monotonic positive functionals. The type of the interpretation is inductively defined by the type of the corresponding symbol:*

- *a symbol* $\mathtt{b}$ *of type* A *(*$A \neq [\alpha]$*) has interpretation* $[\mathtt{b}]_{\mathbb{N}}$ *in* $\mathbb{N}$,
- *a symbol* $\mathtt{b}$ *of type* $[\alpha]$ *has interpretation* $[\mathtt{b}]_{\mathbb{N}}$ *in* $\mathbb{N} \to \mathbb{N}$,
- *a symbol* $\mathtt{b}$ *of type* $A \to B$ *has interpretation* $[\mathtt{b}]_{\mathbb{N}}$ *in* $T_A \to T_B$, *whether* T_A *and* T_B *are the types of the interpretations of the symbols of type* `A` *and, respectively, type* `B`.

The assignments of the constructs $\mathtt{case\ M\ of\ c}_1\ \mathtt{x}_1 : \mathtt{N}_1 | \dots | \mathtt{c}_n\ \mathtt{x}_n : \mathtt{N}_n$ *and* $\mathtt{letRec\ f} = \mathtt{M}$ *are defined as in Subsection 4.1.1.*

We fix the assignment of each constructor symbol by:

- $[\mathtt{c}]_{\mathbb{N}}(X_1, \ldots, X_{ar(\mathtt{c})}) = X_1 + \ldots X_{ar(\mathtt{c})} + 1$, *if* $\mathtt{c} \in \mathcal{C} \setminus \{\mathtt{cons}\}$,
- $[\mathtt{cons}]_{\mathbb{N}}(X, Y)(Z + 1) = 1 + X + Y(Z)$,
- $[\mathtt{cons}]_{\mathbb{N}}(X, Y)(0) = X$.

Once the assignment of each symbol is fixed, we can extend assignments to any expression by structural induction:

- $[\mathtt{x}]_{\mathbb{N}} = X$, *if* $\mathtt{x}$ *is a variable of type* A *(*$A \neq [\alpha]$*) and* X *is a variable in* $\mathbb{N}$,
- $[\mathtt{y}]_{\mathbb{N}}(Z) = Y(Z)$, *if* $\mathtt{y}$ *is a variable of type* $[\alpha]$, *i.e. we associate a unique second order variable* $Y : \mathbb{N} \to \mathbb{N}$ *to each* $\mathtt{y} \in \mathcal{X}$ *of type* $[\alpha]$,
- $[\mathtt{t}\ \mathtt{e_1}\ \ldots\ \mathtt{e_n}]_{\mathbb{N}} = [\mathtt{t}]_{\mathbb{N}}([\mathtt{e_1}]_{\mathbb{N}}, \ldots, [\mathtt{e_n}]_{\mathbb{N}})$, *if* $\mathtt{t}$ *is a constructor or function symbol.*

An assignment is polynomial if all symbols are interpreted as functions bounded by type-2 polynomials.

Consequently, an assignment $[-]_{\mathbb{N}}$ maps any expression to a functional (of the assignment of its free variables).

Example 4.2.2. *The stream constructor* $\mathtt{cons}$ *has type* $\alpha \to [\alpha] \to [\alpha]$. *Consequently, its assignment* $[\mathtt{cons}]_{\mathbb{N}}$ *has type* $(\mathbb{N} \times (\mathbb{N} \to \mathbb{N})) \to (\mathbb{N} \to \mathbb{N})$.[28] *Considering the expression* $\mathtt{cons\ p\ (cons\ q\ r)}$, *with* $\mathtt{p}, \mathtt{q}, \mathtt{r}$ *variables, we obtain that:*

$$\begin{aligned}
[\mathtt{cons\ p\ (cons\ q\ r)}]_{\mathbb{N}} &= [\mathtt{cons}]_{\mathbb{N}}([\mathtt{p}]_{\mathbb{N}}, [\mathtt{cons\ q\ r}]_{\mathbb{N}}) \\
&= [\mathtt{cons}]_{\mathbb{N}}([\mathtt{p}]_{\mathbb{N}}, [\mathtt{cons}]_{\mathbb{N}}([\mathtt{q}]_{\mathbb{N}}, [\mathtt{r}]_{\mathbb{N}})) \\
&= [\mathtt{cons}]_{\mathbb{N}}(P, [\mathtt{cons}]_{\mathbb{N}}(Q, R)) \\
&= F(P, Q, R)
\end{aligned}$$

where $F \in ((\mathbb{N} \to \mathbb{N}) \times \mathbb{N}^2) \to (\mathbb{N} \to \mathbb{N})$ *is the positive functional such that:*

- $F(P, Q, R)(Z + 2) = 1 + P + [\mathtt{cons}]_{\mathbb{N}}(Q, R)(Z + 1) = 2 + P + Q + R(Z)$,
- $F(P, Q, R)(1) = 1 + P + [\mathtt{cons}]_{\mathbb{N}}(Q, R)(0) = 1 + P + Q$,
- $F(P, Q, R)(0) = P$.

Definition 4.2.6 (Interpretation)**.** *The assignment of a term* $\mathtt{M}$ *is an interpretation if for each definition* $\mathtt{f\ p_1 \ldots p_n} = \mathtt{e} \in \mathtt{M}$,

$$[\mathtt{f\ p_1 \ldots p_n}]_{\mathbb{N}} > [\mathtt{e}]_{\mathbb{N}}$$

By extension, a term $\mathtt{M}$ *admits a (polynomial) interpretation if there exists a (polynomial) assignment that is an interpretation of* $\mathtt{M}$.

The following programs are examples of well-founded polynomial programs.

[28] We will use the Cartesian product instead of the arrow for the argument types of a function symbol in the following.

Example 4.2.3. !! s $\underline{\mathtt{n}}$ *computes the* $(n+1)^{th}$ *element of the stream* s

$$\begin{aligned}&!! : [\alpha] \to \mathtt{Nat} \to \alpha\\ &!!\ (\mathtt{cons\ x\ xs})\ \mathtt{0} = \mathtt{x}\\ &!!\ (\mathtt{cons\ x\ xs})\ (\mathtt{y{+}1}) = !!\ \mathtt{xs\ y}\end{aligned}$$

and admits an interpretation $[!!]_{\mathbb{N}}$ *in* $((\mathbb{N} \to \mathbb{N}) \times \mathbb{N}) \to \mathbb{N}$ *defined by* $[!!]_{\mathbb{N}}(Y, N) = Y(N)$. *Indeed, we check that:*

$$\begin{aligned}&[!!\ (\mathtt{cons\ x\ xs})\ (\mathtt{y{+}1})]_{\mathbb{N}} = [\mathtt{cons\ x\ xs}]_{\mathbb{N}}([\mathtt{y}]_{\mathbb{N}} + 1) = 1 + [\mathtt{x}]_{\mathbb{N}} + [\mathtt{xs}]_{\mathbb{N}}([\mathtt{y}]_{\mathbb{N}}),\\ &> [\mathtt{xs}]_{\mathbb{N}}([\mathtt{y}]_{\mathbb{N}}) = [!!\ \mathtt{xs\ y}]_{\mathbb{N}},\end{aligned}$$

and

$$[!!\ (\mathtt{cons\ x\ xs})\ \mathtt{0}]_{\mathbb{N}} = [(\mathtt{cons\ x\ xs})]_{\mathbb{N}}([\mathtt{0}]_{\mathbb{N}}) = [(\mathtt{cons\ x\ xs})]_{\mathbb{N}}(1) = 1 + [\mathtt{x}]_{\mathbb{N}} + [\mathtt{xs}]_{\mathbb{N}}(0) > [\mathtt{x}]_{\mathbb{N}}.$$

In the same way, we let the reader check that `tln`*, which drops the first* $n+1$ *elements of a stream, admits the well-founded interpretation* $[\mathtt{tln}]_{\mathbb{N}}$ *of type* $((\mathbb{N} \to \mathbb{N}) \times \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$ *defined by* $[\mathtt{tln}]_{\mathbb{N}}(Y, N)(Z) = Y(N + Z + 1)$.

$$\begin{aligned}&\mathtt{tln} : [\alpha] \to \mathtt{Nat} \to [\alpha]\\ &\mathtt{tln}\ (\mathtt{cons\ x\ xs})\ \mathtt{0} = \mathtt{xs}\\ &\mathtt{tln}\ (\mathtt{cons\ x\ xs})\ (\mathtt{y{+}1}) = \mathtt{tln\ xs\ y}\end{aligned}$$

Indeed, for the last rule, we just check that $[\mathtt{tln}\ (\mathtt{cons\ x\ xs})\ (\mathtt{y{+}1})]_{\mathbb{N}} > [\mathtt{tln\ xs\ y}]_{\mathbb{N}}$*, that is* $\forall Z \in \mathbb{N} \setminus \{0\},\ [\mathtt{tln}\ (\mathtt{cons\ x\ xs})\ (\mathtt{y{+}1})]_{\mathbb{N}}(Z) > [\mathtt{tln\ xs\ y}]_{\mathbb{N}}(Z)$.

Before stating the main characterization of type-2 Feasible Functionals, we need to restrict the considered interpretations.

Definition 4.2.7 (exp-poly)**.** *We call exp-poly the set of functions generated by the following grammar:*

$$EP := P \mid EP + EP \mid EP \times EP \mid Y(2^{EP}),$$

where P *denotes a first order polynomial and* Y *a second order variable.*

The interpretation of a program is exp-poly if each symbol is interpreted by an exp-poly function.

Theorem 4.2.2 ([FHHP15])**.** $\mathtt{BFF}_2$ *is exactly the set of functions that can be computed by programs that admit a exp-poly interpretation.*

Notice that the exponential is due to the fact that the time bound is encoded in [FHHP15] uses unary numbers rather than binary numbers. Indeed a decrease by one on a binary number does not imply a strict decrease of the interpretation. Consequently, this exponential is restricted to values (type-0 data).

4.2.3 A stream based characterization of polynomial time over the reals

Now we show that the above characterization can be adapted to polynomial time over the real numbers, as any real number can be encoded as a stream of integers.

Indeed, type 2 functionals can be used to represent real functions. Recursive analysis models computations on real numbers as computations on converging sequences of rational numbers. For interested readers, [Wei00] provides an in-depth view of recursive analysis from a computability point of view. Complexity in this model is treated in [Ko91].

We will require a given convergence speed to be able to compute efficiently. A real number $x \in \mathbb{R}^+$ is represented by the sequence $(q_n) \in \mathbb{Q}^{\mathbb{N}}$ if $\forall i \in \mathbb{N}, |x - q_i| < 2^{-i}$. This will be denoted by $(q_n) \rightsquigarrow x$. A function $f : \mathbb{R} \to \mathbb{R}$ will be said to be computed by a machine M if

$$(q_n) \rightsquigarrow x \Rightarrow (M(q_n)) \rightsquigarrow f(x). \tag{4.1}$$

Hence if α is an inductive type describing rational numbers (*e.g.* pairs of binary integers) then a computable real function could be represented by stream programs of type $[\alpha] \to [\alpha]$, where α is an inductive type describing the set of rationals $\mathbb{Q}$. Only programs encoding machines verifying implication (4.1) will make sense in this framework. Property (4.1) is highly semantic and hence difficult to enforce using static analysis techniques.

The above definition implies that if a real function is computable then it must be continuous (see [Wei00]).

Definition 4.2.8 (Complexity of a real function)**.** *$f : \mathbb{N} \to \mathbb{N}$ is a complexity measure for Machine M if and only if $\forall (q_n) \in \mathbb{Q}^{\mathbb{N}}, M(q_n)$ is output in time $f(n)$.*

The continuity property described above has a counterpart in complexity related to the modulus of continuity.

Definition 4.2.9 (Modulus of continuity)**.** *Given $f : \mathbb{R} \to \mathbb{R}$, we say that $m : \mathbb{N} \to \mathbb{N}$ is a modulus of continuity for f if and only if*

$$\forall n \in \mathbb{N}, \forall r, s \in \mathbb{R}, |r - s| < 2^{-m(n)} \implies |f(r) - f(s)| < 2^{-n}.$$

Definition 4.2.10. *The class of functions computed by Machines that have polynomial time complexity is denoted* $\mathtt{FP}(\mathbb{R})$.

Proposition 4.2.1. *If $f \in \mathtt{FP}(\mathbb{R})$ then it has a polynomial modulus of continuity.*

This proposition provides an interesting result for functions in $\mathtt{FP}(\mathbb{R})$. Indeed, it tells us that there are polynomial m and P such that for any real number r and any precision n, it suffices to read the first $m(n)$ digits of the stream representation of r and then to perform $P(n)$ computational steps in order to approximate $f(r)$ with a precision 2^{-n}.

Consequently, a stream based programming language allowed to compute a polynomial number of steps on a stream encoding would allow us to simulate such a behaviour.

Remark that the notion of Size based I/O upper bound of Subsection 4.1.4 can be viewed as sufficient condition to force the program to compute a function with a modulus of continuity. Indeed the size based I/O upper bound was defined as

$$\forall\ \underline{\mathtt{n}} \in \mathtt{Nat},\ \text{if } \mathtt{take}\ \underline{\mathtt{n}}\ \mathtt{s} \Downarrow_v \mathtt{v}\ \text{ implies } \mathtt{lg}\ \mathtt{M\{v/x\}} \Downarrow_v \underline{\mathtt{m}} \text{ then } F(|\underline{\mathtt{n}}|) \geq |\underline{\mathtt{m}}|.$$

If the stream $\mathtt{s}$ represents a real number (*e.g.* using a representation with digits in $\{-1, 0, 1\}$) then $\mathtt{take}\ \underline{\mathtt{n}}\ \mathtt{s}$ provides the $\mathtt{n}$ first elements of $\mathtt{s}$ and could be an approximation of the fractional

part of the real number (the error being bounded by $2^{-\mathtt{n}}$). The length of the output is bounded by $F(\mathtt{n})$ and, hence, the error is bounded by $2^{-F(\mathtt{n})}$. In the case where F is a one-to-one mapping, we obtain a modulus of continuity $m(\mathtt{n}) = F^{-1}(\mathtt{n})$.

Theorem 4.2.3 ([FHHP15]). *If a program $[\alpha] \to [\alpha]$ admitting a polynomial interpretation computes a real function on compact $\mathbb{K} \subseteq \mathbb{R}$, then this function is in $\mathtt{FP}(\mathbb{R})$.*

Theorem 4.2.4 ([FHHP15]). *Any polynomial-time computable real function (defined over $\mathbb{K}$) can be implemented by a program admitting a polynomial interpretation.*

4.2.4 A higher-order characterization of feasible functionals at any type

An alternative characterization of type-2 BFF to functions was provided in [HP17] in terms of functions over natural numbers (that can be viewed as streams/sequences of integers). For that purpose, we consider the higher-order language of Subsection 3.3.2 and the notion of interpretation introduced in Definition 3.3.3 (and Figure 3.16).

Definition 4.2.11 (Order). *Given a term $\mathtt{M}$ of type $\mathtt{T}$, i.e. $\emptyset; \Delta \vdash \mathtt{M} :: \mathtt{T}$, the order of $\mathtt{M}$ is equal to the order of $\mathtt{T}$, denoted $\mathtt{ord}(\mathtt{T})$ and defined inductively by:*

$$\begin{aligned} \mathtt{ord}(\mathtt{b}) &= 0, && \text{if } \mathtt{b} \in \boldsymbol{B}, \\ \mathtt{ord}(\mathtt{T} \longrightarrow \mathtt{T}') &= \max(\mathtt{ord}(\mathtt{T}) + 1, \mathtt{ord}(\mathtt{T}')) && \text{otherwise.} \end{aligned}$$

We extend Definition 4.2.3 to polynomials at any order.

Definition 4.2.12. *We consider types built from the basic type $\overline{\mathbb{N}}$ as follows:*

$$A, B ::= \overline{\mathbb{N}} \mid A \longrightarrow B.$$

Higher-order polynomials are built by the following grammar:

$$P, Q ::= c^{\overline{\mathbb{N}}} \mid X^A \mid +^{\overline{\mathbb{N}} \to \overline{\mathbb{N}} \to \overline{\mathbb{N}}} \mid \times^{\overline{\mathbb{N}} \to \overline{\mathbb{N}} \to \overline{\mathbb{N}}} \mid (P^{A \to B}\ Q^A)^B \mid (\Lambda X^A.P^B)^{A \to B}.$$

where c represents constants in $\mathbb{N}$ and P^A means that P is of type A. A polynomial P^A is of order i if $\mathtt{ord}(A) = i$. When A is explicit from the context, we use the notation P_i to denote a polynomial of order i.

In the above definition, constants of type $\overline{\mathbb{N}}$ cannot be $\top \in \overline{\mathbb{N}}$. By definition, a higher-order polynomial P_i has arguments of order at most $i - 1$. For notational convenience, we will use the application of $+$ and $\times$ with an infix notation as in the following example.

Example 4.2.4. *Here are several examples of polynomials generated by the grammer of Definition 4.2.12:*

- $P_1 = \Lambda X_0.(6 \times X_0^2 + 5)$ *is an order* 1 *polynomial,*
- $Q_1 = \times$ *is an order* 1 *polynomial,*
- $P_2 = \Lambda X_1.\Lambda X_0.(3 \times (X_1\ (6 \times X_0^2 + 5)) + X_0)$ *is an order* 2 *polynomial,*
- $Q_2 = \Lambda X_1.\Lambda X_0.((X_1\ (X_1\ 4)) + (X_1\ X_0))$ *is an order* 2 *polynomial.*

We are now ready to define the class of functions computed by terms admitting an interpretation that is bounded by higher-order polynomials.

$$
\begin{array}{rl}
\text{(Procedures)} \ni P & ::= \textbf{Procedure } v^{\tau_1\times\ldots\times\tau_n\to\mathtt{D}}(v_1^{\tau_1},\ldots,v_n^{\tau_n})\ P^*\ V\ I^*\ \textbf{Return } v_r^{\mathtt{D}} \\
\text{(Declarations)} \ni V & ::= \textbf{Var } v_1^{\mathtt{D}},\ldots,v_n^{\mathtt{D}}; \\
\text{(Instructions)} \ni I & ::= v^{\mathtt{D}} := E;\ |\ \textbf{Loop } v_0^{\mathtt{D}} \textbf{ with } v_1^{\mathtt{D}} \textbf{ do } \{I^*\} \\
\text{(Expressions)} \ni E & ::= 1 \mid v^{\mathtt{D}} \mid v_0^{\mathtt{D}} + v_1^{\mathtt{D}} \mid v_0^{\mathtt{D}} - v_1^{\mathtt{D}} \mid v_0^{\mathtt{D}} \# v_1^{\mathtt{D}} \mid v^{\tau_1\times\ldots\times\tau_n\to\mathtt{D}}(A_1^{\tau_1},\ldots,A_n^{\tau_n}) \\
\text{(Arguments)} \ni A & ::= v \mid \lambda v_1,\ldots,v_n.v(v_1'\ldots,v_m') \quad \text{with } v\notin\{v_1,\ldots,v_n\}
\end{array}
$$

Figure 4.2: BTLP grammar

Definition 4.2.13. *Let* $\mathtt{FP}_i$, $i>0$, *be the class of polynomial functionals at order* i *that consist in functionals computed by closed terms* $\mathtt{M}$ *such that* $\emptyset;\Delta\vdash \mathtt{M}::\mathtt{T}$ *over the basic type* $\mathtt{Nat}$ *of unary numbers and such that:*

- $\mathtt{ord}(\mathtt{T})=i$,
- $[\mathtt{M}]_\rho$ *is bounded by an order* i *polynomial (i.e.* $\exists P_i,\ [\mathtt{M}]_\rho\leq P_i$*).*

We will now introduce Bounded Typed Loop Programs from [IKR02] which provide an extension of the original complexity class $\mathtt{BFF}_2$ to any order using a programming language named BTLP.

Definition 4.2.14 (BTLP)**.** *A Bounded Typed Loop Program (BTLP) is a non-recursive and well-formed procedure defined by the grammar of Figure 4.2.*

The well-formedness assumption is given by the following constraints: each procedure is supposed to be well-typed with respect to simple types over $\mathtt{D}$, the set of natural numbers of dyadic representation over $\{0,1\}$ ($0\equiv\epsilon$, $1\equiv 0$, $2\equiv 1$, $3\equiv 00$, ...). When needed, types are explicitly mentioned in variables' superscript. Each variable of a BTLP procedure is bound by either the procedure declaration parameter list, a local variable declaration, or a lambda abstraction. In a loop statement, the guard variables v_0 and v_1 cannot be assigned to within I^*. In what follows v_1 will be called the *loop bound*.

The operational semantics of BTLP procedures is standard: parameters are passed by CBV. $+$, $-$ and $\#$ denote addition, proper subtraction, and smash function (*i.e.* $x\#y=2^{|x|\times|y|}$, the size $|x|$ of the number x being the size of its dyadic representation), respectively. Each loop statement is evaluated by iterating $|v_0|$-many times the loop body instruction under the following restriction: if an assignment $v:=E$ is to be executed within the loop body, we check if the value obtained by evaluating E is of size smaller than the size of the loop bound $|v_1|$. If not then the result of evaluating this assignment is to assign 0 to v.

Definition 4.2.15 ($\mathtt{BFF}_i$)**.** *For any* $i\geq 1$, $\mathtt{BFF}_i$ *is the class of order* i *functionals computable by a BTLP procedure.*

It is straightforward that $\mathtt{BFF}_1=\mathtt{FP}$ and $\mathtt{BFF}_2=\mathtt{BFF}$.

Now we restrict the domain of $\mathtt{BFF}_i$ classes to inputs in $\mathtt{BFF}_k$ for $k<i$, the obtained classes are named $\mathtt{SFF}$ for Safe Feasible Functionals.

Definition 4.2.16 ($\mathtt{SFF}_i$)**.** $\mathtt{SFF}_1$ *is defined to be the class of order* 1 *functionals computable by a BTLP procedure and, for any* $i\geq 1$, $\mathtt{SFF}_{i+1}$ *is the class of order* $i+1$ *functionals computable by*

a BTLP procedure on the input domain $\mathtt{SFF}_i$*. In other words,*

$$\begin{aligned} \mathrm{SFF}_1 &= \mathrm{BFF}_1, \\ \forall i \geq 1,\ \mathrm{SFF}_{i+1} &= \mathrm{BFF}_{i+1\restriction \mathrm{SFF}_i} \end{aligned}$$

This is not a huge restriction since we want an arbitrary term of a given complexity class at order i to compute over terms that are already in classes of the same family at order k, for $k < i$. Consequently, programs can be built in a constructive way component by component. Another strong argument in favor of this domain restriction is that the partial evaluation of a functional at order i will, at the end, provide a function in $\mathbb{N} \longrightarrow \mathbb{N}$ that is shown to be in $\mathtt{BFF}_1$ (=$\mathtt{FP}$).

We are now ready to provide a definition of Basic Feasible Functionals at any order.

Theorem 4.2.5 ([HP17])**.** *For any order $i \geq 1$, the class of functions in $\mathtt{FP}_i$ over $\mathtt{FP}_k$, $k < i$, is exactly the class of functionals in $\mathtt{SFF}_i$. In other words, $\mathtt{SFF}_i \equiv \mathtt{FP}_{i\restriction(\cup_{k<i}\mathtt{FP}_k)}$, for all $i \geq 1$.*

Example 4.2.5. *We have shown in Example 3.3.7, that the program*

$$\begin{aligned} \mathtt{M} := \mathtt{letRec\ f} = \lambda \mathtt{g}.\lambda \mathtt{x}.\mathtt{case\ x\ of\ c\ y\ z} : \mathtt{c\ (g\ y)\ (f\ g\ z)} \\ \mid \mathtt{nil} : \mathtt{nil} \end{aligned}$$

is such that $\mathtt{ord}(\mathtt{M}) = 2$ *and admits an interpretation bounded by* $\Lambda[\mathtt{g}]_\rho.\Lambda[\mathtt{x}]_\rho.(5 \oplus ([\mathtt{g}]_\rho\ [\mathtt{x}]_\rho)) \times [\mathtt{x}]_\rho$*. This is a second order polynomial as it can be rewritten equivalently as* $\Lambda X_1.\Lambda X_0.(5 + X_1(X_0)) \times X_0$*. Consequently, it belongs to* $\mathtt{FP}_2$ *and hence computes a function of* $\mathtt{SFF}_2$ *when fed with input in* $\mathtt{FP}$*.*

4.3 Coinductive datatypes in the light affine lambda calculus

The categorical notions of algebras and coalgebras [JR97] can provide different forms of recursion and corecursion that can be used to program algorithms in different ways. Moreover, algebras and coalgebras also provide some form of induction and coinduction that we can use to prove program properties.

Despite the fact that coalgebras correspond to infinite data types, interestingly coalgebras (and algebra) can be added to languages that are strongly normalizing while preserving the strong normalization property, as shown by [Hag87]. Moreover, algebras and coalgebras can also be encoded by using parametric polymorphism in strongly normalizing languages such as System F as shown by [Wra93].

We consider the definability of algebras and coalgebras in the Light Affine Lambda Calculus (LALC), a term language for LAL (see Subsection 1.2.4), as studied in [GP15a]. Our aim is to get a better understanding of the expressive power of LALC with respect to the definability of inductive and coinductive data structures, in particular with a focus on infinite data structures like streams. We want to define such kind of data without breaking the polynomial time normalization properties of the type system.

LALC can be seen as a subsystem of System F. However, not surprisingly, the standard System F encoding of (co)algebra cannot be straightforwardly adapted to LALC for technical reasons: variable duplication in the terms enforces the modalities ! and § to appear and to propagate to the functor. Consequently, new types for encoding initial algebras and final coalgebras are required. Consider a functor F over types.

The initial algebra for F can be encoded in LALC by terms of type

$$\forall X.!(F(X) \multimap X) \multimap \S X$$

$$
\begin{array}{rl}
\tau, \sigma ::= & X \mid \mathbf{1} \mid !\tau \mid \S\tau \mid \tau \oplus \sigma \mid \tau \otimes \sigma \mid \tau \multimap \sigma \mid \forall X.\tau \mid \exists X.\tau \\
\mathtt{M}, \mathtt{N}, \mathtt{L} ::= & \mathtt{x} \mid () \mid \lambda \mathtt{x} : \tau.\mathtt{M} \mid \mathtt{M}\,\mathtt{N} \mid \Lambda \mathtt{X}.\mathtt{M} \mid \mathtt{M}\,\tau \mid !\mathtt{M} \mid \S\mathtt{M} \\
& \mid \mathtt{let}\ \S\mathtt{x} : \tau = \mathtt{M}\ \mathtt{in}\ \mathtt{N} \mid \mathtt{let}\ !\mathtt{x} : \tau = \mathtt{M}\ \mathtt{in}\ \mathtt{N} \\
& \mid \langle \mathtt{M}, \mathtt{N} \rangle \mid \mathtt{let}\ \langle \mathtt{x} : \tau_1, \mathtt{y} : \tau_2 \rangle = \mathtt{M}\ \mathtt{in}\ \mathtt{N} \\
& \mid \mathtt{pack}\ (\mathtt{M}, \sigma)\ \mathtt{as}\ \tau \mid \mathtt{unpack}\ \mathtt{M}\ \mathtt{as}\ (\mathtt{X}, \mathtt{x})\ \mathtt{in}\ \mathtt{N} \\
& \mid \mathtt{let}\ () = \mathtt{M}\ \mathtt{in}\ \mathtt{N} \mid \mathtt{inj}_{\mathtt{i}}^{\tau}(\mathtt{M}) \mid \\
& \mid \mathtt{case}\ \mathtt{M}\ \ \mathtt{of}\ \ \{\mathtt{inj}_0^{\tau}(\mathtt{x}) \to \mathtt{N} | \mathtt{inj}_1^{\tau}(\mathtt{x}) \to \mathtt{L}\}
\end{array}
$$

Figure 4.3: Syntax of LALC

whereas they are encoded as $\forall X.(F(X) \to X) \to X$ in System F.

The final coalgebra for the same functor F can instead be encoded by terms of type

$$\exists X.!(X \multimap F(X)) \otimes \S X$$

whereas they are encoded as $\exists X.(X \to F(X)) \times X$ in System F.

Initial algebras and final coalgebras definable in System F are only *weak* (*i.e.* existence but no uniqueness). In the case of LALC the two types above provide even more restricted classes of initial algebras and final coalgebras: the ones that enjoy some distributive properties with the § modality. More precisely, for initial algebras we need functors that *left-distribute* over §, *i.e.* functors F such that $F(\S X) \multimap \S F(X)$. Conversely, for final algebras we need functors that *right-distribute* over §, *i.e.* functors F such that $\S F(X) \multimap F(\S X)$.

Functors that left-distribute over § are quite common in LALC and so we can define several standard inductive data types. Unfortunately, only few functors right-distribute over §. In particular, we cannot encode standard coinductive data structures. The main reason is that the modality § does not *distribute* with respect to the connectives tensor and plus. More precisely, in LALC we cannot derive the distribution laws $\S(A \otimes B) \multimap \S A \otimes \S B$ and $\S(A \oplus B) \multimap \S A \oplus \S B$ for generic A and B. We overcome this situation by adding terms for these distributive laws to the language LALC. Thanks to this extensions we are able to write programs working on infinite streams of Boolean numbers (or of any finite data type) and other infinite data types. A more complete and detailed discussion about the soundness of this extension can be found in [GP15a].

4.3.1 Light affine lambda calculus

The syntax of the Light Affine Lambda Calculus (LALC) is inspired by the type restrictions of LAL that we have described in Subsection 1.2.4. The types and the terms of LALC are presented in Figure 4.3. The multiplicative unit $\mathbf{1}$ is the only basic type. The type constructors consist of the linear implication $\multimap$, the tensor product $\otimes$, and the additive disjunction $\oplus$. There are type variables X, a universal quantifier $\forall X.\tau$, and an existential quantifier $\exists X.\tau$. As in LAL, we also have the two modalities ! and §. Every type constructor comes equipped with a term constructor and a term destructor.

The semantics of LALC is defined in terms of the reduction relation $\to$ described in Figure 4.4 where we use the notations $[\mathtt{M}/\mathtt{x}]$ and $[\tau/\mathtt{X}]$ for the usual capture avoiding substitution on terms and on types, respectively. Let $\dagger, \ddagger$ be modality meta-variables denoting the modalities ! or §. In Figure 4.4, we have omitted several commuting rules. The number of these rules is quite high and their behavior is standard. We provide only the last two rules as representative of this class.

$$
\begin{aligned}
(\lambda \mathtt{x} : \tau.\mathtt{M})\,\mathtt{N} &\to \mathtt{M}[\mathtt{N}/\mathtt{x}] \\
\mathtt{let}\ () = ()\ \mathtt{in}\ \mathtt{M} &\to \mathtt{M} \\
(\Lambda X.\mathtt{M})\,\tau &\to \mathtt{M}[\tau/X] \\
\mathtt{case}\ \mathtt{inj}_1^\tau(\mathtt{M})\ \ \mathtt{of}\ \ \{\mathtt{inj}_0^\tau(\mathtt{x}) \to \mathtt{N}_0 | \mathtt{inj}_1^\tau(\mathtt{x}) \to \mathtt{N}_1\} &\to \mathtt{N}_i[\mathtt{M}/\mathtt{x}] \\
\mathtt{let}\ \langle \mathtt{x} : \tau, \mathtt{y} : \sigma\rangle = \langle \mathtt{N}_0, \mathtt{N}_1\rangle\ \mathtt{in}\ \mathtt{M} &\to \mathtt{M}[\mathtt{N}_0/\mathtt{x}, \mathtt{N}_1/\mathtt{y}] \\
\mathtt{unpack}\ (\mathtt{pack}\ (\mathtt{M}, \sigma)\ \mathtt{as}\ \exists \mathtt{X}.\tau)\ \mathtt{as}\ (\mathtt{X}, \mathtt{x})\ \mathtt{in}\ \mathtt{N} &\to \mathtt{N}[\mathtt{M}/\mathtt{x}, \sigma/X] \\
\mathtt{let}\ \S\mathtt{x} : \S\tau = \S\mathtt{N}\ \mathtt{in}\ \mathtt{M} &\to \mathtt{M}[\mathtt{N}/\mathtt{x}] \\
\mathtt{let}\ !\mathtt{x} :!\tau = !\mathtt{N}\ \mathtt{in}\ \mathtt{M} &\to \mathtt{M}[\mathtt{N}/\mathtt{x}] \\
(\mathtt{let}\ \dagger\mathtt{x} : \dagger\tau = \mathtt{N}\ \mathtt{in}\ \mathtt{M})\mathtt{L} &\to \mathtt{let}\ \dagger\mathtt{x} : \dagger\tau = \mathtt{N}\ \mathtt{in}\ (\mathtt{ML}) \\
\mathtt{let}\ \dagger\mathtt{y} : \dagger\tau = (\mathtt{let}\ \ddagger\mathtt{x} : \ddagger\sigma = \mathtt{N}\ \mathtt{in}\ \mathtt{L})\ \mathtt{in}\ \mathtt{M} &\to \mathtt{let}\ \ddagger\mathtt{x} : \ddagger\sigma = \mathtt{N}\ \mathtt{in}\ (\mathtt{let}\ \dagger\mathtt{y} : \dagger\tau = \mathtt{L}\ \mathtt{in}\ \mathtt{M})
\end{aligned}
$$

Figure 4.4: Semantics of LALC

Let an environment Γ be a map from types to term variables. A typing judgment is of the shape $\Gamma \vdash \mathtt{M} : \tau$, for some typing environment Γ, some term $\mathtt{M}$ and some type τ. The standard typing rules, inherited from LAL, together with the rules for the extra constructs are given in Figure 4.5. As usual, this system uses the notion of discharged formulas, which are expressions of the form $[\tau]_\dagger$. Given a typing environment $\Gamma = \mathtt{x}_1 : \tau_1, \ldots, \mathtt{x}_n : \tau_n$, $[\Gamma]_\dagger$ is a notation for the environment $\mathtt{x}_1 : [\tau_1]_\dagger, \ldots \mathtt{x}_n : [\tau_n]_\dagger$.

We can characterize FP using LALC. This characterization is similar to the one of LAL presented in Theorem 1.2.3. Let the *depth* $d(\mathtt{M})$ of a term $\mathtt{M}$: the maximal number of nested ! or § that can be found in any path of the term syntax tree. Let the data type $\mathbb{B}^*$ be a standard Church encoding of strings of Boolean numbers.

Theorem 4.3.1 (FP soundness and completeness)**.** *Consider a term $\Gamma \vdash \mathtt{M} : \tau$. Then, $\mathtt{M}$ can be reduced to normal form by a TM working in time polynomial in $|\mathtt{M}|$ with exponent proportional to $d(\mathtt{M})$. Conversely, for every function $f : \{0,1\}^* \to \{0,1\}^*$ in FP there exists a natural number n and a term $\mathtt{M} : \mathbb{B}^* \multimap \S^n\mathbb{B}^*$ such that and $\mathtt{M}$ computes f.*

4.3.2 Algebra and coalgebra in System F

Let us start by recalling the standard notions of F-algebras and F-coalgebras.

Definition 4.3.1 (F-Algebra and F-Coalgebra)**.** *Given a category $\mathcal{C}$ and a (endo)functor $F : \mathcal{C} \to \mathcal{C}$:*

- *a* F-algebra *is pair (A, a) of an object $A \in \mathcal{C}$ together with a $\mathcal{C}$-morphism $a : F(A) \to A$,*
- *a* F-coalgebra *is pair (A, a) of an object $A \in \mathcal{C}$ together with a $\mathcal{C}$-morphism $a : A \to F(A)$.*

We can define two categories Alg-F and Coalg-F whose objects are F-algebras and F-coalgebras, respectively, and whose morphisms are defined as follows.

Definition 4.3.2. *A F-algebra homomorphism from the F-algebra (A, a) to the F-algebra (B, b)*

$$\frac{}{\mathtt{x} : \tau \vdash \mathtt{x} : \tau}\ (\text{Ax}) \quad \frac{\Gamma \vdash \mathtt{M} : \tau}{\Gamma, \Delta \vdash \mathtt{M} : \tau}\ (\text{W}) \quad \frac{\Gamma, \mathtt{x} : [\tau]_!, \mathtt{y} : [\tau]_! \vdash \mathtt{M} : \sigma}{\Gamma, \mathtt{z} : [\tau]_! \vdash \mathtt{M}[\mathtt{z}/\mathtt{x}, \mathtt{z}/\mathtt{y}] : \sigma}\ (\text{C})$$

$$\frac{\Gamma, \mathtt{x} : \tau \vdash \mathtt{M} : \sigma}{\Gamma \vdash \lambda \mathtt{x} : \tau . \mathtt{M} : \tau \multimap \sigma}\ (\multimap\text{I}) \quad \frac{\Gamma \vdash \mathtt{M} : \tau \multimap \sigma \qquad \Delta \vdash \mathtt{N} : \tau}{\Gamma, \Delta \vdash \mathtt{M}\,\mathtt{N} : \sigma}\ (\multimap\text{E})$$

$$\frac{\Gamma, \Delta \vdash \mathtt{M} : \tau}{[\Gamma]_!, [\Delta]_\S \vdash \S \mathtt{M} : \S \tau}\ (\S\text{I}) \quad \frac{\Gamma \vdash \mathtt{N} : \S \tau \qquad \Delta, \mathtt{x} : [\tau]_\S \vdash \mathtt{M} : \sigma}{\Gamma, \Delta \vdash \mathtt{let}\ \S \mathtt{x} : \S \tau = \mathtt{N}\ \mathtt{in}\ \mathtt{M} : \sigma}\ (\S\text{E})$$

$$\frac{\Gamma \vdash \mathtt{M} : \tau \qquad \Gamma \subseteq \{\mathtt{x} : \sigma\}}{[\Gamma]_! \vdash !\mathtt{M} : !\tau}\ (!\ \text{I}) \quad \frac{\Gamma \vdash \mathtt{N} : !\tau \qquad \Delta, \mathtt{x} : [\tau]_! \vdash \mathtt{M} : \sigma}{\Gamma, \Delta \vdash \mathtt{let}\ !\mathtt{x} : !\tau = \mathtt{N}\ \mathtt{in}\ \mathtt{M} : \sigma}\ (!\ \text{E})$$

$$\frac{\Gamma \vdash \mathtt{M} : \tau \qquad X \notin FV(\Gamma)}{\Gamma \vdash \Lambda X . \mathtt{M} : \forall X . \tau}\ (\forall\ \text{I}) \quad \frac{\Gamma \vdash \mathtt{M} : \forall X . \tau}{\Gamma \vdash \mathtt{M}\sigma : \tau[\sigma/X]}\ (\forall\ \text{E})$$

$$\frac{\Gamma \vdash \mathtt{M} : \tau \qquad \Delta \vdash \mathtt{N} : \sigma}{\Gamma, \Delta \vdash \langle \mathtt{M}, \mathtt{N} \rangle : \tau \otimes \sigma}\ (\otimes\ \text{I}) \quad \frac{\Gamma \vdash \mathtt{M} : \tau \otimes \sigma \qquad \Delta, \mathtt{x} : \tau, \mathtt{y} : \sigma \vdash \mathtt{N} : \tau'}{\Gamma, \Delta \vdash \mathtt{let}\ \langle \mathtt{x} : \tau, \mathtt{y} : \sigma \rangle = \mathtt{M}\ \mathtt{in}\ \mathtt{N} : \tau'}\ (\otimes\ \text{E})$$

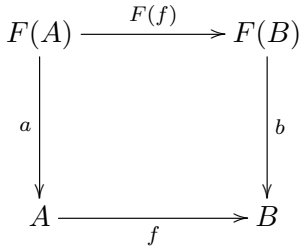
$$\frac{}{\Gamma \vdash () : \mathbf{1}}\ (\mathbf{1}\ \text{I}) \quad \frac{\Gamma \vdash \mathtt{M} : \mathbf{1} \qquad \Delta \vdash \mathtt{N} : \tau}{\Gamma, \Delta \vdash \mathtt{let}\ () = \mathtt{M}\ \mathtt{in}\ \mathtt{N} : \tau}\ (\mathbf{1}\ \text{E})$$

$$\frac{\Gamma \vdash \mathtt{M} : \tau_i}{\Gamma \vdash \mathtt{inj}_i^{\tau_0 \oplus \tau_1}(\mathtt{M}) : \tau_0 \oplus \tau_1}\ (\oplus\ \text{I}) \quad \frac{\Gamma \vdash \mathtt{M} : \tau_0 \oplus \tau_1 \qquad \forall i,\ \Delta, \mathtt{x} : \tau_i \vdash \mathtt{N}_i : \tau}{\Gamma, \Delta \vdash \mathtt{case}\ \mathtt{M}\ \ \mathtt{of}\ \ \{\mathtt{inj}_0^{\tau_0 \oplus \tau_1}(\mathtt{x}) \to \mathtt{N}_0 | \mathtt{inj}_1^{\tau_0 \oplus \tau_1}(\mathtt{x}) \to \mathtt{N}_1\} : \tau}\ (\oplus\ \text{E})$$

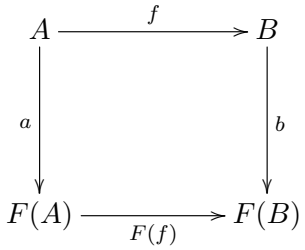
$$\frac{\Gamma \vdash \mathtt{M} : \tau[\sigma/X]}{\Gamma \vdash \mathtt{pack}\ (\mathtt{M}, \sigma)\ \mathtt{as}\ \exists \mathtt{X} . \tau : \exists X . \tau}\ (\exists\ \text{I}) \quad \frac{\Gamma \vdash \mathtt{M} : \exists X . \tau \qquad \Delta, \mathtt{x} : \tau \vdash \mathtt{N} : \sigma}{\Gamma, \Delta \vdash \mathtt{unpack}\ \mathtt{M}\ \mathtt{as}\ (\mathtt{X}, \mathtt{x})\ \mathtt{in}\ \mathtt{N} : \sigma}\ (\exists\ \text{E})$$

Figure 4.5: Type system for LALC

is a morphism $f : A \to B$ making the following diagram commute:



A F-coalgebra homomorphism from the F-coalgebra (A, a) to the F-coalgebra (B, b) is a morphism $f : A \to B$ making the following diagram commute:



To define the traditional inductive and coinductive data types we also need the notions of *initial algebras* and *final coalgebras.*

Definition 4.3.3 (Initial algebra and final coalgebra)**.** *A F-algebra (A, a) is* initial *if for each F-algebra (B, b), there exists a unique F-algebra homomorphism $f : A \to B$. A F-coalgebra (A, a) is* final *if for each F-coalgebra (B, b), there exists a unique F-coalgebra homomorphism $f : B \to A$.*

If the uniqueness condition is not met then the F-algebra (F-coalgebra, respectively) is only weakly initial *(*weakly final*, respectively).*

An initial F-algebra is an initial object in the category Alg-F. Conversely, a final F-coalgebra is a terminal object in the category Coalg-F.

A functor $F(X)$ is definable in System F if $F(X)$ is a type scheme mapping every type A to the type $F(A)$, and if there exists a term F mapping every term of type $A \to B$ to a term of type $F(A) \to F(B)$ and such that it preserves identity and composition. We say that a functor $F(X)$ is covariant if the variable X only appears in covariant positions.

It is a well known result that for any covariant functor $F(X)$ that is definable in System F we can define an algebra that is weakly initial and a coalgebra that is weakly final [JR97].

Proposition 4.3.1 (Weakly Initial Algebra)**.** *Let $F(X)$ be a covariant functor definable in System F and $\mathtt{T} = \forall X.(F(X) \to X) \to X$. Consider the morphisms defined by:*

$$\begin{aligned}
&\mathtt{in}_\mathtt{T} : F(\mathtt{T}) \to \mathtt{T},\\
&\mathtt{in}_\mathtt{T} = \lambda \mathtt{s} : F(\mathtt{T}).\Lambda X.\lambda \mathtt{k} : F(X) \to X.\mathtt{k}(F\ (\mathtt{fold}_\mathtt{T}\ X\ \mathtt{k})\ \mathtt{s}),\\
&\mathtt{fold}_\mathtt{T} : \forall X.(F(X) \to X) \to \mathtt{T} \to X,\\
&\mathtt{fold}_\mathtt{T} = \Lambda X.\lambda \mathtt{k} : F(X) \to X.\lambda \mathtt{t} : \mathtt{T}.\mathtt{t}\ X\ \mathtt{k}.
\end{aligned}$$

Then, $(T, \mathtt{in}_\mathtt{T})$ is a weakly initial F-algebra: for every F-algebra $(A, g : F(A) \to A)$ there is a F-homomorphism $h : T \to A$ defined as $h = \mathtt{fold}_\mathtt{T}\, A\, g$.

We will sometimes write T as $\mu X.F(X)$ when we want to stress the underlying functor F and the fact that T corresponds to the least fixpoint of F.

Example 4.3.1 ([GP15a])**.** *Let us consider a functor defined on types as* $F(X) = \mathbf{1} + X$ *and on terms as:*

$$\lambda \mathtt{f} : X \to Y.\lambda \mathtt{x} : \mathbf{1} + X.\mathtt{case\ x\ of}\{\mathtt{inj}_0^{\mathbf{1}+X}(\mathtt{z}) \to \mathtt{inj}_0^{\mathbf{1}+Y}(()), \mathtt{inj}_1^{\mathbf{1}+X}(\mathtt{z}) \to \mathtt{inj}_1^{\mathbf{1}+Y}(\mathtt{f\,z})\ \}.$$

Let $\mathbb{N} = \mu X.F(X)$*. Proposition 4.3.1 ensures that* $(\mathbb{N}, \mathtt{in}_{\mathbb{N}})$ *is a weak initial algebra: the weakly initial algebra of natural numbers. In particular, we can define* $\underline{0} = \mathtt{in}_{\mathbb{N}}(\mathtt{inj}_0^{\mathbf{1}+\mathbb{N}}(()))$*,* $\underline{n+1} = \mathtt{in}_{\mathbb{N}}(\mathtt{inj}_1^{\mathbf{1}+\mathbb{N}}(\underline{n}))$*, and more in general the successor function as* $\mathtt{succ} = \lambda \mathtt{x}.\mathtt{in}_{\mathbb{N}}(\mathtt{inj}_1^{\mathbf{1}+\mathbb{N}}(\mathtt{x}))$*. We can use the fact that* $\mathbb{N}$ *is a weakly initial algebra to define an addition function. We just need to consider a term like the following (we omit some type for conciseness):*

$$g = \lambda \mathtt{x} : \mathbf{1} + (\mathbb{N} \to \mathbb{N}).\mathtt{case\ x\ of}\{\mathtt{inj}_0(\mathtt{z}) \to \lambda \mathtt{y} : \mathbb{N}.\mathtt{y}, \mathtt{inj}_1(\mathtt{z}) \to \lambda \mathtt{y} : \mathbb{N}.\mathtt{succ}(\mathtt{z\,y})\ \}.$$

Then, Proposition 4.3.1 ensures that we can define $\mathtt{add}$ *as* $\mathtt{fold}_{\mathbb{N}}\,(\mathbb{N} \to \mathbb{N})\,g$*.*

Proposition 4.3.2 (Weakly Final Coalgebra)**.** *Let* F *be a covariant functor definable in System* F *and* $T = \exists X.(X \to F(X)) \times X$*. Consider the morphisms defined by:*

$$\begin{aligned}
&\mathtt{out}_{\mathtt{T}} : \mathtt{T} \to F(\mathtt{T}),\\
&\mathtt{out}_{\mathtt{T}} = \lambda \mathtt{t} : \mathtt{T}.\mathtt{unpack\ t\ as}\ (X, \mathtt{z})\ \mathtt{in}\\
&\qquad \mathtt{let}\ (\mathtt{k}, \mathtt{x}) = \mathtt{z}\ \mathtt{in}\ F(\mathtt{unfold}_{\mathtt{T}}\ X\ \mathtt{k})(\mathtt{k\ x}),\\
&\mathtt{unfold}_{\mathtt{T}} : \forall X.(X \to F(X)) \to X \to \mathtt{T},\\
&\mathtt{unfold}_{\mathtt{T}} = \Lambda X.\lambda \mathtt{k} : X \to F(X).\lambda \mathtt{x} : X.\mathtt{pack}\,((\mathtt{k}, \mathtt{x}), X)\,\mathtt{as\,T}.
\end{aligned}$$

Then, $(T, \mathtt{out}_{\mathtt{T}})$ *is a weakly final* F*-coalgebra: for every* F*-coalgebra* $(A, g : A \to F(A))$ *there is a* F*-homomorphism* $h : A \to T$ *defined as* $h = \mathtt{unfold}_{\mathtt{T}}\,A\,g$*.*

Similarly to the case of F-algebras, we will write T as $\nu X.F(X)$ when we want to stress the underlying functor F and the fact that T corresponds to the greatest fixpoint of F.

Example 4.3.2 ([GP15a])**.** *Let us consider a functor defined on types as* $F(X) = \mathbb{N} \times X$ *and on terms as:*

$$\lambda \mathtt{f} : X \to Y.\lambda \mathtt{x} : \mathbb{N} \times X.\mathtt{let}\ \langle \mathtt{x}_1, \mathtt{x}_2 \rangle = \mathtt{x}\ \mathtt{in}\ \langle \mathtt{x}_1, \mathtt{f\ x}_2 \rangle.$$

Let $\mathbb{N}^\omega = \nu X.F(X)$*. Proposition 4.3.2 ensures that* $(\mathbb{N}^\omega, \mathtt{out}_{\mathbb{N}^\omega})$ *is a weak final coalgebra: the weakly final coalgebra of streams over natural numbers. We can define the usual operations on streams as* $\mathtt{head} = \lambda \mathtt{x} : \mathbb{N}^\omega.\mathtt{let}\ \langle \mathtt{x}_1, \mathtt{x}_2 \rangle = (\mathtt{out}_{\mathbb{N}^\omega}\ \mathtt{x})\ \mathtt{in}\ \mathtt{x}_1$*, and* $\mathtt{tail} = \lambda \mathtt{x} : \mathbb{N}^\omega.\mathtt{let}\ \langle \mathtt{x}_1, \mathtt{x}_2 \rangle = (\mathtt{out}_{\mathbb{N}^\omega}\ \mathtt{x})\ \mathtt{in}\ \mathtt{x}_2$*. We can use the fact that* $\mathbb{N}^\omega$ *is a weakly final coalgebra to define streams. As an example we can define a constant stream of* k*s by using a function:*

$$g = \lambda \mathtt{x} : \mathbf{1}.\mathtt{let}\ () = \mathtt{x}\ \mathtt{in}\ \langle \mathtt{k}, () \rangle.$$

Proposition 4.3.2 ensures that we can define $\mathtt{const} = \mathtt{unfold}_{\mathbb{N}^\omega}\,\mathbf{1}\,g()$*. Similarly, we can define a function that extracts from a stream the elements in even position. This time we need a function:*

$$g = \lambda \mathtt{x} : \mathbb{N}^\omega.\langle \mathtt{hd\,x}, \mathtt{tl}\,(\mathtt{tl\,x}) \rangle.$$

Proposition 4.3.2 ensures that we can define $\mathtt{even} = \mathtt{unfold}_{\mathbb{N}^\omega}\,\mathbb{N}^\omega\,g$*.*

4.3.3 Algebra in the light affine lambda calculus

Now we try to adapt the notions of algebra and coalgebra to the LALC setting. As already mentioned, the candidate type for weakly initial algebra is:

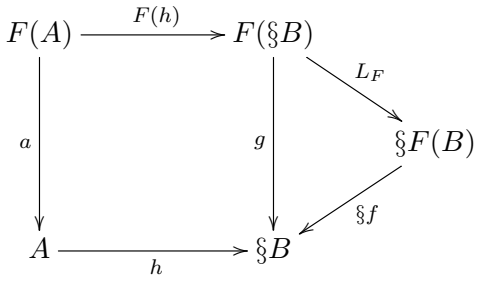
$$\mathtt{T} = \forall X.!(F(X) \multimap X) \multimap \S X.$$

However two restrictions are needed. First, the modality § in the above equation propagates when one wants to build the F-homomorphism to another F-algebra. Consequently, only weakly initial algebras of the shape $(\S B, g : F(\S B) \to \S B)$ can be considered. Second, the functor F has to *left-distribute* over §. This corresponds to require the existence of a morphism:

$$L_F : F(\S X) \multimap \S F(X).$$

Putting these two restrictions together we can formulate an alternative definition of weakly initial algebra as follows.

Definition 4.3.4. *Given a functor F, we say that an F-algebra (A, a) is* weakly initial under § *if for every F-algebra of the form (B, f) there exists an F-algebra $(\S B, g)$ and an F-algebra homomorphism $h : A \to \S B$ making the following diagram commute:*



Theorem 4.3.2 ([GP15a])**.** *Let F be a functor definable in LALC that left-distributes over §, and let $\mathtt{T} = \forall X.!(F(X) \multimap X) \multimap \S X$. Consider the morphisms defined by:*

$$\begin{aligned}
&\mathtt{in_T} : F(\mathtt{T}) \multimap \mathtt{T},\\
&\mathtt{in_T} = \lambda \mathtt{s} : F(\mathtt{T}).\Lambda X.\lambda \mathtt{k} :!(F(X) \multimap X).\mathtt{let}\ !\mathtt{y} :!(F(X) \multimap X) = \mathtt{k}\ \mathtt{in}\\
&\qquad\qquad \mathtt{let}\ \S \mathtt{z} : \S F(X) = L_F(F(\mathtt{fold_T}\ X\ !\mathtt{y})\ \mathtt{s})\ \mathtt{in}\ \S(\mathtt{y}\ \mathtt{z}),\\
&\mathtt{fold_T} : \forall X.!(F(X) \multimap X) \multimap \mathtt{T} \multimap \S X,\\
&\mathtt{fold_T} = \Lambda X.\lambda \mathtt{k} :!(F(X) \multimap X).\lambda \mathtt{p} : T.\mathtt{p}\ X\ \mathtt{k}.
\end{aligned}$$

Then, $(T, \mathtt{in_T})$ is a weakly initial F-algebra under §: for every F-algebra $(B, f : F(B) \multimap B)$ we have an F-algebra $(\S B, g : F(\S B) \to \S B)$ and an F-algebra homomorphism $h : \mathtt{T} \to \S B$ defined as $h = \mathtt{fold_T}\ B\ !f$.

Moreover we can give a characterization of a large class of left-distributing functors.

Lemma 4.3.1. *All the functors built using the following signature left-distribute over §:*

$$F(X) ::= \mathbf{1} \mid X \mid A \mid \S F(X) \mid F(X) \oplus F(X) \mid F(X) \otimes F(X),$$

provided that A is a closed type for which it exists a closed term of type $A \multimap !A$ or type $A \multimap \S A$.

Example 4.3.3. *Consider the functor $F(X) = \mathbf{1} \oplus X$. This is the linear analogous of the functor considered in Example 4.3.1, definable by the same term (in the types annotation, implication is replaced by a linear arrow and $+$ is replaced by $\oplus$). By Lemma 4.3.1, we have that F left-distribute over §, and so by Theorem 4.3.2 we have that $(\mathbb{N}, \mathtt{in}_{\mathbb{N}})$ is a weakly initial F-algebra under §, where by abuse of notation we again use $\mathbb{N}$ to denote $\mu X.F(X)$. Similarly to what we did in Example 4.3.1, we can define natural numbers as inhabitants of this type. Noticing that to the term g defined there we can also give the type $F(\mathbb{N} \multimap \mathbb{N}) \multimap (\mathbb{N} \multimap \mathbb{N})$, we have that $\mathtt{add} = \mathtt{fold}_{\mathbb{N}}\,(\mathbb{N} \multimap \mathbb{N})\,!g$ has type $\mathbb{N} \multimap \S(\mathbb{N} \multimap \mathbb{N})$.*

4.3.4 Coalgebra in the light affine lambda calculus

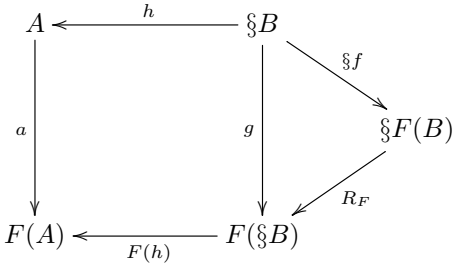
To encode final coalgebra in our setting we consider the following type:

$$\mathtt{T} = \exists X.!(X \multimap F(X)) \otimes \S X.$$

By duality, we encounter problems similar to the ones encountered for the algebra encoding: we need to restrict our study to coalgebra of the shape $(\S B, g : \S B \multimap F(\S B))$. Moreover, we also need the functor F to *right-distribute* over §. This corresponds to require for the existence of the following morphism:

$$R_F : \S F(X) \multimap F(\S X).$$

Definition 4.3.5. *Given a functor F, we say that an F-coalgebra (A, a) is* weakly final under § *if for every F-coalgebra of the form (B, f) there exists an F-coalgebra $(\S B, g)$ and an F-coalgebra homomorphism $h : \S B \to A$ making the following diagram commute:*



Theorem 4.3.3 ([GP15a])**.** *Let F be a functor definable in LALC that right-distribute over §, and let $\mathtt{T} = \exists X.!(X \multimap F(X)) \otimes \S X$. Consider the morphisms defined by:*

$$\begin{aligned}
&\mathtt{out}_{\mathtt{T}} : \mathtt{T} \multimap F(\mathtt{T}),\\
&\mathtt{out}_{\mathtt{T}} = \lambda \mathtt{t} : \mathtt{T}.\mathtt{unpack\ t\ as\ } (X, \mathtt{z}) \mathtt{\ in\ let\ } \langle \mathtt{k} :!(X \multimap F(X)), \mathtt{x} : \S X\rangle = \mathtt{z\ in}\\
&\qquad \mathtt{let\ !u = k\ in\ \ let\ } \S \mathtt{v} = \mathtt{x\ in\ } F\ (\mathtt{unfold}_{\mathtt{T}}\ X\ !\mathtt{u})\ R_F(\S(\mathtt{u\ v})),\\
&\mathtt{unfold}_{\mathtt{T}} : \forall X.!(X \multimap F(X)) \multimap \S X \multimap \mathtt{T},\\
&\mathtt{unfold}_{\mathtt{T}} = \Lambda X.\lambda \mathtt{k} :!(X \multimap F(X)).\lambda \mathtt{x} : \S X.\mathtt{pack}\,(\langle \mathtt{k}, \mathtt{x}\rangle, X)\,\mathtt{as\ T}.
\end{aligned}$$

Then, $(\mathtt{T}, \mathtt{out}_{\mathtt{T}})$ is a weakly final F-coalgebra under §: for every F-coalgebra $(B, f : B \multimap F(B))$ we have an F-coalgebra $(\S B, g : \S B \to F(\S B))$ and an F-coalgebra homomorphism $h : \S B \to \mathtt{T}$ defined as $h = \mathtt{unfold}_{\mathtt{T}}\, B\, !f$.

Sadly, the coalgebra counterpart of Lemma 4.3.1 has a poor expressive power:

Lemma 4.3.2. *All the functors built using the following signature righ-distribute over §:*

$$F(X) ::= \mathbf{1} \mid X \mid A \mid \S F(X),$$

provided that A is a closed type for which it exists a closed term of type $\S A \multimap A$.

$$
\begin{aligned}
\mathtt{M}, \mathtt{N}, \ ::= \ & \ldots \\
& \mid \mathtt{dist}\ \S\langle \mathtt{x} : \tau_1, \mathtt{y} : \tau_2 \rangle = \mathtt{M}\ \mathtt{as}\ \mathtt{z} = \langle \S\mathtt{x}, \S\mathtt{y} \rangle\ \mathtt{in}\ \mathtt{N} \\
& \mid \mathtt{dist}\ \S\mathtt{inj}_{\mathtt{i}}^{\tau \oplus \tau'}(\mathtt{x}) = \mathtt{M}\ \mathtt{as}\ \mathtt{z} = \mathtt{inj}_{\mathtt{i}}^{\S\tau \oplus \S\tau'}(\S\mathtt{x})\ \mathtt{in}\ \mathtt{N}.
\end{aligned}
$$

(a) Syntax of LALC distributions

$$
\begin{aligned}
&\mathtt{dist}\ \S\langle \mathtt{x} : \tau, \mathtt{y} : \sigma \rangle = \S\langle \mathtt{M}_1, \mathtt{M}_2 \rangle\ \mathtt{as}\ \mathtt{z} = \langle \S\mathtt{x}, \S\mathtt{y} \rangle\ \mathtt{in}\ \mathtt{N} \to \mathtt{N}[\langle \S\mathtt{M}_1, \S\mathtt{M}_2 \rangle / \mathtt{z}] && \text{(dis-1)} \\
&\mathtt{dist}\ \S\mathtt{inj}_{\mathtt{i}}^{\tau \oplus \tau'}(\mathtt{x}) = \S\mathtt{inj}_{\mathtt{i}}^{\tau \oplus \tau'}(\mathtt{M})\ \mathtt{as}\ \mathtt{z} = \mathtt{inj}_{\mathtt{i}}^{\S\tau \oplus \S\tau'}(\S\mathtt{x})\ \mathtt{in}\ \mathtt{N} \to \mathtt{N}[\mathtt{inj}_{\mathtt{i}}^{\S\tau \oplus \S\tau'}(\S\mathtt{M})/\mathtt{z}] && \text{(dis-2)}
\end{aligned}
$$

(b) Semantics of LALC distributions

$$
\frac{\Gamma \vdash \mathtt{M} : \S(\tau \otimes \sigma) \qquad \Delta, \mathtt{z} : \S\tau \otimes \S\sigma \vdash \mathtt{N} : \tau'}{\Gamma, \Delta \vdash \mathtt{dist}\ \S\langle \mathtt{x} : \tau, \mathtt{y} : \sigma \rangle = \mathtt{M}\ \mathtt{as}\ \mathtt{z} = \langle \S\mathtt{x}, \S\mathtt{y} \rangle\ \mathtt{in}\ \mathtt{N} : \tau'}\ (\mathtt{d}\otimes)
$$

$$
\frac{\Gamma \vdash \mathtt{M} : \S(\tau \oplus \sigma) \qquad \Delta, \mathtt{z} : \S\tau \oplus \S\sigma \vdash \mathtt{N} : \tau'}{\Gamma, \Delta \vdash \mathtt{dist}\ \S\mathtt{inj}_{\mathtt{i}}^{\tau \oplus \sigma}(\mathtt{x}) = \mathtt{M}\ \mathtt{as}\ \mathtt{z} = \mathtt{inj}_{\mathtt{i}}^{\S\tau \oplus \S\sigma}(\S\mathtt{x})\ \mathtt{in}\ \mathtt{N} : \tau'}\ (\mathtt{d}\oplus)
$$

(c) Typing rules for LALC distributions

Figure 4.6: LALC extended with distributions

This issue is solved in [GP15a] by extending the syntax with explicit distribution constructs. The extended syntax, semantics, and typing rules are provided in Figure 4.6.

Now Lemma 4.3.2 can be extended as follows.

Lemma 4.3.3. *In LALC extended with distributions, all the functors built using the following signature righ-distribute over §:*

$$F(X) ::= \mathbf{1} \mid X \mid A \mid \S F(X) \mid F(X) \otimes F(X) \mid F(X) \oplus F(X),$$

provided that A is a closed type for which it exists a closed term of type $\S A \multimap A$.

Now an open issue solved in [GP15a] was to know whether the soundness of Theorem 4.3.1 remains true in the extended language. The main idea is that the stratification principle observed in LAL or LACL remains valid in the extended framework.

Theorem 4.3.4 (Polynomial Time Soundness of LALC extended with distributions [GP15a])**.** *Consider a type derivation $\Gamma \vdash \mathtt{M} : \tau$ of a term $\mathtt{M}$ of LALC extended with distributions. Then, $\mathtt{M}$ can be reduced to normal form by a TM working in time polynomial in $|\mathtt{M}|$ with exponent proportional to $d(\mathtt{M})$.*

Example 4.3.4 ([GP15a])**.** *We would like to consider streams of natural numbers as in Example 4.3.2. Unfortunately, Lemma 4.3.3 is not enough to show that the functor $F(X) = \mathbb{N} \otimes X$ distributes to the right, as the coercion $\S\mathbb{N} \multimap \mathbb{N}$ does not hold. Nevertheless, we can consider*

streams of every finite type of the shape $\mathbf{1} \oplus \cdots \oplus \mathbf{1}$*, including Boolean numbers. Let us consider the functor defined on types as* $F(X) = \mathbb{B}_2 \otimes X$ *and on terms as:*

$$\lambda \mathtt{f} : X \multimap Y.\lambda \mathtt{x} : \mathbb{B}_2 \otimes X.\mathtt{let}\ \langle \mathtt{x}_1, \mathtt{x}_2 \rangle = \mathtt{x}\ \mathtt{in}\ \langle \mathtt{x}_1, \mathtt{f}\ \mathtt{x}_2 \rangle.$$

This functor right-distributes by Lemma 4.3.3.

Let $\mathbb{B}_2^\omega = \nu X.F(X)$*. Theorem 4.3.3 ensures that* $(\mathbb{B}_2^\omega, \mathtt{out}_{\mathbb{B}_2^\omega})$ *is a weak final coalgebra. We can define the constant stream of* 1 *(as a Boolean number) as follows* $\mathtt{ones} = \mathtt{unfold}\ \mathbf{1}\ !(\lambda \mathtt{x} : \mathbf{1}.\mathtt{let}\ () = \mathtt{x}\ \mathtt{in}\ \langle 1, () \rangle)\ \S()$*.*

As in System F, we can define the usual operations on streams as:

$$\mathtt{head} = \lambda \mathtt{x} : \mathbb{B}_2^\omega.\mathtt{let}\ \langle \mathtt{x}_1, \mathtt{x}_2 \rangle = (\mathtt{out}_{\mathbb{B}_2^\omega}\ \mathtt{x})\ \mathtt{in}\ \mathtt{x}_1,$$
$$\mathtt{tail} = \lambda \mathtt{x} : \mathbb{B}_2^\omega.\mathtt{let}\ \langle \mathtt{x}_1, \mathtt{x}_2 \rangle = (\mathtt{out}_{\mathbb{B}_2^\omega}\ \mathtt{x})\ \mathtt{in}\ \mathtt{x}_2.$$

Unfortunately, using these operations is often inconvenient in presence of linearity, and it is more convenient to use directly the coalgebra structure provided by $\mathtt{out}_{\mathbb{B}_2^\omega}$*. Consider for example the operation that extracts from a stream of Boolean numbers the elements in even position – we have seen a similar operation encoded in System F in Example 4.3.2. We can define this operation by using the term:*

$$\mathtt{M} = \lambda \mathtt{x} : \mathbb{B}_2^\omega.\mathtt{let}\ \langle \mathtt{x}_1, \mathtt{x}_2 \rangle = (\mathtt{out}_{\mathbb{B}_2^\omega}\ \mathtt{x})\ \mathtt{in}\ \mathtt{let}\ \langle \mathtt{x}_{21}, \mathtt{x}_{22} \rangle = (\mathtt{out}_{\mathbb{B}_2^\omega}\ \mathtt{x}_2)\ \mathtt{in}\ \langle \mathtt{x}_1, \mathtt{x}_{22} \rangle.$$

$\mathtt{M}$ *has type* $\mathbb{B}_2^\omega \multimap \mathbb{B}_2 \otimes \mathbb{B}_2^\omega$*. So, by Theorem 4.3.3* $\mathtt{even} = \mathtt{unfold}_{\mathbb{B}_2^\omega}\ \mathbb{B}_2^\omega\ !\mathtt{M}$*. Another interesting example is the term computing the merge of two streams:*

$$\mathtt{M} = \lambda \mathtt{x} : \mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega.\mathtt{let}\ \langle \mathtt{x}_1, \mathtt{x}_2 \rangle = \mathtt{x}\ \mathtt{in}\ \mathtt{let}\ \langle \mathtt{x}_{11}, \mathtt{x}_{12} \rangle = (\mathtt{out}_{\mathbb{B}_2^\omega}\ \mathtt{x}_1)\ \mathtt{in}\ \langle \mathtt{x}_{11}, \langle \mathtt{x}_2, \mathtt{x}_{12} \rangle \rangle.$$

$\mathtt{M}$ *has type* $\mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega \multimap \mathbb{B}_2 \otimes (\mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega)$*. So, by Theorem 4.3.3 again,* $\mathtt{merge} = \mathtt{unfold}_{\mathbb{B}_2^\omega}\ (\mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega)\ !\mathtt{M}$*.*

We can combine algebra and coalgebra examples. For example, consider the inductive function $\mathtt{take}$ *that for a given* n *returns the first* n *elements of a stream as a string:*

$$\%\ = \lambda \mathtt{x} : \mathbf{1} \oplus (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*).\mathtt{case}\ \mathtt{x}\ \mathtt{of}$$
$$\{\mathtt{inj}_0(\mathtt{z}) \to \lambda \mathtt{y} : \mathbb{B}_2^\omega.\mathtt{nil} \mid \mathtt{inj}_1(\mathtt{z}) \to\ \lambda \mathtt{y} : \mathbb{B}_2^\omega.\mathtt{let}\ \langle \mathtt{y}_1, \mathtt{y}_2 \rangle = (\mathtt{out}_{\mathbb{B}_2^\omega}\ \mathtt{y})\ \mathtt{in}\ \mathtt{cons}(\mathtt{y}_1, \mathtt{z}\,\mathtt{y}_2)\}.$$

The term $\mathtt{M}$ *has type* $\mathbf{1} \oplus (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*) \multimap (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*)$*. Consequently, by Theorem 4.3.2,* $\mathtt{take} = \mathtt{fold}_{\mathbb{B}_2^\omega}\ (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*)\ !\mathtt{M}$*.*

Even if we cannot define a stream of the standard inductive natural numbers, we can have a stream of extended natural numbers. Let us define the latter first. Consider the functor $F(X) = \mathbf{1} \oplus X$*. By Lemma 4.3.3, we have that* F *right-distributes over* $\S$*, and so by Theorem 4.3.3 we have that* $(\overline{\mathbb{N}}, \mathtt{out}_{\overline{\mathbb{N}}})$ *is a weakly final* F*-coalgebra under* $\S$*, where* $\overline{\mathbb{N}}$ *denotes* $\nu X.F(X)$*. The inhabitants of the type* $\overline{\mathbb{N}}$ *correspond to the natural numbers extended with a limit element* ∞*. We can think about* $\mathtt{out}_{\overline{\mathbb{N}}}$ *as a predecessor function mapping* 0 *to* $()$*,* n *to* $n-1$ *and* ∞ *to* ∞*. We can define the addition of two extended natural numbers by considering the term:*

$$\mathtt{M} = \lambda \mathtt{x} : \overline{\mathbb{N}} \otimes \overline{\mathbb{N}}.\mathtt{let}\ \langle \mathtt{x}_1, \mathtt{x}_2 \rangle = \mathtt{x}\ \mathtt{in}\ \Big(\mathtt{case}\ (\mathtt{out}_{\overline{\mathbb{N}}}\,\mathtt{x}_1)\ \mathtt{of}$$
$$\mathtt{inj}_0(\mathtt{z}) \to \mathtt{case}\ (\mathtt{out}_{\overline{\mathbb{N}}}\,\mathtt{x}_2)\ \mathtt{of}\ \{\mathtt{inj}_0(\mathtt{z}') \to \mathtt{inj}_0(()) \mid \mathtt{inj}_1(\mathtt{z}') \to \mathtt{inj}_1(\langle \mathtt{z}, \mathtt{z}' \rangle)\}$$
$$\mid \mathtt{inj}_1(\mathtt{z}) \to \mathtt{inj}_1(\langle \mathtt{z}, \mathtt{x}_2 \rangle)\Big).$$

The term $\mathtt{M}$ *has type* $\overline{\mathbb{N}} \otimes \overline{\mathbb{N}} \multimap \mathbf{1} \otimes (\overline{\mathbb{N}} \otimes \overline{\mathbb{N}})$*. So, by Theorem 4.3.3* $\mathtt{add} = \mathtt{unfold}_{\overline{\mathbb{N}}}\ (\overline{\mathbb{N}} \otimes \overline{\mathbb{N}})\ !\mathtt{M}$*. For the extended natural numbers, we have a term* $\mathtt{coer}_{\overline{\mathbb{N}}} : \S\overline{\mathbb{N}} \multimap \overline{\mathbb{N}}$*, this is given by Theorem 4.3.3 as* $\mathtt{coer}_{\overline{\mathbb{N}}} = \mathtt{unfold}_{\overline{\mathbb{N}}}\ \overline{\mathbb{N}}\ !\mathtt{out}_{\overline{\mathbb{N}}}$*.*

4.4 Alternative results on streams and real numbers

In this section, we review most of the alternative results of ICC either based on stream programming languages or characterizing complexity classes over real numbers.

4.4.1 Streams, parsimonious types and non-uniform complexity classes

In [Maz14], Mazza considers an infinitary affine lambda calculus as the completion of a finitary affine lambda calculus. This language is coupled with a *parsimonious* type system to characterize the class of non-uniform polynomial time computable decision problems `P/poly`. `P/poly` is the class of decision problems that can be computed by a family of non-uniform Boolean circuits of polynomial size; a family of circuits (C_n) being uniform if there is a polynomial time algorithm computing C_n for every input n. The intuition behind parsimony is to restrict the infinitary calculus to a subset of terms with a polynomial *"modulus of continuity"*. This line of work takes inspirations from the works about infinitary lambda calculus [Maz12, Maz14] and differential lambda calculus [ER03, ER08].

We present here the characterization introduced by Mazza and Terui in [MT15] that characterizes `P/poly` and also `L/poly` under some restrictions by combining a stream language and a parsimonious type system. An alternative syntax and alternative semantics with indexes simplifying the management of parsimony properties can also be found in [Maz15].

The parsimonious calculus

The essential features of the considered stream programming language are the following. Variables are separated into two different kinds, affine and exponential variables. Exponential variables $x, y, z, \ldots$ correspond to streams. A *box* construct allows us to define infinite streams in a finite syntax: $!_f(u_0, u_1, \ldots, u_{k-1})$ which represents the stream $u_{f(0)} :: u_{f(1)} :: \cdots$ for a given function $f : \mathbb{N} \to \mathbb{N}_k$, with $\mathbb{N}_k = \{0, \ldots, k-1\}$. Terms of the language are defined by the following grammar:

$$t, u ::= \bot \mid a \mid x \mid \lambda a.t \mid t\ u \mid t \otimes u \mid !_f \overline{u} \mid t :: u \mid t[p := u],$$

where $f \in \mathbb{N} \to \mathbb{N}_k$, $k \geq 1$, and where $\overline{u}$ denotes a list of k terms $u_0, \ldots, u_{k-1}$.

Let variables $\mathbf{u}, \mathbf{v}, \ldots$ range over *boxes* of the shape $!_f \overline{u}$ that are basically stream generators. The language includes a stream constructor $::$ as well as a pair constructor $\otimes$ and the corresponding destructors/binders.

We adopt a convention similar to [ADL14] by using explicit substitutions $t[p := u]$, in which a pattern p is either a pair $a \otimes b$ or a stream $a_0 :: a_1 :: \ldots :: a_{n-1} :: x$. In this latter case, we use sometimes the notation $p(x)$ to explicitly mention the exponential variable in the stream. This makes the syntax and semantics lighter by replacing the more verbose *let in* destructor.

In what follows, let $!u$ be a shorthand for the box of one element $!_f u_0$ with $f : \mathtt{N} \to \mathtt{N}_0$ defined by $\forall n \in \mathbb{N},\ f(n) = 0$ and with $u_0 = u$. A term that does not contain non-uniform boxes is called uniform.

Example 4.4.1. *The head and tail functions on streams can be encoded by* $head = \lambda a.b[b :: x := a]$ *and* $tail = \lambda a.c[c \otimes d :=\!!x \otimes \bot][b :: x := a]$. *Notice that the use of the pair constructor and destructor in the tail program is just a syntactical trick allowing us to weaken the affine variable* b. *One would have expected a tail program of the shape* $\lambda a.!x[b :: x := a]$ *but this would require the introduction of an extra weakening rule on affine variables.*

Definition 4.4.1 (Slice). *A slice of a term is obtained by removing all components but one from each box. Let $\mathcal{S}(t)$ be the set of slices of term t defined as follows.*

$$\begin{array}{rcl}
\mathcal{S}(\alpha) & = & \{\alpha\} \qquad \text{if } \alpha \in \{a, x\} \\
\mathcal{S}(!_f(u_0, \ldots, u_{k-1})) & = & \bigcup_{i=0}^{k-1} \{!_f \nu_i ; \nu_i \in \mathcal{S}(u_i)\} \\
\mathcal{S}(t_1 \otimes t_2) & = & \{\tau_1 \otimes \tau_2 \mid \tau_1 \in \mathcal{S}(t_1), \tau_2 \in \mathcal{S}(t_2)\} \\
\mathcal{S}(t_1\ t_2) & = & \{\tau_1\ \tau_2 \mid \tau_1 \in \mathcal{S}(t_1), \tau_2 \in \mathcal{S}(t_2)\} \\
\mathcal{S}(t_1 :: t_2) & = & \{\tau_1 :: \tau_2 \mid \tau_1 \in \mathcal{S}(t_1), \tau_2 \in \mathcal{S}(t_2)\} \\
\mathcal{S}(t_1[p := t_2]) & = & \{\tau_1[p := \tau_2] \mid \tau_1 \in \mathcal{S}(t_1), \tau_2 \in \mathcal{S}(t_2)\} \\
\mathcal{S}(\lambda a.t) & = & \{\lambda a.\tau_1 \mid \tau_1 \in \mathcal{S}(t)\}
\end{array}$$

Example 4.4.2. $\mathcal{S}(!_f(x \otimes \lambda a.a, \lambda a.a \otimes x)) = \{!_f(x \otimes \lambda a.a), !_f(\lambda a.a \otimes x)\}$. *As illustrated by this example, in general, a slice does not belong to the set of (syntactically correct) terms.*

Definition 4.4.2 (Parsimonious term). *A term t is parsimonious if:*

1. *all its slices are affine, i.e. each variable occurs at most once in a slice,*
2. *box subterms do not contain free affine variables,*
3. *all exponential variables belong to a box subterm.*

Example 4.4.3. *To illustrate those points, let us see some (counter)-examples.*

- $\lambda a.a \otimes a$ *is not parsimonious because of point 1. However,* $\lambda a.(\lambda a.a)a$ *is parsimonious.*
- $!_f(x \otimes \lambda a.a, \lambda a.a \otimes x)$ *is parsimonious. Indeed the affine variables a is not free and x occurs exactly once in each slice.*
- $!_f(a, x, y, c)$ *is not parsimonious because of point 2.*
- $x \otimes !_f(y, z)$ *is not parsimonious because of point 3.*

Given a function $f : \mathbb{N} \to \mathbb{N}_k$, let f^{+i}, $i \in \mathbb{N}$, be the function in $\mathbb{N} \to \mathbb{N}_k$ defined by $\forall n \in \mathbb{N},\ f^{+i}(n) = f(n+i)$. Given a box $\mathbf{u} = !_f(u_0, \ldots, u_{k-1})$ and $k \in \mathbb{N}$, let $\mathbf{u}^{+k}$ be the term equal to $!_{f^{+k}}(u_0, \ldots, u_{k-1})$.

Semantics

The one-step reduction corresponding to terms of the language is defined in Figure 4.7 relatively to a finite set sigma σ of stream patterns, *i.e.* $\sigma = \{a_1 :: x_1, \ldots, a_n :: x_n\}$. Let $\mathcal{V}(\sigma)$ be equal to $\{a_1, x_1, \ldots, a_n, x_n\}$.

As described in [MT15], the substitution $\{\{\mathbf{u}/x\}\}$ is non standard on boxes: if x occurs in a box $\mathbf{w} = !_g(w_0, \ldots, w_{l-1})$ and $\mathbf{u} = !_f(u_0, \ldots, u_{k-1})$ then $\mathbf{w}\{\{\mathbf{u}/x\}\} = !_h(v_0, \ldots, v_{lk-1})$ with $v_{ik+j} = w_i\{u_j/x\}$ and $h(n) = g(n)k + f(n)$. Moreover, in the rule (dup) and (aux), it is mandatory that $\{x_1, \ldots, x_n\}$ are precisely the free exponential variables of $u_{f(0)}$. Finally, in the (aux) rule, the notation $t[\mathbf{u}]$ means that the box $\mathbf{u}$ is a subterm of t. If not, then the rules becomes $t[x := v :: w] \overset{\{b_1::x_1,\ldots,b_n::x_n\}}{\to} t$.

As usual, a *context* is a term with at most one occurrence of a special symbol $\langle\cdot\rangle$, called *hole*. We denote by $C\langle t\rangle$ the result of substituting the term t to the hole in C, an operation which may capture variables. The one-step reduction rules are extended contextually: if $t \overset{\sigma}{\to} u$ and C does not bind variables of $\mathcal{V}(\sigma)$ then $C\langle t\rangle \overset{\sigma}{\to} C\langle u\rangle$. If $t \overset{\sigma\cup\{b::x\}}{\to} u$ then $t[p(x) := v] \overset{\sigma}{\to} u[p(b :: x) := v]$.

The system reduction is defined to be the reflexive and transitive closure of $\overset{\emptyset}{\to}$.

$$(\lambda a.t)\ u \xrightarrow{\emptyset} t\{u/a\} \qquad \text{(beta)}$$

$$t[x := \mathbf{u}] \xrightarrow{\emptyset} t\{\{\mathbf{u}/x\}\} \qquad \text{(merge)}$$

$$t[p := v]\ u \xrightarrow{\emptyset} (t\ u)[p := v] \qquad (\text{com}_1)$$

$$t[p := u[q := v]] \xrightarrow{\emptyset} (t[p := u])[q := v] \qquad (\text{com}_2)$$

$$t[a :: p := u :: v] \xrightarrow{\emptyset} t\{u/a\}[p := v] \qquad \text{(cons)}$$

$$t[a \otimes b := u \otimes v] \xrightarrow{\emptyset} t\{u/a, v/b\} \qquad \text{(pair)}$$

$$t[a :: p := \mathbf{u}] \xrightarrow{\{b_1::x_1,\dots,b_n::x_n\}} t\{u_{f(0)}\{b_1/x_1,\dots,b_n/x_n\}/a\}[p := \mathbf{u}^{+1}] \qquad \text{(dup)}$$

$$t[\mathbf{u}][x := v :: w] \xrightarrow{\{b_1::x_1,\dots,b_n::x_n\}} t[u_{f(0)}\{b_1/x_1,\dots,b_n/x_n, v/x\} :: \mathbf{u}^{+1}][x := w] \qquad \text{(aux)}$$

Figure 4.7: One-step reduction of the parsimonious calculus

Example 4.4.4. *Consider the terms head and tail of Example 4.4.1 and let* $!_f(\underline{0},\underline{1})$ *be a stream of Boolean numbers for some function* $f : \mathtt{N} \to \{0,1\}$ *such that* $f(2k) = 0$ *and* $f(2k+1) = 1$*, for each* $k \in \mathtt{N}$*, i.e.* $!_f(\underline{0},\underline{1}) = \underline{0} :: \underline{1} :: \underline{0} :: \underline{1} :: \dots$.

As $head = \lambda a.b[b :: x := a]$*, we have the following reduction.*

$$\begin{aligned} head\ !_f(\underline{0},\underline{1}) &\xrightarrow{\emptyset} b[b :: x := !_f(\underline{0},\underline{1})] && (beta)\\ &\xrightarrow{\emptyset} \underline{0}[x := !_{f^{+1}}(\underline{0},\underline{1})] && (dup) \end{aligned}$$

As $tail = \lambda a.c[c \otimes d :=!x \otimes \bot][b :: x := a]$*, we have the following reduction.*

$$\begin{aligned} tail\ !_f(\underline{0},\underline{1}) &\xrightarrow{\emptyset} c[c \otimes d :=!x \otimes \bot][b :: x := !_f(\underline{0},\underline{1})] && (beta)\\ &\xrightarrow{\emptyset} !x[b :: x := !_f(\underline{0},\underline{1})] && (pair)\\ &\xrightarrow{\emptyset} !x[x := !_{f^{+1}}(\underline{0},\underline{1})] && (dup)\\ &\xrightarrow{\emptyset} !x\{\{!_{f^{+1}}(\underline{0},\underline{1})/x\}\} =!_{f^{+1}}(\underline{0},\underline{1}) && (merge) \end{aligned}$$

The last equality is obtained by definition of non standard substitution. Indeed, in this context, suppose that $!_h(v_0,\dots,v_{lk-1}) =!x\{\{!_{f^{+1}}(\underline{0},\underline{1})/x\}\}$*. We have* $h(n) = g(m)k + f^{+1}(n)$*, with* $g(m) = 0$*,* g *being the function attached with the uniform box* $!x$*, and, consequently,* $h(n) = f^{+1}(n)$ *and* $v_{ik+j} = w_i\{u_j/x\} = x\{u_j/x\}$ *with* $u_0 = \underline{0}$ *and* $u_1 = \underline{1}$.

Parsimonious types

Types are defined inductively by:

$$A, B ::= \alpha \mid A \multimap B \mid A \otimes B \mid\ !A \mid \forall\alpha.A,$$

α being a type variable. As expected, the type $!A$ is for stream data of type A. The type system, $\mathbf{nuPL}_{\forall\ell}$, is adapted from the uniform type system is defined in Figure 4.8. Typing judgments are of the form $\Gamma;\Delta \vdash t : A$ with Δ a typing environment for affine variables and Γ a typing

$$\frac{}{\Gamma;\Delta,a:A\vdash a:A}\ (\mathtt{ax}) \qquad \frac{}{\Gamma;\Delta\vdash \bot:A}\ (\mathtt{coweak})$$

$$\frac{\Gamma;\Delta,a:A\vdash t:B}{\Gamma;\Delta\vdash \lambda a.t:A\multimap B}\ (\multimap\mathtt{I}) \qquad \frac{\Gamma;\Delta\vdash t:A\multimap B \quad \Gamma';\Delta'\vdash u:A}{\Gamma,\Gamma';\Delta,\Delta'\vdash t\ u:A}\ (\multimap\mathtt{E})$$

$$\frac{\Gamma;\Delta\vdash t:A \quad \Gamma';\Delta'\vdash u:B}{\Gamma,\Gamma';\Delta,\Delta'\vdash t\otimes u:A\otimes B}\ (\otimes\mathtt{I}) \qquad \frac{\Gamma;\Delta\vdash u:A\otimes B \quad \Gamma';\Delta',a:A,b:B\vdash t:C}{\Gamma,\Gamma';\Delta,\Delta'\vdash t[a\otimes b:=u]:C}\ (\otimes\mathtt{E})$$

$$\frac{\Gamma,p:A;\Delta,a:A\vdash t:B}{\Gamma,a::p:A;\Delta\vdash t:B}\ (\mathtt{abs}) \qquad \frac{\Gamma;\Delta\vdash t:A \quad \Gamma';\Delta'\vdash u:!A}{\Gamma,\Gamma';\Delta,\Delta'\vdash t::u:!A}\ (\mathtt{coabs})$$

$$\frac{;\overline{a}:\overline{A}\vdash \overline{u}_0:A \quad \dots \quad ;\overline{a}:\overline{A}\vdash \overline{u}_{k-1}:A}{\overline{x}:\overline{A};\vdash !_f\overline{u}\{\overline{x}/\overline{a}\}:!A}\ (!\mathtt{I}) \qquad \frac{\Gamma;\Delta\vdash u:!A \quad \Gamma',p:A;\Delta'\vdash t:B}{\Gamma,\Gamma';\Delta,\Delta'\vdash t[p:=u]:B}\ (!\mathtt{E})$$

$$\frac{\Gamma;\Delta\vdash t:A \quad \alpha\notin FV(\Gamma\cup\Delta)}{\Gamma;\Delta\vdash t:\forall\alpha.A}\ (\forall\mathtt{I}) \qquad \frac{\Gamma;\Delta\vdash t:\forall\alpha.A}{\Gamma;\Delta\vdash t:A[B/\alpha] \quad B \text{ is } !\text{-free}}\ (\forall\mathtt{E})$$

Figure 4.8: Non-uniform parsimonious logic type system

environment for patterns $p(x)$; the variables in Γ and Δ being distinct. In the ($!\mathtt{I}$) rule of Figure 4.8, the variables $\overline{x}$ are fresh. It implies that they do not appear free in each u_i. The type system, **nuPL** is the restriction of $\mathbf{nuPL}_{\forall\ell}$ where rules ($\forall\mathtt{I}$) and ($\forall\mathtt{E}$) are withdrawn.

Example 4.4.5. *The tail program of Example 4.4.1 can be typed in* $\boldsymbol{nuPL}_{\forall\ell}$ *as follows.*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{;e:A\vdash e:A}\,(ax)}{x:A;\vdash !x:!A}\,(!I) \quad \dfrac{}{;\vdash \bot:A}\,(coweak)}{x:A;\vdash !x\otimes\bot:!A\otimes A}\,(!I) \quad \dfrac{}{;c:!A,d:A,b:A\vdash c:!A}\,(ax)}{x:A;b:A\vdash c[c\otimes d:=!x\otimes\bot]:!A}\,(\otimes E) \quad \dfrac{}{;a:!A\vdash a:!A}\,(ax)}{b::x:A;\vdash c[c\otimes d:=!x\otimes\bot]:!A}\,(abs)}{;a:!A\vdash c[c\otimes d:=!x\otimes\bot][b::x:=a]:!A}\,(!E)}{;\vdash \lambda a.c[c\otimes d:=!x\otimes\bot][b::x:=a]:!A\multimap !A}\,(\multimap I)$$

Main results

The type system implies that each typable term is parsimonious. The converse obviously does not hold.

Proposition 4.4.1 ([MT15])**.** *Given a term t, if there are a typing environment Δ and a type A such that $;\Delta\vdash t:A$ then t is parsimonious.*

Hence typable terms have a behavior with a polynomial *"modulus of continuity"*. The restriction to **nuPL** avoid the programmer from being able to compute a simple iteration scheme within the language. Let $[\![\mathbf{nuPL}]\!]$ and $[\![\mathbf{nuPL}_{\forall\ell}]\!]$ be the classes of language decidable by terms typable in **nuPL** and $\mathbf{nuPL}_{\forall\ell}$, respectively.

Theorem 4.4.1 ([MT15]). $[\![\ \mathbf{nuPL}]\!] = \mathtt{P/poly}$ *and* $[\![\ \mathbf{nuPL}_{\forall\ell}]\!] = \mathtt{L/poly}$.

This type discipline was also adapted in [Maz15] to characterize `Logspace` by restricting the language to uniform boxes (mainly constant stream). The parsimonious methodology is very convenient to capture small complexity classes with programs computing over streams. One of its drawback is that its computations on stream behave as a transducer. Hence each stream computation only depends on a finite portion of the input streams, which makes the calculus not straightforwardly adaptable to characterize polynomial time over the reals or higher-order complexity classes such as $\mathtt{BFF}_2$.

4.4.2 Function algebra characterizations of polynomial time over the reals

An interesting function algebra based characterization of the functions computable in polynomial time over real numbers based on Bellantoni and Cook function algebra (see Theorem 1.2.2) has been provided by Bournez, Hainry, and Gomaa [BGH11]. This characterization does not focus precisely on the class $\mathtt{FP}(\mathbb{R})$ introduced in Definition 4.2.10 but on a strict subset $\mathtt{FP}(\mathbb{R})\cap([0,1]\to\mathbb{R}^+)$, the class of functions from $[0,1]$ to $\mathbb{R}^+$ that are computable by a Machine in polynomial time.

Definition 4.4.3. *Let $\mathcal{W}$ be the least class of functions containing the constant functions 0, 1, the binary addition $+(;x,y)=x+y$ and binary substraction $-(;x,y)=x-y$, the projection functions $\pi^n_j(;x_1,\dots,x_n)=x_j$, the conditional function c defined by $c(;x,y,z)=xy+(1-x)z$, the continuous parity function $par(x;)=\max(0,2/(\pi\sin(\pi x))$, and the continuous predecessor function $p(x;)=\int_0^{x-1} par(t;)dt$, and closed under safe composition (`SCOMP`) and Safe Integration (`SI`):*

$$\begin{aligned}
\mathtt{SCOMP}(f,\overline{g},\overline{h})(\overline{x};\overline{y}) &= f(\overline{g}(\overline{x};);\overline{h}(\overline{x};\overline{y})),\\
\mathtt{SI}(f,h_0,h_1)(0,\overline{y};\overline{z}) &= f(\overline{y};\overline{z}),\\
\partial_x\mathtt{SI}(f,h_0,h_1)(x,\overline{y};\overline{z}) &= par(x;)[h_1(p(x;),\overline{y};\overline{z},\mathtt{SI}(f,h_0,h_1)(p(x;),\overline{y};\overline{z}))-\mathtt{SI}(f,h_0,h_1)(2p(x;),\overline{y};\overline{z})]\\
&\quad + par(x-1;)[h_0(p'(x;),\overline{y};\overline{z},\mathtt{SI}(f,h_0,h_1)(p'(x;),\overline{y};\overline{z}))-\mathtt{SI}(f,h_0,h_1)(2p'(x;),\overline{y};\overline{z})],
\end{aligned}$$

with $p'(x;)=p(x-1;)+1$. In other words, $\mathcal{W}=[0,1,+,-,\pi^n_j,c,par,p;\mathtt{SCOMP},\mathtt{SI}]$.

$\mathcal{W}$ is a direct extension of the BC algebra to the real numbers. Indeed, the consumption of one bit i in the BC algebra is replaced by one application of the continuous predecessor function p and the choice of the contextual function h_i is simulated using the parity function *par*. As a consequence, the basic functions are total functions over $\mathbb{R}^+$ and hence any function of the algebra is a total function over $\mathbb{R}^+$. Moreover, the class $\mathcal{W}$ preserves integers (and natural numbers) and, consequently, its restriction to natural numbers provides a characterization of `FP`.

Theorem 4.4.2 ([BGH11]). $\mathcal{W}\cap(\mathbb{N}\to\mathbb{N})=\mathtt{FP}$.

It is (strictly) included in the class of polynomial time computable functions over the real numbers.

Theorem 4.4.3 ([BGH11]). $\mathcal{W}\subsetneq\mathtt{FP}(\mathbb{R})$.

The characterization of $\mathtt{FP}(\mathbb{R}) \cap ([0,1] \to \mathbb{R}^+)$ is obtained as follows.

Theorem 4.4.4 ([BGH11]). *$\mathtt{FP}(\mathbb{R}) \cap ([0,1] \to \mathbb{R}^+)$ is exactly the the class of functions that either are Lipschitz and $\mathcal{W}$-definable or that are n^k-smooth and n^k-$\mathcal{W}$-definable.*

Remember that a function f is Lipschitz if there exists a constant $k \geq 0$ such that for all x, y, $|f(x) - f(y)| \leq k|x - y|$. In the above theorem, we have made an arbitrary choice not to describe the notions of definability and smoothness. Definability is related to function approximation and n^k-smoothness is related to a polynomial modulus of continuity. The full details can be found in [BGH11].

It is worth noticing that related works [BH04, CO07] provide also interesting function algebra characterizations of recursive functions (*i.e.* computable functions) over real numbers.

4.4.3 The BSS model

The papers [BCJdNM05, BCJdNM06] provide function algebra characterizations of complexity classes in the Blum-Shub-Smale (BSS) model, which is another model for describing computations over real numbers and their complexity.

The BSS model [BSS88] consists in RAMs, whose registers can store arbitrary real numbers, that compute constant time arithmetic operations on real numbers following a finite list of instructions. Such machines take inputs from $\mathbb{R}^\infty = \cup_{n=1}^\infty \mathbb{R}^n$ and halt by returning an output in $\mathbb{R}^\infty$ or loop forever. In what follows, let a be an element of $\mathbb{R}$, let $\overline{x}$ be an element of $\mathbb{R}^\infty$ and let $a.\overline{x}$ be the element of $\mathbb{R}^\infty$ whose first component is a followed by $\overline{x}$. Let ϵ be the empty tuple.

Let M be a BSS machine and $[\![M]\!]$ be the function associating each input in $\mathbb{R}^\infty$ to the corresponding output in $\mathbb{R}^\infty$. A function $f : (\mathbb{R}^\infty)^k \to \mathbb{R}^\infty$ is computable in the BSS model if there exists a machine M such that $f = [\![M]\!]$.

A BSS machine runs in polynomial time if there is a polynomial $P \in \mathbb{N}[X]$ such that for any input $\overline{x} \in \mathbb{R}^\infty$ the machine produces an output in $P(|\overline{x}|)$ steps.[29] Let $\mathtt{FP}_\mathbb{R}$ be the class of functions computable by BSS machines in polynomial time.

Theorem 4.4.5 ([BCJdNM05][30]). *The least class of functions containing the constant functions 0 and 1, the head, tail and cons functions over sequences of real numbers defined by $head(;a.\overline{x}) = a$, $tail(;a.\overline{x}) = \overline{x}$, $head(;\epsilon) = tail(;\epsilon) = \epsilon$, $cons(;a.\overline{x},\overline{y}) = a.\overline{y}$, and $cons(;\epsilon,\overline{y}) = \overline{y}$, the projection functions $\pi_i^n(\overline{x_1},\ldots,\overline{x_n}) = \overline{x_i}$, $i \in [1,n]$, the arithmetic operations $\otimes$, $\oplus$ and $\leq$ defined by $\oplus(;a.\overline{x},b.\overline{y}) = (a+b).\overline{y}$, $\otimes(;a.\overline{x},b.\overline{y}) = (a \times b).\overline{y}$, $\leq(;a.\overline{x},b.\overline{y}) = i.\overline{y}$, with $i = 1$ if $a \leq b$, and $i = 0$ otherwise, the conditional function $C(;a.\overline{x},\overline{y},\overline{z}) = \overline{y}$, if $a = 1$, $C(;a.\overline{x},\overline{y},\overline{z}) = \overline{z}$ otherwise, and closed under safe composition* ($\mathtt{SCOMP}$) *and Safe Recursion* ($\mathtt{SR}$)*:*

$$\begin{aligned}
\mathtt{SCOMP}(f,\overline{g},\overline{h})(\overline{x};\overline{y}) &= f(\overline{g}(\overline{x};);\overline{h}(\overline{x};\overline{y})),\\
\mathtt{SR}(f,g)(\epsilon,\overline{y};\overline{z}) &= g(\overline{y};\overline{z}),\\
\mathtt{SR}(f,g)(a.\overline{x},\overline{y};\overline{z}) &= f(\overline{x},\overline{y};\overline{z},\mathtt{SR}(f,g)(\overline{x},\overline{y};\overline{z})),
\end{aligned}$$

is exactly $\mathtt{FP}_\mathbb{R}$. In other words, $[0,1,head,tail,cons,\pi_i^n,\otimes,\oplus,\leq,C;\mathtt{SCOMP},\mathtt{SR}] = \mathtt{FP}_\mathbb{R}$.

[29] The size of $\overline{x}$ is the dimension of the tuple.

[30] Here we consider $\mathbb{R}$ with a ring structure. Hence substraction and division are not included. They could have been considered as the result of [BCJdNM05] holds for arbitrary structures. In this case, for Theorem 4.4.5 to hold, substraction and division have to be added as basic operations in the corresponding function algebra. $+$ and $\times$ denote respectively the standard addition and standard multiplication operations on real numbers.

Characterizations of other interesting complexity classes based on the BSS model such as parallel polynomial time and classes of the polynomial hierarchy can be found in [BCJdNM05] and [BCJdNM06], respectively. For the interested reader, a characterization of $\mathtt{FP}_{\mathbb{R}}$ based on LAL was also studied in [BP06].

Chapter 5

Research perspectives

This chapter presents the research directions in ICC that I would like to explore in the future or that, in my opinion, correspond to the main research issues that should be addressed in the next decades. In Section 5.1 and Section 5.2, I mention some open issues and research directions in the context of probabilistic programs and quantum computing, respectively. In Section 5.3, I discuss the open issues related to languages for real numbers computations and with complexity certificates. In Section 5.4, I mention the issue of finding a decidable theory for type-2 polynomial time complexity. Finally, in Section 5.5, I discuss two typing disciplines of interest, sized types and intersection types, that are used to characterize termination of some lambda-calculi and mention some related open issues of interest from a complexity viewpoint.

5.1 Probabilistic models

The notion of probabilistic programming language (and model) has had a renewed popularity due to the need for such applications in algorithmics, robotics, and cryptography. Current probabilistic languages consist in higher-order functional languages with sampling and conditioning instructions.[31]

Like classical programs, probabilistic programs are in need of tools and techniques for formalizing and reasoning on their semantics. Several semantics aspect of probabilistic programs have already been studied (denotational semantics [Koz79], operational semantics [BDLGS16], uniqueness, normalization, confluence, and standardization of the lambda calculus with choice [Fag19, FRDR19], ...). They also require some verification techniques to be developed to certify properties such as termination. Termination with probability 1 is called, *almost sure termination* and if, in addition, the mean length of a derivation is finite then it is called *positive almost sure termination*. This latter notion has been deeply studied for TRSs with probabilistic rewrite rules by [BG05, BG06, Gna07, ADLY18] and almost sure termination for probabilistic higher-order languages has been studied in [KDLG19, DLG19] .

There are only a few works for verification techniques based on ICC methods. [DLT15] provides a first characterization of PP, the class of decision problems solvable by a probabilistic TM in polynomial time, with an error probability of less than $1/2$ for all instances. A less implicit extension to BPP, the class of decision problems solvable by a probabilistic TM in polynomial time with an error probability less than $1/3$ for all instances, is also considered. Although pioneering,

[31] Conditioning consists in allowing to add information about observed events into the program that may influence the posterior probability distribution.

this work is not satisfactory insofar as the PP class is not really interesting in this context (as error is too close from success) and the characterization of BPP is not sufficiently implicit.

More recently, Avanzini, Moser, and Schaper have developed an average case runtime analysis for imperative languages [AMS19]. It is inspired by the ert-calculus [KKMO16], a method allowing to infer expected runtimes of probabilistic programs. The paper [ADLG19] also provides a sound and complete average case runtime analysis of higher-order functional programs with respect to the average case polytime TMs. Intuitively, the restriction of this class to decision problems is strictly included in ZPP, the class of zero-error polynomial time probabilistic programs, as the non-polynomial executions may be transformed into errors and, consequently, this work is an interesting improvement on existing methods.

As the characterization of [ADLG19] relies on an affine type system, it would be of interest to study extensions of the tiering method in an imperative and probabilistic setting in order to characterize this complexity class and to study to which extent a pure ICC characterization of ZPP can be obtained.

5.2 Quantum computations

Quantum computing is a computational paradigm which takes advantage of quantum mechanics to perform computations. An important number of quantum algorithms (Shor's [Sho97], Grover's [Gro96],...) have been shown to be more efficient than their classical counterpart in terms of time complexity. Several interesting works on the semantics properties of quantum programs have been carried out in recent years [Sel04]. However, while there are plenty of tools to analyze automatically the complexity of classical programs, only a few works have been carried out for quantum programs.

The work by Dal Lago et al. [DLMZ10], presented in Subsection 3.2.3, was to our knowledge the first application of ICC techniques to the quantum paradigm. However, it is not fully satisfactory for two main reasons. First, as already mentioned in Subsection 3.2.3, measurements are not allowed as a basic construct within the programming language. Consequently, this work cannot be straightforwardly adapted to mainstream quantum programming languages such as QPL (Quantum Programming Language) of [Sel04]. Second, most semantics restrictions on the definition of the captured quantum complexity classes are also holding on the notion of computed function or accepted language (*e.g.* Definition 3.2.2) and, hence, this weakens the purity of the characterization. One solution might be to consider the (unpublished) work of [Yam18] which provides the first function algebra characterization of the quantum complexity class (F)BQP. In this work, the semantics requirements are less external in the sense that a function is in FBQP if and only if it can be *"approximated"* polynomially by some function definable in the function algebra. Here the weakness is the extra requirement of this external polynomial that one could get rid off using standard safe-like function algebra. Another study of interest would then be to see whether the tiering approach can be adapted to a fragment of QPL to characterize similar complexity classes.

Last but not least, a well-suited complexity analysis of quantum programs should take into account the entanglement of qubits during program execution. Indeed, most of the algorithms improving the complexity of their classical counterpart (e.g. [Sho97, Gro96]) make use of entanglement to improve the algorithmic speed. Consequently, entanglement seems to be correlated to this speed up and an adequate study of quantum programs complexity should take a measure of entanglement into account. One possible direction is to consider the interesting work of Perdrix which has developed an abstract interpretations based static analysis for getting an

approximation of entangled qubits [Per08].

5.3 Feasible computations over the reals

Some characterizations of the class of real functions computable in polynomial time $\mathtt{FP}(\mathbb{R})$ have been provided in Section 4.2 and Section 4.4 ([FHHP10, BGH11, FHHP15]). They have also shown to be equivalent to other well-know computational models over the reals such as the General Purpose Analog Computer (GPAC)[BGP16]. However, these characterizations suffer from several drawbacks: either they provide a non-natural way to write programs (*i.e.* function algebra with non standard operators and recursion schemata) or they involve criteria that are not tractable (type-2 higher-order polynomials over stream programs).

Characterizing the complexity class $\mathtt{FP}(\mathbb{R})$ is still challenging as, by soundness, programs computing functions of $\mathtt{FP}(\mathbb{R})$ correspond to programs that can give an approximation of the output at precision n in polynomial time in n, which is the intuitive and rational way to understand the complexity of a function over real numbers. For being effective, works characterizing the class $\mathtt{FP}(\mathbb{R})$ should not constrain the programmer in their way of writing down programs but should rather provide a certificate (type) that the program has the good complexity properties. Hence one issue of interest is to consider a standard functional programming language on streams (*e.g.* streams of signed digits) encoding real numbers and to develop a type system sound and complete for $\mathtt{FP}(\mathbb{R})$. One suggestion towards the completion of this work would be to consider the DLAL system of Baillot and Terui introduced in Section 2.1 and to combine it with stream data with good properties such as productivity [Sev17, AM13a, CBGB15].

5.4 A decidable theory for type-2 polynomial time

As mentioned in Subsection 4.2.1, several characterizations of higher complexity, more precisely, type-2 polynomial time complexity were studied in the last decades. All these characterizations suffer from three main problems. They rely on some external or explicit bounds [Con73, Meh76, CK90, IRK01, KS18, KS19]. Consequently, programming in such a paradigm is very hard as program behaviors are unpredictable (in particular the oracles answers cannot be predicted), or they are using machines or non natural function scheme and cannot be used in practice [Coo92, CU93], or they are in need of an undecidable check on constraints (e.g., inequalities over higher-order polynomials) [KC91, KC96, HP17].

A main issue of interest for this complexity theory is hence to provide a tractable characterization based on realistic programming language. One direction is to consider the characterization of [KS18], showing that Oracle Polynomial Time (OPT) (see [Coo92]), together with a restriction on the number of *lookahead revisions* (the number of time the size of an oracle input can increase is bounded by a constant), characterize $\mathtt{BFF}_2$ (under some lambda closure). Such a semantics property could be enforced by techniques such as tiering or light affine type systems and, hence, would allow us to characterize $\mathtt{BFF}_2$ in a tractable manner on a realistic programming (imperative or functional) language.

5.5 Other typing disciplines

Several other typing techniques have been used for termination and complexity purposes.

One of the most successful techniques is *sized types* that have been used to prove correctness properties of reactive systems [HPS96], size based termination [CK01, BGR08], probabilistic termination [DLG19], and space upper bounds [Vas08].

The line of work on the use of dependent types [DLG11, DLP14] for inferring program complexity properties is related to this approach. Indeed, as mentioned by Dal Lago, *"linear dependent types can be seen as a way to inject precision and linearity into sized types"*.

Sized types have been adapted to functional languages to infer resource upper bounds in practice [ADL17]. This methodology can be fully automated and a relative completeness result is also stated but it merely focuses on soundness properties and on extending the expressive power (some examples of programs that were not captured are provided and no characterization of complexity class is studied). In [BG18], sized types have been combined with light linear logic to characterize the full Grzegorczyk's hierarchy, including `FP`. One question of interest is whether such a technique could also be combined to other ICC techniques such as tiering and interpretation (at least for first order). Another question of interest is also whether the work of [BG18] could be transferred to other complexity classes (polynomial space and subpolynomial classes).

Another technique of interest is *intersection types*. Intersection types have been used to show strong normalization of the lambda-calculus for the non-idempotent framework (i.e. provided that the intersection of a type with itself is not the identity) [DC05, BL11, BKV17, DC18]. Characterizations of head normalizing terms and strongly normalizing terms have also been adapted to non-idempotent intersection types and union types of the lambda-mu-calculus in [KV17]. In this setting upper bounds can be derived on the (head)-reduction length. Intersection types have also been applied to the setting of the probabilistic lambda-calculus [BDL18], where the probability of a term to terminate is characterized as the least upper bound of the "weight" of its typing derivations. Non-idempotent intersection types are used for characterizing terms normalizing in elementary time in [BRDR15]. For that purpose, the considered types are also non-associative. The non-associativity entails a stratification that is reminiscent of the stratification in light logics. In this framework, characterizations of the most popular complexity classes remain open issues.

Glossary

`Alogtime` : U_E^* uniform $\mathtt{NC}^1$
BC : Bellantoni and Cook function algebra
`BFF` : Basic Feasible Functionals
$\mathtt{BFF}_2$: Basic Feasible Functionals at order 2
BLL : Bounded Linear Logic
`BQP` : Bounded error Quantum Polynomial time
`BPP` : The class of problems computable in polytime with bounded-error by a probabilistic TM
BSS : Blum-Shub-Smale model
BTLP : Bounded Type Loop Programs
CBN : Call-By-Name
CBV : Call-By-Value
CPS : Continuation-Passing Style
DLAL : Dual Light Affine Logic
DP : Dependency Pair
DPG : Dependency Pair Graph
DPI : Dependency Pair Interpretation
EAL : Elementary Affine Logic
ELL : Elementary Linear Logic
`EQP` : Exact Quantum Polynomial time
`EXPTIME` : The class of problems computable in exponential time by a deterministic TM
`FBQP` : Functional BQP
$\mathtt{FP}(\mathbb{R})$: The class of functions computable in polynomial time over the reals
$\mathtt{FP}_{\mathbb{R}}$: The class of functions computable by BSS machines in polynomial time
FMT : Finite Model Theory
`FP` : The class of functions computable in polynomial time by a deterministic TM
`FPSPACE` : The class of functions computable in polynomial space by a deterministic TM
GPAC : General Purpose Analog Computer
HO : Higher-Order
HOP : Higher-Order Polynomial
I : Interpretation
ICC : Implicit Computational Complexity
`L` : The class of problems computable in logarithmic space by a deterministic TM
`L/poly` : The class of decision problems computable by branching programs of polynomial size
LAL : Light Affine Logic
LALC : Light Affine Lambda Calculus
LL : Linear Logic
LLL : Light Linear Logic
LLPO : Light Lexicographic Path Ordering

`Logspace` : `L`
LPO : Lexicographic Path Ordering
MLSTA : ML Soft Type Assignment
MPO : Multiset Path Ordering
$\mathtt{NC}^k$: Decision problems computable by U_E^* uniform circuits of polynomial size and $\mathtt{log}^k$ depth
$\mathtt{NC}$: $\cup_{k\geq 0}\mathtt{NC}^k$
`NP` : The class of decision problems computable in polynomial time by a non deterministic TM
OTM : Oracle Turing Machine
OO : Object Oriented
OPT : Oracle Polynomial Time
`P` : The class of decision problems computable in polynomial time by a deterministic TM
`P/poly` : The class of decision problems computable by families of polynomial size circuits
PCF : The language of Programming Computable Functions
POP : Product Path Ordering
POP* : Polynomial Path Ordering
`PP` : The class of decision problems computable in polynomial time by a probabilistic TM
`PSPACE` : The class of problems computable in polynomial space by a deterministic TM
QBF :Quantified Boolean Formula
QI : Quasi-interpretation
QPL : Quantum Programming Language
QTM : Quantum Turing Machine
RAM : Random Access Machine
RPO : Recursive Path Ordering
SAL : Soft Affine Logic
SAT : Satisfiability problem
SCP : Size Change Principle
SI : Sup-interpretation
SLL : Soft Linear Logic
SMT : Satisfiability Modulo Theory
sPOP* : small Polynomial Path Ordering
STA : Soft Type Assignment
STTRS : Simply Typed TRS
TM : Turing Machine
TRS : Term Rewrite System / Term Rewriting System
$\mathbf{U}_{\mathbf{E}}^*$: A condition of uniformity for families of Boolean circuits (see [Ruz81])
`ZPP` : Zero-error Probabilistic Polynomial time
`ZQP` : Zero-error Quantum Polynomial time

List of Figures

Bibliography

[AAB+13] Roberto M. Amadio, Nicolas Ayache, François Bobot, et al. Certified Complexity (CerCo). In *FOPARA*, pages 1–18. Springer, 2013.

[AAG+07a] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost analysis of Java bytecode. In *ESOP*, pages 157–172. Springer, 2007.

[AAG+07b] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. COSTA: design and implementation of a cost and termination analyzer for Java bytecode. In *FMCO*, pages 113–132. Springer, 2007.

[AAG+12] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

[AARG12] Nicolas Ayache, Roberto M. Amadio, and Yann Régis-Gianas. Certifying and reasoning on cost annotations in C programs. In *FMICS*, pages 32–46. Springer, 2012.

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[ABT07] Vincent Atassi, Patrick Baillot, and Kazushige Terui. Verification of `Ptime` reducibility for system F terms: type inference in dual light affine logic. *Logical Methods in Computer Science*, 3(4), 2007.

[ACGDZJ04] Roberto M. Amadio, Solange Coupet-Grimal, Silvano Dal Zilio, and Line Jakubiec. A functional scenario for bytecode verification of resource bounds. In *CSL*, pages 265–279. Springer, 2004.

[ACJ+18] Elvira Albert, Jesús Correas, Einar Broch Johnsen, Ka I Pun, and Guillermo Román-Díez. Parallel cost analysis. *Transactions on Computational Logic*, 19(4):31, 2018.

[AD06] Roberto M. Amadio and Frédéric Dabrowski. Feasible reactivity for synchronous cooperative threads. *Electronic Notes in Theoretical Computer Science*, 154(3):33–43, 2006.

[ADL14] Beniamino Accattoli and Ugo Dal Lago. Beta reduction is invariant, indeed. In *CSL and LICS*, number 8, pages 1–10. ACM, 2014.

[ADL17] Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. In *ICFP*, volume 1, pages 1–29. ACM, 2017.

[ADL18] Martin Avanzini and Ugo Dal Lago. On sharing, memoization, and polynomial time. *Information and Computation*, 261:3–22, 2018.

[ADLG19] Martin Avanzini, Ugo Dal Lago, and Alexis Ghyselen. Type-based complexity analysis of probabilistic functional programs. In *LICS*, pages 1–13. IEEE, 2019.

[ADLM15] Martin Avanzini, Ugo Dal Lago, and Georg Moser. Analysing the complexity of functional programs: higher-order meets first-order. In *ICFP*, volume 50, pages 152–164. ACM, 2015.

[ADLY18] Martin Avanzini, Ugo Dal Lago, and Akihisa Yamada. On probabilistic term rewriting. In *FLOPS*, pages 132–148. Springer, 2018.

[ADZ04] Roberto M. Amadio and Silvano Dal Zilio. Resource control for synchronous cooperative threads. In *CONCUR*, pages 68–82. Springer, 2004.

[AEM15] Martin Avanzini, Naohi Eguchi, and Georg Moser. A new order-theoretic characterisation of the polytime computable functions. *Theoretical Computer Science*, 585:3–24, 2015.

[AFRD15] Elvira Albert, Jesús Correas Fernández, and Guillermo Román-Díez. Non-cumulative resource analysis. In *TACAS*, pages 85–100. Springer, 2015.

[AG00] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

[AGGZ07] Elvira Albert, Samir Genaim, and Miguel Gomez-Zamalloa. Heap space analysis for Java bytecode. In *ISMM*, pages 105–116. ACM, 2007.

[AGGZ13] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Heap space analysis for garbage collected languages. *Science of Computer Programming*, 78(9):1427–1448, 2013.

[AKM09] James Avery, Lars Kristiansen, and Jean-Yves Moyen. Static complexity analysis of higher order programs. In *FOPARA*, pages 84–99. Springer, 2009.

[All91] Bill Allen. Arithmetizing uniform `NC`. *Annals of Pure and Applied Logic*, 53(1):1–50, 1991.

[AM08] Martin Avanzini and Georg Moser. Complexity analysis by rewriting. In *FLOPS*, pages 130–146. Springer, 2008.

[AM09] Martin Avanzini and Georg Moser. Dependency pairs and polynomial path orders. In *RTA*, pages 48–62. Springer, 2009.

[AM10a] Martin Avanzini and Georg Moser. Closing the gap between runtime complexity and polytime computability. In *RTA*, volume 6 of *LIPIcs*, pages 33–48. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[AM10b] Martin Avanzini and Georg Moser. Complexity analysis by graph rewriting. In *FLOPS*, pages 257–271. Springer, 2010.

[AM13a] Robert Atkey and Conor McBride. Productive coprogramming with guarded recursion. In *ICFP*, volume 48, pages 197–208. ACM, 2013.

[AM13b] Martin Avanzini and Georg Moser. Tyrolean complexity tool: Features and usage. In *RTA*, volume 21 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[AM16] Martin Avanzini and Georg Moser. A combination framework for complexity. *Information and Computation*, 248:22–55, 2016.

[Ama03] Roberto M. Amadio. Max-plus quasi-interpretations. In *TLCA*, pages 31–45. Springer, 2003.

[Ama05] Roberto M. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65(1-2):29–60, 2005.

[AMS16] Martin Avanzini, Georg Moser, and Michael Schaper. TcT: Tyrolean complexity Tool. In *TACAS*, pages 407–423. Springer, 2016.

[AMS19] Martin Avanzini, Georg Moser, and Michael Schaper. Modular runtime complexity analysis of probabilistic while programs. DICE-FOPARA workshops, 2019.

[AR02] Andrea Asperti and Luca Roversi. Intuitionistic light affine logic. *Transactions on Computational Logic*, 3(1):137–175, 2002.

[Asp98] Andrea Asperti. Light affine logic. In *LICS*, pages 300–308. IEEE, 1998.

[Ava08] Martin Avanzini. POP* and semantic labeling using SAT. In *ESSLLI*, pages 155–166. Springer, 2008.

[B$^+$84] Hendrik P. Barendregt et al. *The Lambda Calculus*, volume 3. North-Holland Amsterdam, 1984.

[BA02] Amir Ben-Amram. General size-change termination and lexicographic descent. In *The essence of computation*, pages 3–17. Springer, 2002.

[BAG13] Amir Ben-Amram and Samir Genaim. On the linear ranking problem for integer linear-constraint loops. In *POPL*, volume 48, pages 51–62. ACM, 2013.

[BAG17] Amir Ben-Amram and Samir Genaim. On multiphase-linear ranking functions. In *CAV*, pages 601–620. Springer, 2017.

[BAGM12] Amir Ben-Amram, Samir Genaim, and Abu Naser Masud. On the termination of integer loops. *Transactions on Programming Languages and Systems*, 34(4):16, 2012.

[Bai02] Patrick Baillot. Checking polynomial time complexity with types. In *Foundations of Information Technology in the Era of Network and Mobile Computing*, pages 370–382. Springer, 2002.

[Bai04a] Patrick Baillot. Stratified coherence spaces: a denotational semantics for light linear logic. *Theoretical Computer Science*, 318(1-2):29–55, 2004.

[Bai04b] Patrick Baillot. Type inference for light affine logic via constraints on words. *Theoretical Computer Science*, 328(3):289–323, 2004.

[Bai08] Patrick Baillot. Logique linéaire, types et complexité implicite. Mémoire d'habilitation à diriger des recherches, 2008. LIPN, UMR 7030 CNRS, Université Paris 13.

[BBRDR18] Patrick Baillot, Erika De Benedetti, and Simona Ronchi Della Rocca. Characterizing polynomial and exponential complexity classes in elementary lambda-calculus. *Information and Computation*, 261(1):55–77, 2018.

[BC92] Spephen Bellantoni and Stephen A. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2(2):97–110, 1992.

[BCJdNM05] Olivier Bournez, Felipe Cucker, Paulin Jacobé de Naurois, and Jean-Yves Marion. Implicit complexity over an arbitrary structure: Sequential and parallel polynomial time. *Journal of Logic and Computation*, 15(1):41–58, 2005.

[BCJdNM06] Olivier Bournez, Felipe Cucker, Paulin Jacobé de Naurois, and Jean-Yves Marion. Implicit complexity over an arbitrary structure: Quantifier alternations. *Information and Computation*, 204(2):210–230, 2006.

[BCMT98] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. Complexity classes and rewrite systems with polynomial interpretation. In *CSL*, pages 372–384. Springer, 1998.

[BCMT01] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, 11(1):33–53, 2001.

[BCR09] Michael J. Burrell, J. Robin B. Cockett, and Brian F. Redmond. Pola: A language for ptime programming. In *FICS*, pages 7–8. Institute of Cybernetics at Tallinn University of Technology, 2009.

[BD10] Guillaume Bonfante and Florian Deloup. Complexity invariance of real interpretations. In *TAMC*, pages 139–150. Springer, 2010.

[BDH15] Guillaume Bonfante, Florian Deloup, and Antoine Henrot. Real or natural number interpretation and their effect on complexity. *Theoretical Computer Science*, 585:25–40, 2015.

[BDL12] Patrick Baillot and Ugo Dal Lago. Higher-order interpretations and program complexity. In *CSL*, volume 16 of *LIPIcs*, pages 62–76. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[BDL16] Patrick Baillot and Ugo Dal Lago. Higher-order interpretations and program complexity. *Information and Computation*, 248:56–81, 2016.

[BDL18] Flavien Breuvart and Ugo Dal Lago. On intersection types and probabilistic lambda calculi. In *PPDP*, volume 8, pages 1–13. ACM, 2018.

[BDLGS16] Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. A lambda-calculus foundation for universal probabilistic programming. In *ICFP*, volume 51, pages 33–46. ACM, 2016.

[BDLM12] Patrick Baillot, Ugo Dal Lago, and Jean-Yves Moyen. On quasi-interpretations, blind abstractions and implicit complexity. *Mathematical Structures in Computer Science*, 22(4):549–580, 2012.

[Bel95] Stephen Bellantoni. Predicative recursion and the polytime hierarchy. In *Feasible Mathematics II*, pages 15–29. Springer, 1995.

[BG05] Olivier Bournez and Florent Garnier. Proving positive almost-sure termination. In *RTA*, pages 323–337. Springer, 2005.

[BG06] Olivier Bournez and Florent Garnier. Proving positive almost sure termination under strategies. In *RTA*, pages 357–371. Springer, 2006.

[BG07] Guillaume Bonfante and Yves Guiraud. Programs as polygraphs: computability and complexity. Technical report, 2007.

[BG08] Guillaume Bonfante and Yves Guiraud. Intensional properties of polygraphs. *Electronic Notes in Theoretical Computer Science*, 203(1):65–77, 2008.

[BG18] Patrick Baillot and Alexis Ghyselen. Combining linear logic and size types for implicit complexity. In *CSL*, volume 119 of *LIPIcs*, pages 1–21. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[BGH11] Olivier Bournez, Walid Gomaa, and Emmanuel Hainry. Algebraic characterizations of complexity-theoretic classes of real functions. *International Journal of Unconventional Computing*, 7(5):331–351, 2011.

[BGP16] Olivier Bournez, Daniel S. Graça, and Amaury Pouly. Polynomial time corresponds to solutions of polynomial ordinary differential equations of polynomial length: The general purpose analog computer and computable analysis are two efficiently equivalent models of computations. In *ICALP*, volume 55 of *LIPIcs*, pages 1–15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[BGR08] Gilles Barthe, Benjamin Grégoire, and Colin Riba. Type-based termination with sized products. In *CSL*, pages 493–507. Springer, 2008.

[BH04] Olivier Bournez and Emmanuel Hainry. Real recursive functions and real extensions of recursive functions. In *MCU*, pages 116–127. Springer, 2004.

[BHMS04] Lennart Beringer, Martin Hofmann, Alberto Momigliano, and Olha Shkaravska. Automatic certification of heap consumption. In *LPAR*, pages 347–362. Springer, 2004.

[Bib77] Kenneth J. Biba. Integrity considerations for secure computer systems. Technical report, Mitre corp rep., 1977.

[BKMO08] Guillaume Bonfante, Reinhard Kahle, Jean-Yves Marion, and Isabel Oitavem. Recursion schemata for $\mathtt{NC}^k$. In *CSL*, pages 49–63. Springer, 2008.

[BKV17] Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Non-idempotent intersection types for the lambda-calculus. *Logic Journal of the IGPL*, 25(4):431–464, 2017.

[BL11] Alexis Bernadet and Stéphane Lengrand. Complexity of strongly normalising λ-terms via non-idempotent intersection types. In *FoSSaCS*, pages 88–107. Springer, 2011.

[Blo94] Stephen Bloch. Function-algebraic characterizations of log and polylog parallel time. *Computational Complexity*, 4(2):175–205, 1994.

[BLO+12] Cristina Borralleras, Salvador Lucas, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Sat modulo linear arithmetic for solving polynomial constraints. *Journal of Automated Reasoning*, 48(1):107–131, 2012.

[BLP76] David E. Bell and Leonard J. La Padula. Secure computer system: unified exposition and multics interpretation. Technical report, Mitre corp Rep., 1976.

[Blu67] Manuel Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the ACM*, 14(2):322–336, 1967.

[BM04] Patrick Baillot and Virgile Mogbil. Soft lambda-calculus: a language for polynomial time computation. In *FoSSaCS*, pages 27–41. Springer, 2004.

[BM10] Patrick Baillot and Damiano Mazza. Linear logic by levels and bounded time complexity. *Theoretical Computer Science*, 411(2):470–503, 2010.

[BM12] Aloïs Brunel and Antoine Madet. Indexed realizability for bounded-time programming with references and type fixpoints. In *APLAS*, pages 264–279. Springer, 2012.

[BMM01] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. On lexicographic termination ordering with space bound certifications. In *PSI*, pages 482–493. Springer, 2001.

[BMM05] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. Quasi-interpretations and small space bounds. In *RTA*, pages 150–164. Springer, 2005.

[BMM11] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. Quasi-interpretations a way to control resources. *Theoretical Computer Science*, 412(25):2776–2796, 2011.

[BMMP05] Guillaume Bonfante, Jean-Yves Marion, Jean-Yves Moyen, and Romain Péchoux. Synthesis of quasi-interpretations. LCC, 2005.

[BMP06] Guillaume Bonfante, Jean-Yves Marion, and Romain Péchoux. A characterization of alternating log time by first order functional programs. In *LPAR*, pages 90–104. Springer, 2006.

[BMP07] Guillaume Bonfante, Jean-Yves Marion, and Romain Péchoux. Quasi-interpretation synthesis by decomposition. In *ICTAC*, pages 410–424. Springer, 2007.

[BN99] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1999.

[BNS00] Stephen Bellantoni, Karl-Heinz Niggl, and Helmut Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104(1-3):17–30, 2000.

[Boa14] Peter van Emde Boas. Machine models and simulations. *Handbook of Theoretical Computer Science*, 1:1–66, 2014.

[Bon06] Guillaume Bonfante. Some programming languages for `Logspace` and `Ptime`. In *AMAST*, pages 66–80. Springer, 2006.

[Bon11] Guillaume Bonfante. Complexité implicite des calculs : interprétation de programmes. Mémoire d'habilitation à diriger des recherches, 2011. LORIA, UMR 7503, INPL.

[BP96] Andrew Barber and Gordon Plotkin. Dual intuitionistic linear logic. Technical report, University of Edinburgh, Department of Computer Science, 1996.

[BP06] Patrick Baillot and Marco Pedicini. An embedding of the BSS model of computation in light affine lambda-calculus. LCC, 2006.

[BRDR15] Erika De Benedetti and Simona Ronchi Della Rocca. Call-by-value, elementary time and intersection types. In *FOPARA*, pages 40–59. Springer, 2015.

[BSS88] Lenore Blum, Mike Shub, and Steve Smale. On a theory of computation over the real numbers; `NP` completeness, recursive functions and universal machines. In *FOCS*, pages 387–397. IEEE, 1988.

[BT04] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. In *LICS*, pages 266–275. IEEE, 2004.

[BT05] Patrick Baillot and Kazushige Terui. A feasible algorithm for typing in elementary affine logic. In *TLCA*, pages 55–70. Springer, 2005.

[BT09] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda calculus. *Information and Computation*, 207(1):41–62, 2009.

[BT10] Aloïs Brunel and Kazushige Terui. Church => Scott = `Ptime`: an application of resource sensitive realizability. In *DICE*, volume 23, pages 31–46. Electronic Proceedings in Theoretical Computer Science, 2010.

[BV97] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, 1997.

[CBGB15] Ranald Clouston, Ales Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. Programming and reasoning with guarded recursion for coinductive types. In *FoSSaCS*, pages 407–421. Springer, 2015.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[CD96] Wei-Ngan Chin and John Darlington. A higher-order removal method. *Lisp and Symbolic Computation*, 9(4):287–322, 1996.

[CHS15] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. *PLDI*, 50(6):467–478, 2015.

[CK90] Stephen A. Cook and Bruce M. Kapron. Characterizations of the basic feasible functionals of finite type. In *Feasible mathematics*, pages 71–96. Springer, 1990.

[CK93] Peter Clote and Jan Krajícek. *Arithmetic, Proof Theory, and Computational Complexity*. Number 23. Oxford University Press, 1993.

[CK01] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2-3):261–300, 2001.

[CL87] Ahlem Ben Cherifa and Pierre Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137–159, 1987.

[CL92] Adam Cichon and Pierre Lescanne. Polynomial interpretations and the complexity of algorithms. In *CADE*, pages 139–147. Springer, 1992.

[Clo90] Peter Clote. Sequential, machine-independent characterizations of the parallel complexity classes $\texttt{Alogtime}$, $\texttt{AC}^k$, $\texttt{NC}^k$ and $\texttt{NC}$. In *Feasible mathematics*, pages 49–69. Springer, 1990.

[Clo97] Peter Clote. Nondeterministic stack register machines. *Theoretical Computer Science*, 178(1-2):37–76, 1997.

[Clo99] Peter Clote. Computation models and function algebras. In *Studies in Logic and the Foundations of Mathematics*, volume 140, pages 589–681. Elsevier, 1999.

[CM00] Adam Cichon and Jean-Yves Marion. The light lexicographic path ordering. Technical report, Loria, 2000.

[CM01] Paolo Coppola and Simone Martini. Typing lambda terms in elementary logic with linear constraints. In *TLCA*, pages 76–90. Springer, 2001.

[CMTU05] Evelyne Contejean, Claude Marché, Ana Paula Tomás, and Xavier Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325, 2005.

[CO07] Manuel L. Campagnolo and Kerry Ojakian. Using approximation to relate computational classes over the reals. In *MCU*, pages 39–61. Springer, 2007.

[Cob65] Alan Cobham. The intrinsic computational difficulty of functions. *North-Holland Publishing*, 1965.

[Col89] Loic Colson. About primitive recursive algorithms. In *ICALP*, pages 194–206. Springer, 1989.

[Con73] Robert L. Constable. Type two computational complexity. In *STOC*, pages 108–121. ACM, 1973.

[Coo92] Stephen A. Cook. Computability and complexity of higher type functions. In *Logic from Computer Science*, pages 51–72. Springer, 1992.

[Coq94] Thierry Coquand. Infinite objects in type theory. In *TYPES*, volume 806 of *LNCS*, pages 62–78. Springer, 1994.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. TERMINATOR: beyond safety. In *CAV*, pages 415–418. Springer, 2006.

[CR80] Stephen A. Cook and Charles Rackoff. Space lower bounds for maze threadability on restricted machines. *SIAM Journal on Computing*, 9(3):636–652, 1980.

[CS76] Ashok Chandra and Larry Stockmeyer. Alternation. In *FOCS*, pages 98–108. IEEE, 1976.

[CS12] Jacek Chrzaszcz and Aleksy Schubert. ML with `Ptime` complexity guarantees. In *CSL*, volume 16 of *LIPIcs*, pages 198–212. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[CS16] Jacek Chrzaszcz and Aleksy Schubert. The role of polymorphism in the characterisation of complexity by soft types. *Information and Computation*, 248:130–149, 2016.

[CT95] Peter Clote and Gaisi Takeuti. First order bounded arithmetic and small boolean circuit complexity classes. In *Feasible mathematics*, pages 154–218. Springer, 1995.

[CU93] Stephen A. Cook and Alasdair Urquhart. Functional interpretations of feasibly constructive arithmetic. *Annals of Pure and Applied Logic*, 63(2):103–200, 1993.

[Dan06] Olivier Danvy. An analytical approach to program as data objects. DSc thesis, 2006. Department of Computer Science, University of Aarhus.

[DC05] Daniel De Carvalho. Intersection types for light affine lambda calculus. *Electronic Notes in Theoretical Computer Science*, 136:133–152, 2005.

[DC18] Daniel De Carvalho. Execution time of λ-terms via denotational semantics and intersection types. *Mathematical Structures in Computer Science*, 28(7):1169–1203, 2018.

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

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

[Der87] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1/2):69–116, 1987.

[Deu85] David Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Mathematical and Physical Sciences*, 400(1818):97–117, 1985.

[Dij80] Edsger W. Dijkstra. On the productivity of recursive definitions. EWD749, 1980.

[DJ03] Vincent Danos and Jean-Baptiste Joinet. Linear logic and elementary time. *Information and Computation*, 183(1):123–137, 2003.

[DLG11] Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. In *LICS*, pages 133–142. IEEE, 2011.

[DLG19] Ugo Dal Lago and Charles Grellois. Probabilistic termination by monadic affine sized typing. *Transactions on Programming Languages and Systems*, 41(2):10:1–10:65, 2019.

[DLH05] Ugo Dal Lago and Martin Hofmann. Quantitative models and implicit complexity. In *FSTTCS*, pages 189–200. Springer, 2005.

[DLH11] Ugo Dal Lago and Martin Hofmann. Realizability models and implicit complexity. *Theoretical Computer Science*, 412(20):2029–2047, 2011.

[DLM09] Ugo Dal Lago and Simone Martini. On constructor rewrite systems and the lambda-calculus. In *ICALP*, pages 163–174. Springer, 2009.

[DLMS10] Ugo Dal Lago, Simone Martini, and Davide Sangiorgi. Light logics and higher-order processes. In *EXPRESS*, volume 41, pages 46–60. Electronic Proceedings in Theoretical Computer Science, 2010.

[DLMS16] Ugo Dal Lago, Simone Martini, and Davide Sangiorgi. Light logics and higher-order processes. *Mathematical Structures in Computer Science*, 26(6):969–992, 2016.

[DLMZ10] Ugo Dal Lago, Andrea Masini, and Margherita Zorzi. Quantum implicit computational complexity. *Theoretical Computer Science*, 411(2):377–409, 2010.

[DLP14] Ugo Dal Lago and Barbara Petit. Linear dependent types in a call-by-value scenario. *Science of Computer Programming*, 84:77–100, 2014.

[DLR15] Norman Danner, Daniel R. Licata, and R. Ramyaa. Denotational cost semantics for functional languages with inductive types. In *ICFP*, volume 50, pages 140–151. ACM, 2015.

[DLS10] Ugo Dal Lago and Ulrich Schöpp. Functional programming in sublinear space. In *ESOP*, pages 205–225. Springer, 2010.

[DLS16] Ugo Dal Lago and Ulrich Schöpp. Computation by interaction for space-bounded functional programming. *Information and Computation*, 248:150–194, 2016.

[DLT15] Ugo Dal Lago and Paolo Parisen Toldin. A higher-order characterization of probabilistic polynomial time. *Information and Computation*, 241:114–141, 2015.

[EGH+10] Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Ariya Isihara, and Jan Willem Klop. Productivity of stream definitions. *Theoretical Computer Science*, 411(4-5):765–782, 2010.

[ER03] Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theoretical Computer Science*, 309(1-3):1–41, 2003.

[ER08] Thomas Ehrhard and Laurent Regnier. Uniformity and the taylor expansion of ordinary lambda-terms. *Theoretical Computer Science*, 403(2-3):347–372, 2008.

[EWZ08] Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning*, 40(2-3):195–220, 2008.

[Fag74] Ronald Fagin. Generalized first-order spectra and polynomial-time recognizable sets. *Complexity of computation*, 7:43–73, 1974.

[Fag19] Claudia Faggian. Probabilistic rewriting: Normalization, termination, and unique normal forms. In *FSCD*, volume 131 of *LIPIcs*, pages 1–25. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

[FGM+07] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. SAT solving for termination analysis with polynomial interpretations. In *SAT*, pages 340–354. Springer, 2007.

[FHHP10] Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, and Romain Péchoux. Interpretation of stream programs: Characterizing type 2 polynomial time complexity. In *ISAAC*, pages 291–303. Springer, 2010.

[FHHP15] Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, and Romain Péchoux. Characterizing polynomial time complexity of stream programs using interpretations. *Theoretical Computer Science*, 585:41–54, 2015.

[FHM+18] Hugo Férée, Samuel Hym, Micaela Mayero, Jean-Yves Moyen, and David Nowak. Formal proof of polynomial-time complexity with quasi-interpretations. In *CPP*, pages 146–157. ACM, 2018.

[FRDR19] Claudia Faggian and Simona Ronchi Della Rocca. Lambda calculus and probabilistic computation. In *LICS*, pages 1–13. IEEE, 2019.

[GAB+17] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, et al. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 2017.

[Geu88] Oliver Geupel. Terminationsbeweise bei Termersetzungssystemen. diplomarbeit, 1988. Technische Universität Dresden, Sektion Mathematik.

[Gie95] Jürgen Giesl. Generating polynomial orderings for termination proofs. In *RTA*, pages 426–431. Springer, 1995.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[Gir98] Jean-Yves Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.

[GM16] Stéphane Gimenez and Georg Moser. The complexity of interaction. In *POPL*, volume 51, pages 243–255. ACM, 2016.

[GMC09] Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *POPL*, volume 44, pages 127–139. ACM, 2009.

[GMRDR08a] Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. A logical account of PSPACE. volume 43, pages 121–131. ACM, 2008.

[GMRDR08b] Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. Soft linear logic and polynomial complexity classes. *Electronic Notes in Theoretical Computer Science*, 205:67–87, 2008.

[Gna07] Isabelle Gnaedig. Induction for positive almost sure termination. In *PPDP*, pages 167–178. ACM, 2007.

[Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931.

[Goe92] Andreas Goerdt. Characterizing complexity classes by general recursive definitions in higher types. *Information and Computation*, 101(2):202–218, 1992.

[GP09a] Marco Gaboardi and Romain Péchoux. Global and local space properties of stream programs. In *FOPARA*, pages 51–66. Springer, 2009.

[GP09b] Marco Gaboardi and Romain Péchoux. Upper bounds on stream I/O using semantic interpretations. In *CSL*, pages 271–286. Springer, 2009.

[GP15a] Marco Gaboardi and Romain Péchoux. Algebras and coalgebras in the light affine lambda calculus. In *ICFP*, volume 50, pages 114–126. ACM, 2015.

[GP15b] Marco Gaboardi and Romain Péchoux. On bounding space usage of streams using interpretation analysis. *Science of Computer Programming*, 111:395–425, 2015.

[Grä91] Erich Grädel. The expressive power of second order Horn logic. In *STACSs*, pages 466–477. Springer, 1991.

[Gra94] Bernhard Gramlich. Generalized sufficient conditions for modular termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 5(3-4):131–158, 1994.

[GRDR07] Marco Gaboardi and Simona Ronchi Della Rocca. A soft type assignment system for λ-calculus. In *CSL*, pages 253–267. Springer, 2007.

[GRDR08] Marco Gaboardi and Simona Ronchi Della Rocca. Type inference for a polynomial lambda calculus. In *TYPES*, pages 136–152. Springer, 2008.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *STOC*, pages 212–219. ACM, 1996.

[Grz53] Andrzej Grzegorczyk. Some classes of recursive functions. Technical report, Instytut Matematyczny Polskiej Akademi Nauk (Warszawa), 1953.

[GSS92] Jean-Yves Girard, Andre Scedrov, and Philip Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1–66, 1992.

[Gul09] Sumit Gulwani. SPEED: Symbolic complexity bound analysis. In *CAV*, pages 51–62. Springer, 2009.

[Gur83] Yuri Gurevich. Algebras of feasible functions. In *FOCS*, pages 210–214. IEEE, 1983.

[Hag87] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In *CTCS*, volume 283, pages 140–57. Springer, 1987.

[HAH11] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *POPL*, volume 46, pages 357–370. ACM, 2011.

[HAH12] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In *CAV*, pages 781–786. Springer, 2012.

[Haj79] Petr Hajek. Arithmetical hierarchy and complexity of computation. *Theoretical Computer Science*, 8(2):227–237, 1979.

[HH10] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In *ESOP*, pages 287–306. Springer, 2010.

[HJ03] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, volume 38, pages 185–197. ACM, 2003.

[HJ06] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *ESOP*, pages 22–37. Springer, 2006.

[HJP10] Matthias Heizmann, Neil Jones, and Andreas Podelski. Size-change termination and transition invariants. In *SAS*, pages 22–50. Springer, 2010.

[HM08] Nao Hirokawa and Georg Moser. Automated complexity analysis based on the dependency pair method. In *IJCAR*, pages 364–379. Springer, 2008.

[HM14a] Nao Hirokawa and Georg Moser. Automated complexity analysis based on context-sensitive rewriting. In *TLCA and RTA*, pages 257–271. Springer, 2014.

[HM14b] Martin Hofmann and Georg Moser. Amortised resource analysis and typed polynomial interpretations. In *TLCA and RTA*, pages 272–286. Springer, 2014.

[HM15] Martin Hofmann and Georg Moser. Multivariate amortised resource analysis for term rewrite systems. In *TLCA*, volume 38 of *LIPIcs*, pages 241–256. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[HMP13] Emmanuel Hainry, Jean-Yves Marion, and Romain Péchoux. Type-based complexity analysis for fork processes. In *FoSSaCS*, pages 305–320. Springer, 2013.

[Hof92] Dieter Hofbauer. Termination proofs by multiset path orderings imply primitive recursive derivation lengths. *Theoretical Computer Science*, 105(1):129–140, 1992.

[Hof99] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *LICS*, pages 464–473. IEEE, 1999.

[Hof00] Martin Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, 104(1-3):113–166, 2000.

[Hof01] Dieter Hofbauer. Termination proofs by context-dependent interpretations. In *RTA*, pages 108–121. Springer, 2001.

[Hof02] Martin Hofmann. The strength of non-size increasing computation. *POPL*, 37(1):260–269, 2002.

[Hof03] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1):57–85, 2003.

[HP15] Emmanuel Hainry and Romain Péchoux. Objects in polynomial time. In *APLAS*, pages 387–404. Springer, 2015.

[HP17] Emmanuel Hainry and Romain Péchoux. Higher order interpretation for higher order complexity. In *LPAR*, EPIC, pages 269–285. easychair, 2017.

[HP18] Emmanuel Hainry and Romain Péchoux. A type-based complexity analysis of object oriented programs. *Information and Computation*, 261(1):78–115, 2018.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *POPL*, pages 410–423. ACM, 1996.

[HR09] Martin Hofmann and Dulma Rodriguez. Efficient type-checking for amortised heap-space analysis. In *CSL*, pages 317–331. Springer, 2009.

[HR13] Martin Hofmann and Dulma Rodriguez. Automatic type inference for amortised heap-space analysis. In *ESOP*, pages 593–613. Springer, 2013.

[HRS13] Martin Hofmann, Ramyaa Ramyaa, and Ulrich Schöpp. Pure pointer programs and tree isomorphism. In *FoSSaCS*, pages 321–336. Springer, 2013.

[HS10] Martin Hofmann and Ulrich Schöpp. Pure pointer programs with iteration. *Transactions on Computational Logic*, 11(4):26, 2010.

[HS15] Jan Hoffmann and Zhong Shao. Automatic static cost analysis for parallel programs. In *ESOP*, pages 132–157. Springer, 2015.

[HW06] Dieter Hofbauer and Johannes Waldmann. Termination of string rewriting with matrix interpretations. In *RTA*, pages 328–342. Springer, 2006.

[IKR02] Robert J. Irwin, Bruce M. Kapron, and James S. Royer. On characterizations of the basic feasible functionals, Part II. Technical report, Syracuse University, 2002.

[Imm86] Neil Immerman. Relational queries computable in polynomial time. *Information and Control*, 68(1-3):86–104, 1986.

[Imm12] Neil Immerman. *Descriptive Complexity*. Springer Science & Business Media, 2012.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[IRK01] Robert J. Irwin, James S. Royer, and Bruce M. Kapron. On characterizations of the basic feasible functionals, Part I. *Journal of Functional Programming*, 11(1):117–153, 2001.

[JdN19] Paulin Jacobé de Naurois. Pointers in recursion: Exploring the tropics. In *FSCD*, volume 131 of *LIPIcs*, pages 1–18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

[JHLH10] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *POPL*, volume 45, pages 223–236. ACM, 2010.

[JK05] Neil Jones and Lars Kristiansen. The flow of data and the complexity of algorithms. In *CIE*, pages 263–274. Springer, 2005.

[JK09] Neil Jones and Lars Kristiansen. A flow calculus of mwp-bounds for complexity analysis. *Transactions on Computational Logic*, 10(4):28, 2009.

[Jon99] Neil Jones. `Logspace` and `Ptime` characterized by programming languages. *Theoretical Computer Science*, 228(1-2):151–174, 1999.

[Jon01] Neil Jones. The expressive power of higher-order types or, life without CONS. *Journal of Functional Programming*, 11(1):55–94, 2001.

[JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[K+01] Jan Willem Klop et al. *Term Rewriting Systems*. Cambridge University Press, 2001.

[Kal43] László Kalmár. Egyszerü példa eldönthetetlen aritmetikai problémára. *Mate és fizikai lapok*, 50:1–23, 1943.

[Kam80] Samuel Kamin. Attempts for generalizing the recursive path orderings. Technical report, Report, Dept. of Computer Science, University of Illinois, 1980.

[KC91] Bruce M. Kapron and Stephen A. Cook. A new characterization of Mehlhorn's polynomial time functionals. In *FOCS*, pages 342–347. IEEE, 1991.

[KC96] Bruce M. Kapron and Steven A. Cook. A new characterization of type-2 feasibility. *SIAM Journal on Computing*, 25(1):117–132, 1996.

[KDLG19] Naoki Kobayashi, Ugo Dal Lago, and Charles Grellois. On the termination problem for probabilistic higher-order recursive programs. In *LICS*, pages 1–14. IEEE, 2019.

[KKMO16] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected runtimes of probabilistic programs. In *ESOP*, pages 364–389. Springer, 2016.

[Kle35] Stephen Kleene. A theory of positive integers in formal logic. Part I. *American Journal of Mathematics*, 57(1):153–173, 1935.

[Kle36] Stephen Kleene. General recursive functions of natural numbers. *Mathematische annalen*, 112(1):727–742, 1936.

[KN85] Mukkai Krishnamoorthy and Paliath Narendran. On recursive path ordering. *Theoretical Computer Science*, 40:323–328, 1985.

[KN04] Lars Kristiansen and Karl-Heinz Niggl. On the computational complexity of imperative programming languages. *Theoretical Computer Science*, 318(1-2):139–161, 2004.

[Ko91] Ker-I Ko. *Complexity Theory of Real Functions*. Birkhäuser, 1991.

[KO92] Masahito Kurihara and Azuma Ohuchi. Modularity of simple termination of term rewriting systems with shared constructors. *Theoretical Computer Science*, 103(2):273–282, 1992.

[Koz79] Dexter Kozen. Semantics of probabilistic programs. In *SFCS*, pages 101–114. IEEE, 1979.

[KP84] Brian W. Kernighan and Rob Pike. *The Unix Programming Environment*. Prentice Hall, 1984.

[KS18] Bruce M. Kapron and Florian Steinberg. Type-two polynomial-time and restricted lookahead. In *LICS*, pages 579–588. IEEE, 2018.

[KS19] Bruce M. Kapron and Florian Steinberg. Type-two iteration with bounded query revision. In *DICE-FOPARA*, volume 298, pages 61–74. Electronic Proceedings in Theoretical Computer Science, 2019.

[KSvG+12] Rody Kersten, Olha Shkaravska, Bernard van Gastel, Manuel Montenegro, and Marko C. J. D. van Eekelen. Making resource analysis practical for real-time Java. In *JTRES*, pages 135–144. ACM, 2012.

[KV03] Lars Kristiansen and Paul Voda. Complexity classes and fragments of C. *Information Processing Letters*, 88(5):213–218, 2003.

[KV17] Delia Kesner and Pierre Vial. Types as resources for classical natural deduction. In *FSCD*, volume 84 of *LIPIcs*, pages 1–17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[Laf04] Yves Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318(1-2):163–180, 2004.

[Lan79] Dallas Lankford. On proving term rewriting systems are Noetherien. Technical report, Department of Mathematics, Louisiana Technical University, 1979.

[Lau88] Clemens Lautemann. A note on polynomial interpretation. *Bulletin of the EATCS*, 36:129–130, 1988.

[Lei95] Daniel Leivant. Ramified recurrence and computational complexity I: Word recurrence and poly-time. In *Feasible mathematics*, pages 320–343. Springer, 1995.

[Lei98] Daniel Leivant. A characterization of NC by tree recurrence. In *FOCS*, pages 716–724. IEEE, 1998.

[LJBA01] Chin Soon Lee, Neil Jones, and Amir Ben-Amram. The size-change principle for program termination. In *POPL*, volume 36, pages 81–92. ACM, 2001.

[LM93] Daniel Leivant and Jean-Yves Marion. Lambda calculus characterizations of poly-time. In *TLCA*, pages 274–288. Springer, 1993.

[LM94] Daniel Leivant and Jean-Yves Marion. Ramified recurrence and computational complexity II: substitution and poly-space. In *CSL*, pages 486–500. Springer, 1994.

[LM00] Daniel Leivant and Jean-Yves Marion. A characterization of alternating log time by ramified recurrence. *Theoretical Computer Science*, 236(1-2):193–208, 2000.

[LM13] Daniel Leivant and Jean-Yves Marion. Evolving graph-structures and their implicit computational complexity. In *ICALP*, pages 349–360. Springer, 2013.

[LTdF06] Olivier Laurent and Lorenzo Tortora de Falco. Obsessional cliques: a semantic characterization of bounded time complexity. In *LICS*, pages 179–188. IEEE, 2006.

[Luc05] Salvador Lucas. Polynomials over the reals in proofs of termination: from theory to practice. *RAIRO-Theoretical Informatics and Applications*, 39(3):547–586, 2005.

[Luc07] Salvador Lucas. Practical use of polynomials over the reals in proofs of termination. In *PPDP*, pages 39–50. ACM, 2007.

[MA11] Antoine Madet and Roberto M. Amadio. An elementary affine λ-calculus with multithreading and side effects. In *TLCA*, pages 138–152. Springer, 2011.

[Mad12] Antoine Madet. A polynomial time λ-calculus with multithreading and side effects. In *PPDP*, pages 55–66. ACM, 2012.

[Mar11] Jean-Yves Marion. A type system for complexity flow analysis. In *LICS*, pages 123–132. IEEE, 2011.

[Mau03] François Maurel. Nondeterministic light logics and NP-time. In *TLCA*, pages 241–255. Springer, 2003.

[Maz12] Damiano Mazza. An infinitary affine lambda-calculus isomorphic to the full lambda-calculus. In *LICS*, pages 471–480. IEEE, 2012.

[Maz14] Damiano Mazza. Non-uniform polytime computation in the infinitary affine lambda-calculus. In *ICALP*, pages 305–317. Springer, 2014.

[Maz15] Damiano Mazza. Simple parsimonious types and logarithmic space. In *CSL*, volume 41 of *LIPIcs*, pages 24–40. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[Meh76] Kurt Mehlhorn. Polynomial and abstract subrecursive classes. *Journal of Computer and System Sciences*, 12(2):147–178, 1976.

[MM00] Jean-Yves Marion and Jean-Yves Moyen. Efficient first order functional program interpreter with time bound certifications. In *LPAR*, pages 25–42. Springer, 2000.

[MN70] Zohar Manna and Steven Ness. On the termination of markov algorithms. In *HICSS*, pages 789–792. IEEE, 1970.

[MO04] Andrzej Murawski and Luke Ong. On an interpretation of safe recursion in light affine logic. *Theoretical Computer Science*, 318(1-2):197–223, 2004.

[Moy09] Jean-Yves Moyen. Resource control graphs. *Transactions on Computational Logic*, 10(4):29, 2009.

[MP06] Jean-Yves Marion and Romain Péchoux. Resource analysis by sup-interpretation. In *FLOPS*, pages 163–176. Springer, 2006.

[MP08a] Jean-Yves Marion and Romain Péchoux. Analyzing the implicit computational complexity of object-oriented programs. In *FSTTCS*, volume 2 of *LIPIcs*, pages 316–327. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

[MP08b] Jean-Yves Marion and Romain Péchoux. A characterization of $\mathtt{NC}^k$ by first order functional programs. In *TAMC*, pages 136–147. Springer, 2008.

[MP08c] Jean-Yves Marion and Romain Péchoux. Characterizations of polynomial complexity classes with a better intensionality. In *PPDP*, pages 79–88. ACM, 2008.

[MP09] Jean-Yves Marion and Romain Péchoux. Sup-interpretations, a semantic method for static analysis of program resources. *Transactions on Computational Logic*, 10(4):27, 2009.

[MP14] Jean-Yves Marion and Romain Péchoux. Complexity information flow in a multi-threaded imperative language. In *TAMC*, pages 124–140. Springer, 2014.

[MS08] Georg Moser and Andreas Schnabl. Proving quadratic derivational complexities using context dependent interpretations. In *RTA*, pages 276–290. Springer, 2008.

[MS09] Georg Moser and Andreas Schnabl. The derivational complexity induced by the dependency pair method. In *RTA*, pages 255–269. Springer, 2009.

[MS11] Georg Moser and Andreas Schnabl. Termination Proofs in the Dependency Pair Framework May Induce Multiple Recursive Derivational Complexity. In *RTA*, volume 10 of *LIPIcs*, pages 235–250. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[MSW08] Georg Moser, Andreas Schnabl, and Johannes Waldmann. Complexity analysis of term rewriting based on matrix and context dependent interpretations. In *RTA*, volume 2 of *LIPIcs*, pages 304–315. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

[MT15] Damiano Mazza and Kazushige Terui. Parsimonious types and non-uniform computation. In *ICALP*, pages 350–361. Springer, 2015.

[Mül74] Helmut Müller. *Klassifizierungen der primitiv-rekursiven Funktionen.* PhD thesis, Verlag nicht ermittelbar, 1974.

[NCH18] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. Bounded expectations: resource analysis for probabilistic programs. In *PLDI*, volume 53, pages 496–512. ACM, 2018.

[NEG13] Lars Noschinski, Fabian Emmes, and Jürgen Giesl. Analyzing innermost runtime complexity of term rewriting by dependency pairs. *Journal of Automated Reasoning*, 51(1):27–56, 2013.

[Nig00] Karl-Heinz Niggl. The μ-measure as a tool for classifying computational complexity. *Archive for Mathematical Logic*, 39(7):515–539, 2000.

[NM10] Friedrich Neurauter and Aart Middeldorp. Polynomial interpretations over the reals do not subsume polynomial interpretations over the integers. In *RTA*, volume 6 of *LIPIcs*, pages 243–258. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[NW06] Karl-Heinz Niggl and Henning Wunderlich. Certifying polynomial time and linear/polynomial space for imperative programs. *SIAM Journal on Computing*, 35(5):1122–1147, 2006.

[NZM10] Friedrich Neurauter, Harald Zankl, and Aart Middeldorp. Revisiting matrix interpretations for polynomial derivational complexity of term rewriting. In *LPAR*, pages 550–564. Springer, 2010.

[Oit01] Isabel Oitavem. Implicit characterizations of `PSPACE`. In *PTCS*, pages 170–190. Springer, 2001.

[Pap03] Christos Papadimitriou. *Computational Complexity*. John Wiley and Sons Ltd., 2003.

[Péc13] Romain Péchoux. Synthesis of sup-interpretations: a survey. *Theoretical Computer Science*, 467:30–52, 2013.

[Per08] Simon Perdrix. Quantum entanglement analysis based on abstract interpretation. In *SAS*, pages 270–282. Springer, 2008.

[Per18] Matthieu Perrinel. Paths-based criteria and application to linear logic subsystems characterizing polynomial time. *Information and Computation*, 261:23–54, 2018.

[Pét67] Rózsa Péter. *Recursive Functions*. Academic Press, 1967.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT press, 1991.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT press, 2002.

[Pit98] François Pitt. A quantifier-free theory based on a string algebra for $\mathtt{NC}^1$. In *Proof complexity and feasible arithmetics*, DIMACS, pages 229–252. AMS, 1998.

[Pla78] David Plaisted. A recursively defined ordering for proving termination of term rewriting systems. Technical Report 943, Department of Computer Science, University of Illinois at Urbana-Champaign, 1978.

[PR04] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS*, pages 32–41. IEEE, 2004.

[Red07] Brian F. Redmond. Multiplexor categories and models of soft linear logic. In *LFCS*, pages 472–485. Springer, 2007.

[Rei08] Omer Reingold. Undirected connectivity in log-space. *Journal of the ACM*, 55(4):17, 2008.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference*, volume 2, pages 717–740. ACM, 1972.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

[Rit63] Robert Ritchie. Classes of predictably computable functions. *Transactions of the American Mathematical Society*, 106(1):139–173, 1963.

[Ros87] Harvey Rose. *Subrecursion: Functions and Hierarchies*. Oxford University Press, 1987.

[Rov99] Luca Roversi. A p-time completeness proof for light logics. In *CSL*, pages 469–483. Springer, 1999.

[Ruz81] Walter L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22(3):365–383, 1981.

[San90] David Sands. Complexity analysis for a lazy higher-order language. In *ESOP*, pages 361–376. Springer, 1990.

[San91] David Sands. Time analysis, cost equivalence and program refinement. In *FSTTCS*, pages 25–39. Springer, 1991.

[Sav98] John E. Savage. *Models of Computation*, volume 136. Addison-Wesley Reading, MA, 1998.

[Saz80] Vladimir Sazonov. Polynomial computability and recursivity in finite domains. *Elektronische Informationsverarbeitung und Kybernetik*, 16(7):319–323, 1980.

[Sch69] Helmut Schwichtenberg. Rekursionszahlen und die Grzegorczyk-Hierarchie. *Archiv für mathematische Logik und Grundlagenforschung*, 12(1-2):85–97, 1969.

[Sch07] Ulrich Schöpp. Stratified bounded affine logic for logarithmic space. In *LICS*, volume 7, pages 411–420. IEEE, 2007.

[Sel04] Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[Sev17] Paula Severi. A light modality for recursion. In *FoSSaCS*, pages 499–516. Springer, 2017.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[Sij89] Ben A. Sijtsma. On the productivity of recursive list definitions. *Transactions on Programming Languages and Systems*, 11(4):633–649, 1989.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, Boston, 2006.

[SKvE10] Olha Shkaravska, Rody Kersten, and Marko C. J. D. van Eekelen. Test-based inference of polynomial loop-bound functions. In *PPDP*, pages 99–108. ACM, 2010.

[SS05] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *CSFW*, pages 255–269. IEEE, 2005.

[Ste92] Joachim Steinbach. Proving polynomials positive. In *FSTTCS*, pages 191–202. Springer, 1992.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *POPL*, pages 355–364. ACM, 1998.

[SV06] Peter Selinger and Benoit Valiron. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science*, 16(3):527–552, 2006.

[SvET13] Olha Shkaravska, Marko C. J. D. van Eekelen, and Alejandro Tamalet. Collected size semantics for strict functional programs over general polymorphic lists. In *FOPARA*, pages 143–159. Springer, 2013.

[SvEvK09] Olha Shkaravska, Marko C. J. D. van Eekelen, and Ron van Kesteren. Polynomial size analysis of first-order shapely functions. *Logical Methods in Computer Science*, 5(2), 2009.

[SvKvE07] Olha Shkaravska, Ron van Kesteren, and Marko C. J. D. van Eekelen. Polynomial size analysis of first-order functions. In *TLCA*, pages 351–365. Springer, 2007.

[SW03] Davide Sangiorgi and David Walker. *The Pi-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2003.

[Tar51] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951.

[Ter01] Kazushige Terui. Light affine lambda calculus and polytime strong normalization. In *LICS*, pages 209–220. IEEE, 2001.

[Var82] Moshe Vardi. The complexity of relational query languages. In *STOC*, pages 137–146. ACM, 1982.

[Vas08] Pedro Vasconcelos. *Space cost analysis using sized types*. PhD thesis, University of St Andrews, 2008.

[VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

[VJFH15] Pedro Vasconcelos, Steffen Jost, Mário Florido, and Kevin Hammond. Type-based allocation analysis for co-recursion in lazy functional languages. In *ESOP*, pages 787–811. Springer, 2015.

[VW96] Heribert Vollmer and Klaus Wagner. Recursion theoretic characterizations of complexity classes of counting functions. *Theoretical Computer Science*, 163(1-2):245–258, 1996.

[Wal10] Johannes Waldmann. Polynomially bounded matrix interpretations. In *RTA*, volume 6 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[Wal15] Johannes Waldmann. Matrix interpretations on polyhedral domains. In *RTA*, volume 36 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[Wei95] Andreas Weiermann. Termination proofs for term rewriting systems by lexicographic path orderings imply multiply recursive derivation lengths. *Theoretical Computer Science*, 139(1-2):355–362, 1995.

[Wei00] Klaus Weihrauch. *Computable Analysis: An Introduction*. Springer, 2000.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT press, 1993.

[Wra93] Gavin C. Wraith. A note on categorical datatypes. In *CTCS*, volume 389, pages 118–127. Springer, 1993.

[Yam01] Toshiyuki Yamada. Confluence and termination of simply typed term rewriting systems. In *RTA*, pages 338–352. Springer, 2001.

[Yam18] Tomoyuki Yamakami. A schematic definition of quantum polynomial time computability. *arXiv preprint arXiv:1802.02336*, 2018.

[Zoo] Complexity Zoo. https://complexityzoo.uwaterloo.ca/Complexity_Zoo.

Résumé

In this thesis, we explore the different results obtained by the implicit complexity community over the past thirty years. These results generally consist in characterizing a given complexity class on a fixed programming language using a static analysis technique (interpretation, tiering, type system, light logic, ...). After listing the main techniques and highlighting their similarities, we will first study the intensionality results obtained for some these techniques, i.e. results that make it possible to compare the expressive power of these techniques or to extend the number of captured programs. Then, we will study crossovers, extensions of these techniques to other programming paradigms. We will then focus our analysis on the results obtained by using these techniques in the context of infinite data types such as streams, infinite sequences, real numbers, and higher-order functions. Finally, we will conclude this thesis by presenting some of the most interesting open questions in the field, including extensions to probabilistic models or quantum computing.

Abstract

Dans ce mémoire, nous explorons les différents résultats obtenus par la communauté de la complexité implicite au cours de ces trente dernières années. Ces résultats permettent en général de caractériser une classe de complexité donnée sur un langage de programmation fixé en utilisant une technique d'analyse statique (interprétations, tiering, systèmes de types, logiques légères, ...). Après avoir listé les principales techniques et avoir mis en avant leurs similitudes, nous étudierons dans un premier temps, les résultats d'intensionalité obtenus pour certaines de ces techniques, c'est-à-dire des résultats permettant de comparer le pouvoir d'expression de ces techniques ou permettant d'étendre le nombre de programmes capturés. Puis, nous étudierons des *"crossovers"*, des extensions de ces techniques à d'autres paradigmes de programmation. Nous focaliserons ensuite notre analyse sur les résultats obtenus en utilisant ces techniques dans le cadre de domaines infinis tels que les streams, les suites infinies, les nombres réels et les fonctions d'ordre supérieur. Enfin, nous conclurons ce mémoire en exposant certaines des questions ouvertes les plus intéressantes du domaine, dont des extensions à des modèles probabilistes ou de l'informatique quantique.