ICC and Probabilistic Classes

Paolo Parisen Toldin

Alma Mater Studiorum - Università di Bologna

DOTTORATO DI RICERCA IN INFORMATICA
Ciclo: XXV

Settore Concorsuale di afferenza: 01/B1
Settore Scientifico disciplinare: INF01

Implicit Computational Complexity and Probabilistic Classes

Presentata da: Paolo Parisen Toldin

Coordinatore Dottorato:
Maurizio Gabbrielli

Supervisore:
Simone Martini

Esame finale anno 2012

Dedicated to my parents Pietro and Maria Valeria
and to my brothers Francesco and Matteo
for supporting and sustaining me during these years
and particularly in the last hard times.

None of this could have been done without them.

Acknowledgement

In my opinion, this is one of the most difficult part of a thesis. No matter how long is the list of people you would like to thank, you will always end up forgetting someone. During these three years of PhD I met a lot of people. Some of them were professors or researcher or just postdocs and PhD students like me. Independently from who they were, all of them gave me something. They gave me ideas, different points of view, support and they taught me what is the meaning of "doing research". Looking back to these passed years, I think that PhD is not only a starting point for getting involved in research but also a school of hard knocks.

First of all I would like to thank Ugo Dal Lago, he followed me a lot during these three years, even though sometimes I did not deserve, and I learned a lot of things from him. I cannot also forget to thank Simone Martini, my advisor, for all the hints he gave me, for his support and his availability. Of course, without them, I wouldn't be here writing this thesis. I would like to thank both of them for telling me the right words at the right time. I express my gratitude also to Jean-Yves Moyen, professor in Paris XIII. I met and work with him during my period abroad. He open my eyes to new interesting research direction in which I would like to go on in the future.

I would like also to thank the reviewers of my thesis, Jean Yves Marion, Luca Roversi and Patrick Baillot, for all of the suggestions about my thesis.

Even if they are not co-author or strictly related with my PhD, I would like to thank, in random order: Jacopo Mauro, Michael Lienhardt, Valentina Varano, Lorenzo Coviello, Emanuele Coviello, Marco Solieri, Giulio Pellitta, Matteo Serpelloni, Valeria dal Degan, Luca Simonetti, Davide Benetti, Gianluca Rigon, Manuela Campan, Alice Zanivan and Arianna Zanivan. Friends are the ones who make your life more tasty. Their support was and it is still nowadays fundamental. For all of those I didn't say "thank" explicitly, thank you again.

Contents

Chapter 1

Introduction

> In omnibus autem negotiis, prius, quam aggrediare, adhibenda est præparatio diligens.
>
> Marcus Tullius Cicero

Logic, from greek λογος can be defined as the field that studies the principles and rules of reasoning. By starting from axioms it gives tools for inferring the truthfulness or falsity of statements. In the fields of computer science, it is widely used for different application but all of them deal with properties of programs.

Computational Complexity is that kind of research field that focus on the minimal and necessary quantity of resources (usually these are time and space) for solving a definite kind of decidable problem. Talking about problems means talking about functions. Indeed, a problem, in its decisional form, can be seen as a function. It has some well defined inputs and it asks for an output of "yes" or "no". There are infinite ways to implement such function and compute the solution. We need an algorithm, so a finite procedure that step-by-step computes basic operations, and we would like to find the best one. For every problem we have many ways to compute the correct answer. There are intrinsic limits in every problem, such that for every algorithm there is no way to compute the solution in time or space less than a specific amount (respect to the size of inputs). Sometimes, these limits are not known.

In this thesis we investigate Implicit Computational Complexity applied to Probabilistic complexity classes. ICC (Implicit Computational Complexity) is a research field that tries to combine Computational Complexity with logic. By using the Curry–Howard correspondence (proofs-as-programs correspondence and formulae-as-types correspondence) we can easily talk about step of calculus in a program and verify properties by using the tools of logic. We are mainly interested in complexity properties, such as the execution time. We present programming languages and methodologies in order to capture the complexity and expressive power of Probabilistic Polynomial Time Class.

All of these works find application in all of those systems where space and time resources (memory and CPU time) are important; knowing a priori the space and time needed to execute a given program is a precious information. While talking about "systems" we are not only consider critical systems or embedded systems having, in real terms, time and space constrains; we are also considering usual systems like cellular phones or tablet devices, that are general purpose systems (so that we can easily run new programs).

In general, it is a challenging task to find efficient algorithms for computationally expensive problems. Nowadays, efficient solutions to computationally hard problems are probabilistic rather than deterministic. A lot of problems are indeed solved with techniques and methodologies built on statistical analysis and probability, e.g. algorithms based on Monte Carlo and Las Vegas schemes are widely used not only in physics but also in other fields. From this point of view, it appears interesting to study implicit characterisations of probabilistic complexity classes in order to develop new programming languages able to internalise bounds both on complexity and on error probability. These frameworks will allow the development of statistical methods for static analysis of algorithms' complexity. All of these works are correlated with the problem of finding efficient algorithms for solving problems.

There are many ways in computer science to focus on this problem and the logical one is one of the most interesting. Indeed, logic is a formalism that allows you to work more easily on complex systems because it gives a higher point of view. There are also several approaches in ICC; they vary from recursion theory to proof theory and to model theory. We followed the path dealing with recursion theory by using restrictions on usage of recursion.

1.1 Contributions

The main contribution produced in this thesis is the extension of ICC techniques to the probabilistic complexity classes. We mainly focus on **PP** (which stands for Probabilistic Polynomial Time) class, showing a syntactical characterisation of **PP** and a static complexity analyser able to recognise if an imperative program computes in Probabilistic Polynomial Time. We tried to go deeply and get characterisations of other probabilistic polynomial classes, such as **BPP**, **RP**, **ZPP**, but it seems that a syntactical characterisation of these "semantic classes" is really hard problem and would imply the solution of some old open problems. We show parametric characterisation of these classes.

The first work, syntactical characterisation of **PP**, is mainly based on a work of M. Hofmann [21]. His work presents a characterisation of the class `P` (polynomial time) by semantical proof. We extend his work to the Probabilistic Polynomial Time class, giving a syntactical and constructive proof and obtaining also subject reduction. The second work, static analyser for Probabilistic Polynomial Time, is mainly based on a work of Neil D.Jones and Lars Kristiansen [25]. We extend and adjust the analysis in order to achieve soundness and completeness for **PP**. Moreover, our analysis runs in polynomial time and it is quite efficiently. Some benchmarks are also presented in order to show that even if in the worst case, of our static analyser, is bounded by a polynomial of degree five, the average case seems to grow with a rate that is less than linear in the size of number of variables used in the program.

1.2 Thesis outline

In this subsection we give an overview of the contents presented in each chapter. Basically, the thesis is divided into two parts. The first one talks about ICC applied to a variation of lambda calculus and the second part deals with imperative programming languages and methodologies for inferring the running time of a program. In real terms, the thesis is so subdivided:

Chapter 2 We give an introduction about Computational Complexity and stochastic computations. We present the concept of the Probabilistic Turing Machine and we show how it works.

Chapter 3 We introduce the topic of Implicit Computational Complexity. We give an overview of it and then we proceed by introducing fundamental papers such as [11], [6], [21].

Chapter 4 One of the main original contribution is presented. We show how it is correlated with the previously introduced papers and which are its main points.

Chapter 5 We present a new topic, about static analysers for time complexity inference. We show how ICC is strictly correlated with this different topic. We focus on the imperative paradigm and we present fundamental papers, such as [32] and [25], used for developing our analysis.

Chapter 6 Second original contribution is proposed, showing its application to Probabilistic Polynomial Time and presenting its performance with some benchmarks.

Chapter 7 Conclusions and future develops.

Chapter 2

Computational Complexity

> Chance phenomena, considered
> collectively and on a grand scale,
> create non-random regularity.
>
> Andrey Kolmogorov

Computer Science, as a science, is based, principally, on Computability Theory, a research field that was born around 1930. Its main purpose is to understand what is actually computable and what is not. It is for this reason that nowadays it is possible to give a formal definition of the intuitive idea of computable function. Its discoveries let us know the limits and the potentiality of Computer Science.

Computability Theory started to be developed without referring to a specific real calculator. Initial works about λ-calculus [10] and Turing Machine [39] were published few years before the construction of the first modern programmable calculating machine (the famous "Z3" by Konrad Zuse in 1941).

Computational Complexity theory is a research branch that came from Computability Theory. Its main purpose is to classify computational problems in sets, called computational sets. They are classified according to the "difficulty" to solve them. The measure of difficulty is the quantity of resources time and space that is required for solving a particular kind of problem. The subjects of the analysis in Computational Complexity theory are the so called "computational problems". These are problems, mathematically formalised, for which we are searching an algorithm to solve them.

Should be clear that not all the problems are considered by this research field. First we need to separate the problems that are formalisable from the ones that cannot be. Then

we need to separate the ones for which exists an algorithm ("decidable problem") from the others ("non decidable").

An example of undecidable problem is the famous "halting problem". The halting problem is formalised in the following way. We have as input a program C and we would like to know if the program terminates or not. There is no algorithm able to solve this problem. The proof is quite easy and uses the technique of diagonalization.

Proof: We indicate with $\downarrow$ the property that a program terminates. On the contrary, the symbol $\uparrow$ indicates that a program does not terminate. Suppose that there is an algorithm f that on input C and i (the input of the program C) is telling "yes" (expressed by the value 1) or "no" (expressed by the value 0) according on the termination of the program $C(i)$.

$$f(C,i) = \begin{cases} 1 & \text{if } C(i)\downarrow \\ 0 & \text{if } C(i)\uparrow \end{cases}$$

If so, we are also able to write down an algorithm g behaving in slight different way.

$$g(i) = \begin{cases} 0 & \text{if f(i,i)=0} \\ \uparrow & \text{otherwise} \end{cases}$$

But what is the expected result of applying g with itself? $g(g)$ terminates only if $g(g)$ does not terminates and viceversa. Here is the paradox. We can conclude that there is no algorithm able to solve the halting problem. □

Famous results in this field are the hierarchy theorems. First we need to introduce some notions and some well known complexity classes and then we will be able to understand the relation between then.

2.1 Turing Machines

In 1937, Alan Turing [39] presented a theoretical formalism for an automatic machine. The so called "Turing Machine" (TM in the following) is a model machine that operates over an, hypothetical, infinite tape. The tape is subdivided in cells where a symbol from

a finite alphabet can be read or written. The machine has a head able to move along the whole tape and able to read and write symbols. The behaviour of a TM is well determined by a finite set of instructions. For every combination of symbol read and state machine, the TM evolves in (possibly) new state, overwrites with new symbol the cell pointed out by the head and could move the head left or right. This is called "transition function".

Space and time consumption are defined, respectively, as the number of steps required for a TM to reach a final state and as the number of cells written at least once during its computation.

There are several kind of Turing Machine. The usual one works with one tape, but an easy extension uses more tapes. The computational power does not change. There are two particular Turing Machines interesting for the work presented in this thesis. The first one is the Non Deterministic Turing Machine (NTM in the following) and the latter one is the Probabilistic Turing Machine (PTM in the following).

We introduce briefly the NTM, while we leave the PTM for a better introduction later. A Non Deterministic Turing Machine is a TM where the transition function works differently. Instead of having a single output, the function can lead the machine to different configurations. One can imagine as the NTM branches into many copies of itself, where each of it computes a different transition. So, instead of having, as in a TM, one possible computational path, the NTM has more computational paths: a tree. If any branch of the tree stops in an accepting configuration, we say that a NTM accepts the input. Viceversa, an NTM rejects the input is all of its paths lead to a rejecting configuration. Of course, a NTM does not represent an implementable model machine, but it is useful for describing and classifying particular problems.

We can easily define the following complexity sets:

- **L** is the class of problems that are solvable by a Turing Machine in logarithmic space.
- **NL** is the class of problems that are solvable by a non deterministic Turing Machine in logarithmic space.
- **P**: is the class of problems solvable by a Turing Machine in polynomial time.
- **NP**: is the class of problems solvable by a non deterministic Turing Machine in polynomial time.

- **PSPACE**: is the class of problems that are solvable by a Turing Machine in polynomial space.
- **NPSPACE**: is the class of problems that are solvable by a non deterministic Turing Machine in polynomial space.
- **EXP**: is the class of problems that are solvable by a Turing Machine in exponential time.

Is quite clear that from the previous definition, some inclusions between this classes hold. Indeed, a TM can be seen as a particular case of a NTM. Moreover, if a Turing Machine (deterministic or not) is working in polynomial time, it cannot use more than polynomial space (modulo the number of the possible tapes).

$$\begin{aligned}
\mathbf{PTIME} &\subseteq \mathbf{EXP} \\
\mathbf{PTIME} &\subseteq \mathbf{NP} \\
\mathbf{PTIME} &\subseteq \mathbf{PSPACE} \\
\mathbf{L} &\subseteq \mathbf{PSPACE} \subseteq \mathbf{NPSPACE} \\
\mathbf{L} \subseteq \mathbf{NL} &\subseteq \mathbf{NPSPACE}
\end{aligned}$$

Let a "proper complexity function" be a non decreasing function $f : \mathbf{N} \to \mathbf{N}$ such that there exists a TM able to produce, on every input of length n, $f(n)$ symbols in time bounded by $n+f(n)$ and space bounded by $f(n)$. Given a function $f : \mathbf{N} \to \mathbf{N}$, $\mathbf{TIME}(f)$ is the complexity class of all of those languages decidable by a TM in time bounded by function f. Similar, given a function f, $\mathbf{NTIME}(f)$ is the complexity class of all of those languages decidable by a NTM in time bounded by the function f. Clearly, we can define the class of languages $\mathbf{SPACE}(f)$ and $\mathbf{NSPACE}(f)$ with the obvious meaning. An important result in literature shows that is it possible to extend the releations between complexity classes by proving the following two theorems:

Theorem 2.1 (Non Deterministic Time versus Deterministic Space) *Given a proper complexity function f on input n, if a problem is solvable by a NTM in time $f(n)$, then can be solved by a TM in space $f(n)$.*

$$\mathbf{NTIME}(f(n)) \subseteq \mathbf{SPACE}(f(n))$$

Theorem 2.2 (Non Deterministic Space versus Deterministic Time) *Given a proper complexity function f on input n, if a problem is solvable by a NTM in space $f(n)$, then can be solved by a TM in time $m^{\log n + f(n)}$, where $m > 1$ is a constant.*

$$\mathbf{NSPACE}(f(n)) \subseteq \mathbf{TIME}(m^{\log n + f(n)})$$

From all the previous observations, we can easily create the well known hierarchy:

$$\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{PTIME} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{NPSPACE} \subseteq \mathbf{EXP}$$

In order to know if there are or not tight inclusions between these sets we need to prove more theorems. These are called hierarchy theorems. What happens if we give more computational time to a TM? is it able to compute more function? Consider the following problem:

Definition 2.1 (Halting problem in fixed number of steps) *Given a proper complexity function $f(n) \geq n$, define*

$$\mathcal{H}_f = \{(M; x) | M \text{ accepts } x \text{ within } f(|x|) \text{ steps } \}$$

Notice that the condition for being accepted in the language, requires the machine M to halt before $f(|x|)$ steps. If it requires more than these steps, the pair (M, x) is rejected. The set $\mathcal{H}_f$ is so decidable.

We can prove the following properties:

$$\mathcal{H}_f \in \mathbf{TIME}(f(n)^3)$$
$$\mathcal{H}_f \notin \mathbf{TIME}(f(\lfloor \frac{n}{2} \rfloor))$$

We have all the ingredients to present the following main result. Knowing that $\mathcal{H}_f \in \mathbf{TIME}(f(n)^3)$ but not in $\mathbf{TIME}(f(\lfloor \frac{n}{2} \rfloor))$ we can put $n = 2m + 1$ and obtain that there is a problem solvable in $\mathbf{TIME}(f(2m+1)^3)$ that cannot be solvable in $\mathbf{TIME}(f(m))$

Corollary 2.1 *The class of problems solvable in polynomial time is strictly included in the class of problem solvable in exponential time by a Turing Machine.*

$$\mathbf{PTIME} \subset \mathbf{EXP}$$

Proof: Clearly every polynomial $p(n)$ is definitely minor than 2^n. So we can have the following chain

$$\mathbf{PTIME} \subseteq \mathbf{TIME}(2^n) \subset \mathbf{TIME}((2^{2n+1})^3) \subseteq \mathbf{EXP}$$

□

Given a proper complexity function f, we can prove in similar way that $\mathbf{SPACE}(f(n)) \subset \mathbf{SPACE}(f(n)\log f(n))$ and easily conclude with the following corollary.

Corollary 2.2 *The class of problems solvable in logarithmic space is strictly included in the class of problems solvable in polynomial space by a Turing Machine.*

$$\mathbf{L} \subset \mathbf{PSPACE}$$

There is another well known theorem that we haven't yet introduced. We have seen the relation between **PSPACE** and **NPSPACE**. Clearly, $\mathbf{PSPACE} \subseteq \mathbf{NPSPACE}$. Even if could seem counteractive, it is also true that $\mathbf{NPSPACE} \subseteq \mathbf{PSPACE}$. Every problem in **NPSPACE** can be solved with a Turing Machine in quadratic space respect to the one required by the non deterministic solution.

Theorem 2.3 *For every proper complexity function f it holds that:*

$$\mathbf{NPSPACE}(f(n)) \subseteq \mathbf{PSPACE}(f(n)^2)$$

Proof: Let M be a NTM working on space $f(n)$. The graph of all the possible configurations $G(M, x)$ has $O(k^{f(|n|)})$ nodes. So, knowing if x is a positive instance for the given problem or not is equal to solve a reachability problem on this kind of graph. It has been proved in literature [35] that reachability problem belongs to **L**. It can be solved in space $(\log n)^2$. So, we can solve our reachability problem by using savitch solution and get the result in space $O((\log k^{f(|n|)})^2)$ that is $O(f(|n|)^2)$. □

All of these results lead us to the final chain of relations:

$$\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{PTIME} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE} \subseteq \mathbf{EXP}$$

$$\mathbf{L} \subset \mathbf{PSPACE}$$

$$\mathbf{PTIME} \subset \mathbf{EXP}$$

There are more complexity sets than these and for most of them is not well clear the relation between each other. There is an online database containing quite all the classes introduced in literature; it's called "*the complexity zoo*"[1]. We are interested mainly on probabilistic algorithms and their complexity classes and we are going to introduce them.

2.2 Stochastic Computations

Randomised computation is central to several areas of theoretical computer science, including cryptography, analysis of computation dealing with uncertainty and incomplete knowledge agent systems. In the context of computational complexity, there are some complexity classes that deal with probabilistic computations. Some of them are nowadays considered as very closely corresponding to the informal notion of feasibility. In particular, a complexity class called **BPP**, which stands for "Bound Probabilistic Polynomial Time", is consider the right candidate containing all the feasible problems. A solution to a problem in **BPP** can be computed in polynomial time up to any given degree of precision: **BPP** is the set of problems which can be solved by a probabilistic Turing machine working in polynomial time with a probability of error bounded by a constant strictly smaller than $1/2$.

2.2.1 Probabilistic Turing Machines

There are two ways to think about a Probabilistic Turing Machine. One definition says that a Probabilistic Turing Machine is a particular deterministic Turing Machine working on two tapes, where one tapes is a read-only-once tape with random $0, 1$ values and the other is the usual working tape. The other definition describes a Probabilistic Turing Machine as a non deterministic Turing Machine with two transition functions. At each steps, the machine according to a probability distribution decides which transition function it has to apply.

In the following we will use the latter definition, because of its easy of use. Our Probabilistic Turing Machines will use a fair tossing coin. It does not matter if the Probabilistic Turing Machine works with $0/1$ fair tossing coin or something else with different probability distribution. It has been shown [17] that the expressiveness of a Probabilistic Turing Machine does not change while changing the probability distribution.

Formally, a Probabilistic Turing Machine is a tuple $M = (Q, q_0, F, \Sigma, \sqcup, \delta)$, where Q is the finite set of states of the machine; q_0 is the initial state; F is the set of final states of M; Σ is the finite alphabet of the tape; $\sqcup \in \Sigma$ is the symbol for empty string; $\delta \subseteq (Q \times \Sigma) \times (Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\})$ is the transition function of M. For each pair $(q, s) \in Q \times \Sigma$, there are exactly two triples (r_1, t_1, d_1) and (r_2, t_2, d_2) such that $((q, s), (r_1, t_1, d_1)) \in \delta$ and $((q, s), (r_1, t_1, d_1)) \in \delta$.

Definition 2.2 *We say that a Probabilistic Turing Machine M on input x runs in time $p(|x|)$ if $M(x)$, for every possible computational path, requires at most $p(|x|)$ steps to terminate.*

Definition 2.3 *We say that a Probabilistic Turing Machine M on input x runs in space $q(|x|)$ if $M(x)$, for every possible computational path, requires at most $q(|x|)$ worktape cells during its execution.*

While dealing with probability and random computation it is reasonable asking if the PTM could answer in a wrong way. In a PTM running in time t the number of possible outcomes is 2^t. Thus, the probability for a PTM M to answer "yes" is exactly the fraction of outcomes that return "yes". So, we can define the probability error of a Probabilistic Turing Machine.

Definition 2.4 *Let M be a Probabilistic Turing Machine for a language L. Let $E[x \in L]$ the expected answer "yes" or "no" concerning $x \in L$: $E[x \in L]$ returns "yes" iff $x \in L$. Let $\epsilon_1 \in [0, 1]$ representing a probability. We say that M on input x is working with probability error ϵ if the fraction of computational paths of $M(x)$ not leading to answer $E[x \in L]$ is less than ϵ, that is $P(M(x) = \neg E[x \in L]) \leq \epsilon$.*

We can extend the notion of probability error to the whole machine, independently from which computation it is performing. In this case we have to introduce the notions of *Two-sided error*, *One-Sided error*, *Zero-sided error*.

Definition 2.5 (Two-sided error) *Let M be a Probabilistic Turing Machine for a language L. We say that M is working with probability error $\epsilon > 0$ if for every string x we have that $P(M(x) = \neg E[x \in L]) \leq \epsilon$.*

Notice that in the definition of Two-sided error we have defined a unique probability error that hold both for false positive and false negative answers. However, many probabilistic algorithms tend to give, in at least one side, the right answer. That means, as example, that they could provide right answer when $x \in L$ and in the other case they could give false positive answer, according to a certain probability error. Moreover, some probabilistic algorithm, such as the Random Quicksort, are able to return correct solutions without error. This leads us to define the "One-sided error" and the "Zero-sided error".

Definition 2.6 (One-sided error) *Let M be a Probabilistic Turing Machine for a language L. We say that M is working with one-sided error $\epsilon > 0$ if one of the following holds:*

- *if $x \in L$, $P(M(x) = \text{"no"}) \leq \epsilon$ and if $x \notin L$, $P(M(x) = \text{"yes"}) = 0$.*
- *if $x \in L$, $P(M(x) = \text{"no"}) = 0$ and if $x \notin L$, $P(M(x) = \text{"yes"}) = \leq \epsilon$.*

Definition 2.7 (Zero-sided error) *Let M a Probabilistic Turing Machine for a language L. We say that M is working with zero-sided error is for every string in input x we have that $P(M(x) = \neg E[x \in L]) = 0$. The machine cannot give wrong answer, even if it is working with a random source.*

Example 2.1 *L*et's see some example on how a Probabilistic Turing Machine works. Suppose we want to describe PTM that with probability 1/2 flips the bit read. The transition relation δ would be the following one:

$$\{(q_0, 1), (q_0, 1, \rightarrow), (q_0, 0, \rightarrow)\} \in \delta$$
$$\{(q_0, 0), (q_0, 0, \rightarrow), (q_0, 1, \rightarrow)\} \in \delta$$
$$\{(q_0, \sqcup), (q_f, \sqcup, \downarrow), (q_f, \sqcup, \downarrow)\} \in \delta$$
$$\{(q_f, \sqcup), (q_f, \sqcup, \downarrow), (q_f, \sqcup, \downarrow)\} \in \delta$$

The machine is working with ternary alphabet $0, 1, \sqcup$ and the state q_f represents the final state. As can be easily checked, our PTM scans all the tape till the end and with probability 1/2 flips the bits read.

□

2.2.2 Probabilistic Polytime Hierarchy

Based on Probabilistic Turing Machine, we can define complexity classes. The most important probabilistic polynomial classes are presented in Figure 2.1. Note that we emphasised some sets with a dotted line. They represent semantic sets [33].

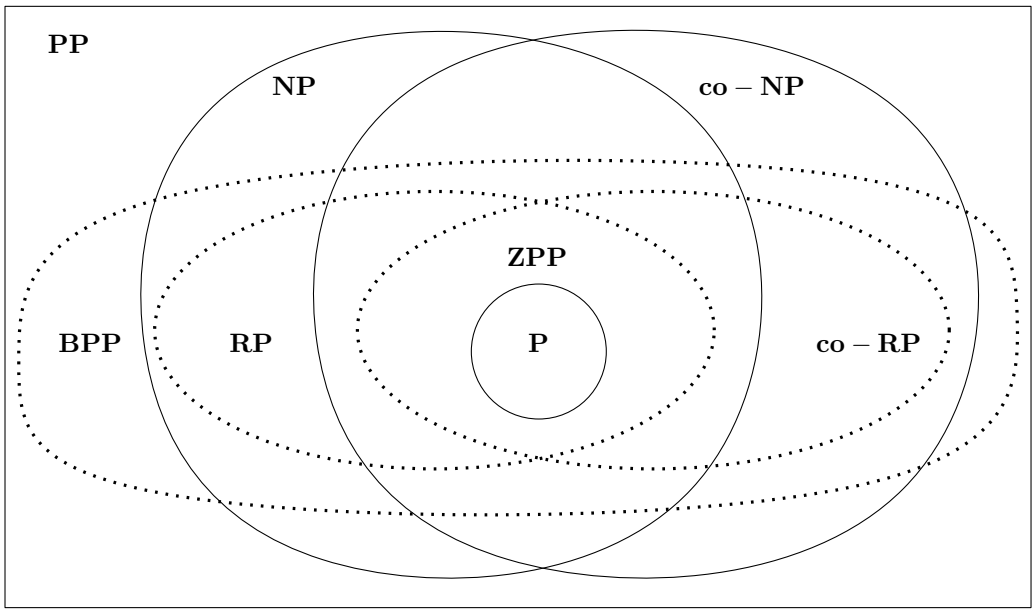


Figure 2.1: Probability Polytime Hierarchy

- The outermost class is called **PP**, which stands for *Probabilistic Polynomial Time*. It describes problems for which there is a random algorithm solving the problem with probability greater than $\frac{1}{2}$. It is important to notice that no restriction is made about *"how much"* is greater than $\frac{1}{2}$. Therefore, the probability error both for "yes" and "no" answers is strictly less than $\frac{1}{2}$.
- **co-NP** is the complementary class of **NP**.
- **RP** is the set of problems for which exists a polytime Monte Carlo algorithm that if the right answer is "no" then it answers without errors, otherwise it answers with an error $e < \frac{1}{2}$ [33]. Formally speaking, given a Probabilistic Turing Machine M for the language $\mathcal{L} \in \mathbf{RP}$ on all possible input x, $x \in \mathcal{L} \Rightarrow P(M(x) = \text{"yes"}) > \frac{1}{2}$ and $x \notin \mathcal{L} \Rightarrow P(M(x) = \text{"yes"}) = 0$
- **co-RP** is the complementary class of **RP**. Given a Probabilistic Turing Machine M for the language $\mathcal{L} \in$ **co-RP** on all possible inputs x, $x \in \mathcal{L} \Rightarrow P(M(x) = \text{"yes"}) = 1$

and $x \notin \mathcal{L} \Rightarrow P(M(x) = \text{“yes”}) \leq \frac{1}{2}$.

- **ZPP** is the set of problems for which a Las Vegas algorithm exists, solving them in polynomial time [33]. Indeed, it is recognised as the intersection set of **RP** with **co-RP**. PRIME [1] was discovered to be in **ZPP** [2].
- **BPP** is the set of decision problems solvable by a probabilistic Turing machine in polynomial time with an error $e \leq \frac{1}{3}$ for all possible answers. Formally speaking, given a Probabilistic Turing Machine M for the language $\mathcal{L} \in \mathbf{BPP}$ on all possible input x, $x \in \mathcal{L} \Rightarrow P(M(x) = \text{“yes”}) > \frac{2}{3}$ and $x \notin \mathcal{L} \Rightarrow P(M(x) = \text{“yes”}) < \frac{1}{3}$.

Let's focus not on the relations between probabilistic classes and the other well known sets. The definitions of probabilistic polynomial classes that we have seen before give rise to the following hierarchy.

$$\mathbf{PTIME} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP} \subseteq \mathbf{BPP} \subseteq \mathbf{PP}$$

$$\mathbf{PTIME} \subseteq \mathbf{ZPP} \subseteq \mathbf{CO-RP} \subseteq \mathbf{BPP} \subseteq \mathbf{PP}$$

Nowadays, it is not known if there are strict inclusions between these classes. There are a lot of conjectures about the equivalence of some classes. Some researchers believe that **BPP** = **PTIME**, but no evidence has not yet been found. Moreover it has been shown that there are a lot of consequences if some claim will be solved. It has been proved [26] that if **NP** $\subseteq$ **BPP**, then **RP** = **NP**.

2.3 Semantic Classes

Classes as **RP**, **co-RP**, **ZPP** are considered semantic classes, differently from syntactic classes as **PTIME** or **NP**. Given a Turing Machine N defining a language l, it is not so difficult to check if the language l belongs to these sets or not. This verification is not easy if we are considering semantic classes.

Consider, as example, the class **RP**. Given a Turing Machine, it is not easy to check if it characterises a language in **RP**. Indeed, for TM N to define a language l in **RP** it has the property that on all inputs it outputs unanimously “no” or “yes” on majority. Most non-deterministic TM behave differently in at least some inputs.

[1] The problem asks to distinguish prime numbers from composite numbers and of resolving the latter in to their prime factors.

One of the main problems with semantic classes is that there are no known complete languages for them. The standard complete language for syntactic sets is the following:

$$\{(M, x) : M \in \mathcal{M} \text{ and } M(x) = \text{“yes”}\}$$

where $\mathcal{M}$ is a set of TM sufficient to define the class.

Note that **PP** is a syntactic and not a semantic class. Indeed, any non-deterministic polynomially time bounded TM defines a language in this class. No other properties are required to belong to **PP**, since the statement on the probability error is too weak (“accept on majority”).

Chapter 3

Implicit Computational Complexity

> Logic and mathematics seem to be the only domains where self evidence manages to rise above triviality
>
> Willard van Orman Quine

3.1 Introduction

Implicit computational complexity (ICC) combines computational complexity, mathematical logic, and formal systems to give a machine independent account of complexity phenomena.

The "machine independent characterisation has not to be confused with the so called Structural Complexity. The main aim of Structural Complexity is the study of the relations between various complexity classes and their global properties [20]. Structural Complexity focuses on logical implications among certain unsolved problems, that have to do with complexity classes, and explores the power of various resource bounded reductions and the properties of the corresponding complete languages in order to understand the internal logical structure of the classes. It gives independent characterisations of complexity sets but the main aim is different.

In ICC the main purpose is to characterise the complexity classes so that it will be possible to develop new languages able to internalise complexity bounds. The known results go in the direction of creating new languages that have bounds in complexity, for example as the bounds in polynomial time or log space.

Many fields of mathematical logic influence this research field. We can include for sure recursion theory, proof theory (by using the Curry-Howard isomorphism) and model theory. This variety of fields reflects the variety of different approaches in ICC.

In literature we can find approaches that use Girard's Linear Logic [18] and its subsystems, others that work directly on functions, by limiting primitive recursion, and others that work by implementing these features in λ-calculus. A wide community of ICC works on the first approach and has produced a lot of key works. We would like to recall few of them that were able to characterise the class **PTIME**, such as Light Linear Logic of J.Y. Girard [19], Soft Linear Logic of Y. Lafont[28] and Light Affine Linear Logic of A. Asperti [4]. Concerning the limitation on recursion, there is a good survey from M.Hofmann [22].

ICC has been successfully applied to the characterisation of a variety of complexity classes, especially in the sequential and parallel modes of computation (e.g., **FP** [6, 29], **PSPACE** [30, 16], **L** [24], **NC** [9]). Its techniques, however, may be applied also to non-standard paradigms, like quantum computation [13] and concurrency [12]. Among the many characterisations of the class **FP** of functions computable in polynomial time, we can find Hofmann's *safe linear recursion* [21], a higher-order generalisation of Bellantoni and Cook's *safe recursion* [5] in which linearity plays a crucial role.

3.2 The Intrinsic Computational Difficulty of Functions

In 1965 Cobham published a paper called "The Intrinsic Computational Difficulty of Functions" [11]. His main purpose was to understand how it was possible to restrict the primitive recursion in order to capture only polynomial time computable function.

Recalling the definition of primitive recursion

$$f(0, \overline{y}) = g(\overline{y}) \tag{3.1}$$

$$g(\mathbf{S}(x), \overline{y}) = h(y, f(x, \overline{y}), \overline{y}) \tag{3.2}$$

we can easily check that if we are using unary encoding the number of recursion calls needed to execute the recursion are exponentially correlated with the length of y. So, Cobham defined "Recursion on notation":

$$f(0, \overline{y}) = g(\overline{y}) \tag{3.3}$$

$$f(x, \overline{y}) = h(x, f(\lfloor x/2 \rfloor, \overline{y}), \overline{y}) \tag{3.4}$$

This solution, unfortunately, does not solve the problem. Indeed, we are still able to write down non polynomial functions. Consider the following example.

Define $r(n, y)$ as:

$$r(0, y) = y \tag{3.5}$$

$$r(x, y) = 4 \cdot r(\lfloor x/2 \rfloor, y) \tag{3.6}$$

It is easy to check that

$$r(x, y) = (2^{|x|})^2 - y$$

and hence,

$$|r(x, y)| = 2(|x| + 1) + |y|$$

Now, if we define a function e in the following way

$$q(0) = 1 \tag{3.7}$$

$$q(z) = r(q(\lfloor z/2 \rfloor), 1), \text{ if } z > 0 \tag{3.8}$$

we obtain a function of super-polynomial growth. Indeed:

$$|q(z)| = |r(q(\lfloor z/2 \rfloor), 1)| = 2(|q(\lfloor z/2 \rfloor)| + 1) + 1 \geq 2^{|z|}$$

The solution proposed by Cobham is to give a polynomial bound a priori on the function that we could write with this system. In order to do that it is necessary to define a basic function that could give us a polynomial bound. This function is the so called "smash function" $x\#y = 2^{|x| \cdot |y|}$.

The functions f of Cobham's system can be defined starting from basic functions as:

$$\mathbf{S}_0(x) = 2x \tag{3.9}$$

$$\mathbf{S}_1(x) = 2x + 1 \tag{3.10}$$

$$x\#y = 2^{|x| \cdot |y|} \tag{3.11}$$

and the "limited recursion on notation":

$$f(\overline{x}, 0) = g(\overline{y})$$

$$f(\overline{x}, \mathbf{S}_i y) = h_i(\overline{x}, y, f(\overline{x}, y))$$

$$f(\overline{x}, y) \leq r(\overline{x}, y)$$

where g, h_i, r are previously defined functions of the system.

3.3 Safe recursion

We are going to present here a fundamental step concerning implicit characterisation of class P by using functions and particularly kind of composition of functions. This work is called "A new recursion-theoretic characterisation of the polytime functions" by S. Bellantoni and S. Cook [6]. A very similar work was made at the same time by D. Leivan "Stratified functional programs and computational complexity" [29] in 1993. Basic idea is, more or less, the same. They both provide a way to restrict the functional recursion.

3.3.1 Class *B*

We define a class "B" of functions [6], working on binary strings. Each function in this class has two kinds of inputs. The first one is the so called "normal" and the latter is called "safe". We are allowed to make recursion on values that are "normal" but not on the ones that are "safe". We can "promote" a variable "normal" to "safe" but not viceversa.

The class is defined as the smallest class containing the following functions

1. (Constant) 0 (the zero constant function).
2. (Projection) $\pi_i^{n+m}(x_1, x_n; x_{n+1}, x_{n+m}) = x_i$, where $1 \leq i \leq n+m$.
3. (Successor) $\mathbf{S}_i(; a) = 2a + i$, where $i \in \{0, 1\}$.
4. (Predecessor) $\mathbf{p}(; 0) = 0$ and $\mathbf{p}(; ai) = a$, where $i \in \{0, 1\}$.
5. (Conditional)
$$\mathbf{C}(; a, b, c) = \begin{cases} b \text{ if } a \bmod 2 \\ c \text{ otherwise} \end{cases}$$

and closed under

6. (Predicative Recursion on Notation) We define a new function f in the following way
$$f(0, \overline{x}; \overline{a}) = g(\overline{x}; \overline{a})$$
$$f(yi, \overline{x}; \overline{a}) = h_i(y, \overline{x}; \overline{a}, f(y, \overline{x}; \overline{a})) \text{ for } yi \neq 0$$

where $i \in \{0, 1\}$ and where g and h_i are functions in B.

7. (Safe Composition) We define a new function f in the following way:

$$f(\overline{x}; \overline{a}) = g(\overline{h}(\overline{x};); \overline{r}(\overline{x}; \overline{a}))$$

where g, $\overline{h}$, $\overline{r}$ are functions in B.

The functions that are polytime are exactly the ones in B that have no safe inputs. Recall that the inputs on the left side (respect to the semicolons), are the "normal" ones, the other are the "safe" ones. The term "safe" is used here to intend that is safe to substitute larger values without compromise polynomiality of the function.

Let's take a look at the definitions. First of all, let's see how to promote variables from normal to safe but not viceversa. Looking at rule for Safe Composition we can see that we can shift a normal variable to the right side but it is not allowed to shift a safe variable to the left side.

Predicative Recursion on Notation ensures that values computed stay on the right side. This means that the depth of the subrecursions computed by h_i cannot depend on values recursively computed; it is a way to prevent possible complexity blowup.

Functions defined by rules $3, 4, 5$ operate only on safe inputs. Of course, this is not restrictive, since we can promote normal variables to safe position.

3.3.2 *B* contains PTIME

Bellantoni & Cook prove the inclusion of the function class B into the function class **PTIME** by referring to Cobham characterisation **FPTIME** in [11]. Indeed it has been proved in [34] and [38] that Cobham functions are all the polytime functions.

Lemma 3.1 *Let f be any polytime function. There is a function f' in B and a monotone polynomial p_f such that $f(\overline{a}) = f'(w; \overline{a})$, for all $\overline{a}$ and for all w.*

Proof: By induction on the length of the derivation of f as a function in the Cobham class.

- If f is constant, projection or successor function, then f is easily defined as the corresponding constant, projection or successor of the class B and in this case p_f is 0.

- If f is a composition $g(\overline{h}(\overline{x}))$, then f' is defined as the safe composition in B. $f'(w;\overline{x}) = g'(w;\overline{h}(w;\overline{x}))$. Knowing that functions $\overline{h}$ are in the Cobham class, they have a bound in their values. Call them $\overline{q}_h$. So, for every $h_i \in \overline{h}$ we have $h_i(\overline{x}) \leq q_{h_i}(\overline{x})$.

  We define our polynomial bound $p_f(|\overline{x}|)$ as $p_g(\overline{q}_h(|\overline{x}|)) + \sum_j p_{h_j}(|\overline{x}|)$.

  It's easy to check that property holds.

- If f is defined by recursion on notation, then we are in the following case:

$$\begin{aligned} f(\overline{x}, 0) &= g(\overline{y}) \\ f(\overline{x}, \mathbf{S}_i y) &= h_i(\overline{x}, y, f(\overline{x}, y)) \\ f(\overline{x}, y) &\leq r(\overline{x}, y) \end{aligned}$$

  for some $r(\overline{x}, y)$ in Cobham class.

  By induction hypothesis we get functions g', h'_1, h'_2. In order to define function f' and having a more readable proof, we need to define some new functions. Notice also that we cannot easily define f' by recursion on the variable x, since it would appear on right side, in normal position; this is not allowed by definition of Predicative Recursion on Notation. The new functions are the following ones:

$$\begin{aligned} \mathbf{P}(0; b) &= b \\ \mathbf{P}(ai; b) &= \mathbf{p}(; \mathbf{P}(a; b)) \\ \mathbf{P}'(a, b;) &= \mathbf{P}(a; b) \\ \mathbf{Y}(c, w; y) &= \mathbf{P}(\mathbf{P}'(c, w;); y) \\ \mathbf{PAR}(; a) &= \mathbf{C}(; a, 0, 1) \\ \mathbf{I}(c, w; y) &= \mathbf{PAR}(; \mathbf{Y}(c, w; y)) \\ \vee(0; a) &= \mathbf{PAR}(; a) \\ \vee(xi; a) &= \mathbf{C}(; \vee(x; a), \mathbf{PAR}(; \mathbf{P}(xi; a)), 1) \end{aligned}$$

  As Bellantoni and Cook explained in [6], $\mathbf{P}(a; b)$ is a function that takes $|a|$ predecessors of b. Function $\mathbf{Y}(c, w; y)$ produces in output the in input y with $|w| - |c|$

rightmost bits deleted. In this way, the output of the function $\mathbf{Y}$, depending on input c, may vary from the entire string y down to the basic string 0. Function $\mathbf{I}(c, w; y)$ is needed in recursion in order to look into y and choose which function h_i should be applied. The last function, called $\vee$, simply implements the logical OR between the rightmost $|x|$ bits of input a.

We can now move on and finally define function f'.

$$\begin{aligned}
\hat{h}(w; i, a, \overline{b}, c) &= \mathbf{C}(; i, h'_0(w; a, \overline{b}, c), h'_1(w; a, \overline{b}, c)) \\
\hat{f}(0, w; \overline{x}, y) &= 0 \\
\hat{f}(ci, w; \overline{x}, y) &= \mathbf{C}(; \vee(w; \mathbf{Y}(c\, w; y)), g'(w; \overline{x}), \hat{h}(w; \mathbf{I}(c, w; y), \mathbf{Y}(c, w; y), \overline{x}, \hat{f}(c, w; \overline{x}, y))) \\
f'(w; \overline{x}, y) &= \hat{f}(w, w; \overline{x}, y)
\end{aligned}$$

Knowing that f is in Cobham class, it has also a monotone polynomial bound q_f and for the same reason, also functions h_i and g have monotone polynomials bound. So, let $q_h = q_{h_0} + q_{h_1}$, we define p_f as:

$$p_f(\overline{|x|}, |y|) = p_h(\overline{|x|}, |y|, q_f(\overline{|x|}, |y|)) + p_g(\overline{|x|}) + |y| + 1$$

Next step is to prove by induction that $\hat{f}(u, w; y, \overline{x}) = f(\mathbf{Y}(u, w; y), \overline{x})$, by fixing y and $\overline{x}$. Formally, the induction will be made on the size of parameter u and we will prove that for $|w| - |y| \leq |u| \leq |w|$ we get $\hat{f}(u, w; y, \overline{x}) = f(\mathbf{Y}(u, w; y), \overline{x})$.

Notice that $\mathbf{Y}(w, w; y)$ computes y and so $f'(w; \overline{x}, y)$ is by substitution equal to $f(y, \overline{x})$. Let's see how the result varies by varying $|w|$ (recall $|w| - |y| \leq |u| \leq |w|$).

The expression $|w| - |y|$ gives a value greater or equal than 1 and so it is also the value $|u|$. So, it exists a z and a $j \in \{0, 1\}$ such that $u = zj$. It does not matter which is the value j because its size does not change: $\mathbf{Y}(z1, w; y) = \mathbf{Y}(zj, w; y)$.

Recall that $|w| \geq |\mathbf{Y}(z1, w; y)|$ and so, the function $\vee(w; \mathbf{Y}(z1, w; y))$ gives 0 if $\mathbf{Y}(z1, w; y)$ computes to 0, otherwise it gives 1.

In the particularly case where $|u| = |zj| = |w| - |y|$ we have that function $\mathbf{Y}(z1, w; y)$ computes to 0 and so, by definition of $\hat{f}$ we get $\hat{f}(zj, w; \overline{y}) = g'(w; \overline{x})$; by induction we get $g(\overline{x})$ that is $f(0, \overline{x}) = f(\mathbf{Y}(z1, w; y), \overline{x})$.

So, in the last case, we have $|zj| > |w| - |y|$ and we can assume $\hat{f}(z, w; \overline{x}, y) = f(\mathbf{Y}(z, w; y)\overline{x})$. Now we use monotonicity of polynomials p_f and p_{h_i} to get a prerequisite for applying induction hypothesis.

$$\begin{aligned} |w| &\geq p_f(\overline{|x|}, |y|) \\ &\geq p_{h_i}(|\mathbf{Y}(z, w; y)|, \overline{|x|}, q_f(|\mathbf{Y}(z, w; y)|, \overline{|x|})) \\ &\geq p_{h_i}(|\mathbf{Y}(z, w; y)|, \overline{|x|}, |f(\mathbf{Y}(z, w; y), \overline{x})|) \end{aligned}$$

We are therefore allowed to apply induction hypothesis on h_i. So, we get:

$$\begin{aligned} h_i'(w; \mathbf{Y}(z, w; y), \overline{x}, \hat{f}(z, w; \overline{x}, y)) &= h_i'(w; \mathbf{Y}(z, w; y), \overline{x}, \hat{f}(\mathbf{Y}(z, w; y)\overline{x})) \\ &= h_i(\mathbf{Y}(z, w; y), \overline{x}, \hat{f}(\mathbf{Y}(z, w; y)\overline{x})) \end{aligned}$$

We are going to use this equality for the last equality. First recall that the condition $|w| - |y| < |zj| \leq |w|$ implies that y is not zero and, moreover, also $\mathbf{Y}(z1, w; y)$ is not 0. Knowing that $\mathbf{Y}(z1, w; y)$ is $\mathbf{S}_{\mathbf{I}(z,w;y)}\mathbf{Y}(z, w; y)$ we can conclude and get:

$$\begin{aligned} \hat{f}(zj, w; \overline{x}, y) &= h'_{\mathbf{I}(z,w;y)}(w; \mathbf{Y}(z, w; y), \overline{x}, \hat{f}(z, w; \overline{x}, y)) \\ &= h'_{\mathbf{I}(z,w;y)}(\mathbf{Y}(z, w; y), \overline{x}, \hat{f}(\mathbf{Y}(z, w; y)\overline{x})) \qquad = f(\mathbf{Y}(zj, w; y), \overline{x}) \end{aligned}$$

that is exactly the sub-thesis we were proving.

- If f is the smash function $x\#y$, then we can redefine it using recursion on notation. Define g as

$$g(0, y) = y$$

$$g(xi) = g(x, y)0 \quad \text{where } xi \text{ is not } 0$$

(note that $g(x, y)0$ means the result of $g(x, y)$ concatenated to 0) and f as

$$f(0, y) = 1$$

$$f(xi, y) = g(y, f(x, y)) \quad \text{where } xi \text{ is not } 0$$

we can easily check that bounding polynomials are $p_f(|x|, |y|) = |x| \cdot |y| + 1$ and $p_g(|x|, |y|) = |x| + |y|$. We have brought it back to a case already analysed. So, we can use the technique shown in presence of recursion on notation and we are done.

This concludes the proof. □

We have proved that for any function f in Cobham's class we are able to build a function f' in class B, computing the same result, with an extra argument w satisfying a particular inequality. In the following theorem we will get rid of this extra argument.

Theorem 3.1 *Let $f(\overline{x})$ be a polytime function, then $f(\overline{x};)$ is in B.*

Proof: The proof is quite easy to understand, even if is a bit technical. By theorem 3.1 we can get polynomial p_f and function f'. We construct a bound-function $q(\overline{x};)$ in B such that it satisfies all the hypothesis of the thesis of lemma 3.1, namely $|q(\overline{x};)| \geq p_f(\overline{|x|})$ in order to get $f(\overline{x};) = f'(q(\overline{x};); \overline{x})$.

In order to get this bound-function we define a function that concatenates strings. We define three functions: $\oplus^2$, $\oplus^k$, $\#$ in such way:

$$\begin{aligned}
\oplus^2(0; y) &= y \\
\oplus^2(xi; y) &= \mathbf{S}_i(; \oplus^2(x; y)), \text{ where } xi \neq 0 \\
\oplus^k(x_1, \ldots, x_{k-1}; x_k) &= \oplus^2(x_1; \oplus^{k-1}(x_2, \ldots, x_{k-1}; x_k)) \\
\#(0;) &= 0 \\
\#(xi;) &= \oplus^2(xi; \#(x;)), \text{ where } xi \neq 0
\end{aligned}$$

It can be checked that the size of $\#x$ is $|x|(|x| + 1)/2$. So, let n, m three constants such that $(\sum_j |x_j|)^n + m \geq p_f(|\overline{x}|)$, for any x. If we compose function $\#()$ with itself a constant number of time, we could easily get a function $q(x;)$ such that $|x;| \geq (|x|)^n + n$. Finally we create function $g(\overline{x};)$ defined as $q(\oplus^k(x_1, \ldots, x_{k-1}; x_k);)$ that is greater than $p_f(\overline{x})$. So, we have found a desired value greater than the polynomial p_f, as desired.

□

3.3.3 B is in PTIME

We have proved that every polytime function can be expressed in B. We are now going to prove that every function in B is polytime. In such way we will be able to conclude that B characterises the polytime functions. Should be an expected result; indeed all the basic functions are polytime and the recursion, the key problem, is a controlled recursion, where we are allowed to apply such recursion only on specific terms, the normal ones.

Also this theorem is subdivided in two lemmas: one controlling the size of the output and one controlling the number of required steps.

Lemma 3.2 *Let f be a function in B. There is a monotone polynomial p_f such that $|f(\overline{x};\overline{y})| \leq p_f(\overline{|x|}) + \max_i(|y_i|)$, for all $\overline{x},\overline{y}$.*

Proof: By induction on the derivation of function f in class B.

- If f is a constant, projection, successor, predecessor or conditional function, then it is easy to check that there is a polynomial bound ad is $1 + \sum_i |x_i|$.
- If f is defined as a Predicative Recursion on Notation we are in the following case:

$$f(0,\overline{x};\overline{y}) = g(\overline{x};\overline{y})$$
$$f(zi,\overline{x};\overline{y}) = h_i(z,\overline{x};\overline{y},f(z,\overline{x};\overline{y})) \text{ for } zi \neq 0$$

  By applying induction hypothesis on g, h_0, h_1 we get

$$|f(0,\overline{x};\overline{y})| = p_g(|\overline{x}|) + \max_i(|y_i|)$$
$$|f(zi,\overline{x};\overline{y})| = |p_h(|z|,|\overline{x}|)| + \max(\max_i(|y_i|),|f(z,\overline{x};\overline{y})|)$$

  Where p_h is defined as the sum of p_{h_0} and p_{h_1}.

  First case is trivial, we focus on the latter one. We define our polynomial bound as

$$p_f(|z|,\overline{|x|}) = |z| \cdot p_h(|z|,|\overline{x}|) + p_g(|\overline{x}|)$$

  and we assume $|f(z,\overline{x};\overline{y})| \leq p_f(|z|,|\overline{x}|) + \max_i(|y_i|)$. We are going to use this inequality in the second step of the following calculus.

$$\begin{aligned}
|f(zi,\overline{x};\overline{y})| &= |p_h(|z|,|\overline{x}|)| + \max(\max_j(|y_j|), |f(z,\overline{x};\overline{y})|) \\
&\leq |p_h(|z|,|\overline{x}|)| + \max(\max_j(|y_j|), p_f(|z|,|\overline{x}|) + \max_j(|y_j|)) \\
&\leq |p_h(|z|,|\overline{x}|)| + p_f(|z|,|\overline{x}|) + \max_j(|y_j|) \\
&\leq |p_h(|z|,|\overline{x}|)| + |z| \cdot p_h(|z|,|\overline{x}|) + p_g(|\overline{x}|) + \max_j(|y_j|) \\
&\leq |zi| \cdot p_h(|z|,|\overline{x}|) + p_g(|\overline{x}|) + \max_j(|y_j|) \\
&\leq p_f(|zi|,|\overline{x}|) + \max_j(|y_j|)
\end{aligned}$$

as desired.

- If f is defined as a Safe Composition, then we have $f(\overline{x};\overline{y}) = g(\overline{h}(\overline{x};),\overline{r}(\overline{x};\overline{y}))$ and so we get, by induction hypothesis applied to g, h, r:

$$\begin{aligned}
|f(\overline{x};\overline{y})| &= |g(\overline{h}(\overline{x};),\overline{r}(\overline{x};\overline{y}))| \\
&\leq p_g(\overline{|h(\overline{x};)|}) + \max_i(|r_i(\overline{x};\overline{y})|) \\
&\leq p_g(\overline{p_h}(\overline{x};)) + \max_i(|r_i(\overline{x};\overline{y})|) \\
&\leq p_g(\overline{p_h}(\overline{x};)) + \max_i(p_{r_i}(|\overline{x}|) + \max_j(|\overline{y}|)) \\
&\leq p_g(\overline{p_h}(\overline{x};)) + \sum_i(p_{r_i}(|\overline{x}|)) + \max_j(|\overline{y}|)
\end{aligned}$$

So, $f(\overline{x};\overline{y})$ is bounded by a polynomial where the safe argument appears alone only inside a max operator, and normal argument appears in a sub-polynomial, as desired.

This concludes the proof □

Finally we get the last theorem, proving that every function in B is computable in polytime.

Theorem 3.2 *Let $f(\overline{x};\overline{y})$ be a function in B. Then $f(\overline{x};\overline{y})$ is polytime.*

Proof: By induction on the structure of function f.

- If f is constant, projection, successor, predecessor, conditional function, then it's trivial to check that these function can be computed in polytime.

- If f is defined as safe composition, it's easy to check because by using induction we can observe that composition of polynomial time functions is still a polynomial time function.

- If f is defined by Predicative Recursion on Notation, then notice that it could be executed in polynomial time if the result is polynomially bounded and the number of steps and base function are polytime. In our case, lemma 3.2 makes us sure to get polynomial bounds. So, Predicative Recursion on Notation can be evaluated in polytime.

This concludes the proof. □

3.4 Safe linear recursion

Instead of introducing Safe Linear Recursion (in the following SLR) of Martin Hofmann [21], we are going to present a slight different variation of it, where the proofs of soundness and completeness are made with syntactical means.

In this new version of SLR some restrictions have to be made to SLR if one wants to be able to prove polynomial time soundness easily and operationally. And what one obtains at the end is indeed quite similar to (a variation of) Bellantoni, Niggl and Schwichtenberg calculus RA [7, 36]. Actually, the main difference between this version and SLR deals with linearity: keeping the size of reducts under control during normalisation is very difficult in presence of higher-order duplication. For this reason, the two function spaces $A \to B$ and $A \multimap B$ of original SLR collapse to just one here, and arguments of a higher-order type can *never* be duplicated. This constraint allows us to avoid an exponential blowup in the size of terms and results in a reasonably simple system for which polytime soundness can be proved explicitly, by studying the combinatorics of reduction. Another consequence of the just described modification is subject reduction, which can be easily proved in our system, contrarily to what happens in original SLR [21].

3.4.1 The Syntax and Basic Properties of SLR

SLR is a fairly standard Curry-style lambda calculus with constants for the natural numbers, branching and recursion. Its type system, on the other hand, is based on ideas

coming from linear logic (some variables can appear at most once in terms) and on a distinction between modal and non modal variables.

Let us introduce the category of types first:

Definition 3.1 (Types) *The* types *of* SLR *are generated by the following grammar:*

$$A ::= \mathbf{N} \mid \Box A \to A \mid \blacksquare A \to A.$$

Types different from $\mathbf{N}$ *are denoted with metavariables like* H *or* G. $\mathbf{N}$ *is the only* base type.

There are two function spaces in SLR. Terms which can be typed with $\blacksquare A \to B$ are such that the result (of type B) can be computed in constant time, independently on the size of the argument (of type A). On the other hand, computing the result of functions in $\Box A \to B$ requires polynomial time in the size of their argument.

A notion of subtyping is used in SLR to capture the intuition above by stipulating that the type $\blacksquare A \to B$ is a subtype of $\Box A \to B$. Subtyping is best formulated by introducing aspects:

Definition 3.2 (Aspects) *An* aspect *is either* $\Box$ *or* $\blacksquare$*: the first is the* modal *aspect, while the second is the* non modal *one. Aspects are partially ordered by the binary relation* $\{(\Box, \Box), (\Box, \blacksquare), (\blacksquare, \blacksquare)\}$*, noted* $<:$.

Subtyping rules are in Figure 3.1.

$$\frac{}{A <: A}\ (\text{S-Refl}) \qquad \frac{A <: B \qquad B <: C}{A <: C}\ (\text{S-Trans})$$

$$\frac{B <: A \qquad C <: D \qquad b <: a}{aA \to C <: bB \to D}\ (\text{S-Sub})$$

Figure 3.1: Subtyping rules.

SLR's terms are those of an applied lambda calculus with primitive recursion and branching, in the style of Gödel's T:

Definition 3.3 (Terms) *Terms and constants are defined as follows:*

$$t ::= x \mid c \mid ts \mid \lambda x : aA.t \mid \texttt{case}_A\ t\ \texttt{zero}\ s\ \texttt{even}\ r\ \texttt{odd}\ q \mid \texttt{recursion}_A\ t\, s\, r;$$
$$c ::= n \mid \mathtt{S}_0 \mid \mathtt{S}_1 \mid \mathtt{P}.$$

Here, x ranges over a denumerable set of variables and n ranges over the natural numbers seen as constants of base type. Every constant c has its naturally defined type, that we indicate with type(c). As an example, type$(n) = \mathbf{N}$ for every n, while type$(\mathtt{S}_0) = \blacksquare\mathbf{N} \to \mathbf{N}$. The size $|t|$ of any term t can be easily defined by induction on t:

$$\begin{aligned}
|x| &= 1;\\
|ts| &= |t| + |s|;\\
|\lambda x : aA.t| &= |t| + 1;\\
|\texttt{case}_A\ t\ \texttt{zero}\ s\ \texttt{even}\ r\ \texttt{odd}\ q| &= |t| + |s| + |r| + |q| + 1;\\
|\texttt{recursion}_A\ t\, s\, r| &= |t| + |s| + |r| + 1;\\
|n| &= \lceil \log_2(n) \rceil;\\
|\mathtt{S}_0| = |\mathtt{S}_1| &= |\mathtt{P}| = 1.
\end{aligned}$$

A term is said to be explicit *if it does not contain any instance of* `recursion`*. As usual, terms are considered modulo α-conversion. Free (occurrences of) variables and capture-avoiding substitution can be defined in a standard way.*

Arguments are passed to functions following a mixed scheme in SLR: arguments of base type are evaluated before being passed to functions, while arguments of a higher-order type are passed to functions possibly unevaluated, in a call-by-name fashion. Let's first of all define the one-step reduction relation:

Definition 3.4 (Reduction) *The* one-step reduction relation $\to$ *is a binary relation between terms and terms. It is defined by the axioms in Figure 3.2 and can be applied everywhere, except as the second or third argument of a recursion. A term t is in* normal form *if t cannot appear as the left-hand side of a pair in $\to$. NF is the set of terms in normal form.*

In this case, a multistep reduction relation is be defined by simply taking the transitive and reflective closure of $\to$, since a term can reduce in multiple steps into just one term.

$$
\begin{aligned}
\texttt{case}_A\ 0\ \texttt{zero}\ t\ \texttt{even}\ s\ \texttt{odd}\ r &\to t;\\
\texttt{case}_A\ (\mathtt{S}_0 n)\ \texttt{zero}\ t\ \texttt{even}\ s\ \texttt{odd}\ r &\to s;\\
\texttt{case}_A\ (\mathtt{S}_1 n)\ \texttt{zero}\ t\ \texttt{even}\ s\ \texttt{odd}\ r &\to r;\\
\texttt{recursion}_A\ 0\, g\, f &\to g;\\
\texttt{recursion}_A\ n\, g\, f &\to f n(\texttt{recursion}_\tau\ \lfloor\frac{n}{2}\rfloor\, g\, f);\\
\mathtt{S}_0 n &\to 2\cdot n;\\
\mathtt{S}_1 n &\to 2\cdot n+1;\\
\mathtt{P}0 &\to 0;\\
\mathtt{P}n &\to \lfloor\frac{n}{2}\rfloor;\\
(\lambda x : a\mathbf{N}.t)n &\to t[x/n];\\
(\lambda x : aH.t)s &\to t[x/s];\\
(\lambda x : aA.t)sr &\to (\lambda x : aA.tr)s;
\end{aligned}
$$

Figure 3.2: One-step reduction rules.

Definition 3.5 (Contexts) *A* context Γ *is a finite set of assignments of types and aspects to variables, in the form* $x : aA$. *As usual, we require contexts not to contain assignments of distinct types and aspects to the same variable. The union of two disjoint contexts* Γ *and* Δ *is denoted as* Γ, Δ. *In doing so, we implicitly assume that the variables in* Γ *and* Δ *are pairwise distinct. The union* Γ, Δ *is sometimes denoted as* $\Gamma; \Delta$. *This way we want to stress that all types appearing in* Γ *are base types. With the expression* $\Gamma <: a$ *we mean that any aspect* b *appearing in* Γ *is such that* $b <: a$.

Typing rules are in Figure 3.3. Observe how rules with more than one premise are designed

$$\frac{x : aA \in \Gamma}{\Gamma \vdash x : A}\ (\text{T-Var-Aff}) \qquad \frac{\Gamma \vdash t : A \qquad A <: B}{\Gamma \vdash t : B}\ (\text{T-Sub})$$

$$\frac{\Gamma, x : aA \vdash t : B}{\Gamma \vdash \lambda x : aA.t : aA \to B}\ (\text{T-Arr-I}) \qquad \frac{}{\Gamma \vdash c : \mathit{type}(c)}\ (\text{T-Const-Aff})$$

$$\frac{\begin{array}{cc}\Gamma; \Delta_1 \vdash t : \mathbf{N} & \Gamma; \Delta_3 \vdash r : A \\ \Gamma; \Delta_2 \vdash s : A & \Gamma; \Delta_4 \vdash q : A \qquad A \text{ is } \Box\text{-free}\end{array}}{\Gamma; \Delta_1, \Delta_2, \Delta_3, \Delta_4 \vdash \mathtt{case}_A\ t\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q : A}\ (\text{T-Case})$$

$$\frac{\begin{array}{ccc}\Gamma_1; \Delta_1 \vdash t : \mathbf{N} & & \\ \Gamma_1, \Gamma_2; \Delta_2 \vdash s : A & & \Gamma_1; \Delta_1 <: \Box \\ \Gamma_1, \Gamma_2; \vdash r : \Box\mathbf{N} \to \blacksquare A \to A & & A \text{ is } \Box\text{-free}\end{array}}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash \mathtt{recursion}_A\ t\, s\, r : A}\ (\text{T-Rec})$$

$$\frac{\Gamma; \Delta_1 \vdash t : aA \to B \qquad \Gamma; \Delta_2 \vdash s : A \qquad \Gamma, \Delta_2 <: a}{\Gamma; \Delta_1, \Delta_2 \vdash (ts) : B}\ (\text{T-Arr-E})$$

Figure 3.3: Type rules

in such a way as to guarantee that whenever $\Gamma \vdash t : A$ can be derived and $x : aH$ is in Γ, then x can appear free at most once in t. If $y : a\mathbf{N}$ is in Γ, on the other hand, then y can appear free in t an arbitrary number of times.

Definition 3.6 *A* first-order term *of arity* k *is a closed, well typed term of type* $a_1\mathbf{N} \to a_2\mathbf{N} \to \ldots a_k\mathbf{N} \to \mathbf{N}$ *for some* $a_1, \ldots, a_k$.

Example 3.1 Let's see some examples. Two terms that we are able to type in our system and one that is not possible to type.

As we will see in Chapter 3.4.5 we are able to type addition and multiplication.

Sometimes, however, it is more convenient to work in unary notation. Given a natural number i, its *unary encoding* is simply the numeral that, written in binary notation, is 1^i. Given a natural number i we will refer to its encoding $\underline{i}$. The type in which unary encoded natural numbers will be written, is just $\mathbf{N}$, but for reason of clarity we will use the symbol $\mathbf{U}$ instead.

Addition gives in output a number (recall that we are in unary notation) such that the resulting length is the sum of the input lengths.

$$\begin{aligned}\mathsf{add} \equiv & \lambda x : \Box\mathbf{N}.\lambda y : \blacksquare\mathbf{N}.\\ & \mathtt{recursion}_{\mathbf{N}}\ x\, y\, (\lambda x : \Box\mathbf{N}.\lambda y : \blacksquare\mathbf{N}.\mathtt{S}_1 y) : \Box\mathbf{N} \to \blacksquare\mathbf{N} \to \mathbf{N}\end{aligned}$$

We are also able to define multiplication. The operator is, as usual, defined by applying a sequence of additions.

$$\begin{aligned}\mathsf{mult} \equiv & \lambda x : \Box\mathbf{N}.\lambda y : \Box\mathbf{N}.\\ & \mathtt{recursion}_{\mathbf{N}}\ (\mathtt{P}x)\, y\, (\lambda x : \Box\mathbf{N}.\lambda z : \blacksquare\mathbf{N}.\mathsf{add} yz) : \Box\mathbf{N} \to \Box\mathbf{N} \to \mathbf{N}\end{aligned}$$

Now that we have multiplication, why not insert it in a recursion and get an exponential? As it will be clear from the next example, the restriction on the aspect of the iterated function save us from having an exponential growth. Are we able to type the following term?

$$\lambda h : \Box\mathbf{N}.\mathtt{recursion}_{\mathbf{N}}\ h\, (11)\, (\lambda x : \Box\mathbf{N}.\lambda y : \blacksquare\mathbf{N}.\mathsf{mult}(y, y))$$

The answer is negative: the operator mult requires input of aspect $\Box$, while the iterator function need to have type $\Box\mathbf{N} \to \blacksquare\mathbf{N} \to \mathbf{N}$.

□

3.4.2 Subject Reduction

The first property we are going to prove about SLR is preservation of types under reduction, the so-called Subject Reduction Theorem. The proof of it is going to be very standard and, as usual, amounts to proving substitution lemmas. Preliminary to that is a technical lemma saying that weakening is derivable (since the type system is affine):

Lemma 3.3 (Weakening Lemma) *If $\Gamma \vdash t : A$, then $\Gamma, x : bB \vdash t : A$ whenever x does not appear in Γ.*

Proof: By induction on the structure of the typing derivation for t.

- If last rule was (T-Var-Aff) or (T-Const-Aff), we are allowed to add whatever we want in the context. This case is trivial.
- If last rule was (T-Sub) or (T-Arr-I), the thesis is proved by using induction hypothesis on the premise.
- Suppose that the last rule was:

$$\frac{\begin{array}{ccc}\Gamma;\Delta_1 \vdash u : N & \Gamma;\Delta_3 \vdash r : A & \\ \Gamma;\Delta_2 \vdash s : A & \Gamma;\Delta_4 \vdash q : A & A \text{ is } \Box\text{-free}\end{array}}{\Gamma;\Delta_1,\Delta_2,\Delta_3,\Delta_4 \vdash \mathtt{case}_A\ u\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q : A}\ (\text{T-Case})$$

  If $B \equiv \mathbf{N}$ we can easily do it by applying induction hypothesis on every premises and add x to Γ. Otherwise, we can do it by applying induction hypothesis on just one premise and the thesis is proved.

- Suppose that the last rule was:

$$\frac{\begin{array}{cc}\Gamma_1;\Delta_1 \vdash q : \mathbf{N} & \\ \Gamma_1,\Gamma_2;\Delta_2 \vdash s : A & \Gamma_1;\Delta_1 <: \Box \\ \Gamma_1,\Gamma_2;\vdash r : \Box\mathbf{N} \to \blacksquare A \to A & A \text{ is } \Box\text{-free}\end{array}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2 \vdash \mathtt{recursion}_A\ q\, s\, r : A}\ (\text{T-Rec})$$

  Suppose that $B \equiv \mathbf{N}$, we have the following cases:

  - If $b \equiv \Box$, we can do it by applying induction hypothesis on all the premises and add x in Γ_1.
  - If $b \equiv \blacksquare$ we apply induction hypothesis on $\Gamma_1,\Gamma_2;\Delta_2 \vdash s : A$ and on $\Gamma_1,\Gamma_2;\vdash r : \Box\mathbf{N} \to \blacksquare A \to A$.

  Otherwise we apply induction hypothesis on $\Gamma_1;\Delta_1 \vdash q : \mathbf{N}$ or on $\Gamma_1,\Gamma_2;\Delta_2 \vdash s : A$ and we are done.

- Suppose that the last rule was:

$$\frac{\Gamma;\Delta_1 \vdash r : aA \to B \qquad \Gamma;\Delta_2 \vdash s : A \qquad \Gamma,\Delta_2 <: a}{\Gamma;\Delta_1,\Delta_2 \vdash (rs) : B}\ (\text{T-Arr-E})$$

  If $B \equiv \mathbf{N}$ we have to apply induction hypothesis on all the premises. Otherwise we apply induction hypothesis on just one premise and the thesis is proved.

This concludes the proof. □

Two substitution lemmas are needed in SLR. The first one applies when the variable to be substituted has a non-modal type:

Lemma 3.4 (■-Substitution Lemma) *Let* $\Gamma;\Delta \vdash t : A$. *Then*

1. *if* $\Gamma = x : \blacksquare\mathbf{N}, \Theta$, *then* $\Theta; \Delta \vdash t[x/n] : A$ *for every* n;
2. *if* $\Delta = x : \blacksquare H, \Theta$ *and* $\Gamma; \Xi \vdash s : H$, *then* $\Gamma; \Theta, \Xi \vdash t[x/s] : A$.

Proof: By induction on a type derivation of t.

- If the last rule is (T-Var-Aff) or (T-Arr-I) or (T-Sub) or (T-Const-Aff) the proof is trivial.
- If the last rule is (T-Case). By applying induction hypothesis on the interested term we can easily derive the thesis.
- If the last rule is (T-Rec), our derivation will have the following appearance:

$$\frac{\begin{array}{cc} \Gamma_2; \Delta_4 \vdash q : \mathbf{N} & \\ \Gamma_2, \Gamma_3; \Delta_5 \vdash s : B & \Gamma_2; \Delta_4 <: \Box \\ \Gamma_2, \Gamma_3; \vdash r : \Box\mathbf{N} \to \blacksquare B \to B & B \text{ is } \Box\text{-free} \end{array}}{\Gamma_2, \Gamma_3; \Delta_4, \Delta_5 \vdash \mathtt{recursion}_B\ q\, s\, r : B}\ (\text{T-Rec})$$

  By definition, $x : \blacksquare A$ cannot appear in $\Gamma_2; \Delta_4$. If it appears in Δ_5 we can simply apply induction hypothesis and prove the thesis. We will focus on the most interesting case: it appears in Γ_3 and so $A \equiv \mathbf{N}$. In that case, by the induction hypothesis applied to (type derivations for) s and r, we obtain that:

$$\begin{array}{c} \Gamma_2, \Gamma_4; \Delta_5 \vdash s[x/n] : B \\ \Gamma_2, \Gamma_4; \vdash r[x/n] : \Box\mathbf{N} \to \blacksquare B \to B \end{array}$$

  where $\Gamma_3 \equiv \Gamma_4, x : \blacksquare\mathbf{N}$.
- If the last rule is (T-Arr-E),

$$\frac{\Gamma; \Delta_4 \vdash t : aC \to B \qquad \Gamma; \Delta_5 \vdash s : C \qquad \Gamma, \Delta_5 <: a}{\Gamma, \Delta_4, \Delta_5 \vdash (ts) : B}\ (\text{T-Arr-E})$$

  If $x : A$ is in Γ then we apply induction hypothesis on both branches, otherwise it is either in Δ_4 or in Δ_5 and we apply induction hypothesis on the corresponding branch. We arrive to the thesis by applying (T-Arr-E) at the end.

This concludes the proof. □

Notice how two distinct substitution statements are needed, depending on the type of the substituted variable being a base or a higher-order type. Substituting a variable of a modal type requires an additional hypothesis on the term being substituted:

Lemma 3.5 (□-Substitution Lemma) *Let* $\Gamma; \Delta \vdash t : A$. *Then*

1. *if* $\Gamma = x : \Box\mathbf{N}, \Theta$, *then* $\Theta; \Delta \vdash t[x/n] : A$ *for every* n;
2. *if* $\Delta = x : \Box H, \Theta$ *and* $\Gamma; \Xi \vdash s : H$ *where* $\Gamma, \Xi <: \Box$, *then* $\Gamma; \Theta, \Xi \vdash t[x/s] : A$.

Proof: By induction on the derivation.

- If last rule is (T-Var-Aff) or (T-Arr-I) or (T-Sub) or (T-Const-Aff) the proof is trivial.
- If last rule is (T-Case). By applying induction hypothesis on the interested term we can easily derive the thesis.
- If last rule is (T-Rec), our derivation will have the following appearance:

$$\frac{\begin{array}{cc}\Gamma_2; \Delta_4 \vdash q : \mathbf{N} & \\ \Gamma_2, \Gamma_3; \Delta_5 \vdash s : B & \Gamma_2; \Delta_4 <: \Box \\ \Gamma_2, \Gamma_3; \vdash r : \Box\mathbf{N} \to \blacksquare B \to B & B \text{ is } \Box\text{-free}\end{array}}{\Gamma_2, \Gamma_3; \Delta_4, \Delta_5 \vdash \mathtt{recursion}_B\ q\, s\, r : B}\ (\text{T-Rec})$$

  By definition $x : \Box A$ can appear in $\Gamma_1; \Delta_4$. If so, by applying induction hypothesis we can derive easily the proof. In the other cases, we can proceed as in Lemma 3.4. We will focus on the most interesting case, where $x : \Box A$ appears in Γ_2 and so $A \equiv \mathbf{N}$. In that case, by the induction hypothesis applied to (type derivations for) s and r, we obtain that:

$$\begin{array}{c}\Gamma_4, \Gamma_3; \Delta_5 \vdash s[x/n] : B \\ \Gamma_4, \Gamma_3; \vdash r[x/n] : \Box\mathbf{N} \to \blacksquare B \to B\end{array}$$

  where $\Gamma_2 \equiv \Gamma_4, x : \Box\mathbf{N}$.
- If last rule is (T-Arr-E),

$$\frac{\Gamma; \Delta_4 \vdash t : aC \to B \qquad \Gamma; \Delta_5 \vdash s : C \qquad \Gamma, \Delta_5 <: a}{\Gamma, \Delta_4, \Delta_5 \vdash (ts) : B}\ (\text{T-Arr-E})$$

  If $x : A$ is in Γ then we apply induction hypothesis on both branches, otherwise it is either in Δ_4 or in Δ_5 and we apply induction hypothesis on the relative branch. We prove our thesis by applying (T-Arr-E) at the end.

This concludes the proof. □

Substitution lemmas are necessary ingredients when proving subject reduction. In particular, they allow to prove that types are preserved along beta reduction steps, the other reduction steps being very easy. We get:

Theorem 3.3 (Subject Reduction) *Suppose that* $\Gamma \vdash t : A$. *If* $t \to t_1$, *then it holds that* $\Gamma \vdash t_1 : A$.

Proof: By induction on the derivation for term t. We will check the last rule.

- If last rule is (T-Var-Aff) or (T-Const-Aff). The thesis is trivial.
- If last rule is (T-Sub). The thesis is trivial.
- If last rule is (T-Arr-I). The term cannot reduce since it is a value.
- If last rule is (T-Case).

$$\frac{\begin{array}{ccc}\Gamma;\Delta_1 \vdash s : N & & \Gamma;\Delta_3 \vdash q : A \\ \Gamma;\Delta_2 \vdash r : A & \Gamma;\Delta_4 \vdash u : A & A \text{ is } \Box\text{-free}\end{array}}{\Gamma;\Delta_1,\Delta_2,\Delta_3,\Delta_4 \vdash \mathtt{case}_A\ s\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u : A}\ (\text{T-Case})$$

  Our final term could reduce in two ways. Either we do β-reduction on s, r, q or u, or we choose one of branches in the case. In all the cases, the proof is trivial.

- If last rule is (T-Rec).

$$\frac{\begin{array}{ccc}\rho : \Gamma_1;\Delta_1 \vdash s : \mathbf{N} & & \\ \mu : \Gamma_1,\Gamma_2;\Delta_2 \vdash r : A & & \Gamma_1;\Delta_1 <: \Box \\ \nu : \Gamma_1,\Gamma_2;\vdash q : \Box\mathbf{N} \to \blacksquare A \to A & & A \text{ is } \Box\text{-free}\end{array}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2 \vdash \mathtt{recursion}_A\ s\, r\, q : A}\ (\text{T-Rec})$$

  Our term could reduce in three ways. We could evaluate s (trivial), we could be in the case where $s \equiv 0$ (trivial) and the other case is where we unroll the recursion (so, where s is a value $n \geq 1$). We are going to focus on this last option. The term rewrites to $qn(\mathtt{recursion}_\tau\ \lfloor \frac{n}{2} \rfloor\, r\, q)$. We could set up the following derivation.

$$\pi \equiv \frac{\dfrac{}{\Gamma_1;\Delta_1 \vdash \lfloor \frac{n}{2} \rfloor : \mathbf{N}}\ (\text{T-Const-Aff}) \qquad \mu : \Gamma_1,\Gamma_2;\Delta_2 \vdash r : A}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2 \vdash \mathtt{recursion}_\tau\ \lfloor \frac{n}{2} \rfloor\, r\, q : A}\ (\text{T-Rec})$$

$$\sigma \equiv \frac{\nu : \emptyset;\Gamma_1,\Gamma_2 \vdash q : \Box\mathbf{N} \to \blacksquare A \to A \qquad \dfrac{}{\emptyset;\emptyset \vdash n : \mathbf{N}}\ (\text{T-Const-Aff})}{\emptyset;\Gamma_1,\Gamma_2 \vdash qn : \blacksquare A \to A}\ (\text{T-ARR-E})$$

  By gluing the two derivation with the rule (T-Arr-E) we obtain:

$$\frac{\begin{array}{c}\sigma : \Gamma_1,\Gamma_2;\vdash qn : \blacksquare A \to A \\ \pi : \Gamma_1,\Gamma_2;\Delta_1,\Delta_2 \vdash \mathtt{recursion}_\tau\ \lfloor \frac{n}{2} \rfloor\, r\, q : A\end{array}}{\Gamma_1,\Gamma_2,\Gamma_3;\Delta_1,\Delta_2 \vdash qn(\mathtt{recursion}_\tau\ \lfloor \frac{n}{2} \rfloor\, r\, q) : A}\ (\text{T-Arr-E})$$

  Notice that in the derivation ν we put Γ_1, Γ_2 on the left side of ";" and also on the right side. Recall the definition 3.5, about ";". We would stress out that all the variable on the left side have base type, as Γ_1, Γ_2 have. The two contexts could also be *"shifted"* on the right side because no constrains has been set on the variables on the right side.

- If last rule was (T-Sub) we have the following derivation:

$$\frac{\Gamma \vdash s : A \qquad A <: B}{\Gamma \vdash s : B}\ (\text{T-Sub})$$

  If s reduces to r we can apply induction hypothesis on the premises and having the following derivation:

$$\frac{\Gamma \vdash r : A \qquad A <: B}{\Gamma \vdash r : B}\ (\text{T-Sub})$$

- If last rule was (T-Arr-E), we could have different cases.
  - Cases where on the left part of our application we have $\mathtt{S}_i$, $\mathtt{P}$ is trivial.
  - Let's focus on the case where on the left part we find a λ-abstraction. We will consider the case only where we apply the substitution. The other case are trivial. We could have two possibilities:
    - First of all, we can be in the following situation:

$$\frac{\Gamma;\Delta_1 \vdash \lambda x : \blacksquare A.r : aC \to B \qquad \Gamma;\Delta_2 \vdash s : C \qquad \Gamma,\Delta_2 <: a}{\Gamma,\Delta_1,\Delta_2 \vdash (\lambda x : \blacksquare A.r)s : B}\ (\text{T-Arr-E})$$

      where $C <: A$ and $a <: \blacksquare$. We have that $(\lambda x : \blacksquare A.r)s$ rewrites to $r[x/s]$. By looking at rules in Figure 3.3 we can deduce that $\Gamma;\Delta_1 \vdash \lambda x : \blacksquare A.r : aC \to B$ derives from $\Gamma; x : \blacksquare A, \Delta_1 \vdash r : D$ (with $D <: B$). For the reason that $C <: A$ we can apply (T-Sub) rule to $\Gamma;\Delta_2 \vdash s : C$ and obtain $\Gamma;\Delta_2 \vdash s : A$ By applying Lemma 3.4, we get to

$$\Gamma,\Delta_1,\Delta_2 \vdash r[x/s] : D$$

      from which the thesis follows by applying (T-Sub).
    - But we can even be in the following situation:

$$\frac{\Gamma;\Delta_1 \vdash \lambda x : \square A.r : \square C \to B \qquad \Gamma;\Delta_2 \vdash s : C \qquad \Gamma,\Delta_2 <: \square}{\Gamma,\Delta_1,\Delta_2 \vdash (\lambda x : \square A.r)s : B}\ (\text{T-Arr-E})$$

      where $C <: A$. We have that $(\lambda x : \square A.r)s$ rewrites in $r[x/s]$. We behave as in the previous point, by applying Lemma 3.5, and we are done.
  - Another interesting case of application is where we perform a so-called "swap". $(\lambda x : aA.q)sr$ rewrites in $(\lambda x : aA.qr)s$. From a typing derivation with conclusion $\Gamma,\Delta_1,\Delta_2,\Delta_3 \vdash (\lambda x : aA.q)sr : C$ we can easily extract derivations for the following:

$$\Gamma;\Delta_1, x : aA \vdash q : bD \to E$$

$$\Gamma;\Delta_3 \vdash r : B$$

$$\Gamma;\Delta_2 \vdash s : F$$

where $B <: D$, $E <: C$ and $A <: F$ and $\Gamma, \Delta_3 <: b$ and $\Gamma, \Delta_2 <: a$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \Delta_3 <: b \quad \Gamma; \Delta_3 \vdash r : B}{\Gamma; \Delta_1, x : aA \vdash q : bD \to E}\,(\text{T-Arr-E})}{\Gamma; \Delta_1, \Delta_3, x : aA \vdash qr : E}\,(\text{T-Arr-I})}{\Gamma; \Delta_1, \Delta_3, \vdash \lambda x : aA.qr : aA \to E}\,(\text{T-Sub}) \qquad \Gamma, \Delta_2 <: a}{\Gamma; \Delta_1, \Delta_3, \vdash \lambda x : aA.qr : aF \to C} \qquad \Gamma; \Delta_2 \vdash s : F}{\Gamma, \Delta_1, \Delta_2, \Delta_3 \vdash (\lambda x : aA.qr)s : C}\,(\text{T-Arr-E})$$

- All the other cases can be brought back to cases that we have considered.

This concludes the proof. □

Example 3.2 In the following example we consider an example similar to one by Hofmann [21]. Let f be a variable of type $\blacksquare\mathbf{N} \to \mathbf{N}$. The function $h \equiv \lambda g : \blacksquare(\blacksquare\mathbf{N} \to \mathbf{N}).\lambda x : \blacksquare\mathbf{N}.(f(gx))$ gets type $\blacksquare(\blacksquare\mathbf{N} \to \mathbf{N}) \to \blacksquare\mathbf{N} \to \mathbf{N}$. Thus the function $(\lambda r : \blacksquare(\blacksquare\mathbf{N} \to \mathbf{N}).hr)\mathtt{S}_1$ takes type $\blacksquare\mathbf{N} \to \mathbf{N}$. Let's now execute β reductions, by passing the argument $\mathtt{S}_1$ to the function h and we obtain the following term: $\lambda x : \blacksquare\mathbf{N}.(f(\mathtt{S}_1 x))$ It's easy to check that the type has not changed. □

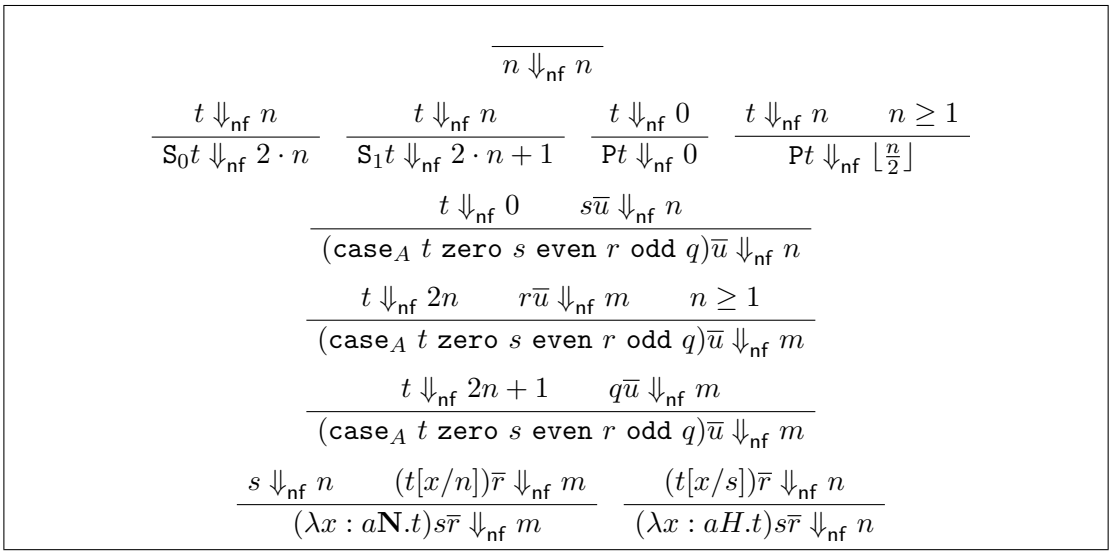
3.4.3 Polytime Soundness

The most difficult (and interesting!) result about this new version of SLR is definitely polytime soundness: every (instance of) a first-order term can be reduced to a numeral in a polynomial number of steps by a deterministic Turing machine. Polytime soundness can be proved, following [7], by showing that:

- Any explicit term of base type can be reduced to its normal form with very low time complexity;
- Any term (non necessarily of base type) can be put in explicit form in polynomial time.

By gluing these two results together, we obtain what we need, namely an effective and efficient procedure to compute the normal forms of terms. Formally, two notions of evaluation for terms correspond to the two steps defined above:

- On the one hand, we need a ternary relation $\Downarrow_{\mathsf{nf}}$ between closed terms of type $\mathbf{N}$ and numerals. Intuitively, $t \Downarrow_{\mathsf{nf}} n$ holds when t is explicit and rewrites to n. The inference rules for $\Downarrow_{\mathsf{nf}}$ are defined in Figure 3.4;
- On the other hand, we need a ternary relation $\Downarrow_{\mathsf{rf}}$ between terms of non modal type and terms. We can derive $t \Downarrow_{\mathsf{rf}} s$ only if t can be transformed into s. The inference rules for $\Downarrow_{\mathsf{rf}}$ are in Figure 3.5.



$$\frac{}{n \Downarrow_{\mathsf{nf}} n}$$

$$\frac{t \Downarrow_{\mathsf{nf}} n}{\mathtt{S}_0 t \Downarrow_{\mathsf{nf}} 2 \cdot n} \qquad \frac{t \Downarrow_{\mathsf{nf}} n}{\mathtt{S}_1 t \Downarrow_{\mathsf{nf}} 2 \cdot n + 1} \qquad \frac{t \Downarrow_{\mathsf{nf}} 0}{\mathtt{P} t \Downarrow_{\mathsf{nf}} 0} \qquad \frac{t \Downarrow_{\mathsf{nf}} n \qquad n \geq 1}{\mathtt{P} t \Downarrow_{\mathsf{nf}} \lfloor \frac{n}{2} \rfloor}$$

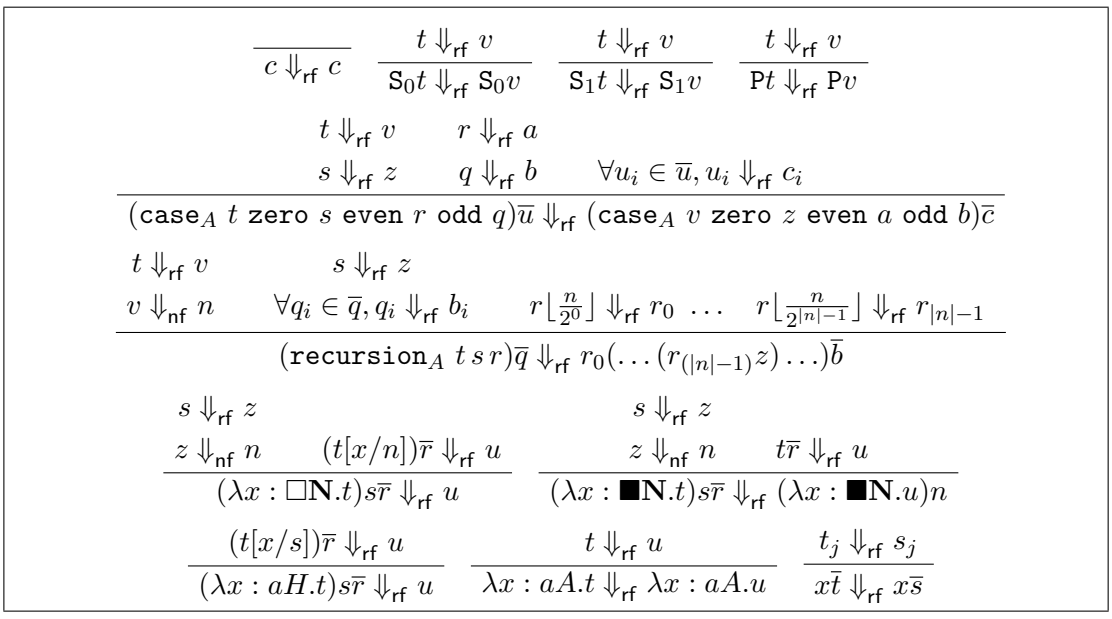
$$\frac{t \Downarrow_{\mathsf{nf}} 0 \qquad s\overline{u} \Downarrow_{\mathsf{nf}} n}{(\mathtt{case}_A\ t\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q)\overline{u} \Downarrow_{\mathsf{nf}} n}$$

$$\frac{t \Downarrow_{\mathsf{nf}} 2n \qquad r\overline{u} \Downarrow_{\mathsf{nf}} m \qquad n \geq 1}{(\mathtt{case}_A\ t\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q)\overline{u} \Downarrow_{\mathsf{nf}} m}$$

$$\frac{t \Downarrow_{\mathsf{nf}} 2n+1 \qquad q\overline{u} \Downarrow_{\mathsf{nf}} m}{(\mathtt{case}_A\ t\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q)\overline{u} \Downarrow_{\mathsf{nf}} m}$$

$$\frac{s \Downarrow_{\mathsf{nf}} n \qquad (t[x/n])\overline{r} \Downarrow_{\mathsf{nf}} m}{(\lambda x : a\mathbf{N}.t)s\overline{r} \Downarrow_{\mathsf{nf}} m} \qquad \frac{(t[x/s])\overline{r} \Downarrow_{\mathsf{nf}} n}{(\lambda x : aH.t)s\overline{r} \Downarrow_{\mathsf{nf}} n}$$

Figure 3.4: The relation $\Downarrow_{\mathsf{nf}}$: Inference Rules



$$\frac{}{c \Downarrow_{\mathsf{rf}} c} \qquad \frac{t \Downarrow_{\mathsf{rf}} v}{\mathtt{S}_0 t \Downarrow_{\mathsf{rf}} \mathtt{S}_0 v} \qquad \frac{t \Downarrow_{\mathsf{rf}} v}{\mathtt{S}_1 t \Downarrow_{\mathsf{rf}} \mathtt{S}_1 v} \qquad \frac{t \Downarrow_{\mathsf{rf}} v}{\mathtt{P} t \Downarrow_{\mathsf{rf}} \mathtt{P} v}$$

$$\frac{\begin{array}{lll} t \Downarrow_{\mathsf{rf}} v & r \Downarrow_{\mathsf{rf}} a & \\ s \Downarrow_{\mathsf{rf}} z & q \Downarrow_{\mathsf{rf}} b & \forall u_i \in \overline{u}, u_i \Downarrow_{\mathsf{rf}} c_i \end{array}}{(\mathtt{case}_A\ t\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q)\overline{u} \Downarrow_{\mathsf{rf}} (\mathtt{case}_A\ v\ \mathtt{zero}\ z\ \mathtt{even}\ a\ \mathtt{odd}\ b)\overline{c}}$$

$$\frac{\begin{array}{lll} t \Downarrow_{\mathsf{rf}} v & s \Downarrow_{\mathsf{rf}} z & \\ v \Downarrow_{\mathsf{nf}} n & \forall q_i \in \overline{q}, q_i \Downarrow_{\mathsf{rf}} b_i & r\lfloor \frac{n}{2^0} \rfloor \Downarrow_{\mathsf{rf}} r_0 \ \ldots \ \ r\lfloor \frac{n}{2^{|n|-1}} \rfloor \Downarrow_{\mathsf{rf}} r_{|n|-1} \end{array}}{(\mathtt{recursion}_A\ t\, s\, r)\overline{q} \Downarrow_{\mathsf{rf}} r_0(\ldots(r_{(|n|-1)}z)\ldots)\overline{b}}$$

$$\frac{\begin{array}{ll} s \Downarrow_{\mathsf{rf}} z & \\ z \Downarrow_{\mathsf{nf}} n & (t[x/n])\overline{r} \Downarrow_{\mathsf{rf}} u \end{array}}{(\lambda x : \square\mathbf{N}.t)s\overline{r} \Downarrow_{\mathsf{rf}} u} \qquad \frac{\begin{array}{ll} s \Downarrow_{\mathsf{rf}} z & \\ z \Downarrow_{\mathsf{nf}} n & t\overline{r} \Downarrow_{\mathsf{rf}} u \end{array}}{(\lambda x : \blacksquare\mathbf{N}.t)s\overline{r} \Downarrow_{\mathsf{rf}} (\lambda x : \blacksquare\mathbf{N}.u)n}$$

$$\frac{(t[x/s])\overline{r} \Downarrow_{\mathsf{rf}} u}{(\lambda x : aH.t)s\overline{r} \Downarrow_{\mathsf{rf}} u} \qquad \frac{t \Downarrow_{\mathsf{rf}} u}{\lambda x : aA.t \Downarrow_{\mathsf{rf}} \lambda x : aA.u} \qquad \frac{t_j \Downarrow_{\mathsf{rf}} s_j}{x\overline{t} \Downarrow_{\mathsf{rf}} x\overline{s}}$$

Figure 3.5: The relation $\Downarrow_{\mathsf{rf}}$: Inference Rules

Moreover, a third ternary relation $\Downarrow$ between closed terms of type $\mathbf{N}$ and numerals can be defined by the rule below:

$$\frac{t \Downarrow_{\mathsf{rf}} s \qquad s \Downarrow_{\mathsf{nf}} n}{t \Downarrow n}$$

A peculiarity of the just introduced relations with respect to similar ones is the following: whenever a statement in the form $t \Downarrow_{\mathsf{nf}} s$ is an immediate premise of another statement $r \Downarrow_{\mathsf{nf}} q$, then t needs to be structurally smaller than r, provided all numerals are assumed to have the same internal structure. A similar but weaker statement holds for $\Downarrow_{\mathsf{rf}}$. This relies on the peculiarities of SLR, and in particular on the fact that variables of higher-order types can appear free at most once in terms, and that terms of base types cannot be passed to functions without having been completely evaluated. In other words, the just described operational semantics is structural in a very strong sense, and this allows to prove properties about it by induction on the structure of *terms*, as we will experience in a moment.

We need to introduce a new definition of size, call it $|t|_{\mathsf{w}}$. It is a definition of size, where all numerals have size equal to 1. Formally, it is defined in the following way:

$$\begin{aligned}
|x|_{\mathsf{w}} &= 1 \\
|ts|_{\mathsf{w}} &= |t|_{\mathsf{w}} + |s|_{\mathsf{w}} \\
|\lambda x : aA.t|_{\mathsf{w}} &= |t|_{\mathsf{w}} + 1 \\
|\mathtt{case}_A\ t\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q|_{\mathsf{w}} &= |t|_{\mathsf{w}} + |s|_{\mathsf{w}} + |r|_{\mathsf{w}} + |q|_{\mathsf{w}} + 1 \\
|\mathtt{recursion}_A\ t\, s\, r|_{\mathsf{w}} &= |t|_{\mathsf{w}} + |s|_{\mathsf{w}} + |r|_{\mathsf{w}} + 1 \\
|n|_{\mathsf{w}} &= 1 \\
|\mathtt{S}_0|_{\mathsf{w}} = |\mathtt{S}_1|_{\mathsf{w}} &= |\mathtt{P}|_{\mathsf{w}} = 1
\end{aligned}$$

It's now time to analyze how big derivations for $\Downarrow_{\mathsf{nf}}$ and $\Downarrow_{\mathsf{rf}}$ can be with respect to the size of the underlying term. Let us start with $\Downarrow_{\mathsf{nf}}$ and prove that, since it can only be applied to explicit terms, the sizes of derivations must be very small:

Proposition 3.1 *Suppose that $\vdash t : \mathbf{N}$, where t is explicit. Then for every $\pi : t \Downarrow_{\mathsf{nf}} m$ it holds that*

1. $|\pi| \leq 2 \cdot |t|$;

2. If $s \in \pi$, then $|s| \leq 2 \cdot |t|^2$;

Proof: Given any term t, $|t|_{\mathsf{w}}$ and $|t|_{\mathsf{n}}$ are defined, respectively, as the size of t where every numeral counts for 1 and the maximum size of the numerals that occur in t. On the other hand, $|\cdot|_{\mathsf{n}}$ is defined as follows: $|x|_{\mathsf{n}} = 0$, $|ts|_{\mathsf{n}} = \max\{|t|_{\mathsf{n}}, |s|_{\mathsf{n}}\}$, $|\lambda x : aA.t|_{\mathsf{n}} = |t|_{\mathsf{n}}$, $|\mathtt{case}_A\ t\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q|_{\mathsf{n}} = \max\{|t|_{\mathsf{n}}, |s|_{\mathsf{n}}, |r|_{\mathsf{n}}, |q|_{\mathsf{n}}\}$, $|\mathtt{recursion}_A\ t\, s\, r|_{\mathsf{n}} = \max\{|t|_{\mathsf{n}}, |s|_{\mathsf{n}}, |r|_{\mathsf{n}}\}$, $|n|_{\mathsf{n}} = \lceil \log_2(n) \rceil$, and $|\mathtt{S}_0|_{\mathsf{n}} = |\mathtt{S}_1|_{\mathsf{n}} = |\mathtt{P}|_{\mathsf{n}} = 0$. Clearly, $|t| \leq |t|_{\mathsf{w}} \cdot |t|_{\mathsf{n}}$. We prove the following strengthening of the statements above by induction on $|t|_{\mathsf{w}}$:

1. $|\pi| \leq |t|_{\mathsf{w}}$;
2. If $s \in \pi$, then $|s|_{\mathsf{w}} \leq |t|_{\mathsf{w}}$ and $|s|_{\mathsf{n}} \leq |t|_{\mathsf{n}} + |t|_{\mathsf{w}}$;

Some interesting cases:

- Suppose t is $\mathtt{S}_i s$. Depending on $\mathtt{S}_i$ we could have two different derivations:

$$\frac{\rho : s \Downarrow_{\mathsf{nf}} n}{\mathtt{S}_0 s \Downarrow_{\mathsf{nf}} 2 \cdot n} \qquad \frac{\rho : s \Downarrow_{\mathsf{nf}} n}{\mathtt{S}_1 s \Downarrow_{\mathsf{nf}} 2 \cdot n + 1}$$

  Suppose we are in the case where $\mathtt{S}_i \equiv \mathtt{S}_0$. Then, for every $r \in \pi$,

$$\begin{aligned} |\pi| = |\rho| + 1 &\leq |s|_{\mathsf{w}} + 1 = |t|_{\mathsf{w}}; \\ |r|_{\mathsf{w}} &\leq |s|_{\mathsf{w}} \leq |t|_{\mathsf{w}} \\ |r|_{\mathsf{n}} &\leq |s|_{\mathsf{n}} + |s|_{\mathsf{w}} + 1 = |s|_{\mathsf{n}} + |t|_{\mathsf{w}} \\ &= |t|_{\mathsf{n}} + |t|_{\mathsf{w}} \end{aligned}$$

  The case where $\mathtt{S}_i \equiv \mathtt{S}_1$ is proved in the same way.

- Suppose t is $\mathtt{P}s$.

$$\frac{\rho : s \Downarrow_{\mathsf{nf}} 0}{\mathtt{P}s \Downarrow_{\mathsf{nf}} 0} \qquad \frac{\rho : s \Downarrow_{\mathsf{nf}} n \qquad n \geq 1}{\mathtt{P}s \Downarrow_{\mathsf{nf}} \lfloor \frac{n}{2} \rfloor}$$

  We focus on case where $n > 1$, the other case is similar. For every $r \in \pi$ we have

$$\begin{aligned} |\pi| = |\rho| + 1 &\leq |s|_{\mathsf{w}} + 1 = |t|_{\mathsf{w}} \\ |r|_{\mathsf{w}} &\leq |s|_{\mathsf{w}} \leq |t|_{\mathsf{w}} \\ |r|_{\mathsf{n}} &\leq |s|_{\mathsf{n}} + |s|_{\mathsf{w}} + 1 = |s|_{\mathsf{n}} + |t|_{\mathsf{w}} \\ &= |t|_{\mathsf{n}} + |t|_{\mathsf{w}} \end{aligned}$$

- Suppose t is n.

$$\frac{}{n \Downarrow_{\mathsf{nf}} n}$$

By knowing $|\pi| = 1$, $|n|_{\mathsf{w}} = 1$ and $|n|_{\mathsf{n}} = |n|$, the proof is trivial.

- Suppose that t is $(\lambda y : a\mathbf{N}.s)r\overline{q}$. All derivations π for t are in the following form:

$$\frac{\rho : r \Downarrow_{\mathsf{nf}} o \qquad \mu : (s[y/o])\overline{q} \Downarrow_{\mathsf{nf}} m}{t \Downarrow_{\mathsf{nf}} m}$$

Then, for every $u \in \pi$,

$$\begin{aligned}
|\pi| &\leq |\rho| + |\mu| + 1 \leq |r|_{\mathsf{w}} + |s[y/o]\overline{q}|_{\mathsf{w}} + 1 \\
&= |r|_{\mathsf{w}} + |s\overline{q}|_{\mathsf{w}} + 1 \leq |t|_{\mathsf{w}}; \\
|u|_{\mathsf{n}} &\leq \max\{|r|_{\mathsf{n}} + |r|_{\mathsf{w}}, |s[y/o]\overline{q}|_{\mathsf{n}} + |s[y/o]\overline{q}|_{\mathsf{w}}\} \\
&= \max\{|r|_{\mathsf{n}} + |r|_{\mathsf{w}}, |s[y/o]\overline{q}|_{\mathsf{n}} + |s\overline{q}|_{\mathsf{w}}\} \\
&= \max\{|r|_{\mathsf{n}} + |r|_{\mathsf{w}}, \max\{|s\overline{q}|_{\mathsf{n}}, |o|\} + |s\overline{q}|_{\mathsf{w}}\} \\
&= \max\{|r|_{\mathsf{n}} + |r|_{\mathsf{w}}, |s\overline{q}|_{\mathsf{n}} + |s\overline{q}|_{\mathsf{w}}, |o| + |s\overline{q}|_{\mathsf{w}}\} \\
&\leq \max\{|r|_{\mathsf{n}} + |r|_{\mathsf{w}}, |s\overline{q}|_{\mathsf{n}} + |s\overline{q}|_{\mathsf{w}}, |r|_{\mathsf{n}} + |r|_{\mathsf{w}} + |s\overline{q}|_{\mathsf{w}}\} \\
&\leq \max\{|r|_{\mathsf{n}}, |s\overline{q}|_{\mathsf{n}}\} + |r|_{\mathsf{w}} + |s\overline{q}|_{\mathsf{w}} \\
&\leq \max\{|r|_{\mathsf{n}}, |s\overline{q}|_{\mathsf{n}}\} + |t|_{\mathsf{w}} \\
&= |t|_{\mathsf{n}} + |t|_{\mathsf{w}}; \\
|u|_{\mathsf{w}} &\leq \max\{|r|_{\mathsf{w}}, |s[y/o]\overline{q}|_{\mathsf{w}}, |t|_{\mathsf{w}}\} \\
&= \max\{|r|_{\mathsf{w}}, |s\overline{q}|_{\mathsf{w}}, |t|_{\mathsf{w}}\} \leq |t|_{\mathsf{w}}.
\end{aligned}$$

If $u \in \pi$, then either $u \in \rho$ or $u \in \mu$ or simply $u = t$. This, together with the induction hypothesis, implies $|u|_{\mathsf{w}} \leq \max\{|r|_{\mathsf{w}}, |s[y/o]\overline{q}|_{\mathsf{w}}, |t|_{\mathsf{w}}\}$. Notice that $|s\overline{q}|_{\mathsf{w}} = |s[y/o]\overline{q}|_{\mathsf{n}}$ holds because any occurrence of y in s counts for 1, but also o itself counts for 1 (see the definition of $|\cdot|_{\mathsf{w}}$ above). More generally, duplication of *numerals* for a variable in t does not make $|t|_{\mathsf{w}}$ bigger.

- Suppose t is $(\lambda y : aH.)r\overline{q}$. Without loosing generality we can say that it derives from the following derivation:

$$\frac{\rho : (s[y/r])\overline{q} \Downarrow_{\mathsf{nf}} n}{(\lambda y : aH.s)r\overline{q} \Downarrow_{\mathsf{nf}} n}$$

For the reason that y has type H we can be sure that it appears at most once in s. So, $|s[y/r]| \leq |sr|$ and, moreover, $|s[y/r]\overline{q}|_{\mathsf{w}} \leq |sr\overline{q}|_{\mathsf{w}}$ and $|s[y/r]\overline{q}|_{\mathsf{n}} \leq |sr\overline{q}|_{\mathsf{n}}$. We have, for

all $u \in \rho$:

$$\begin{aligned} |\pi| &= |\rho| + 1 \leq |s[y/r]\overline{q}|_{\mathsf{w}} + 1 \leq |t|_{\mathsf{w}} \\ |u|_{\mathsf{w}} &\leq |s[y/r]\overline{q}|_{\mathsf{w}} \leq |sr\overline{q}|_{\mathsf{w}} \leq |t|_{\mathsf{w}} \\ |u|_{\mathsf{n}} &\leq |s[y/r]\overline{q}|_{\mathsf{n}} + |s[y/r]\overline{q}|_{\mathsf{w}} \leq |sr\overline{q}|_{\mathsf{n}} + |sr\overline{q}|_{\mathsf{w}} \leq |t|_{\mathsf{n}} + |t|_{\mathsf{w}} \end{aligned}$$

and this means that the same inequalities hold for every $u \in \pi$.

- Suppose t is $\mathtt{case}_A\ s\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u$. We could have three possible derivations:

$$\frac{\rho : s \Downarrow_{\mathsf{nf}} 0 \qquad \mu : r\overline{v} \Downarrow_{\mathsf{nf}} n}{(\mathtt{case}_A\ s\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u)\overline{v} \Downarrow_{\mathsf{nf}} n}$$

$$\frac{\rho : s \Downarrow_{\mathsf{nf}} 2n \qquad \mu : q\overline{v} \Downarrow_{\mathsf{nf}} m \qquad n \geq 1}{(\mathtt{case}_A\ s\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u)\overline{v} \Downarrow_{\mathsf{nf}} m}$$

$$\frac{\rho : s \Downarrow_{\mathsf{nf}} 2n+1 \qquad \mu : u\overline{v} \Downarrow_{\mathsf{nf}} m}{(\mathtt{case}_A\ s\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u)\overline{v} \Downarrow_{\mathsf{nf}} m}$$

  we will focus on the case where the value of s is odd. All the other cases are similar. For all $z \in \pi$ we have:

$$\begin{aligned} |\pi| &\leq |\rho| + |\mu| + 1 \\ &\leq |s|_{\mathsf{w}} + |u\overline{v}|_{\mathsf{w}} + 1 \leq |t|_{\mathsf{w}} \\ |z|_{\mathsf{w}} &\leq |s|_{\mathsf{w}} + |r|_{\mathsf{w}} + |q|_{\mathsf{w}} + |u\overline{v}|_{\mathsf{w}} \leq |t|_{\mathsf{w}} \\ |z|_{\mathsf{n}} &= \max\{|s|_{\mathsf{n}} + |s|_{\mathsf{w}}, |u\overline{v}|_{\mathsf{n}} + |u\overline{v}|_{\mathsf{w}}, |r|_{\mathsf{n}}, |q|_{\mathsf{n}}\} \\ &\leq \max\{|s|_{\mathsf{n}}, |u\overline{v}|_{\mathsf{n}}, |r|_{\mathsf{n}}, |q|_{\mathsf{n}}\} + |s|_{\mathsf{w}} + |u\overline{v}|_{\mathsf{w}} \\ &\leq |t|_{\mathsf{w}} + |t|_{\mathsf{n}} \end{aligned}$$

This concludes the proof. □

As opposed to $\Downarrow_{\mathsf{nf}}$, $\Downarrow_{\mathsf{rf}}$ unrolls instances of primitive recursion, and thus cannot have the very simple combinatorial behavior of $\Downarrow_{\mathsf{nf}}$. Fortunately, however, everything stays under control:

Proposition 3.2 *Suppose that $x_1 : \Box\mathbf{N}, \ldots, x_i : \Box\mathbf{N} \vdash t : A$, where A is $\Box$-free type. Then there are polynomials p_t and q_t such that for every $n_1, \ldots, n_i$ and for every $\pi : t[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} s$ it holds that:*

1. *$|\pi| \leq p_t(\sum_i |n_i|)$;*
2. *If $s \in \pi$, then $|s| \leq q_t(\sum_i |n_i|)$.*

Proof: The following strengthening of the result can be proved by induction on the structure of a type derivation μ for t: if $x_1 : \Box\mathbf{N}, \ldots, x_i : \Box\mathbf{N}, y_1 : \blacksquare A_1, \ldots, y_j : \blacksquare A_j \vdash t : A$, where A is positively $\Box$-free and $A_1, \ldots, A_j$ are negatively $\Box$-free. Then there are polynomials p_t and q_t such that for every $n_1, \ldots, n_i$ and for every $\pi : t[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} s$ it holds that

1. $|\pi| \leq p_t(\sum_i |n_i|)$;
2. If $s \in \pi$, then $|s| \leq q_t(\sum_i |n_i|)$.

In defining positively and negatively $\Box$-free types, let us proceed by induction on types:

- $\mathbf{N}$ is both positively and negatively $\Box$-free;
- $\Box A \to B$ is *not* positively $\Box$-free, and is negatively $\Box$-free whenever A is positively $\Box$-free and B is negatively $\Box$-free;
- $C = \blacksquare A \to B$ is positively $\Box$-free if A is negatively and B is positively $\Box$-free. C is negatively $\Box$-free if A is positively $\Box$-free and B is negatively $\Box$-free.

Please observe that if A is positively $\Box$-free and $B <: A$, then B is positively $\Box$-free. Conversely, if A is negatively $\Box$-free and $A <: B$, then B is negatively $\Box$-free. This can be easily proved by induction on the structure of A. We are ready to start the proof, now. Let us consider some cases, depending on the shape of μ

- If the only typing rule in μ is (T-Const-Aff), then $t \equiv c$, $p_t(x) \equiv 1$ and $q_t(x) \equiv 1$. The thesis is proved.
- If the last rule was (T-Var-Aff) then $t \equiv x$, $p_t(x) \equiv 1$ and $q_t(x) \equiv x$. The thesis is proved
- If the last rule was (T-Arr-I) then $t \equiv \lambda x : \blacksquare A.s$. Notice that the aspect is $\blacksquare$ because the type of our term has to be positively $\Box$-free. So, we have the following derivation:

$$\frac{\rho : s[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} v}{\lambda x : aA.s[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} \lambda x : aA.v}$$

  If the type of t is positively $\Box$-free, then also the type of s is positively $\Box$-free. We can apply induction hypothesis. Define p_t and q_t as:

$$p_t(x) \equiv p_s(x) + 1$$
$$q_t(x) \equiv q_s(x) + 1$$

Indeed, we have:

$$\begin{aligned} |\pi| &\equiv |\rho| + 1 \\ &\leq p_s(\sum_i |n_i|) + 1 \end{aligned}$$

- If last rule was (T-Sub) then we have a typing derivation that ends in the following way:

$$\frac{\Gamma \vdash t : A \qquad A <: B}{\Gamma \vdash t : B}$$

we can apply induction hypothesis on $t : A$ because if B is positively $\Box$-free, then also A will be too. Define $p_{t:B}(x) \equiv p_{t:A}(x)$ and $q_{t:B}(x) \equiv q_{t:A}(x)$.

- If the last rule was (T-Case). Suppose $t \equiv (\mathtt{case}_A\ s\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u)$. The constraints on the typing rule (T-Case) ensure us that the induction hypothesis can be applied to s, r, q, u. The definition of $\Downarrow_{\mathsf{rf}}$ tells us that any derivation of $t[\overline{x}/\overline{n}]$ must have the following shape:

$$\frac{\begin{array}{cc} \rho : s[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} z & \nu : q[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} b \\ \mu : r[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} a & \sigma : u[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} c \end{array}}{t[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} (\mathtt{case}_A\ z\ \mathtt{zero}\ a\ \mathtt{even}\ b\ \mathtt{odd}\ c)}$$

Let us now define p_t and q_t as follows:

$$\begin{aligned} p_t(x) &= p_s(x) + p_r(x) + p_q(x) + p_u(x) + 1 \\ q_t(x) &= q_s(x) + q_r(x) + q_q(x) + q_u(x) + 1 \end{aligned}$$

We have:

$$\begin{aligned} |\pi| &\leq |\rho| + |\mu| + |\nu| + |\sigma| + 1 \\ &\leq p_s(\sum_i |n_i|) + p_r(\sum_i |n_i|) + p_q(\sum_i |n_i|) + p_u(\sum_i |n_i|) + 1 \\ &= p_t(\sum_i |n_i|). \end{aligned}$$

Similarly, if $z \in \pi$, it is easy to prove that $|z| \leq q_z(\sum_i |n_i|)$.

- If the last rule was (T-Rec). Suppose $t \equiv (\mathtt{recursion}_A\ s\ r\ q)$. By looking at the typing rule (figure 3.3) for (T-Rec) we are sure to be able to apply induction hypothesis on

s, r, q. Definition of $\Downarrow_{\mathsf{rf}}$ ensure also that any derivation for $t[\overline{x}/\overline{n}]$ must have the following shape:

$$\frac{\begin{array}{c}\rho : s[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} z \quad \mu : z[\overline{x}/\overline{n}] \Downarrow_{\mathsf{nf}} n \\ \nu : r[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} a \\ \varrho_0 : qz[\overline{x}, z/\overline{n}, \lfloor \frac{n}{2^0} \rfloor] \Downarrow^0_{\mathsf{rf}} q_0 \\ \cdots \\ \varrho_{|n|-1} : qz[\overline{x}, z/\overline{n}, \lfloor \frac{n}{2^{|n|-1}} \rfloor] \Downarrow_{\mathsf{rf}} q_{|n|-1}\end{array}}{(\mathtt{recursion}_A\ s\, r\, q)[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} q_0(\ldots(q_{(|n|-1)}a)\ldots)}$$

Notice that we are able to apply $\Downarrow_{\mathsf{nf}}$ on term z because, by definition, s has only free variables of type $\Box\mathbf{N}$ (see figure 3.3). So, we are sure that z is a closed term of type $\mathbf{N}$ and we are able to apply the $\Downarrow_{\mathsf{nf}}$ algorithm.
Let define p_t and q_t as follows:

$$\begin{aligned} p_t(x) &\equiv p_s(x) + 2 \cdot q_s(x) + p_r(x) + 2 \cdot q_s(x)^2 \cdot p_q(x + 2 \cdot q_s(x)^2) + 1 \\ q_t(x) &\equiv q_s(x) + q_r(x) + 2 \cdot q_s(x)^2 + q_q(x + 2 \cdot q_s(x)^2) \end{aligned}$$

Notice that $|z|$ is bounded by $q_s(x)$. Notice that by applying theorem 3.1 on μ (z has no free variables) we have that every $v \in \mu$ is s.t.$v \leq 2 \cdot |z|^2$.
We have:

$$\begin{aligned} |\pi| &\leq |\rho| + |\mu| + |\nu| + \sum_i (|\varrho_i|) + 1 \\ &\leq p_s(\sum_i |n_i|) + 2 \cdot |z| + p_r(\sum_i |n_i|) + |n| \cdot p_{qz}(\sum_i |n_i| + |n|) + 1 \\ &\leq p_s(\sum_i |n_i|) + 2 \cdot q_s(\sum_i |n_i|) + p_r(\sum_i |n_i|) + \\ &\quad + 2 \cdot q_s(\sum_i |n_i|)^2 \cdot p_{qz}(\sum_i |n_i| + 2 \cdot q_s(\sum_i |n_i|)^2) + 1 \end{aligned}$$

Similarly, for every $z \in \pi$:

$$\begin{aligned} |z| &\leq q_s(\sum_i |n_i|) + 2 \cdot q_s(\sum_i |n_i|)^2 + q_r(\sum_i |n_i|) + q_{qz}(\sum_i |n_i| + |n|) \\ &\leq q_s(\sum_i |n_i|) + 2 \cdot q_z(\sum_i |n_i|)^2 + q_r(\sum_i |n_i|) + q_{qz}(\sum_i |n_i| + 2 \cdot q_s(\sum_i |n_i|)^2) \end{aligned}$$

- In the following cases the last rule is (T-Arr-E).
- $t \equiv x\overline{s}$. In this case, obviously, the free variable x has type $\blacksquare A_i$ $(1 \leq i \leq j)$. By definition x is negatively $\square$-free. This it means that every term in $\overline{s}$ has a type that is positively $\square$-free. By knowing that the type of x is negatively $\square$-free, we conclude that the type of our term t is $\square$-free (because is both negatively and positively $\square$-free at the same time).

  Definition of $\Downarrow_{\mathsf{rf}}$ ensures us that the derivation will have the following shape:

$$\frac{\rho_i : s_j[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} r_j}{x\overline{s}[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} x\overline{r}}$$

  We define p_t and q_t as:

$$p_t(x) \equiv \sum_j p_{s_j}(x) + 1$$
$$q_t(x) \equiv \sum_j q_{s_j}(x) + 1$$

  Indeed we have

$$\begin{aligned} |\pi| &\leq \sum_j |\rho_j| + 1 \\ &\leq \sum_j \{p_{s_j}(\sum_i |n_i|)\} + 1 \end{aligned}$$

  Similarly, if $z \in \pi$, it is easy to prove that $|z| \leq q_z(\sum_i |n_i|)$.
- If $t \equiv \mathsf{S}_0 s$, then s have type $\mathbf{N}$ in the context Γ. The derivation π has the following form

$$\frac{\rho : s[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} z}{\mathsf{S}_0 s[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} \mathsf{S}_0 z}$$

  Define $p_t(x) = p_s(x) + 1$ and $q_t(x) = q_s(x) + 1$. One can easily check that, by induction hypothesis

$$\begin{aligned} |\pi| &\leq |\rho| + 1 \leq p_s(\sum_i |n_i|) + 1 \\ &= p_t(\sum_i |n_i|). \end{aligned}$$

  Analogously, if $r \in \pi$ then

$$|s| \leq q_s(\sum_i |n_i|) + 1 \leq q_t(\sum_i |n_i|).$$

- If $t \equiv \mathtt{S}_1 s$ or $t \equiv \mathtt{P}s$, then we can proceed exactly as in the previous case.
- Cases where we have on the left side a case or a recursion with some arguments, is trivial: can be brought back to cases that we have considered.
- If t is $(\lambda x : \Box\mathbf{N}.s)r\overline{q}$, then we have the following derivation:

$$\dfrac{\begin{array}{c}\rho : r[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} a \\ \mu : a[\overline{x}/\overline{n}] \Downarrow_{\mathsf{nf}} n \qquad \nu : (s[x/n])\overline{q}[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} v\end{array}}{(\lambda x : \Box\mathbf{N}.s)r\overline{q}[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} v}$$

  By hypothesis t is positively $\Box$-free and so also r (whose type is $\mathbf{N}$) and $s\overline{q}$ are positively $\Box$-free. So, we are sure that we are able to use induction hypothesis.
  Let p_t and q_t be:

$$p_t(x) \equiv p_r(x) + 2 \cdot q_r(x) + p_{s\overline{q}}(x + 2 \cdot q_r(x)^2) + 1$$
$$q_t(x) \equiv q_{s\overline{q}}(x + 2 \cdot q_r(x)^2) + q_r(x) + 2 \cdot q_r(x)^2 + 1$$

  We have:

$$\begin{aligned}|\pi| &\equiv |\rho| + |\mu| + |\nu| + 1 \\ &\leq p_r(\sum_i |n_i|) + 2 \cdot |a| + p_{s\overline{q}}(\sum_i |n_i| + |n|) + 1 \\ &\leq p_r(\sum_i |n_i|) + 2 \cdot q_r(\sum_i |n_i|) + p_{s\overline{q}}(\sum_i |n_i| + 2 \cdot q_r(\sum_i |n_i|)^2) + 1\end{aligned}$$

  By construction, remember that s has no free variables of type $\blacksquare\mathbf{N}$. For theorem 3.1 (z has no free variables) we have $v \in \mu$ is s.t. $|v| \leq 2 \cdot |a|^2$. By applying induction hypothesis we have that every $v \in \rho$ is s.t. $|v| \leq q_r(\sum_i |n_i|)$, every $v \in \nu$ is s.t.

$$\begin{aligned}|v| &\leq q_{s\overline{q}}(\sum_i |n_i| + |n|) \\ &\leq q_{s\overline{q}}(\sum_i |n_i| + 2 \cdot |a|^2) \\ &\leq q_{s\overline{q}}(\sum_i |n_i| + 2 \cdot q_r(\sum_i |n_i|)^2)\end{aligned}$$

  We can prove the second point of our thesis by setting $q_t(\sum_i |n_i|)$ as $q_{s\overline{q}}(\sum_i |n_i| + 2 \cdot q_r(\sum_i |n_i|)^2) + q_r(\sum_i |n_i|) + 2 \cdot q_r(\sum_i |n_i|)^2 + 1$.
- If t is $(\lambda x : \blacksquare\mathbf{N}.s)r\overline{q}$, then we have the following derivation:

$$\frac{\rho : r[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} a \qquad \mu : a[\overline{x}/\overline{n}] \Downarrow_{\mathsf{nf}} n \qquad \nu : s\overline{q}[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} u}{(\lambda x : \blacksquare\mathbf{N}.s)r\overline{q}[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} (\lambda x : \blacksquare\mathbf{N}.u)n}$$

By hypothesis we have t that is positively $\Box$-free. So, also r and a (whose type is $\mathbf{N}$) and $s\overline{q}$ are positively $\Box$-free. We define p_t and q_t as:

$$p_t(x) \equiv p_r(x) + 2 \cdot q_r(x) + p_{s\overline{q}}(x) + 1;$$
$$q_t(x) \equiv q_r(x) + 2 \cdot q_r(x)^2 + q_{s\overline{q}}(x) + 1.$$

We have:

$$\begin{aligned} |\pi| &\equiv |\rho| + |\mu| + |\nu| + 1 \\ &\leq p_r(\sum_i |n_i|) + 2 \cdot q_r(\sum_i |n_i|) + p_{s\overline{q}}(\sum_i |n_i|) + 1 \end{aligned}$$

Similarly, if $z \in \pi$, it is easy to prove that $|z| \leq q_t(\sum_i |n_i|)$.

- If t is $(\lambda x : aH.s)r\overline{q}$, then we have the following derivation:

$$\frac{\rho : (s[x/r])\overline{q}[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} v}{(\lambda x : aH.s)r\overline{q}[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}} v}$$

By hypothesis we have t that is positively $\Box$-free. So, also $s\overline{q}$ is positively $\Box$-free. r has a higher-order type H and so we are sure that $|(s[x/r])\overline{q}| < |(\lambda x : aH.s)r\overline{q}|$. Define p_t and q_t as:

$$p_t(x) \equiv p_{(s[x/r])\overline{q}}(x) + 1;$$
$$q_t(x) \equiv q_{(s[x/r])\overline{q}}(x) + 1.$$

By applying induction hypothesis we have:

$$|\pi| \equiv |\rho| + 1 \leq p_{(s[x/r])\overline{q}}(\sum_i |n_i|) + 1$$

By using induction we are able also to prove the second point of our thesis.

This concludes the proof. □

Following the definition of $\Downarrow$, it is quite easy to obtain, given a first order term t, of arity k, a deterministic Turing machine that, when receiving on input (an encoding of) $n_1 \ldots n_k$, produces on output the expected value m. Indeed, $\Downarrow_{\mathsf{rf}}$ and $\Downarrow_{\mathsf{nf}}$ are designed in a very algorithmic way. Moreover, the obtained Turing machine works in polynomial time, due to propositions 3.1 and 3.2. Formally:

Theorem 3.4 (Soundness) *Suppose t is a first order term of arity k. Then there is a deterministic Turing machine M_t running in polynomial time such that M_t on input $n_1 \dots n_k$ returns exactly the expected value m.*

Proof: By propositions 3.1 and 3.2. □

3.4.4 Polytime Completeness

In the previous section, we proved that the behavior of any SLR first-order term can be somehow simulated by a deterministic polytime Turing machine. What about the converse? In this section, we prove that any deterministic polynomial time Turing machine (DTM in the following) can be encoded in SLR.

To facilitate the encoding, we extend our system with pairs and projections. All the proofs in previous sections remain valid. Base types now comprise not only natural numbers but also pairs of base types:

$$G := \mathbf{N} \mid G \times G.$$

Terms now contain a binary construct $\langle \cdot, \cdot \rangle$ and two unary constructs $\pi_1(\cdot)$ and $\pi_2(\cdot)$, which can be given a type by the rules below:

$$\frac{\Gamma; \Delta_1 \vdash t : G \qquad \Gamma; \Delta_2 \vdash s : F}{\Gamma; \Delta_1, \Delta_2 \vdash \langle t, s \rangle : G \times F}$$

$$\frac{\Gamma \vdash t : G \times F}{\Gamma \vdash \pi_1(t) : G} \qquad \frac{\Gamma \vdash t : G \times F}{\Gamma \vdash \pi_2(t) : F}$$

As syntactic sugar, we will use $\langle t_1 \dots, t_i \rangle$ (where $i \geq 1$) for the term

$$\langle t_1, \langle t_2, \dots \langle t_{i-1}, t_i \rangle \dots \rangle \rangle.$$

For every $n \geq 1$ and every $1 \leq i \leq n$, we can easily build a term π_i^n which extracts the i-th component from tuples of n elements: this can be done by composing $\pi_1(\cdot)$ and $\pi_2(\cdot)$. With a slight abuse on notation, we sometimes write π_i for π_i^n.

3.4.5 Unary Natural Numbers and Polynomials

Natural numbers in SLR are represented in binary. In other words, the basic operations allowed on them are $\mathtt{S}_0$, $\mathtt{S}_1$ and $\mathtt{P}$, which correspond to appending a binary digit to the

right of the number (seen as a binary string) or stripping the rightmost such digit. This is even clearer if we consider the length $|n|$ of a numeral n, which is only logarithmic in n.

For every numeral n, we can extract the unary encoding of its length:

$$\mathsf{encode} \equiv \lambda t : \Box\mathbf{N}.\mathtt{recursion}_{\mathbf{U}}\ t\, 0\, (\lambda x : \Box\mathbf{U}.\lambda y : \blacksquare\mathbf{U}.\mathtt{S}_1 y) : \Box\mathbf{N} \to \mathbf{U}$$

Predecessor and successor functions are defined in our language, simply as $\mathtt{P}$ and $\mathtt{S}_1$. We need to show how to express polynomials and in order to do this we will define the operators $\mathsf{add} : \Box\mathbf{U} \to \blacksquare\mathbf{U} \to \mathbf{U}$ and $\mathsf{mult} : \Box\mathbf{U} \to \Box\mathbf{U} \to \mathbf{U}$. We define add as

$$\begin{aligned}\mathsf{add} \equiv & \lambda x : \Box\mathbf{U}.\lambda y : \blacksquare\mathbf{U}.\\ & \mathtt{recursion}_{\mathbf{U}}\ x\, y\, (\lambda x : \Box\mathbf{U}.\lambda y : \blacksquare\mathbf{U}.\mathtt{S}_1 y) : \Box\mathbf{U} \to \blacksquare\mathbf{U} \to \mathbf{U}\end{aligned}$$

Similarly, we define mult as

$$\begin{aligned}\mathsf{mult} \equiv & \lambda x : \Box\mathbf{U}.\lambda y : \Box\mathbf{U}.\\ & \mathtt{recursion}_{\mathbf{U}}\ (\mathtt{P}x)\, y\, (\lambda x : \Box\mathbf{U}.\lambda z : \blacksquare\mathbf{U}.\mathsf{add} y z) : \Box\mathbf{U} \to \Box\mathbf{U} \to \mathbf{U}\end{aligned}$$

The following is quite easy:

Lemma 3.6 *Every polynomial of one variable with natural coefficients can be encoded as a term of type* $\Box\mathbf{U} \to \mathbf{U}$.

Proof: Simply, turn add into a term of type $\Box\mathbf{U} \to \Box\mathbf{U} \to \mathbf{U}$ by way of subtyping and then compose add and mult has much as needed to encode the polynomial at hand. □

3.4.6 Finite Sets

Any finite, linearly ordered set $F = (|F|, \sqsubseteq_F)$ can be naturally encoded as an "initial segment" of $\mathbf{N}$: if $|F| = \{a_0, \ldots, a_i\}$ where $a_i \sqsubseteq_F a_j$ whenever $i \leq j$, then a_i is encoded simply by the natural number whose binary representation is 10^i. For reasons of clarity, we will denote $\mathbf{N}$ as $\mathbf{F}_F$. We can do some case analysis on an element of $\mathbf{F}_F$ by the combinator

$$\mathsf{switch}_A^F : \blacksquare\mathbf{F}_F \to \underbrace{\blacksquare A \to \ldots \to \blacksquare A}_{i \text{ times}} \to \blacksquare A \to A$$

where A is a $\Box$-free type and i is the cardinality of $|F|$. The term above can be defined by induction on i:

- If $i = 0$, then it is simply $\lambda x : \blacksquare\mathbf{F}_F.\lambda y : \blacksquare A.y$.
- If $i \geq 1$, then it is the following:

$$\lambda x : \blacksquare\mathbf{F}_F.\lambda y_0 : \blacksquare A. \ldots \lambda y_i : \blacksquare A.\lambda z \blacksquare A.$$

$$(\mathtt{case}_A\ x\ \mathtt{zero}(\lambda h : \blacksquare A.h)$$

$$\mathtt{even}\ (\lambda h : \blacksquare A.\mathsf{switch}_A^E(\mathtt{P}x)y_1 \ldots y_i h)$$

$$\mathtt{odd}\ (\lambda h : \blacksquare A.y_0)$$

where E is the subset of F of those elements with positive indices.

3.4.7 Strings

Suppose $\Sigma = \{a_0, \ldots, a_i\}$ is a finite alphabet. Elements of Σ can be encoded following the just described scheme, but how about strings in Σ^*? We can somehow proceed similarly: the string $a_{j_1} \ldots a_{j_k}$ can be encoded as the natural number

$$10^{j_1}10^{j_2} \ldots 10^{j_k}.$$

Whenevery we want to emphasize that a natural number is used as a string, we write $\mathbf{S}_\Sigma$ instead of $\mathbf{N}$. It is easy to build a term $\mathsf{append}_\Sigma : \blacksquare(\mathbf{S}_\Sigma \times \mathbf{F}_\Sigma) \to \mathbf{S}_\Sigma$ which appends the second argument to the first argument. Similarly, one can define a term $\mathsf{tail}_\Sigma : \blacksquare\mathbf{S}_\Sigma \to \mathbf{S}_\Sigma \times \mathbf{F}_\Sigma$ which strips off the rightmost character a from the argument string and returns a together with the rest of the string; if the string is empty, a_0 is returned, by convention.

We also define a function $\mathsf{NtoS}_\Sigma : \square\mathbf{N} \to \mathbf{S}_\Sigma$ that takes a natural number and produce in output an encoding of the corresponding string in Σ^* (where i_0 and i_1 are the indices of 0 and 1 in Σ):

$$\mathsf{NtoS}_\Sigma \equiv \lambda x : \square\mathbf{N}.\mathtt{recursion}_{\mathbf{S}_\Sigma}\ x\ \sqcup$$

$$\lambda x : \blacksquare\mathbf{N}.\lambda y : \blacksquare\mathbf{S}.\mathtt{case}_{\mathbf{N}}\ x\ \mathtt{zero}\ \mathsf{append}_\Sigma\langle y, 10^{i_0}\rangle$$

$$\mathtt{even}\ \mathsf{append}_\Sigma\langle y, 10^{i_1}\rangle$$

$$\mathtt{odd}\ \mathsf{append}_\Sigma\langle y, 10^{i_1}\rangle : \square\mathbf{N} \to \mathbf{S}$$

Similarly, one can write a term $\mathsf{StoN}_\Sigma : \square\mathbf{S}_\Sigma \to \mathbf{N}$.

3.4.8 Deterministic Turing Machines

Let M be a deterministic Turing machine $M = (Q, q_0, F, \Sigma, \sqcup, \delta)$, where Q is the finite set of states of the machine; q_0 is the initial state; F is the set of final states of M; Σ is the

finite alphabet of the tape; $\sqcup \in \Sigma$ is the symbol for empty string; $\delta \subseteq (Q \times \Sigma) \times (Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\})$ is the transition function of M. For each pair $(q, s) \in Q \times \Sigma$, there is exactly one triple (r_1, t_1, d_1) such that $((q, s), (r_1, t_1, d_1)) \in \delta$. Configurations of M can be encoded as follows:

$$\langle t_{left}, t, t_{right}, s \rangle : \mathbf{S}_\Sigma \times \mathbf{F}_\Sigma \times \mathbf{S}_\Sigma \times \mathbf{F}_Q,$$

where t_{left} represents the left part of the main tape, t is the symbol read from the head of M, t_{right} the right part of the main tape; s is the state of our Turing Machine. Let the type $\mathbf{C}_M$ be a shortcut for $\mathbf{S}_\Sigma \times \mathbf{F}_\Sigma \times \mathbf{S}_\Sigma \times \mathbf{F}_Q$.

Suppose that M on input x runs in time bounded by a polynomial $p : \mathbb{N} \to \mathbb{N}$. Then we can proceed as follows:

- encode the polynomial p by using function $\mathsf{encode}, \mathsf{add}, \mathsf{mult}, \mathsf{dec}$ so that at the end we will have a function $\underline{p} : \Box\mathbf{N} \to \mathbf{U}$;
- write a term $\underline{\delta} : \blacksquare\mathbf{C}_M \to \mathbf{C}_M$ which mimicks δ.
- write a term $\mathsf{init}_M : \blacksquare\mathbf{S}_\Sigma \to \mathbf{C}_M$ which returns the initial configuration for M corresponding to the input string.

The term of type $\Box\mathbf{N} \to \mathbf{N}$ which has exactly the same behavior as M is the following:

$$\lambda x : \Box\mathbf{N}.\mathsf{StoN}_\Sigma(\mathtt{recursion}_{\mathbf{C}_M}\ (\underline{p}\ x)\,(\mathsf{init}_M\ (\mathsf{NtoS}_\Sigma(x)))\,(\lambda y : \blacksquare\mathbf{N}.\lambda z : \blacksquare\mathbf{C}_M.\underline{\delta}\ z)).$$

We then get a faithful encoding of PPTM into RSLR, which will be useful in the forthcoming section:

Theorem 3.5 *Suppose M is a deterministic Turing machine running in polynomial time such that for every n_i returns m_i. Then there is a first order term t such that for every n_i, tn_i evaluates to m_i.*

Definitions

We assume, without loss of generality that the right part of the tape is represented in reverse order. The alphabet of the tapes is $\{0, 1\}$. We define the current state of the machine as a numeral and we will use $\mathbf{Q}$ to refer to its type, even if $\mathbf{Q} \equiv \mathbf{N}$.

Every state q_i is encoded by considering the binary representation of i. E.g. the state q_3 of our machine is encoded as the numeral 11.

We define Turing Machines configurations in the following way

$$\langle x_{left}, x, x_{right}, Q\rangle : \mathbf{S} \times B \times \mathbf{S} \times \mathbf{Q}$$

where x_{left} represents the left part of the main tape, x is the symbol that is read from the head of our TM, x_{right} the right part of the main tape; similarly y represents the value read from the random tape. Q is the state of our Turing Machine. Let the type $\mathbf{C}$ be a shortcut for $\mathbf{S} \times B \times \mathbf{S} \times \mathbf{Q}$.

Basic Functions

We define two functions that take a configuration and give out a new configuration with the head of the first tape moved left or right.

$$\begin{aligned}\mathsf{shiftsx} \equiv \lambda x : \blacksquare\mathbf{C}.\langle & \mathsf{addString}\,(\pi_1 x)(\pi_2 x),\\ & \mathsf{vstring}\,(\pi_3 x),\\ & \mathsf{PP}(\pi_3 x),\\ & (\pi_4 x)\rangle : \blacksquare\mathbf{C} \to \mathbf{C}\end{aligned}$$

$$\begin{aligned}\mathsf{shiftdx} \equiv \lambda x : \blacksquare\mathbf{C}.\langle & \mathsf{PP}(\pi_1 x),\\ & \mathsf{vstring}\,(\pi_1 x),\\ & \mathsf{addString}\,(\pi_3 x)(\pi_2 x),\\ & (\pi_4 x)\rangle : \blacksquare\mathbf{C} \to \mathbf{C}\end{aligned}$$

Encoding input

The init function that creates the starting configuration so, will be the following one:

$$\mathsf{init} \equiv \lambda z : \square\mathbf{N}.(\lambda x : \blacksquare\mathbf{S}.\langle \mathsf{PP}x, \mathsf{vstring}\,x, \underline{\sqcup}, \underline{q_0}\rangle(\mathsf{NtoS}_z)) : \square\mathbf{N} \to \mathbf{C}$$

where $\underline{q_0}$ is the encoding of the init state of our Turing machine. Without loss of generality we decide that the starting position of our heads is at the left of the tapes and are set on the first element (from the right).

We say that $\underline{q_{err}}$ is the encoding of one of (we don't care which one) the halting states of the machine.

Encoding the transition function

Our $\underline{\delta}$ function will take a configuration and will produce in output a new configuration.

$\underline{\delta} \equiv \lambda x : \blacksquare\mathbf{C}.$

$\mathsf{switch}_{\mathbf{C}}^{\{0,\ldots,3\}}\mathbf{BtoS}(\pi_2)$

$\mathsf{switch}_{\mathbf{C}}^{\{0,\ldots,n\}}(\pi_4 x)$ // here the value of π_2 is 0.

// Here is the case where π_4 is q_0; apply $\mathsf{shiftdx}$ or $\mathsf{shiftsx}$ or the identity on the tapes, with the relative new state, according to the original δ function.

... // here are the other cases.

$\langle \pi_1 x, \pi_2 x, \pi_3 x, \pi_4 x, \pi_5 x, \pi_6 x, q_{err} \rangle$ // default value

... // here are the other three cases.

$\langle \pi_1 x, \pi_2 x, \pi_3 x, \pi_4 x, \pi_5 x, \pi_6 x, q_{err} \rangle$ // default value

$: \blacksquare\mathbf{C} \to \mathbf{C}$

Obviously, the definition of this function strictly depends on how is made the function δ of our Turing machine. In the previous description of δ function we introduce some comments in order to make more readable the function.

Example 3.3 Suppose that our deterministic Turing machine has a δ function such that $\delta(q_i, 1)$ gives $(q_j, 0, -1)$.

So, our $\underline{\delta}$ will be encoded in this way:

$\underline{\delta} \equiv \lambda x : \blacksquare\mathbf{C}.$

$\mathsf{switch}_{\mathbf{C}}^{\{0,\ldots,3\}}\mathbf{BtoS}(\pi_2)$

... // three cases ahead

$\mathsf{switch}_{\mathbf{C}}^{\{0,\ldots,n\}}(\pi_4 x)$

... // k cases ahead

$$\langle \pi_1 x, 1, \pi_3 x, \underline{q_k} \rangle$$
$$\dots$$
$$\langle \pi_1 x, \pi_2 x, \pi_3 x, \pi_4 x, \pi_5 x, \pi_6 x, q_{err} \rangle$$
$$\langle \pi_1 x, \pi_2 x, \pi_3 x, \pi_4 x, \pi_5 x, \pi_6 x, q_{err} \rangle$$

□

Encoding the whole program

Finally we will encode a function result that takes a configuration and gives out 1 or 0 if the configuration is in an accepting state.

$$\begin{aligned}\mathsf{result} \equiv & \lambda x : \blacksquare\mathbf{C}.\mathbf{if}_{\mathbf{N}}\,(\pi_7 x = \underline{q_i}) \\ & \textbf{then}\ \text{write 0 if } \underline{q_i} \text{ is an accepting state, 1 otherwise} \\ & \textbf{else}\ \text{consider all the other cases} \\ & \textbf{default}\,\underline{0} : \blacksquare\mathbf{C} \to \mathbf{N}\end{aligned}$$

In the end we will encode our Turing machine in the following way:

$$\begin{aligned}\underline{M} \equiv \lambda x : \square\mathbf{N}.\mathsf{result}\,(&\mathtt{recursion}_{\mathbf{C}} \\ & (\underline{p}(\mathsf{encode}(x))) \\ & (\mathsf{init}\,x) \\ & \underline{\delta}) : \square\mathbf{N} \to \mathbf{N}\end{aligned}$$

Chapter 4

A Higher-Order Characterization of Probabilistic Polynomial Time

In this chapter we are going to present the probabilistic extension of our SLR (presented in the section 3.4) called RSLR. Probabilistic polynomial time computations, seen as oracle computations, were showed to be amenable to implicit techniques since the early days of ICC, by a relativization of Bellantoni and Cook's safe recursion [5]. They were then studied again in the context of formal systems for security, where probabilistic polynomial time computation plays a major role [23, 40]. These two systems are built on Hofmann's work SLR [21], by adding a random choice operator to the calculus. The system in [23], however, lacks higher-order recursion, and in both papers the characterization of the probabilistic classes is obtained by semantic means. While this is fine for completeness, we think it is not completely satisfactory for soundness — we know from the semantics that for any term of a suitable type its normal form *may be* computed within the given bounds, but no notion of evaluation is given for which computation time is *guaranteed* to be bounded.

4.1 Related Works

We discuss here in more details the relations of RSLR system to the previous work SLR we already cited.

More than ten years ago, Mitchell, Mitchell, and Scedrov [23] introduced OSLR, a type system that characterizes oracle polynomial time functionals. Even if inspired by SLR, OSLR does not admit primitive recursion on higher-order types, but only on base types. The main theorem shows that terms of type $\Box\mathbf{N}^m \to \mathbf{N}^n \to \mathbf{N}$ define precisely the *oracle*

polynomial time functionals, which constitute a class related but different from the ones we are interested in here. Finally, inclusion in the polynomial time class is proved without studying reduction from an operational viewpoint, but only via semantics: it is not clear for *which* notion of evaluation, computation time is guaranteed to be bounded.

Recently, Zhang's [40] introduced a further system (CSLR) which builds on OSLR and allows higher-order recursion. The main interest of the paper are applications to the verification of security protocols. It is stated that CSLR defines exactly those functions that can be computed by probabilistic Turing machines in polynomial time, via a suitable variation of Hofmann's techniques as modified by Mitchell et al. This is again a purely semantic proof, whose details are missing in [40].

Finally, both works are derived from Hofmann's one, and as a consequence they both have potential problems with subject reduction. Indeed, as Hofmann showed in his work [21], subject reduction does not hold in SLR, and hence is problematic in both OSLR and CSLR.

4.2 RSLR: An Informal Account

Many things are similar to the one presented in the previous section. We are using same restrictions already presented in the section 3.4. By adding probabilities to reduction steps we need to prove more theorems to insure confluence of possible terms in output.

We extend the grammar with a new constant called `rand`. Once evaluated, this new constant gives 1 or 0 with the same probability $\frac{1}{2}$.

4.3 On the Difficulty of Probabilistic ICC

Differently from most well known complexity classes such as **P**, **NP** and **L**, the probabilistic hierarchy contains so-called "semantic classes", like **BPP** and **ZPP**. A semantic class is a complexity class defined on top of a class of algorithms which cannot be easily enumerated: a probabilistic polynomial time Turing machine does not *necessarily* solve a problem in **BPP** nor in **ZPP**. For most semantic classes, including **BPP** and **ZPP**, the existence of complete problems and the possibility to prove hierarchy theorems are both open. Indeed, researchers in the area have proved the existence of such results for other probabilistic classes, but not for those we are interested and given in [15].

Now, having a "truly implicit" system I for a complexity class C means that we have a way to enumerate a set of programs solving problems in C (for every problem there is at least one program that solves it). The presence or absence of complete problems is deeply linked with the possibility to have a real ICC system for these semantic classes. In our case the "semantic information" in **BPP** and **ZPP**, that is the probability error, seems to be an information that is impossible to capture with syntactical restrictions. We need to execute the program in order to check if the error bound is correct or not.

4.4 The Syntax and Basic Properties of RSLR

Also RSLR is a fairly standard Curry-style lambda calculus. We have constants for the natural numbers, branching and recursion. As SLR presented in 3.4, its type system takes ideas from linear logic. Indeed, some variables can appear at most once in a term.

Definition 4.1 (Types) *The types of* RSLR *are exactly the ones presented in definition 3.1. We still have* $\mathbf{N}$ *as the only* base type *and we still have arrow types. All of these have the same meaning we have already explained.*

In RSLR we have again the notion of subtyping as explained before. In such way we are able to say that the type $\blacksquare A \to B$ is a subtype of $\square A \to B$.

Definition 4.2 (Aspects) *An* aspect *is either* $\square$ *or* $\blacksquare$*: the first is the* modal *aspect, while the second is the* non modal *one. Aspects are partially ordered by the binary relation* $\{(\square,\square),(\square,\blacksquare),(\blacksquare,\blacksquare)\}$*, noted* $<:$.

Defining subtyping, then, merely consists in generalizing $<:$ to a partial order on types in which only structurally identical types can be compared. Subtyping rules are in Figure 4.1. Please observe that (S-Sub) is contravariant in the aspect a.

$$\frac{}{A <: A}\ (\text{S-Refl}) \qquad \frac{A <: B \qquad B <: C}{A <: C}\ (\text{S-Trans})$$

$$\frac{B <: A \qquad C <: D \qquad b <: a}{aA \to C <: bB \to D}\ (\text{S-Sub})$$

Figure 4.1: Subtyping rules.

RSLR's terms are those of an applied lambda calculus with primitive recursion and branching, in the style of Gödel's T:

Definition 4.3 (Terms) *Terms and constants are defined as follows:*

$$t ::= x \mid c \mid ts \mid \lambda x : aA.t \mid \mathtt{case}_A\ t\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q \mid \mathtt{recursion}_A\ t\, s\, r;$$

$$c ::= n \mid \mathtt{S}_0 \mid \mathtt{S}_1 \mid \mathtt{P} \mid \mathtt{rand}.$$

Here, x ranges over a denumerable set of variables and n ranges over the natural numbers seen as constants of base type. Every constant c has its naturally defined type, that we indicate with $type(c)$. Formally, $type(n) = \mathbf{N}$ for every n, $type(\mathtt{rand}) = \mathbf{N}$, while $type(\mathtt{S}_0) = type(\mathtt{S}_1) = type(\mathtt{P}) = \blacksquare\mathbf{N} \to \mathbf{N}$. The size $|t|$ of any term t can be easily defined by induction on t (where, by convention, we stipulate that $\log_2(0) = 0$):

$$\begin{aligned}
|x| &= 1;\\
|ts| &= |t| + |s|;\\
|\lambda x : aA.t| &= |t| + 1;\\
|\mathtt{case}_A\ t\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q| &= |t| + |s| + |r| + |q| + 1;\\
|\mathtt{recursion}_A\ t\, s\, r| &= |t| + |s| + |r| + 1;\\
|n| &= \lfloor \log_2(n) \rfloor + 1;\\
|\mathtt{S}_0| = |\mathtt{S}_1| = |\mathtt{P}| = |\mathtt{rand}| &= 1.
\end{aligned}$$

Notice that the size of n is exactly the length of the number n in binary representation. Size of 5, as an example, is $\lfloor \log_2(5) \rfloor + 1 = 3$, while 0 only requires one binary digit to be represented, and its size is thus 1. As usual, terms are considered modulo α-conversion. Free (occurrences of) variables and capture-avoiding substitution can be defined in a standard way.

Definition 4.4 (Explicit term) *A term is said to be* explicit *if it does not contain any instance of* `recursion`.

The main peculiarity of RSLR with respect to similar calculi is the presence of a constant for random, binary choice, called `rand`, which evolves to either 0 or 1 with probability $\frac{1}{2}$. Although the calculus is in Curry-style, variables are explicitly assigned a type and an aspect in abstractions. This is for technical reasons that will become apparent soon.

Note 4.5 *The presence of terms which can (probabilistically) evolve in different ways makes it harder to define a confluent notion of reduction for* RSLR*. To see why, consider a term like*

$$t = (\lambda x : \blacksquare\mathbf{N}.(t_{\oplus} x x))\texttt{rand}$$

*where $t_{\oplus}$ is a term computing $\oplus$ on natural numbers seen as booleans (*0 *stands for "false" and everything else stands for "true"):*

$$t_{\oplus} = \lambda x : \blacksquare\mathbf{N}.\texttt{case}_{\blacksquare\mathbf{N}\to\mathbf{N}}\ x\ \texttt{zero}\ s_{\oplus}\ \texttt{even}\ r_{\oplus}\ \texttt{odd}\ r_{\oplus};$$

$$s_{\oplus} = \lambda y : \blacksquare\mathbf{N}.\texttt{case}_{\mathbf{N}}\ y\ \texttt{zero}\ 0\ \texttt{even}\ 1\ \texttt{odd}\ 1;$$

$$r_{\oplus} = \lambda y : \blacksquare\mathbf{N}.\texttt{case}_{\mathbf{N}}\ y\ \texttt{zero}\ 1\ \texttt{even}\ 0\ \texttt{odd}\ 0.$$

If we evaluate t in a call-by-value fashion, `rand` *will be fired* before *being passed to $t_{\oplus}$ and, as a consequence, the latter will be fed with two identical natural numbers, returning* 0 *with probability* 1*. If, on the other hand,* `rand` *is passed unevaluated to $t_{\oplus}$, the four possible combinations on the truth table for $\oplus$ will appear with equal probabilities and the outcome will be* 0 *or* 1 *with probability $\frac{1}{2}$. In other words, we need to somehow restrict our notion of reduction if we want it to be consistent, i.e. confluent.*

For the just explained reasons, arguments are passed to functions following a mixed scheme in RSLR*: arguments of base type are evaluated before being passed to functions, while arguments of an higher-order type are passed to functions possibly unevaluated, in a call-by-name fashion.*

In our system higher-order terms cannot be duplicated and this guarantees that if a term duplicates then it has no `rand` *inside. The counterexample no more longer works.*

Let's first of all define the one-step reduction relation:

Definition 4.6 (Reduction) *The* one-step reduction relation $\to$ *is a binary relation between terms and sequences of terms. It is defined by the axioms in Figure 4.2 and can be applied in any contexts, except in the second and third argument of a recursion. A term t is in* normal form *if t cannot appear as the left-hand side of a pair in $\to$. NF is the set of terms in normal form.*

Notice the little but significant difference between rules in figure 4.2 and rules of SLR presented in figure 3.2. On the right side of the arrow now we can have more than one

$$
\begin{aligned}
\mathtt{case}_A\ 0\ \mathtt{zero}\ t\ \mathtt{even}\ s\ \mathtt{odd}\ r &\to t;\\
\mathtt{case}_A\ (\mathtt{S}_0 n)\ \mathtt{zero}\ t\ \mathtt{even}\ s\ \mathtt{odd}\ r &\to s;\\
\mathtt{case}_A\ (\mathtt{S}_1 n)\ \mathtt{zero}\ t\ \mathtt{even}\ s\ \mathtt{odd}\ r &\to r;\\
\mathtt{recursion}_A\ 0\, g\, f &\to g;\\
\mathtt{recursion}_A\ n\, g\, f &\to f n(\mathtt{recursion}_\tau\ \lfloor \frac{n}{2} \rfloor\, g\, f);\\
\mathtt{S}_0 n &\to 2 \cdot n;\\
\mathtt{S}_1 n &\to 2 \cdot n + 1;\\
\mathtt{P}0 &\to 0;\\
\mathtt{P}n &\to \lfloor \frac{n}{2} \rfloor;\\
(\lambda x : a\mathbf{N}.t)n &\to t[x/n];\\
(\lambda x : aH.t)s &\to t[x/s];\\
(\lambda x : aA.t)sr &\to (\lambda x : aA.tr)s;\\
\mathtt{rand} &\to 0, 1;
\end{aligned}
$$

Figure 4.2: One-step reduction rules.

term. Informally, $t \to s_1, \ldots, s_n$ means that t can evolve in one-step to each of $s_1, \ldots, s_n$ with the same probability $\frac{1}{n}$. As a matter of fact, n can be either 1 or 2.

A multistep reduction relation will not be defined by simply taking the transitive and reflective closure of $\to$, since a term can reduce in multiple steps to many terms with different probabilities. Multistep reduction puts in relation a term t to a probability distribution on terms $\mathscr{D}_t$ such that $\mathscr{D}_t(s) > 0$ only if s is a normal form to which t reduces. Of course, if t is itself a normal form, $\mathscr{D}_t$ is well defined, since the only normal form to which t reduces is t itself, so $\mathscr{D}_t(t) = 1$. But what happens when t is *not* in normal form? Is $\mathscr{D}_t$ a well-defined concept? Let us start by formally defining $\rightsquigarrow$:

Definition 4.7 (Multistep Reduction) *The binary relation $\rightsquigarrow$ between terms and probability distributions is defined by the rules in Figure 4.3.*

$$\frac{t \to t_1, \ldots, t_n \qquad t_i \rightsquigarrow \mathscr{D}_i}{t \rightsquigarrow \sum_{i=1}^{n} \frac{1}{n} \mathscr{D}_i} \qquad \frac{t \in NF}{t \rightsquigarrow \mathscr{D}_t}$$

Figure 4.3: Multistep Reduction: Inference Rules

In Section 4.6, we will prove that for every t there is at most one $\mathscr{D}$ such that $t \rightsquigarrow \mathscr{D}$. We are finally able to present the type system. Preliminary to that is the definition of a proper notion of a context.

Definition 4.8 (Contexts) *A* context Γ *is a finite set of assignments of types and aspects to variables, i.e., of expressions in the form $x : aA$. As usual, we require contexts not to contain assignments of distinct types and aspects to the same variable. The union of two disjoint contexts Γ and Δ is denoted as Γ, Δ. In doing so, we implicitly assume that the variables in Γ and Δ are pairwise distinct. The expression $\Gamma; \Delta$ denotes the union Γ, Δ, but is only defined when all types appearing in Γ are base types. As an example, it is perfectly legitimate to write $x : a\mathbf{N}; y : b\mathbf{N}$, while the following is an ill-defined expression:*

$$x : a(b\mathbf{N} \to \mathbf{N}); y : c\mathbf{N},$$

the problem being the first assignment, which appears on the left of ";" but which assigns the higher-order *type $b\mathbf{N} \to \mathbf{N}$ (and the aspect a) to x. This notation is particularly helpful*

when giving typing rules. With the expression $\Gamma <: a$ *we mean that any aspect* b *appearing in* Γ *is such that* $b <: a$.

Typing rules are in Figure 4.4. Observe how rules with more than one premise are designed

$$\frac{x : aA \in \Gamma}{\Gamma \vdash x : A}\ (\text{T-Var-Aff}) \qquad \frac{\Gamma \vdash t : A \qquad A <: B}{\Gamma \vdash t : B}\ (\text{T-Sub})$$

$$\frac{\Gamma, x : aA \vdash t : B}{\Gamma \vdash \lambda x : aA.t : aA \to B}\ (\text{T-Arr-I}) \qquad \frac{}{\Gamma \vdash c : \mathit{type}(c)}\ (\text{T-Const-Aff})$$

$$\frac{\begin{array}{cc}\Gamma;\Delta_1 \vdash t : \mathbf{N} & \Gamma;\Delta_3 \vdash r : A \\ \Gamma;\Delta_2 \vdash s : A & \Gamma;\Delta_4 \vdash q : A \qquad A \text{ is } \Box\text{-free}\end{array}}{\Gamma;\Delta_1,\Delta_2,\Delta_3,\Delta_4 \vdash \mathtt{case}_A\ t\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q : A}\ (\text{T-Case})$$

$$\frac{\begin{array}{cc}\Gamma_1;\Delta_1 \vdash t : \mathbf{N} & \\ \Gamma_1,\Gamma_2;\Delta_2 \vdash s : A & \Gamma_1;\Delta_1 <: \Box \\ \Gamma_1,\Gamma_2;\vdash r : \Box\mathbf{N} \to \blacksquare A \to A & A \text{ is } \Box\text{-free}\end{array}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2 \vdash \mathtt{recursion}_A\ t\, s\, r : A}\ (\text{T-Rec})$$

$$\frac{\Gamma;\Delta_1 \vdash t : aA \to B \qquad \Gamma;\Delta_2 \vdash s : A \qquad \Gamma,\Delta_2 <: a}{\Gamma;\Delta_1,\Delta_2 \vdash (ts) : B}\ (\text{T-Arr-E})$$

Figure 4.4: Type rules

in such a way as to guarantee that whenever $\Gamma \vdash t : A$ can be derived and $x : aH$ is in Γ, then x can appear free at most once in t. If $y : a\mathbf{N}$ is in Γ, on the other hand, then y can appear free in t an arbitrary number of times.

Definition 4.9 *A* first-order term *of arity* k *is a closed, well typed term of type* $a_1\mathbf{N} \to a_2\mathbf{N} \to \ldots a_k\mathbf{N} \to \mathbf{N}$ *for some* $a_1, \ldots, a_k$.

4.5 Subject Reduction

Also RSLR preserves the types under reduction. The so called Subject Reduction Theorem still holds. We will not re-do all the proofs already done in section 3.4.2, we repropose here the main statement.

Theorem 4.1 (Subject Reduction) *Suppose that* $\Gamma \vdash t : A$. *If* $t \to t_1 \dots t_j$, *then for every* $i \in \{1, \dots, j\}$, *it holds that* $\Gamma \vdash t_i : A$.

Proof: Proof of this theorem is similar to one of theorem 3.3, even if the definition of reduction step is different. □

4.6 Confluence

In view of the peculiar notion of reduction given in Definition 4.6, let us go back to the counterexample to confluence given in the Introduction. The term $t = (\lambda x : \blacksquare\mathbf{N}.(t_{\oplus} x x))\mathtt{rand}$ cannot be reduced to $t_{\oplus}\,\mathtt{rand}\,\mathtt{rand}$ anymore, because only numerals can be passed to functions as arguments of base types. The only possibility is reducing t to the sequence

$$(\lambda x : \blacksquare\mathbf{N}.(t_{\oplus} x x))0, (\lambda x : \blacksquare\mathbf{N}.(t_{\oplus} x x))1$$

Both terms in the sequence can be further reduced to 0. In other words, $t \rightsquigarrow \{0^1\}$.

More generally, the phenomenon of non-convergence of final distributions can no longer happen in RSLR. Technically, this is due to the impossibility of duplicating terms that can evolve in a probabilistically nontrivial way, i.e., terms containing occurrences of `rand`. In the above example and in similar cases we have to evaluate the argument before firing the β-redex — it is therefore not possible to obtain two different distributions. RSLR can also handle correctly the case where `rand` is within an argument t of higher-order type: terms of higher-order type cannot be duplicated and so neither any occurrences of `rand` inside them.

Confluence of our system is proved by first showing a kind of confluence for the single step arrow; then we show the confluence for the multistep arrow. This allows us to certify the confluence of our system.

Lemma 4.1 *Let* t *be a well typed term in* RSLR*; if* $t \to v$ *and* $t \to z$ *(*v *and* z *distinct) then exactly one of the following holds:*

- $\exists a$ *s.t.* $v \to a$ *and* $z \to a$
- $v \to z$
- $z \to v$

Proof: By induction on the structure of the typing derivation for the term t.

- If t is a constant or a variable, the theorem is easily proved. The premise is always false, so the theorem is always valid. Remember that $\texttt{rand} \to 0, 1$.
- If last rule was T-Sub or T-Arr-I, by applying induction hypothesis the case is easily proved.
- If last rule was T-Case. Our derivation will have the following shape:

$$\frac{\begin{array}{ccc}\Gamma;\Delta_1 \vdash s : N & \Gamma;\Delta_3 \vdash q : A & \\ \Gamma;\Delta_2 \vdash r : A & \Gamma;\Delta_4 \vdash u : A & A \text{ is } \Box\text{-free}\end{array}}{\Gamma;\Delta_1,\Delta_2,\Delta_3,\Delta_4 \vdash \texttt{case}_A\ s\ \texttt{zero}\ r\ \texttt{even}\ q\ \texttt{odd}\ u : A}\ (\text{T-Case})$$

  We could have reduced one of the following s, r, q, u terms or a combination of them. In the first case we prove by applying induction hypothesis and in the latter case we can easily find a s.t. $v \to a$ and $z \to a$: is the term where we apply both reductions. Last case is where from one part we reduce the case, selecting a branch and from the other part we reduce one of the subterms. As can be easily seen, it is trivial to prove this case; we can easily find a common confluent term.
- If last rule was T-Rec, our derivation will have the following shape:

$$\frac{\begin{array}{ccc}\Gamma_2;\Delta_4 \vdash q : \mathbf{N} & & \\ \Gamma_2,\Gamma_3;\Delta_5 \vdash s : B & \Gamma_2;\Delta_4 <: \Box & \\ \Gamma_2,\Gamma_3;\vdash r : \Box\mathbf{N} \to \blacksquare B \to B & B \text{ is } \Box\text{-free} & \end{array}}{\Gamma_2,\Gamma_3;\Delta_4,\Delta_5 \vdash \texttt{recursion}_B\ q\, s\, r : B}\ (\text{T-Rec})$$

  By definition, we can have reduction only on q or, if q is a value, we can reduce the recursion by unrolling it. In both cases the proof is trivial.
- If last rule was T-Arr-E. Our term could have different shapes but the only interesting cases are the following ones. The other cases can be easily brought back to cases that we have considered.
  - Our derivation will end in the following way:

$$\frac{\Gamma;\Delta_1 \vdash \lambda x : aA.r : bC \to B \qquad \Gamma;\Delta_2 \vdash s : C \qquad \Gamma,\Delta_2 <: b}{\Gamma,\Delta_1,\Delta_2 \vdash (\lambda x : aA.r)s : B}\ (\text{T-Arr-E})$$

    where $C <: A$ and $b <: a$. We have that $(\lambda x : aA.r)s$ rewrites in $r[x/s]$; if $A \equiv \mathbf{N}$ then s is a value, otherwise we are able to make the substitution whenever we want. If we reduce only on s or only on r we can easily prove our thesis by applying induction hypothesis.

    The interesting cases are when we perform the substitution on one hand and on the other hand we make a reduction step on one of the two possible terms s or r.

Suppose $(\lambda x : aA.r)s \to r[x/s]$ and $(\lambda x : aA.r)s \to (\lambda x : aA.r)s'$, where $s \to s'$. Let a be $r[x/s']$. We have that $(\lambda x : aA.r)s' \to a$ and $r[x/s] \to a$. Indeed if A is $\mathbf{N}$, s is a value (we are making substitutions) but no reduction could be made on s, otherwise there is at least one occurrence of s in $r[x/s]$ and by executing one reduction step we are able to have a.

Suppose $(\lambda x : aA.r)s \to r[x/s]$ and $(\lambda x : aA.r)s \to (\lambda x : aA.r')s$, where $r \to r'$. As we have shown in the previous case, we are able to find a confluent term for both terms.

- The other interesting case is when we perform the so called "swap". $(\lambda x : aA.q)sr$ rewrites in $(\lambda x : aA.qr)s$. If the reduction steps are made only on q or s or r by applying induction hypothesis we have the thesis. In all the other cases, where we perform one step on subterms and we perform, on the other hand, the swap, it's easy to find a confluent term a.

□

Lemma 4.2 *Let t be a well typed term in* RSLR*; if $t \to v_1, v_2$ and $t \to z$ then one of the following statement is valid:*

- $\exists a_1, a_2$ *s.t.* $v_1 \to a_1$ *and* $v_2 \to a_2$ *and* $z \to a_1, a_2$
- $\forall i. v_i \to z$
- $z \to a_1, a_2$

Proof: By induction on the structure of typing derivation for the term t.

- t cannot be a constant or a variable. Indeed if t is `rand`, rand reduces in $0, 1$ and this differs from our hypothesis.
- If last rule was T-Sub or T-Arr-I, the thesis is easily proved by applying induction hypothesis.
- If last rule was T-Case, our derivation will have the following shape:

$$\frac{\begin{array}{ccc}\Gamma;\Delta_1 \vdash s : N & \Gamma;\Delta_3 \vdash q : A & \\ \Gamma;\Delta_2 \vdash r : A & \Gamma;\Delta_4 \vdash u : A & A \text{ is } \Box\text{-free}\end{array}}{\Gamma;\Delta_1,\Delta_2,\Delta_3,\Delta_4 \vdash \mathtt{case}_A\ s\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u : A}\ (\text{T-Case})$$

If we perform the two reductions on the single subterms we could be in the following case (all the other cases are similar). for example, if t rewrites in $\mathtt{case}_A\ s'\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u$ and $\mathtt{case}_A\ s''\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u$ and also $t \to \mathtt{case}_A\ s\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u'$.

It is easy to check that if the two confluent terms are $a_1 = \mathtt{case}_A\ s'\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u'$ and $a_2 = \mathtt{case}_A\ s''\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u'$ the thesis is valid.

Another possible case is where on one hand we perform a reduction by selecting a branch and on the other case we make a reduction on one branch. As example, $t \to q$ and $r \to r_1, r_2$. This case is trivial.

- If last rule was T-Rec, our derivation will have the following shape:

$$\frac{\begin{array}{ccc}\Gamma_2;\Delta_4 \vdash q : \mathbf{N} & & \\ \Gamma_2,\Gamma_3;\Delta_5 \vdash s : B & & \Gamma_2;\Delta_4 <: \Box \\ \Gamma_2,\Gamma_3;\vdash r : \Box\mathbf{N} \to \blacksquare B \to B & & B \text{ is } \Box\text{-free}\end{array}}{\Gamma_2,\Gamma_3;\Delta_4,\Delta_5 \vdash \mathtt{recursion}_B\ q\ s\ r : B}\ (\text{T-Rec})$$

By definition, we can have reduction only on q. By applying induction hypothesis the thesis is proved.

- If last rule was T-Arr-E. Our term could have different shapes but the only interesting cases are the following ones. The other cases can be easily brought back to cases that we have considered.
  - Our derivation will end in the following way:

$$\frac{\Gamma;\Delta_1 \vdash \lambda x : aA.r : bC \to B \qquad \Gamma;\Delta_2 \vdash s : C \qquad \Gamma,\Delta_2 <: b}{\Gamma,\Delta_1,\Delta_2 \vdash (\lambda x : aA.r)s : B}\ (\text{T-Arr-E})$$

    where $C <: A$ and $b <: a$. We have that $(\lambda x : aA.r)s$ rewrites in $r[x/s]$; if $A \equiv \mathbf{N}$ then s is a value, otherwise we are able to make the substitution whenever we want. If we reduce only on s or only on r we can easily prove our thesis by applying induction hypothesis.

    The interesting cases are when we perform the substitution on one hand and on the other hand we make a reduction step on one of the two possible terms s or r.

    Suppose $(\lambda x : aA.r)s \to r[x/s]$ and $(\lambda x : aA.r)s \to (\lambda x : aA.r)s', (\lambda x : aA.r)s''$, where $s \to s', s''$. Let a_1 be $r[x/s']$ and a_2 be $r[x/s'']$.

    We have that $(\lambda x : aA.r)s' \to a_1$, $(\lambda x : aA.r)s'' \to a_2$ and $r[x/s] \to a_1, a_2$. Indeed if A is $\mathbf{N}$ then s is a value (because we are making substitutions) and we cannot have the reductions on s, otherwise there is at least one occurrence of s in $r[x/s]$ and by performing one reduction step on the subterm s we are able to have a_1, a_2.

    Suppose $(\lambda x : aA.r)s \to r[x/s]$ and $(\lambda x : aA.r)s \to (\lambda x : aA.r')s, (\lambda x : aA.r'')s$, where $r \to r', r''$. As we have shown in the previous case, we are able to find a confluent term for both terms.

- The other interesting case is when we perform the so called "swap". $(\lambda x : aA.q)sr$ rewrites in $(\lambda x : aA.qr)s$. If the reduction steps are made only on q or s or r by applying induction hypothesis we have the thesis. In all the other cases, where we perform one step on subterms and we perform, on the other hand, the swap, it's easy to find a confluent term a.

□

Lemma 4.3 *Let t be a well typed term in* RSLR*; if $t \to v_1, v_2$ and $t \to z_1, z_2$ (v_1, v_2 and z_1, z_2 different) then $\exists a_1, a_2, a_3, a_4$ s.t. $v_1 \to a_1, a_2$ and $v_2 \to a_3, a_4$ and $\exists i. z_i \to a_1, a_3$ and $z_{1-i} \to a_2, a_4$.*

Proof: By induction on the structure of typing derivation for term t.

- If t is a variable or a constant the thesis is trivial.
- If last rule was (T-Sub) or (T-Arr-I) the thesis is trivial, by applying induction hypothesis.
- If last rule was (T-Case) our derivation will have the following shape:

$$\frac{\begin{array}{ccc}\Gamma;\Delta_1 \vdash s : N & \Gamma;\Delta_3 \vdash q : A & \\ \Gamma;\Delta_2 \vdash r : A & \Gamma;\Delta_4 \vdash u : A & A \text{ is } \Box\text{-free}\end{array}}{\Gamma;\Delta_1,\Delta_2,\Delta_3,\Delta_4 \vdash \mathtt{case}_A\ s\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u : A}\ (\text{T-Case})$$

  Also this case is easy to prove. Indeed if the reduction steps are made only on single subterms: s or r or q or u we can prove by using induction hypothesis. Otherwise we are in the case where one reduction step is made on some subterm and the other is made considering a different subterm. Suppose $s \to s', s''$ and $q \to q', q''$. We could have two possible reduction. One is $t \to \mathtt{case}_A\ s'\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u, \mathtt{case}_A\ s''\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u$ and the other is $t \to \mathtt{case}_A\ s\ \mathtt{zero}\ r\ \mathtt{even}\ q'\ \mathtt{odd}\ u, \mathtt{case}_A\ s\ \mathtt{zero}\ r\ \mathtt{even}\ q''\ \mathtt{odd}\ u$.
  It is easy to find the common confluent terms: are the ones in which we have performed both $s \to s', s''$ and $q \to q', q''$.
- If last rule was (T-Rec) our derivation will have the following shape:

$$\frac{\begin{array}{ccc}\Gamma_2;\Delta_4 \vdash q : \mathbf{N} & & \\ \Gamma_2,\Gamma_3;\Delta_5 \vdash s : B & \Gamma_2;\Delta_4 <: \Box & \\ \Gamma_2,\Gamma_3;\vdash r : \Box\mathbf{N} \to \blacksquare B \to B & B \text{ is } \Box\text{-free} & \end{array}}{\Gamma_2,\Gamma_3;\Delta_4,\Delta_5 \vdash \mathtt{recursion}_B\ q\ s\ r : B}\ (\text{T-Rec})$$

  By definition, we can have reduction only on q. By applying induction hypothesis the thesis is proved.

- If last rule was (T-ARR-E). Our term could have different shapes but all of them are trivial or can be easily brought back to cases that we have considered. Also the case where we consider the so called "swap" and the usual application with a lambda abstraction are not interesting in this lemma. Indeed, we cannot consider the "swap" or the substitution case because the reduction relation gives only one term on the right side of the arrow $\rightarrow$.

□

It is not trivial to prove confluence for $\rightsquigarrow$. For this purpose we will prove our statement on a different definition of multistep arrow. This new definition is laxer than the standard one. Being able to prove our theorems for this new definition, allows us to conclude that these theorems hold also for $\rightsquigarrow$.

Definition 4.10 *In order to prove the following statements we define a new multistep reduction arrow $\Rightarrow$ as in Figure 4.5. As usual, $\mathscr{G}_t$ is the distribution that associates to the*

$$\frac{t \rightarrow t_1, \ldots, t_n \qquad t_i \Rightarrow \mathscr{D}_i}{t \Rightarrow \sum_{i=1}^{n} \frac{1}{n} \mathscr{D}_i} \qquad \frac{}{t \Rightarrow \mathscr{G}_t}$$

Figure 4.5: New Multistep Reduction: Inference Rules

term t probability 1. With this relation, distribution are functions $\mathscr{D} : \Lambda \rightarrow [0, 1]$. It is easy to check that if $t \rightsquigarrow \mathscr{D}$ then $t \Rightarrow \mathscr{D}$ (but not vice-versa).

Definition 4.11 (Size of distribution derivation) *We define the size of a derivation $t \Rightarrow \mathscr{D}$, written $|t \Rightarrow \mathscr{D}|$, in an inductive way. If the last rule was the axiom, $|t \Rightarrow \mathscr{G}_t| = 0$; otherwise, $|t \Rightarrow \sum_{i=1}^{n} \frac{1}{n} \mathscr{D}_i| = \max_i |t_i \Rightarrow \mathscr{D}_i| + 1$.*

Lemma 4.4 *If $t \Rightarrow \mathscr{D}$, be $\mathscr{D} \equiv \{M_1^{\alpha_1}, \ldots, M_n^{\alpha_n}\}$, and if for all i $M_i \Rightarrow \mathscr{E}_i$ then $t \Rightarrow \sum_i \alpha_i \mathscr{E}_i$ and $|t \Rightarrow \sum_i \alpha_i \mathscr{E}_i| \leq |t \Rightarrow \mathscr{D}| + \max_i |M_i \Rightarrow \mathscr{E}_i|$.*

Proof: By induction on the structure of the derivation for $t \Rightarrow \mathscr{D}$.

- If last rule was the axiom, then $t \Rightarrow \mathscr{G}_t$. Suppose $t \Rightarrow \mathscr{E}$. The thesis is easily proved.
- The derivation finishes with the following rule:

$$\frac{t \rightarrow t_1, \ldots, t_n \qquad t_i \Rightarrow \mathscr{D}_i}{t \Rightarrow \sum_{i=1}^{n} \frac{1}{n} \mathscr{D}_i}$$

Let's analyze all the possible cases, depending on the value n.

- If $n \equiv 1$.

$$\frac{t \to t_1 \qquad t_1 \Rightarrow \mathscr{D}}{t \Rightarrow \mathscr{D}}$$

By using induction hypothesis on the premise, we prove our thesis.

- If $n \equiv 2$.

$$\frac{t \to t_1, t_2 \qquad t_1 \Rightarrow \mathscr{D}_1 \qquad t_2 \Rightarrow \mathscr{D}_2}{t \Rightarrow \frac{1}{2}(\mathscr{D}_1 + \mathscr{D}_2)}$$

Be $\mathscr{D} \equiv \{M_1^{\alpha_1}, \ldots, M_n^{\alpha_n}\}$ and for all i $M_i \Rightarrow \mathscr{E}_i$. By construction, we have some elements that belong to $\mathscr{D}_1$, other to $\mathscr{D}_2$ and some element that belongs to both of them. Without loosing generality, let's say that elements $M_1, \ldots, M_m$ belongs to $\mathscr{D}_1$ and elements $M_o, \ldots, M_n$, where $1 \leq o \leq m \leq n$.
So, we have that $\mathscr{D}_1 \equiv \{M_1^{2\alpha_1}, \ldots, M_{o-1}^{2\alpha_{o-1}}, M_o^{\alpha_o}, \ldots, M_m^{\alpha_m}\}$ and we have that $\mathscr{D}_2$ is $\{M_o^{\alpha_o}, \ldots, M_m^{\alpha_m}, M_{m+1}^{2\alpha_m}, \ldots, M_n^{2\alpha_n}\}$.
By applying induction hypothesis on the two premises we have that $t_1 \Rightarrow \mathscr{P}_1$ and $t_2 \Rightarrow \mathscr{P}_2$, where $\mathscr{P}_1 \equiv \sum_{i=1}^{m-1} 2\alpha_i \mathscr{E}_i + \sum_{i=m}^{o} \alpha_i \mathscr{E}_i$ and $\mathscr{P}_2 \equiv \sum_{i=m}^{o} \alpha_i \mathscr{E}_i + \sum_{i=o+1}^{n} 2\alpha_i \mathscr{E}_i$
So, we can derive that $t \Rightarrow \frac{1}{2}(\mathscr{P}_1 + \mathscr{P}_2)$ that is our thesis.
Concerning the bound on the derivation, the induction hypothesis applied to the premises gives us $|t_1 \Rightarrow \mathscr{P}_1| \leq |t_1 \Rightarrow \mathscr{D}_1| + \max_{0,\ldots,m} |M_i \Rightarrow \mathscr{E}_i|$ and $|t_2 \Rightarrow \mathscr{P}_2| \leq |t_2 \Rightarrow \mathscr{D}_2| + \max_{o,\ldots,n} |M_i \Rightarrow \mathscr{E}_i|$. We have:

$$\begin{aligned}
|t \Rightarrow \sum_i \alpha_i \mathscr{E}_i| &\equiv \max\{\mathscr{P}_1, \mathscr{P}_2\} + 1 \\
&\leq \max\{|t_1 \Rightarrow \mathscr{D}_1| + \max_{0,\ldots,m} |M_i \Rightarrow \mathscr{E}_i|, |t_2 \Rightarrow \mathscr{D}_2| + \max_{o,\ldots,n} |M_i \Rightarrow \mathscr{E}_i|\} + 1 \\
&\leq \max\{|t_1 \Rightarrow \mathscr{D}_1|, |t_2 \Rightarrow \mathscr{D}_2|\} + 1 + \max\{\max_{o,\ldots,n} |M_i \Rightarrow \mathscr{E}_i|, \max_{0,\ldots,m} |M_i \Rightarrow \mathscr{E}_i|\} \\
&\leq |t \Rightarrow \mathscr{D}| + \max_i |M_i \Rightarrow \mathscr{E}_i|
\end{aligned}$$

and the lemma is proved.

□

Theorem 4.2 (Multistep Confluence) *Let t be a closed, typable, term. Then if $t \rightsquigarrow \mathscr{D}$ and $t \rightsquigarrow \mathscr{E}$ then $\mathscr{D} \equiv \mathscr{E}$. $\mathscr{D} \equiv \mathscr{E}$ means that the two distributions are exactly the same distributions.*

Proof:

We are going to prove the following strengthening of the thesis: Be t a closed term. If $t \Rightarrow \mathscr{D}$ and $t \Rightarrow \mathscr{E}$, be $\mathscr{D} \equiv \{M_1^{p_1}, \cdots, M_n^{p_n}\}$ and $\mathscr{E} \equiv \{N_1^{q_1}, \cdots, N_k^{q_k}\}$ then there exist $\mathscr{L}_1, \ldots, \mathscr{L}_n, \mathscr{J}_1, \ldots, \mathscr{J}_k$ such that $M_1 \Rightarrow \mathscr{L}_1, \cdots, M_n \Rightarrow \mathscr{L}_n$ and $N_1 \Rightarrow \mathscr{J}_1, \cdots, N_k \Rightarrow \mathscr{J}_k$, $\max_i(|M_i \Rightarrow \mathscr{L}_i|) \leq |t \Rightarrow \mathscr{E}|$, $\max_j(|N_j \Rightarrow \mathscr{J}_j|) \leq |t \Rightarrow \mathscr{D}|$ and $\sum_i(p_i \times \mathscr{L}_i) \equiv \sum_j(q_j \times \mathscr{J}_j)$.

We are going to prove it by induction on the sum of the length of the two derivations of $t \Rightarrow \mathscr{D}$ and $t \Rightarrow \mathscr{E}$.

- If both derivations end with the axiom rule,we are in the following case:

$$\frac{}{t \Rightarrow \mathscr{G}_{t1}} \quad \frac{}{t \Rightarrow \mathscr{G}_{t2}}$$

  we can associate to t the distribution $\mathscr{G}_t$ and the thesis is proved.
- If t is `rand`, it's easy to check the validity of the thesis (independently from the structure of the two derivations).
- If only one of the derivation consists of the axiom rule, we are in the following case:

$$\frac{t \to t_1, \ldots, t_n \qquad t_i \Rightarrow \mathscr{D}_i}{t \Rightarrow \sum_{i=1}^n \frac{1}{n}\mathscr{D}_i} \quad \frac{}{t \Rightarrow \mathscr{G}_t}$$

  If $\mathscr{D} \equiv \sum_{i=1}^n \frac{1}{n}\mathscr{D}_i \equiv \{M_1^{p_1}, \cdots, M_n^{p_n}\}$ and $\mathscr{G}_t \equiv \{t^1\}$, then it's easy to find the "confluent" distribution. For each M_i we associate the relative $\mathscr{G}_{M_i}$ and to t we associate $\mathscr{D}$. The thesis is proved.
- Otherwise we are in the case where the sum of the two lengths is more than 2 and so, where the last rule, for both derivations, is not the axiom one.

$$\frac{t \to t_1, \ldots, t_n \qquad t_i \Rightarrow \mathscr{D}_i}{t \Rightarrow \sum_{i=1}^n \frac{1}{n}\mathscr{D}_i} \quad \frac{t \to s_1, \ldots, s_m \qquad s_i \Rightarrow \mathscr{E}_i}{t \Rightarrow \sum_{i=1}^m \frac{1}{m}\mathscr{E}_i}$$

  - If $t_1, \ldots, t_n$ is equal to $s_1, \ldots, s_m$ (modulo sort) then by using induction hypothesis we are done. Let's consider the most interesting case, where the terms on the right hand side of $\to$ are different.
  - If $n = m = 1$. By lemma 4.1 we could have three possible configurations:
    - $t_1 \to s_1$. We have that $t_1 \Rightarrow \mathscr{D}_1$ and $t_1 \Rightarrow \mathscr{E}_1$. So the thesis is derived by induction.
    - $s_1 \to t_1$. Same as before.

- $\exists r$ s.t. $t_1 \to r$ and $s_1 \to r$. Be $\mathscr{D} \equiv \{M_1^{p_1}, \cdots, M_n^{p_n}\}$ and $\mathscr{E} \equiv \{N_1^{q_1}, \cdots, N_k^{q_k}\}$. By using axiom rule, we can associate a distribution to r; let's call it $\mathscr{P}$, such that $r \Rightarrow \mathscr{P}$. So, $t_1 \Rightarrow \mathscr{D}_1$ and $t_1 \Rightarrow \mathscr{P}$. By induction exist $\mathscr{L}_1, \ldots, \mathscr{L}_n, \mathscr{K}$ such that $M_1 \Rightarrow \mathscr{L}_1, \cdots, M_n \Rightarrow \mathscr{L}_n$ and $r \Rightarrow \mathscr{K}$, $\max_i(|M_i \Rightarrow \mathscr{L}_i|) \leq |t \Rightarrow \mathscr{P}|$ and $|r \Rightarrow \mathscr{K}| \leq |t \Rightarrow \mathscr{D}|$ and $\sum_i (p_i \times \mathscr{L}_i) \equiv \mathscr{K}$.
  Similar we have that there exist $\mathscr{J}_1, \ldots, \mathscr{J}_k, \mathscr{H}$ such that $N_1 \Rightarrow \mathscr{J}_1, \cdots, N_k \Rightarrow \mathscr{J}_k$ and $r \Rightarrow \mathscr{H}$, $\max_i(|N_i \Rightarrow \mathscr{J}_i|) \leq |t \Rightarrow \mathscr{P}|$ and $|r \Rightarrow \mathscr{H}| \leq |t \Rightarrow \mathscr{E}|$ and $\sum_i (q_i \times \mathscr{J}_i) \equiv \mathscr{H}$.
  Merging the two disambiguation, we obtain that $|r \Rightarrow \mathscr{K}| + |r \Rightarrow \mathscr{H}| \leq |t \Rightarrow \mathscr{D}| + |t \Rightarrow \mathscr{E}|$. Be $\mathscr{K} \equiv \{P_1^{\gamma_1}, \ldots, P_o^{\gamma_o}\}$ and $\mathscr{H} \equiv \{Q_1^{\delta_1}, \ldots, Q_p^{\delta_p}\}$
  We can apply induction hypothesis and obtain that exist $\mathscr{Q}_1, \ldots, \mathscr{Q}_o, \mathscr{R}_1, \ldots, \mathscr{R}_p$ such that $P_1 \Rightarrow \mathscr{Q}_1, \cdots, P_n \Rightarrow \mathscr{Q}_o$ and $Q_1 \Rightarrow \mathscr{R}_1, \cdots, Q_k \Rightarrow \mathscr{R}_k$, $\max_i(|P_i \Rightarrow \mathscr{Q}_i|) \leq |r \Rightarrow \mathscr{H}|$ and $\max_j(|Q_j \Rightarrow \mathscr{R}_j|) \leq |r \Rightarrow \mathscr{K}|$ and $\sum_i(\gamma_i \times \mathscr{Q}_i) \equiv \sum_j(\delta_j \times \mathscr{R}_j)$.
  Notice that the cardinality of $\mathscr{D}$ and $\mathscr{K}$ may differ but for sure they have the same terms with non zero probability. Similarly, $\mathscr{E}$ and $\mathscr{H}$ have the same terms with non zero probability.
  By using lemma 4.4 and using transitive property of equality we obtain that $t \Rightarrow \sum_i p_i \mathscr{Q}_i \equiv \sum_i \gamma_i \mathscr{Q}_i = \sum_j \delta_j \mathscr{R}_j$ and $t \Rightarrow \sum_i q_i \mathscr{R}_i \equiv \sum_j \delta_j \mathscr{R}_j$. Moreover we have:

$$\max_i(|M_i \Rightarrow \mathscr{Q}_i|) \leq |r \Rightarrow \mathscr{H}| \leq |t \Rightarrow \mathscr{E}|$$
$$\max_i(|N_i \Rightarrow \mathscr{R}_i|) \leq |r \Rightarrow \mathscr{K}| \leq |t \Rightarrow \mathscr{D}|$$

  The thesis is proved.

- If $n = 2$ and $m = 1$. By lemma 4.2 we could have three possible configurations:
  - $\forall i. t_i \to s_1$. If so, $t_1 \Rightarrow \mathscr{E}$ and $t_2 \Rightarrow \mathscr{E}$ (recall that $m = 1$, so $s_1 \Rightarrow \mathscr{E}$). Be $\mathscr{D} \equiv \{M_1^{\alpha_1}, \ldots, M_n^{\alpha_n}\}$ and $\mathscr{E} \equiv \{N_1^{\beta_1}, \ldots, N_k^{\beta_k}\}$. By construction, we have some elements that belong to $\mathscr{D}_1$, other to $\mathscr{D}_2$ and some element that belong to both of them. Without loosing generality, let's say that elements $M_1, \ldots, M_m$ belongs

to $\mathscr{D}_1$ and elements $M_o, \ldots, M_n$, where $1 \leq o \leq m \leq n$.

So, we have that $\mathscr{D}_1 \equiv \{M_1^{2\alpha_1}, \ldots, M_{o-1}^{2\alpha_{o-1}}, M_o^{\alpha_o}, \ldots, M_m^{\alpha_m}\}$ and we have that $\mathscr{D}_2$ is $\{M_o^{\alpha_o}, \ldots, M_m^{\alpha_m}, M_{m+1}^{2\alpha_m}, \ldots, M_n^{2\alpha_n}\}$.

By using induction we have that there exist $\mathscr{L}_1, \ldots, \mathscr{L}_n$, $\mathscr{J}_1, \ldots, \mathscr{J}_k$ such that $M_1 \Rightarrow \mathscr{L}_1, \cdots, M_n \Rightarrow \mathscr{L}_n$ and $N_1 \Rightarrow \mathscr{J}_1, \cdots, N_k \Rightarrow \mathscr{J}_k$, $\max_{0\leq i\leq m}(|M_i \Rightarrow \mathscr{L}_i|) \leq |t \Rightarrow \mathscr{E}|$, $\max_j(|N_j \Rightarrow \mathscr{J}_j|) \leq |t_1 \Rightarrow \mathscr{D}_1|$, $\max_{o\leq i\leq n}(|M_i \Rightarrow \mathscr{L}_i|) \leq |t \Rightarrow \mathscr{E}|$, $\max_j(|N_j \Rightarrow \mathscr{J}_j|) \leq |t_2 \Rightarrow \mathscr{D}_2|$, $\sum_{i=1}^{m-1} 2\alpha_i\mathscr{L}_i + \sum_{i=m}^{o} \alpha_i\mathscr{L}_1 \equiv \sum_j(\beta_j \times \mathscr{J}_j)$ and $\sum_{i=m}^{o} \alpha_i\mathscr{L}_i + \sum_{i=o+1}^{n} 2\alpha_i\mathscr{L}_i \equiv \sum_j(\beta_j \times \mathscr{J}_j)$.

Merging all, we have that there exist $\mathscr{L}_1, \ldots, \mathscr{L}_n$, $\mathscr{J}_1, \ldots, \mathscr{J}_k$ such that $M_1 \Rightarrow \mathscr{L}_1, \ldots, M_n \Rightarrow \mathscr{L}_n$ and $N_1 \Rightarrow \mathscr{J}_1, \ldots, N_k \Rightarrow \mathscr{J}_k$, $\max_i(|M_i \Rightarrow \mathscr{L}_i|) \leq |t \Rightarrow \mathscr{E}|$, $\max_j(|N_j \Rightarrow \mathscr{J}_j|) \leq |t \Rightarrow \mathscr{D}|$, $\sum_i(p_i \times \mathscr{L}_i) \equiv \sum_j(q_j \times \mathscr{J}_j)$.

- $s \to t_1, t_2$. We have that $s \Rightarrow \frac{1}{2}(\mathscr{D}_1 + \mathscr{D}_2)$ and $s \Rightarrow \mathscr{E}$. By applying the induction hypothesis we prove our thesis. Notice that $|s \Rightarrow \mathscr{D}| = |t \Rightarrow \mathscr{D}|$.
- $\exists a_1, a_2$ s.t. $t_1 \to a_1$ and $t_2 \to a_2$ and $s_1 \to a_1, a_2$. Be $\mathscr{D} \equiv \{M_1^{\alpha_1}, \ldots, M_n^{\alpha_n}\}$ and $\mathscr{E} \equiv \{N_1^{\beta_1}, \ldots, N_k^{\beta_k}\}$. By construction, we have some elements that belong to $\mathscr{D}_1$, other to $\mathscr{D}_2$ and some element that belong to both of them. Without loosing generality, let's say that elements $M_1, \ldots, M_m$ belongs to $\mathscr{D}_1$ and elements $M_o, \ldots, M_n$, where $1 \leq o \leq m \leq n$.

  So, we have that $\mathscr{D}_1 \equiv \{M_1^{2\alpha_1}, \ldots, M_{o-1}^{2\alpha_{o-1}}, M_o^{\alpha_o}, \ldots, M_m^{\alpha_m}\}$ and we have that $\mathscr{D}_2$ is $\{M_o^{\alpha_o}, \ldots, M_m^{\alpha_m}, M_{m+1}^{2\alpha_m}, \ldots, M_n^{2\alpha_n}\}$.

  By using the axiom rule, we associate to every a_i a distribution $\mathscr{P}_i$ s.t. $a_i \Rightarrow \mathscr{P}_i$. Be $\mathscr{P}_1 \equiv \{P_1^{\gamma_1}, \ldots, P_o^{\gamma_o}\}$ and be $\mathscr{P}_2 \equiv \{Q_1^{\delta_1}, \ldots, Q_p^{\delta_o}\}$.

  So, we have, for all i, $t_i \Rightarrow \mathscr{D}_i$ and $t_i \Rightarrow \mathscr{P}_i$, $s \Rightarrow \mathscr{E}$ and $s \Rightarrow \frac{1}{2}(\mathscr{P}_1 + \mathscr{P}_2)$.

  By applying induction hypothesis on all the three cases we have that there exist $\mathscr{L}_1, \ldots, \mathscr{L}_n$, $\mathscr{J}_1, \ldots, \mathscr{J}_k, \mathscr{K}, \mathscr{H}, \mathscr{Q}, \mathscr{R}$ such that $M_1 \Rightarrow \mathscr{L}_1, \cdots, M_n \Rightarrow \mathscr{L}_n$, $N_1 \Rightarrow \mathscr{J}_1, \cdots, N_k \Rightarrow \mathscr{J}_k$, and $a_1 \Rightarrow \mathscr{K}$ and $a_2 \Rightarrow \mathscr{H}$ and $a_1 \Rightarrow \mathscr{Q}$ and $a_2 \Rightarrow \mathscr{R}$ such that:

  - $\max_{1\leq i\leq m}(|M_i \Rightarrow \mathscr{L}_i|) \leq |t_1 \Rightarrow \mathscr{P}_1|$,
    $|a_1 \Rightarrow \mathscr{K}| \leq |t_1 \Rightarrow \mathscr{D}_1|$,
    $\sum_{i=1}^{m-1} 2\alpha_i\mathscr{L}_i + \sum_{i=m}^{o} \alpha_i\mathscr{L}_i \equiv \mathscr{K}$
  - $\max_{o\leq i\leq n}(|M_i \Rightarrow \mathscr{L}_i|) \leq |t_2 \Rightarrow \mathscr{P}_2|$,
    $|a_2 \Rightarrow \mathscr{H}| \leq |t_2 \Rightarrow \mathscr{D}_2|$,

$\sum_{i=m}^{o} \alpha_i \mathscr{L}_i + \sum_{i=o+1}^{n} 2\alpha_i \mathscr{L}_i \equiv \mathscr{H}$

- $\max_i(|N_i \Rightarrow \mathscr{J}_i|) \leq |s \Rightarrow \frac{1}{2}(\mathscr{P}_1 + \mathscr{P}_2)|$,
  $\max\{|a_1 \Rightarrow \mathscr{Q}|, |a_2 \Rightarrow \mathscr{R}|\} \leq |s \Rightarrow \mathscr{E}|$
  $\sum_i \beta_i \mathscr{J}_i \equiv \frac{1}{2}(\mathscr{Q} + \mathscr{R})$

  Notice that $|a_1 \Rightarrow \mathscr{Q}|+|a_1 \Rightarrow \mathscr{K}| < |t \Rightarrow \mathscr{D}|+|t \Rightarrow \mathscr{E}|$. Moreover, notice also that the following inequality holds: $|a_2 \Rightarrow \mathscr{R}|+|a_2 \Rightarrow \mathscr{H}| < |t \Rightarrow \mathscr{D}|+|t \Rightarrow \mathscr{E}|$. We are allowed to apply, again, induction hypothesis and have a confluent distribution for both cases. Lemma 4.4 then allows us to connect the first two main derivations and by transitive property of equality we have the thesis.

- If $n = 1$ and $m = 2$. This case is similar to the previous one.
- If $n = m = 2$. By lemma 4.3 we have: $\exists a_1, a_2, a_3, a_4$ s.t. $t_1 \to a_1, a_2$ and $t_2 \to a_3, a_4$ and $\exists i.s_i \to a_1, a_3$ and $s_{1-i} \to a_2, a_4$.

  At each a_i we associate, by using the axiom rule, the relative distribution $\mathscr{P}_i$ s.t. $a \Rightarrow \mathscr{P}_i$.

  Without loosing generality, let's say that elements $M_1, \ldots, M_m$ belongs to $\mathscr{D}_1$ and elements $M_o, \ldots, M_n$ to $\mathscr{D}_2$, where $1 \leq o \leq m \leq n$; $N_1, \ldots, M_p$ belongs to $\mathscr{E}_1$ and elements $N_q, \ldots, N_k$ to $\mathscr{E}_2$, where $1 \leq q \leq p \leq k$.

  So, we have that $\mathscr{D}_1 \equiv \{M_1^{2\alpha_1}, \ldots, M_{o-1}^{2\alpha_{o-1}}, M_o^{\alpha_o}, \ldots, M_m^{\alpha_m}\}$ and we have that $\mathscr{D}_2$ is $\{M_o^{\alpha_o}, \ldots, M_m^{\alpha_m}, M_{m+1}^{2\alpha_m}, \ldots, M_n^{2\alpha_n}\}$ and $\mathscr{E}_1 \equiv \{N_1^{2\beta_1}, \ldots, N_{q-1}^{2\beta_{q-1}}, N_q^{\beta_q}, \ldots, N_p^{\beta_p}\}$ and $\mathscr{E}_2 \equiv \{N_q^{\beta_q}, \ldots, N_p^{\beta_p}, N_{q+1}^{2\beta_{p+1}}, \ldots, N_k^{2\beta_k}\}$ (some elements may overlap).

  This case is very similar to two previous ones. We have that $t_1 \Rightarrow \mathscr{D}_1$ and $t_1 \Rightarrow \frac{1}{2}(\mathscr{P}_1+\mathscr{P}_2)$, $t_2 \Rightarrow \mathscr{D}_2$ and $t_2 \Rightarrow \frac{1}{2}(\mathscr{P}_3+\mathscr{P}_4)$, $s_1 \Rightarrow \mathscr{E}_1$ and $s_1 \Rightarrow \frac{1}{2}(\mathscr{P}_1+\mathscr{P}_3)$, $s_2 \Rightarrow \mathscr{E}_2$ and $s_2 \Rightarrow \frac{1}{2}(\mathscr{P}_2 + \mathscr{P}_4)$. We can apply the induction hypothesis to the four cases and have that there exist $\mathscr{L}_1, \ldots, \mathscr{L}_n, \mathscr{J}_1, \ldots, \mathscr{J}_k, \mathscr{K}_1, \mathscr{K}_2, \mathscr{K}_3, \mathscr{K}_4, \mathscr{H}_1, \mathscr{H}_2, \mathscr{H}_3, \mathscr{H}_4$ such that $M_1 \Rightarrow \mathscr{L}_1, \cdots, M_n \Rightarrow \mathscr{L}_n$, $N_1 \Rightarrow \mathscr{J}_1, \cdots, N_k \Rightarrow \mathscr{J}_k$, $a_i \Rightarrow \mathscr{K}_i$ and $a_i \Rightarrow \mathscr{H}_i$ such that:

  - $\max_{1 \leq i \leq m}(|M_i \Rightarrow \mathscr{L}_i|) \leq |t_1 \Rightarrow \frac{1}{2}(\mathscr{P}_1 + \mathscr{P}_2)|$,
    $\max\{|a_1 \Rightarrow \mathscr{K}_1|, |a_2 \Rightarrow \mathscr{K}_2|\} \leq |t_1 \Rightarrow \mathscr{D}_1|$,
    $\sum_{i=1}^{m-1} 2\alpha_i \mathscr{L}_i + \sum_{i=m}^{o} \alpha_i \mathscr{L}_i \equiv \frac{1}{2}(\mathscr{K}_1 + \mathscr{K}_2)$
  - $\max_{o \leq i \leq n}(|M_i \Rightarrow \mathscr{L}_i|) \leq |t_2 \Rightarrow \frac{1}{2}(\mathscr{P}_3 + \mathscr{P}_4)|$,
    $\max\{|a_3 \Rightarrow \mathscr{K}_3|, |a_4 \Rightarrow \mathscr{K}_4|\} \leq |t_2 \Rightarrow \mathscr{D}_2|$,
    $\sum_{i=m}^{o} \alpha_i \mathscr{L}_i + \sum_{i=o+1}^{n} 2\alpha_i \mathscr{L}_i \equiv \frac{1}{2}(\mathscr{K}_3 + \mathscr{K}_4)$

- $\max_{1\leq i\leq p}(|N_i \Rightarrow \mathscr{J}_i|) \leq |s \Rightarrow \frac{1}{2}(\mathscr{P}_1 + \mathscr{P}_3)|$,
  $\max\{|a_1 \Rightarrow \mathscr{H}_1|, |a_3 \Rightarrow \mathscr{H}_3|\} \leq |s_1 \Rightarrow \mathscr{E}_1|$
  $\sum_{i=1}^{q-1} 2\beta_i \mathscr{J}_i + \sum_{i=q}^{p} \beta_i \mathscr{J}_i \equiv \frac{1}{2}(\mathscr{H}_1 + \mathscr{H}_2)$
- $\max_{q\leq i\leq k}(|N_i \Rightarrow \mathscr{J}_i|) \leq |s \Rightarrow \frac{1}{2}(\mathscr{P}_2 + \mathscr{P}_4)|$,
  $\max\{|a_2 \Rightarrow \mathscr{H}_2|, |a_4 \Rightarrow \mathscr{H}_4|\} \leq |s_2 \Rightarrow \mathscr{E}_2|$
  $\sum_{i=q}^{p} \beta_i \mathscr{J}_i + \sum_{i=p+1}^{k} 2\beta_i \mathscr{J}_i \equiv \frac{1}{2}(\mathscr{H}_2 + \mathscr{H}_4)$

Now, notice that for all i, $|a_i \Rightarrow \mathscr{K}_i| + |a_i \Rightarrow \mathscr{H}_i| \leq |t \Rightarrow \mathscr{D}| + |t \Rightarrow \mathscr{E}|$. As we have done in the previous cases, we are now able to apply the induction hypothesis on the four cases. Then we use the lemma 4.4 and find confluent distributions. Sum everything and we are able to prove our thesis.

This concludes the proof. □

Example 4.1 Consider again the term

$$t = (\lambda x : \blacksquare\mathbf{N}.(t_\oplus x x))\texttt{rand}$$

where $t_\oplus$ is a term computing $\oplus$ on natural numbers seen as booleans (0 stands for "false" and everything else stands for "true"):

$$t_\oplus = \lambda x : \blacksquare\mathbf{N}.\texttt{case}_{\blacksquare\mathbf{N}\to\mathbf{N}}\ x\ \texttt{zero}\ s_\oplus\ \texttt{even}\ r_\oplus\ \texttt{odd}\ r_\oplus;$$
$$s_\oplus = \lambda y : \blacksquare\mathbf{N}.\texttt{case}_{\mathbf{N}}\ y\ \texttt{zero}\ 0\ \texttt{even}\ 1\ \texttt{odd}\ 1;$$
$$r_\oplus = \lambda y : \blacksquare\mathbf{N}.\texttt{case}_{\mathbf{N}}\ y\ \texttt{zero}\ 1\ \texttt{even}\ 0\ \texttt{odd}\ 0.$$

In order to simplify reading, let us define:

- $f \equiv (t_\oplus x x)$
- $g_0 \equiv (\texttt{case}_{\blacksquare\mathbf{N}\to\mathbf{N}}\ 0\ \texttt{zero}\ s_\oplus\ \texttt{even}\ r_\oplus\ \texttt{odd}\ r_\oplus)$
- $g_1 \equiv (\texttt{case}_{\blacksquare\mathbf{N}\to\mathbf{N}}\ 1\ \texttt{zero}\ s_\oplus\ \texttt{even}\ r_\oplus\ \texttt{odd}\ r_\oplus)$
- $h_0 \equiv \texttt{case}_{\mathbf{N}}\ 0\ \texttt{zero}\ 0\ \texttt{even}\ 1\ \texttt{odd}\ 1$
- $h_1 \equiv \texttt{case}_{\mathbf{N}}\ 1\ \texttt{zero}\ 1\ \texttt{even}\ 0\ \texttt{odd}\ 0$

We can produce the following derivation tree:

$$\pi_0 : \dfrac{(\lambda x : \blacksquare\mathbf{N}.f)0 \to t_\oplus 00 \qquad \dfrac{t_\oplus 0\,0 \to g_0 0 \qquad \dfrac{g_0 0 \to s_\oplus 0 \qquad \dfrac{s_\oplus 0 \to h_0 \qquad \dfrac{h_0 \to 0 \qquad 0 \rightsquigarrow \{0^1\}}{h_0 \rightsquigarrow \{0^1\}}}{s_\oplus 0 \rightsquigarrow \{0^1\}}}{g_0 0 \rightsquigarrow \{0^1\}}}{(\lambda x : \blacksquare\mathbf{N}.\texttt{case}_{\blacksquare\mathbf{N}\to\mathbf{N}}\ x\ \texttt{zero}\ s_\oplus\ \texttt{even}\ r_\oplus\ \texttt{odd}\ r_\oplus)0\,0 \rightsquigarrow \{0^1\}}}{(\lambda x : \blacksquare\mathbf{N}.f)0 \rightsquigarrow \{0^1\}}$$

$$\pi_1 : \dfrac{(\lambda x : \blacksquare\mathbf{N}.f)1 \to t_\oplus 11 \qquad \dfrac{t_\oplus 1\,1 \to g_1 1 \qquad \dfrac{g_1 1 \to r_\oplus 1 \qquad \dfrac{r_\oplus 1 \to h_1 \qquad \dfrac{h_1 \to 0 \qquad 0 \rightsquigarrow \{0^1\}}{h_1 \rightsquigarrow \{0^1\}}}{r_\oplus 1 \rightsquigarrow \{0^1\}}}{g_1 1 \rightsquigarrow \{0^1\}}}{(\lambda x : \blacksquare\mathbf{N}.\mathtt{case}_{\blacksquare\mathbf{N}\to\mathbf{N}}\ x\ \mathtt{zero}\ s_\oplus\ \mathtt{even}\ r_\oplus\ \mathtt{odd}\ r_\oplus)1\,1 \rightsquigarrow \{0^1\}}}{(\lambda x : \blacksquare\mathbf{N}.f)1 \rightsquigarrow \{0^1\}}$$

$$\dfrac{(\lambda x : \blacksquare\mathbf{N}.f)\mathtt{rand} \to (\lambda x : \blacksquare\mathbf{N}.f)0, (\lambda x : \blacksquare\mathbf{N}.f)1 \qquad \pi_0 : (\lambda x : \blacksquare\mathbf{N}.f)0 \rightsquigarrow \{0^1\} \qquad \pi_1 : (\lambda x : \blacksquare\mathbf{N}.f)1 \rightsquigarrow \{0^1\}}{(\lambda x : \blacksquare\mathbf{N}.(t_\oplus x x))\mathtt{rand} \rightsquigarrow \{0^1\}}$$

□

4.7 Probabilistic Polytime Soundness

The most difficult (and interesting!) result about RSLR is definitely polytime soundness: every (instance of) a first-order term can be reduced to a numeral in a polynomial number of steps by a probabilistic Turing machine. Polytime soundness can be proved, following [7], by showing that:

- Any explicit term of base type can be reduced to its normal form with very low time complexity;
- Any term (non necessarily of base type) can be put in explicit form in polynomial time.

By gluing these two results together, we obtain what we need, namely an effective and efficient procedure to compute the normal forms of terms. Formally, two notions of evaluation for terms correspond to the two steps defined above:

- On the one hand, we need a ternary relation $\Downarrow_{\mathsf{nf}}$ between closed terms of type $\mathbf{N}$, probabilities and numerals. Intuitively, $t \Downarrow_{\mathsf{nf}}^{\alpha} n$ holds when t is explicit and rewrites to n with probability α. The inference rules for $\Downarrow_{\mathsf{nf}}$ are defined in Figure 4.6;
- On the other hand, we need a ternary relation $\Downarrow_{\mathsf{rf}}$ between terms of non modal type, probabilities and terms. We can derive $t \Downarrow_{\mathsf{rf}}^{\alpha} s$ only if t can be transformed into s with probability α consistently with the reduction relation. The inference rules for $\Downarrow_{\mathsf{rf}}$ are in Figure 4.7.

Moreover, a third ternary relation $\Downarrow$ between closed terms of type $\mathbf{N}$, probabilities and numerals can be defined by the rule below:

$$\dfrac{t \Downarrow_{\mathsf{rf}}^{\alpha} s \qquad s \Downarrow_{\mathsf{nf}}^{\beta} n}{t \Downarrow^{\alpha\beta} n}$$

$$\frac{}{n \Downarrow_{\mathsf{nf}}^{1} n} \qquad \frac{}{\mathtt{rand} \Downarrow_{\mathsf{nf}}^{1/2} 0} \qquad \frac{}{\mathtt{rand} \Downarrow_{\mathsf{nf}}^{1/2} 1}$$

$$\frac{t \Downarrow_{\mathsf{nf}}^{\alpha} n}{\mathtt{S}_0 t \Downarrow_{\mathsf{nf}}^{\alpha} 2 \cdot n} \qquad \frac{t \Downarrow_{\mathsf{nf}}^{\alpha} n}{\mathtt{S}_1 t \Downarrow_{\mathsf{nf}}^{\alpha} 2 \cdot n + 1} \qquad \frac{t \Downarrow_{\mathsf{nf}}^{\alpha} 0}{\mathtt{P} t \Downarrow_{\mathsf{nf}}^{\alpha} 0} \qquad \frac{t \Downarrow_{\mathsf{nf}}^{\alpha} n \qquad n \geq 1}{\mathtt{P} t \Downarrow_{\mathsf{nf}}^{\alpha} \lfloor \frac{n}{2} \rfloor}$$

$$\frac{t \Downarrow_{\mathsf{nf}}^{\alpha} 0 \qquad s\overline{u} \Downarrow_{\mathsf{nf}}^{\beta} n}{(\mathtt{case}_A\ t\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q)\overline{u} \Downarrow_{\mathsf{nf}}^{\alpha\beta} n}$$

$$\frac{t \Downarrow_{\mathsf{nf}}^{\alpha} 2n \qquad r\overline{u} \Downarrow_{\mathsf{nf}}^{\beta} m \qquad n \geq 1}{(\mathtt{case}_A\ t\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q)\overline{u} \Downarrow_{\mathsf{nf}}^{\alpha\beta} m}$$

$$\frac{t \Downarrow_{\mathsf{nf}}^{\alpha} 2n+1 \qquad q\overline{u} \Downarrow_{\mathsf{nf}}^{\beta} m}{(\mathtt{case}_A\ t\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q)\overline{u} \Downarrow_{\mathsf{nf}}^{\alpha\beta} m}$$

$$\frac{s \Downarrow_{\mathsf{nf}}^{\alpha} n \qquad (t[x/n])\overline{r} \Downarrow_{\mathsf{nf}}^{\beta} m}{(\lambda x : a\mathbf{N}.t)s\overline{r} \Downarrow_{\mathsf{nf}}^{\alpha\beta} m} \qquad \frac{(t[x/s])\overline{r} \Downarrow_{\mathsf{nf}}^{\beta} n}{(\lambda x : aH.t)s\overline{r} \Downarrow_{\mathsf{nf}}^{\beta} n}$$

Figure 4.6: The relation $\Downarrow_{\mathsf{nf}}$: Inference Rules

A peculiarity of the just introduced relations with respect to similar ones is the following: whenever a statement in the form $t \Downarrow_{\mathsf{nf}}^{\alpha} s$ is an immediate premise of another statement $r \Downarrow_{\mathsf{nf}}^{\beta} q$, then t needs to be structurally smaller than r, provided all numerals are assumed to have the same internal structure. A similar but weaker statement holds for $\Downarrow_{\mathsf{rf}}$. This relies on the peculiarities of RSLR, and in particular on the fact that variables of higher-order types can appear free at most once in terms, and that terms of base types cannot be passed to functions without having been completely evaluated. In other words, the just described operational semantics is structural in a very strong sense, and this allows to prove properties about it by induction on the structure of *terms*, as we will experience in a moment.

Before starting to study the combinatorial properties of $\Downarrow_{\mathsf{rf}}$ and $\Downarrow_{\mathsf{nf}}$, it is necessary to show that, at least, $\Downarrow$ is adequate as a way to evaluate lambda terms. In the following, the size $|\pi|$ of any derivation π (for any formal system) is simply the number of distinct rule occurrences in π.

Theorem 4.3 (Adequacy) *For every term t such that $\vdash t : \mathbf{N}$, the following two conditions are equivalent:*

1. *There are j distinct derivations $\pi_1 : t \Downarrow^{\alpha_1} n_1, \ldots, \pi_j : t \Downarrow^{\alpha_j} n_j$ such that $\sum_{i=1}^{j} \alpha_i = 1$;*

$$\frac{}{c \Downarrow_{\mathsf{rf}}^{1} c} \qquad \frac{t \Downarrow_{\mathsf{rf}}^{\alpha} v}{\mathtt{S}_0 t \Downarrow_{\mathsf{rf}}^{\alpha} \mathtt{S}_0 v} \qquad \frac{t \Downarrow_{\mathsf{rf}}^{\alpha} v}{\mathtt{S}_1 t \Downarrow_{\mathsf{rf}}^{\alpha} \mathtt{S}_1 v} \qquad \frac{t \Downarrow_{\mathsf{rf}}^{\alpha} v}{\mathtt{P}t \Downarrow_{\mathsf{rf}}^{\alpha} \mathtt{P}v}$$

$$\frac{\begin{array}{lll} t \Downarrow_{\mathsf{rf}}^{\alpha} v & r \Downarrow_{\mathsf{rf}}^{\gamma} a & \\ s \Downarrow_{\mathsf{rf}}^{\beta} z & q \Downarrow_{\mathsf{rf}}^{\delta} b & \forall u_i \in \overline{u}, u_i \Downarrow_{\mathsf{rf}}^{\epsilon_i} c_i \end{array}}{(\mathtt{case}_A\ t\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q)\overline{u} \Downarrow_{\mathsf{rf}}^{\alpha\beta\gamma\delta\prod_i \epsilon_i} (\mathtt{case}_A\ v\ \mathtt{zero}\ z\ \mathtt{even}\ a\ \mathtt{odd}\ b)\overline{c}}$$

$$\frac{\begin{array}{lll} t \Downarrow_{\mathsf{rf}}^{\alpha} v & n > 0 & s \Downarrow_{\mathsf{rf}}^{\gamma} z \\ v \Downarrow_{\mathsf{nf}}^{\beta} n & \forall q_i \in \overline{q}, q_i \Downarrow_{\mathsf{rf}}^{\delta_i} b_i & r\lfloor \frac{n}{2^0} \rfloor \Downarrow_{\mathsf{rf}}^{\gamma_0} r_0 \ \ldots \ r\lfloor \frac{n}{2^{|n|-1}} \rfloor \Downarrow_{\mathsf{rf}}^{\gamma_{|n|-1}} r_{|n|-1} \end{array}}{(\mathtt{recursion}_A\ t\, s\, r)\overline{q} \Downarrow_{\mathsf{rf}}^{\alpha\beta\gamma(\prod_j \gamma_j)(\prod_i \delta_i)} r_0(\ldots(r_{(|n|-1)}z)\ldots)\overline{b}}$$

$$\frac{\begin{array}{ll} t \Downarrow_{\mathsf{rf}}^{\alpha} v & s \Downarrow_{\mathsf{rf}}^{\gamma} z \\ v \Downarrow_{\mathsf{nf}}^{\beta} 0 & \forall q_i \in \overline{q}, q_i \Downarrow_{\mathsf{rf}}^{\delta_i} b_i \end{array}}{(\mathtt{recursion}_A\ t\, s\, r)\overline{q} \Downarrow_{\mathsf{rf}}^{\alpha\beta\gamma(\prod_i \delta_i)} z\overline{b}}$$

$$\frac{\begin{array}{ll} s \Downarrow_{\mathsf{rf}}^{\alpha} z & \\ z \Downarrow_{\mathsf{nf}}^{\gamma} n & (t[x/n])\overline{r} \Downarrow_{\mathsf{rf}}^{\beta} u \end{array}}{(\lambda x : \Box\mathbf{N}.t)s\overline{r} \Downarrow_{\mathsf{rf}}^{\alpha\gamma\beta} u} \qquad \frac{\begin{array}{ll} s \Downarrow_{\mathsf{rf}}^{\alpha} z & \\ z \Downarrow_{\mathsf{nf}}^{\gamma} n & t\overline{r} \Downarrow_{\mathsf{rf}}^{\beta} u \end{array}}{(\lambda x : \blacksquare\mathbf{N}.t)s\overline{r} \Downarrow_{\mathsf{rf}}^{\alpha\gamma\beta} (\lambda x : \blacksquare\mathbf{N}.u)n}$$

$$\frac{(t[x/s])\overline{r} \Downarrow_{\mathsf{rf}}^{\beta} u}{(\lambda x : aH.t)s\overline{r} \Downarrow_{\mathsf{rf}}^{\beta} u} \qquad \frac{t \Downarrow_{\mathsf{rf}}^{\beta} u}{\lambda x : aA.t \Downarrow_{\mathsf{rf}}^{\beta} \lambda x : aA.u} \qquad \frac{t_j \Downarrow_{\mathsf{rf}}^{\alpha_j} s_j}{x\overline{t} \Downarrow_{\mathsf{rf}}^{\prod_i \alpha_i} x\overline{s}}$$

Figure 4.7: The relation $\Downarrow_{\mathsf{rf}}$: Inference Rules

2. $t \rightsquigarrow \mathscr{D}$*, where for every* m*,* $\mathscr{D}(m) = \sum_{n_i=m} \alpha_i$*.*

Proof: Implication 1 $\Rightarrow$ 2 can be proved by an induction on $\sum_{k=1}^{j} |\pi_k|$. About the converse, just observe that, *some* derivations like the ones required in Condition 1 need to exist. This can be formally proved by induction on $|t|_{\mathsf{w}}$, where $|\cdot|_{\mathsf{w}}$ is defined as follows: $|x|_{\mathsf{w}} = 1$, $|ts|_{\mathsf{w}} = |t|_{\mathsf{w}} + |s|_{\mathsf{w}}$, $|\lambda x : aA.t|_{\mathsf{w}} = |t|_{\mathsf{w}} + 1$, $|\mathtt{case}_A\ t\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q|_{\mathsf{w}} = |t|_{\mathsf{w}} + |s|_{\mathsf{w}} + |r|_{\mathsf{w}} + |q|_{\mathsf{w}} + 1$, $|\mathtt{recursion}_A\ t\, s\, r|_{\mathsf{w}} = |t|_{\mathsf{w}} + |s|_{\mathsf{w}} + |r|_{\mathsf{w}} + 1$, $|n|_{\mathsf{w}} = 1$, $|\mathtt{S}_0|_{\mathsf{w}} = |\mathtt{S}_1|_{\mathsf{w}} = |\mathtt{P}|_{\mathsf{w}} = |\mathtt{rand}|_{\mathsf{w}} = 1$. Thanks to multistep confluence, we can conclude. □

It's now time to analyse how big derivations for $\Downarrow_{\mathsf{nf}}$ and $\Downarrow_{\mathsf{rf}}$ can be with respect to the size of the underlying term. Let us start with $\Downarrow_{\mathsf{nf}}$ and prove that, since it can only be applied to explicit terms, the sizes of derivations must be very small:

Proposition 4.1 *Suppose that* $\vdash t : \mathbf{N}$*, where* t *is explicit. Then for every* $\pi : t \Downarrow_{\mathsf{nf}}^{\alpha} m$ *it holds that*

1. $|\pi| \leq 2 \cdot |t|$*;*

2. If $s \in \pi$*, then* $|s| \leq 2 \cdot |t|^2$*;*

Proof: Given any term t, $|t|_{\mathsf{w}}$ and $|t|_{\mathsf{n}}$ are defined, respectively, as the size of t where every numeral counts for 1 and the maximum size of the numerals that occur in t. For a formal definition of $|\cdot|_{\mathsf{w}}$, see the proof of Theorem 4.3. On the other hand, $|\cdot|_{\mathsf{n}}$ is defined as follows: $|x|_{\mathsf{n}} = 1$, $|ts|_{\mathsf{n}} = \max\{|t|_{\mathsf{n}}, |s|_{\mathsf{n}}\}$, $|\lambda x : aA.t|_{\mathsf{n}} = |t|_{\mathsf{n}}$, $|\mathtt{case}_A\ t\ \mathtt{zero}\ s\ \mathtt{even}\ r\ \mathtt{odd}\ q|_{\mathsf{n}} = \max\{|t|_{\mathsf{n}}, |s|_{\mathsf{n}}, |r|_{\mathsf{n}}, |q|_{\mathsf{n}}\}$, $|\mathtt{recursion}_A\ t\, s\, r|_{\mathsf{n}} = \max\{|t|_{\mathsf{n}}, |s|_{\mathsf{n}}, |r|_{\mathsf{n}}\}$, $|n|_{\mathsf{n}} = \lfloor \log_2(n+2) \rfloor$, and $|\mathtt{S}_0|_{\mathsf{n}} = |\mathtt{S}_1|_{\mathsf{n}} = |\mathtt{P}|_{\mathsf{n}} = |\mathtt{rand}|_{\mathsf{n}} = 1$. It holds that $|t| \leq |t|_{\mathsf{w}} \cdot |t|_{\mathsf{n}}$. It can be proved by structural induction on term t. We prove the following strengthening of the statements above by induction on $|t|_{\mathsf{w}}$:

1. $|\pi| \leq |t|_{\mathsf{w}}$;
2. If $s \in \pi$, then $|s|_{\mathsf{w}} \leq |t|_{\mathsf{w}}$ and $|s|_{\mathsf{n}} \leq |t|_{\mathsf{n}} + |t|_{\mathsf{w}}$;

First we prove that the strengthening holds. From the first case of the strengthening thesis we can deduce the first case of the main thesis. Notice indeed that $|t| \leq |t|_{\mathsf{w}} \cdot |t|_{\mathsf{n}}$. Regarding the latter point, notice that $|s| \leq |s|_{\mathsf{w}} \cdot |s|_{\mathsf{n}} \leq |t|_{\mathsf{w}} \cdot (|t|_{\mathsf{n}} + |t|_{\mathsf{w}}) \leq |t|^2 + |t| \leq 2 \cdot |t|^2$. Some interesting cases:

- Suppose t is $\mathtt{rand}$. We could have two derivations:

$$\frac{}{\mathtt{rand} \Downarrow_{\mathsf{nf}}^{1/2} 0} \qquad \frac{}{\mathtt{rand} \Downarrow_{\mathsf{nf}}^{1/2} 1}$$

The thesis is easily proved.

- Suppose t is $\mathtt{S}_i s$. Depending on $\mathtt{S}_i$ we could have two different derivations:

$$\frac{\rho : s \Downarrow_{\mathsf{nf}}^{\alpha} n}{\mathtt{S}_0 s \Downarrow_{\mathsf{nf}}^{\alpha} 2 \cdot n} \qquad \frac{\rho : s \Downarrow_{\mathsf{nf}}^{\alpha} n}{\mathtt{S}_1 s \Downarrow_{\mathsf{nf}}^{\alpha} 2 \cdot n + 1}$$

Suppose we are in the case where $\mathtt{S}_i \equiv \mathtt{S}_0$. Then, for every $r \in \pi$,

$$\begin{aligned} |\pi| = |\rho| + 1 &\leq |s|_{\mathsf{w}} + 1 = |t|_{\mathsf{w}}; \\ |r|_{\mathsf{w}} &\leq |s|_{\mathsf{w}} \leq |t|_{\mathsf{w}} \\ |r|_{\mathsf{n}} &\leq |s|_{\mathsf{n}} + |s|_{\mathsf{w}} + 1 = |s|_{\mathsf{n}} + |t|_{\mathsf{w}} \\ &= |t|_{\mathsf{n}} + |t|_{\mathsf{w}} \end{aligned}$$

The case where $\mathtt{S}_i \equiv \mathtt{S}_1$ is proved in the same way.

- Suppose t is $\mathtt{P}s$.

$$\frac{\rho : s \Downarrow_{\mathsf{nf}}^{\alpha} 0}{\mathtt{P}s \Downarrow_{\mathsf{nf}}^{\alpha} 0} \qquad \frac{\rho : s \Downarrow_{\mathsf{nf}}^{\alpha} n \qquad n \geq 1}{\mathtt{P}s \Downarrow_{\mathsf{nf}}^{\alpha} \lfloor \frac{n}{2} \rfloor}$$

We focus on case where $n > 1$, the other case is similar. For every $r \in \pi$ we have

$$\begin{aligned} |\pi| = |\rho| + 1 &\leq |s|_{\mathsf{w}} + 1 = |t|_{\mathsf{w}} \\ |r|_{\mathsf{w}} &\leq |s|_{\mathsf{w}} \leq |t|_{\mathsf{w}} \\ |r|_{\mathsf{n}} &\leq |s|_{\mathsf{n}} + |s|_{\mathsf{w}} + 1 = |s|_{\mathsf{n}} + |t|_{\mathsf{w}} \\ &= |t|_{\mathsf{n}} + |t|_{\mathsf{w}} \end{aligned}$$

- Suppose t is n.

$$\frac{}{n \Downarrow_{\mathsf{nf}}^{1} n}$$

By knowing $|\pi| = 1$, $|n|_{\mathsf{w}} = 1$ and $|n|_{\mathsf{n}} = |n|$, the proof is trivial.

- Suppose that t is $(\lambda y : a\mathbf{N}.s)r\overline{q}$. All derivations π for t are in the following form:

$$\frac{\rho : r \Downarrow_{\mathsf{nf}}^{\alpha} o \qquad \mu : (s[y/o])\overline{q} \Downarrow_{\mathsf{nf}}^{\beta} m}{t \Downarrow_{\mathsf{nf}}^{\alpha\beta} m}$$

Then, for every $u \in \pi$,

$$\begin{aligned}
|\pi| &\leq |\rho| + |\mu| + 1 \leq |r|_{\mathsf{w}} + |s[y/o]\overline{q}|_{\mathsf{w}} + 1 \\
&= |r|_{\mathsf{w}} + |s\overline{q}|_{\mathsf{w}} + 1 \leq |t|_{\mathsf{w}}; \\
|u|_{\mathsf{n}} &\leq \max\{|r|_{\mathsf{n}} + |r|_{\mathsf{w}}, |s[y/o]\overline{q}|_{\mathsf{n}} + |s[y/o]\overline{q}|_{\mathsf{w}}\} \\
&= \max\{|r|_{\mathsf{n}} + |r|_{\mathsf{w}}, |s[y/o]\overline{q}|_{\mathsf{n}} + |s\overline{q}|_{\mathsf{w}}\} \\
&= \max\{|r|_{\mathsf{n}} + |r|_{\mathsf{w}}, \max\{|s\overline{q}|_{\mathsf{n}}, |o|\} + |s\overline{q}|_{\mathsf{w}}\} \\
&= \max\{|r|_{\mathsf{n}} + |r|_{\mathsf{w}}, |s\overline{q}|_{\mathsf{n}} + |s\overline{q}|_{\mathsf{w}}, |o| + |s\overline{q}|_{\mathsf{w}}\} \\
&\leq \max\{|r|_{\mathsf{n}} + |r|_{\mathsf{w}}, |s\overline{q}|_{\mathsf{n}} + |s\overline{q}|_{\mathsf{w}}, |r|_{\mathsf{n}} + |r|_{\mathsf{w}} + |s\overline{q}|_{\mathsf{w}}\} \\
&\leq \max\{|r|_{\mathsf{n}}, |s\overline{q}|_{\mathsf{n}}\} + |r|_{\mathsf{w}} + |s\overline{q}|_{\mathsf{w}} \\
&\leq \max\{|r|_{\mathsf{n}}, |s\overline{q}|_{\mathsf{n}}\} + |t|_{\mathsf{w}} \\
&= |t|_{\mathsf{n}} + |t|_{\mathsf{w}}; \\
|u|_{\mathsf{w}} &\leq \max\{|r|_{\mathsf{w}}, |s[y/o]\overline{q}|_{\mathsf{w}}, |t|_{\mathsf{w}}\} \\
&= \max\{|r|_{\mathsf{w}}, |s\overline{q}|_{\mathsf{w}}, |t|_{\mathsf{w}}\} \leq |t|_{\mathsf{w}}.
\end{aligned}$$

If $u \in \pi$, then either $u \in \rho$ or $u \in \mu$ or simply $u = t$. This, together with the induction hypothesis, implies $|u|_{\mathsf{w}} \leq \max\{|r|_{\mathsf{w}}, |s[y/o]\overline{q}|_{\mathsf{w}}, |t|_{\mathsf{w}}\}$. Notice that $|s\overline{q}|_{\mathsf{w}} = |s[y/o]\overline{q}|_{\mathsf{n}}$ holds because any occurrence of y in s counts for 1, but also o itself counts for 1 (see the definition of $|\cdot|_{\mathsf{w}}$ above). More generally, duplication of *numerals* for a variable in t does not make $|t|_{\mathsf{w}}$ bigger.

- Suppose t is $(\lambda y : aH.s)r\overline{q}$. Without loosing generality we can say that it derives from the following derivation:

$$\frac{\rho : (s[y/r])\overline{q} \Downarrow^{\beta}_{\mathsf{nf}} n}{(\lambda y : aH.s)r\overline{q} \Downarrow^{\beta}_{\mathsf{nf}} n}$$

For the reason that y has type H we can be sure that it appears at most once in s. So, $|s[y/r]| \leq |sr|$ and, moreover, $|s[y/r]\overline{q}|_{\mathsf{w}} \leq |sr\overline{q}|_{\mathsf{w}}$ and $|s[y/r]\overline{q}|_{\mathsf{n}} \leq |sr\overline{q}|_{\mathsf{n}}$. We have, for all $u \in \rho$:

$$\begin{aligned}
|\pi| = |\rho| + 1 &\leq |s[y/r]\overline{q}|_{\mathsf{w}} + 1 \leq |t|_{\mathsf{w}} \\
|u|_{\mathsf{w}} &\leq |s[y/r]\overline{q}|_{\mathsf{w}} \leq |sr\overline{q}|_{\mathsf{w}} \leq |t|_{\mathsf{w}} \\
|u|_{\mathsf{n}} &\leq |s[y/r]\overline{q}|_{\mathsf{n}} + |s[y/r]\overline{q}|_{\mathsf{w}} \leq |sr\overline{q}|_{\mathsf{n}} + |sr\overline{q}|_{\mathsf{w}} \leq |t|_{\mathsf{n}} + |t|_{\mathsf{w}}
\end{aligned}$$

and this means that the same inequalities hold for every $u \in \pi$.

- Suppose t is $\mathtt{case}_A\ s\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u$. We could have three possible derivations:

$$\frac{\rho : s \Downarrow^{\alpha}_{\mathsf{nf}} 0 \qquad \mu : r\overline{v} \Downarrow^{\beta}_{\mathsf{nf}} n}{(\mathtt{case}_A\ s\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u)\overline{v} \Downarrow^{\alpha\beta}_{\mathsf{nf}} n}$$

$$\frac{\rho : s \Downarrow^{\alpha}_{\mathsf{nf}} 2n \qquad \mu : q\overline{v} \Downarrow^{\beta}_{\mathsf{nf}} m \qquad n \geq 1}{(\mathtt{case}_A\ s\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u)\overline{v} \Downarrow^{\alpha\beta}_{\mathsf{nf}} m}$$

$$\frac{\rho : s \Downarrow^{\alpha}_{\mathsf{nf}} 2n+1 \qquad \mu : u\overline{v} \Downarrow^{\beta}_{\mathsf{nf}} m}{(\mathtt{case}_A\ s\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u)\overline{v} \Downarrow^{\alpha\beta}_{\mathsf{nf}} m}$$

  we will focus on the case where the value of s is odd. All the other cases are similar. For all $z \in \pi$ we have:

$$\begin{aligned}
|\pi| &\leq |\rho| + |\mu| + 1 \\
&\leq |s|_{\mathsf{w}} + |u\overline{v}|_{\mathsf{w}} + 1 \leq |t|_{\mathsf{w}} \\
|z|_{\mathsf{w}} &\leq |s|_{\mathsf{w}} + |r|_{\mathsf{w}} + |q|_{\mathsf{w}} + |u\overline{v}|_{\mathsf{w}} \leq |t|_{\mathsf{w}} \\
|z|_{\mathsf{n}} &= \max\{|s|_{\mathsf{n}} + |s|_{\mathsf{w}}, |u\overline{v}|_{\mathsf{n}} + |u\overline{v}|_{\mathsf{w}}, |r|_{\mathsf{n}}, |q|_{\mathsf{n}}\} \\
&\leq \max\{|s|_{\mathsf{n}}, |u\overline{v}|_{\mathsf{n}}, |r|_{\mathsf{n}}, |q|_{\mathsf{n}}\} + |s|_{\mathsf{w}} + |u\overline{v}|_{\mathsf{w}} \\
&\leq |t|_{\mathsf{w}} + |t|_{\mathsf{n}}
\end{aligned}$$

This concludes the proof. □

As opposed to $\Downarrow_{\mathsf{nf}}$, $\Downarrow_{\mathsf{rf}}$ unrolls instances of primitive recursion, and thus cannot have the very simple combinatorial behaviour of $\Downarrow_{\mathsf{nf}}$. Fortunately, however, everything stays under control:

Proposition 4.2 *Suppose that $x_1 : \Box\mathbf{N}, \ldots, x_i : \Box\mathbf{N} \vdash t : A$, where A is $\Box$-free type. Then there are polynomials p_t and q_t such that for every $n_1, \ldots, n_i$ and for every $\pi : t[\overline{x}/\overline{n}] \Downarrow^{\alpha}_{\mathsf{rf}} s$ it holds that:*

1. *$|\pi| \leq p_t(\sum_i |n_i|)$;*
2. *If $s \in \pi$, then $|s| \leq q_t(\sum_i |n_i|)$.*

Proof: The following strengthening of the result can be proved by induction on the structure of a type derivation μ for t: if $x_1 : \Box\mathbf{N}, \ldots, x_i : \Box\mathbf{N}, y_1 : \blacksquare A_1, \ldots, y_j : \blacksquare A_j \vdash t : A$, where A is positively $\Box$-free and $A_1, \ldots, A_j$ are negatively $\Box$-free (formal definition below). Then there are polynomials p_t and q_t such that for every $n_1, \ldots, n_i$ and for every $\pi : t[\overline{x}/\overline{n}] \Downarrow^{\alpha}_{\mathsf{rf}} s$ it holds that

1. $|\pi| \leq p_t(\sum_i |n_i|)$;
2. If $s \in \pi$, then $|s| \leq q_t(\sum_i |n_i|)$.

In defining positively and negatively $\Box$-free types, let us proceed by induction on types:

- $\mathbf{N}$ is both positively and negatively $\Box$-free;
- $\Box A \to B$ is *not* positively $\Box$-free, and is negatively $\Box$-free whenever A is positively $\Box$-free and B is negatively $\Box$-free;
- $C = \blacksquare A \to B$ is positively $\Box$-free if A is negatively and B is positively $\Box$-free. C is negatively $\Box$-free if A is positively $\Box$-free and B is negatively $\Box$-free.

Please observe that if A is positively $\Box$-free and $B <: A$, then B is positively $\Box$-free. Conversely, if A is negatively $\Box$-free and $A <: B$, then B is negatively $\Box$-free. This can be easily proved by induction on the structure of A. We are ready to start the proof, now. Let us consider some cases, depending on the shape of μ

- If the only typing rule in μ is (T-Const-Aff), then $t \equiv c$, $p_t(x) \equiv 1$ and $q_t(x) \equiv 1$. The thesis is proved.
- If the last rule was (T-Var-Aff) then $t \equiv x$, $p_t(x) \equiv 1$ and $q_t(x) \equiv x$. The thesis is proved
- If the last rule was (T-Arr-I) then $t \equiv \lambda x : \blacksquare A.s$. Notice that the aspect is $\blacksquare$ because the type of our term has to be positively $\Box$-free. So, we have the following derivation:

$$\frac{\rho : s[\overline{x}/\overline{n}] \Downarrow^{\beta}_{\mathsf{rf}} v}{\lambda x : aA.s[\overline{x}/\overline{n}] \Downarrow^{\beta}_{\mathsf{rf}} \lambda x : aA.v}$$

If the type of t is positively $\Box$-free, then also the type of s is positively $\Box$-free. We can apply induction hypothesis. Define p_t and q_t as:

$$p_t(x) \equiv p_s(x) + 1$$
$$q_t(x) \equiv q_s(x) + 1$$

Indeed, we have:

$$\begin{aligned} |\pi| &\equiv |\rho| + 1 \\ &\leq p_s(\sum_i |n_i|) + 1 \end{aligned}$$

- If last rule was (T-Sub) then we have a typing derivation that ends in the following way:

$$\frac{\Gamma \vdash t : A \qquad A <: B}{\Gamma \vdash t : B}$$

we can apply induction hypothesis on $t : A$ because if B is positively $\Box$-free, then also A will be too. Define $p_{t:B}(x) \equiv p_{t:A}(x)$ and $q_{t:B}(x) \equiv q_{t:A}(x)$.

- If the last rule was (T-Case). Suppose $t \equiv (\mathtt{case}_A\ s\ \mathtt{zero}\ r\ \mathtt{even}\ q\ \mathtt{odd}\ u)$. The constraints on the typing rule (T-Case) ensure us that the induction hypothesis can be applied to s, r, q, u. The definition of $\Downarrow_{\mathsf{rf}}$ tells us that any derivation of $t[\overline{x}/\overline{n}]$ must have the following shape:

$$\frac{\begin{array}{cc} \rho : s[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\alpha} z & \nu : q[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\gamma} b \\ \mu : r[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\beta} a & \sigma : u[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\delta} c \end{array}}{t[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\alpha\beta\gamma\delta} (\mathtt{case}_A\ z\ \mathtt{zero}\ a\ \mathtt{even}\ b\ \mathtt{odd}\ c)}$$

Let us now define p_t and q_t as follows:

$$\begin{aligned} p_t(x) &= p_s(x) + p_r(x) + p_q(x) + p_u(x) + 1 \\ q_t(x) &= q_s(x) + q_r(x) + q_q(x) + q_u(x) + 1 \end{aligned}$$

We have:

$$\begin{aligned} |\pi| &\leq |\rho| + |\mu| + |\nu| + |\sigma| + 1 \\ &\leq p_s(\sum_i |n_i|) + p_r(\sum_i |n_i|) + p_q(\sum_i |n_i|) + p_u(\sum_i |n_i|) + 1 \\ &= p_t(\sum_i |n_i|). \end{aligned}$$

Similarly, if $z \in \pi$, it is easy to prove that $|z| \leq q_z(\sum_i |n_i|)$.

- If the last rule was (T-Rec). We consider the most interesting case, where the first term computes to a value greater than 0. Suppose $t \equiv (\mathtt{recursion}_A\ s\, r\, q)$. By looking at the typing rule (figure 4.4) for (T-Rec) we are sure to be able to apply induction

hypothesis on s, r, q. Definition of $\Downarrow_{\mathsf{rf}}$ ensure also that any derivation for $t[\overline{x}/\overline{n}]$ must have the following shape:

$$\frac{\begin{array}{c}\rho : s[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\alpha} z \quad \mu : z[\overline{x}/\overline{n}] \Downarrow_{\mathsf{nf}}^{\beta} n \\ \nu : r[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\gamma} a \\ \varrho_0 : qy[\overline{x}, y/\overline{n}, \lfloor \frac{n}{2^0} \rfloor] \Downarrow_{\mathsf{rf}}^{\gamma_0} q_0 \\ \cdots \\ \varrho_{|n|-1} : qy[\overline{x}, y/\overline{n}, \lfloor \frac{n}{2^{|n|-1}} \rfloor] \Downarrow_{\mathsf{rf}}^{\gamma_{|n|-1}} q_{|n|-1}\end{array}}{(\mathtt{recursion}_A\ s\, r\, q)[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\alpha\beta\gamma(\prod_j \gamma_j)} q_0(\ldots(q_{(|n|-1)}a)\ldots)}$$

Notice that we are able to apply $\Downarrow_{\mathsf{nf}}$ on term z because, by definition, s has only free variables of type $\Box\mathbf{N}$ (see figure 4.4). So, we are sure that z is a closed term of type $\mathbf{N}$ and we are able to apply the $\Downarrow_{\mathsf{nf}}$ algorithm.
Let define p_t and q_t as follows:

$$\begin{aligned} p_t(x) &\equiv p_s(x) + 2 \cdot q_s(x) + p_r(x) + 2 \cdot q_s(x)^2 \cdot p_q(x + 2 \cdot q_s(x)^2) + 1 \\ q_t(x) &\equiv q_s(x) + q_r(x) + 2 \cdot q_s(x)^2 + q_q(x + 2 \cdot q_s(x)^2) \end{aligned}$$

Notice that $|z|$ is bounded by $q_s(x)$. Notice that by applying theorem 4.1 on μ (z has no free variables) we have that every $v \in \mu$ is s.t.$v \leq 2 \cdot |z|^2$.
We have:

$$\begin{aligned} |\pi| &\leq |\rho| + |\mu| + |\nu| + \sum_i (|\varrho_i|) + 1 \\ &\leq p_s(\sum_i |n_i|) + 2 \cdot |z| + p_r(\sum_i |n_i|) + |n| \cdot p_{qy}(\sum_i |n_i| + |n|) + 1 \\ &\leq p_s(\sum_i |n_i|) + 2 \cdot q_s(\sum_i |n_i|) + p_r(\sum_i |n_i|) + \\ &\quad + 2 \cdot q_s(\sum_i |n_i|)^2 \cdot p_{qy}(\sum_i |n_i| + 2 \cdot q_s(\sum_i |n_i|)^2) + 1 \end{aligned}$$

Similarly, for every $w \in \pi$:

$$\begin{aligned} |w| &\leq q_s(\sum_i |n_i|) + 2 \cdot q_s(\sum_i |n_i|)^2 + q_r(\sum_i |n_i|) + q_{qy}(\sum_i |n_i| + |n|) \\ &\leq q_s(\sum_i |n_i|) + 2 \cdot q_z(\sum_i |n_i|)^2 + q_r(\sum_i |n_i|) + q_{qy}(\sum_i |n_i| + 2 \cdot q_s(\sum_i |n_i|)^2) \end{aligned}$$

- In the following cases the last rule is (T-Arr-E).
- $t \equiv x\overline{s}$. In this case, obviously, the free variable x has type $\blacksquare A_i$ $(1 \leq i \leq j)$. By definition x is negatively $\square$-free. This it means that every term in $\overline{s}$ has a type that is positively $\square$-free. By knowing that the type of x is negatively $\square$-free, we conclude that the type of our term t is $\square$-free (because is both negatively and positively $\square$-free at the same time).

  Definition of $\Downarrow_{\mathsf{rf}}$ ensures us that the derivation will have the following shape:

$$\frac{\rho_i : s_j[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\alpha_j} r_j}{x\overline{s}[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\prod_i \alpha_i} x\overline{r}}$$

  We define p_t and q_t as:

$$p_t(x) \equiv \sum_j p_{s_j}(x) + 1$$
$$q_t(x) \equiv \sum_j q_{s_j}(x) + 1$$

  Indeed we have

$$\begin{aligned} |\pi| &\leq \sum_j |\rho_j| + 1 \\ &\leq \sum_j \{p_{s_j}(\sum_i |n_i|)\} + 1 \end{aligned}$$

  Similarly, if $z \in \pi$, it is easy to prove that $|z| \leq q_z(\sum_i |n_i|)$.
- If $t \equiv \mathtt{S}_0 s$, then s have type $\mathbf{N}$ in the context Γ. The derivation π has the following form

$$\frac{\rho : s[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\alpha} z}{\mathtt{S}_0 s[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\alpha} \mathtt{S}_0 z}$$

  Define $p_t(x) = p_s(x) + 1$ and $q_t(x) = q_s(x) + 1$. One can easily check that, by induction

hypothesis

$$|\pi| \leq |\rho| + 1 \leq p_s(\sum_i |n_i|) + 1$$
$$= p_t(\sum_i |n_i|).$$

Analogously, if $r \in \pi$ then

$$|s| \leq q_s(\sum_i |n_i|) + 1 \leq q_t(\sum_i |n_i|).$$

- If $t \equiv \mathtt{S}_1 s$ or $t \equiv \mathtt{P}s$, then we can proceed exactly as in the previous case.
- Cases where we have on the left side a case or a recursion with some arguments, is trivial: can be brought back to cases that we have considered.
- If t is $(\lambda x : \Box\mathbf{N}.s)r\overline{q}$, then we have the following derivation:

$$\frac{\begin{array}{c}\rho : r[\overline{x}/\overline{n}] \Downarrow^{\alpha}_{\mathsf{rf}} a \\ \mu : a[\overline{x}/\overline{n}] \Downarrow^{\gamma}_{\mathsf{nf}} n \qquad \nu : (s[x/n])\overline{q}[\overline{x}/\overline{n}] \Downarrow^{\beta}_{\mathsf{rf}} v\end{array}}{(\lambda x : \Box\mathbf{N}.s)r\overline{q}[\overline{x}/\overline{n}] \Downarrow^{\alpha\gamma\beta}_{\mathsf{rf}} v}$$

By hypothesis t is positively $\Box$-free and so also r (whose type is $\mathbf{N}$) and $s\overline{q}$ are positively $\Box$-free. So, we are sure that we are able to use induction hypothesis.
Let p_t and q_t be:

$$p_t(x) \equiv p_r(x) + 2 \cdot q_r(x) + p_{s\overline{q}}(x + 2 \cdot q_r(x)^2) + 1$$
$$q_t(x) \equiv q_{s\overline{q}}(x + 2 \cdot q_r(x)^2) + q_r(x) + 2 \cdot q_r(x)^2 + 1$$

We have:

$$\begin{aligned}|\pi| &\equiv |\rho| + |\mu| + |\nu| + 1 \\ &\leq p_r(\sum_i |n_i|) + 2 \cdot |a| + p_{s\overline{q}}(\sum_i |n_i| + |n|) + 1 \\ &\leq p_r(\sum_i |n_i|) + 2 \cdot q_r(\sum_i |n_i|) + p_{s\overline{q}}(\sum_i |n_i| + 2 \cdot q_r(\sum_i |n_i|)^2) + 1\end{aligned}$$

By construction, remember that s has no free variables of type $\blacksquare\mathbf{N}$. For theorem 4.1 (z has no free variables) we have $v \in \mu$ is s.t. $|v| \leq 2 \cdot |a|^2$. By applying induction

hypothesis we have that every $v \in \rho$ is s.t. $|v| \leq q_r(\sum_i |n_i|)$, every $v \in \nu$ is s.t.

$$\begin{aligned} |v| &\leq q_{s\overline{q}}(\sum_i |n_i| + |n|) \\ &\leq q_{s\overline{q}}(\sum_i |n_i| + 2 \cdot |a|^2) \\ &\leq q_{s\overline{q}}(\sum_i |n_i| + 2 \cdot q_r(\sum_i |n_i|)^2) \end{aligned}$$

We can prove the second point of our thesis by setting $q_t(\sum_i |n_i|)$ as $q_{s\overline{q}}(\sum_i |n_i| + 2 \cdot q_r(\sum_i |n_i|)^2) + q_r(\sum_i |n_i|) + 2 \cdot q_r(\sum_i |n_i|)^2 + 1$.

- If t is $(\lambda x : \blacksquare\mathbf{N}.s)r\overline{q}$, then we have the following derivation:

$$\frac{\rho : r[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\alpha} a \qquad \mu : a[\overline{x}/\overline{n}] \Downarrow_{\mathsf{nf}}^{\gamma} n \qquad \nu : s\overline{q}[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\beta} u}{(\lambda x : \blacksquare\mathbf{N}.s)r\overline{q}[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\alpha\gamma\beta} (\lambda x : \blacksquare\mathbf{N}.u)n}$$

By hypothesis we have t that is positively $\Box$-free. So, also r and a (whose type is $\mathbf{N}$) and $s\overline{q}$ are positively $\Box$-free. We define p_t and q_t as:

$$\begin{aligned} p_t(x) &\equiv p_r(x) + 2 \cdot q_r(x) + p_{s\overline{q}}(x) + 1; \\ q_t(x) &\equiv q_r(x) + 2 \cdot q_r(x)^2 + q_{s\overline{q}}(x) + 1. \end{aligned}$$

We have:

$$\begin{aligned} |\pi| &\equiv |\rho| + |\mu| + |\nu| + 1 \\ &\leq p_r(\sum_i |n_i|) + 2 \cdot q_r(\sum_i |n_i|) + p_{s\overline{q}}(\sum_i |n_i|) + 1 \end{aligned}$$

Similarly, if $z \in \pi$, it is easy to prove that $|z| \leq q_t(\sum_i |n_i|)$.

- If t is $(\lambda x : aH.s)r\overline{q}$, then we have the following derivation:

$$\frac{\rho : (s[x/r])\overline{q}[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\beta} v}{(\lambda x : aH.s)r\overline{q}[\overline{x}/\overline{n}] \Downarrow_{\mathsf{rf}}^{\beta} v}$$

By hypothesis we have t that is positively $\Box$-free. So, also $s\overline{q}$ is positively $\Box$-free. r has an higher-order type H and so we are sure that $|(s[x/r])\overline{q}| < |(\lambda x : aH.s)r\overline{q}|$. Define p_t and q_t as:

$$\begin{aligned} p_t(x) &\equiv p_{(s[x/r])\overline{q}}(x) + 1; \\ q_t(x) &\equiv q_{(s[x/r])\overline{q}}(x) + 1. \end{aligned}$$

By applying induction hypothesis we have:

$$|\pi| \equiv |\rho| + 1 \leq p_{(s[x/r])\overline{q}}(\sum_i |n_i|) + 1$$

By using induction we are able also to prove the second point of our thesis.
This concludes the proof. □

Following the definition of $\Downarrow$, it is quite easy to obtain, given a first order term t, of arity k, a probabilistic Turing machine that, when receiving on input (an encoding of) $n_1 \dots n_k$, produces on output m with probability equal to $\mathscr{D}(m)$, where $\mathscr{D}$ is the (unique!) distribution such that $t \rightsquigarrow \mathscr{D}$. Indeed, $\Downarrow_{\mathsf{rf}}$ and $\Downarrow_{\mathsf{nf}}$ are designed in a very algorithmic way. Moreover, the obtained Turing machine works in polynomial time, due to propositions 4.1 and 4.2. Formally:

Theorem 4.4 (Soundness) *Suppose t is a first order term of arity k. Then there is a probabilistic Turing machine M_t running in polynomial time such that M_t on input $n_1 \dots n_k$ returns m with probability exactly $\mathscr{D}(m)$, where $\mathscr{D}$ is a probability distribution such that $tn_1 \dots n_k \rightsquigarrow \mathscr{D}$.*

Proof: By propositions 4.1 and 4.2. □

4.8 Probabilistic Polytime Completeness

In this section, we prove that any probabilistic polynomial time Turing machine (PPTM in the following) can be encoded in RSLR. The encoding works in similar way as the one done in 3.4.4. We still need to extend types with pair of base types. Natural numbers, strings, and everything need for the encoding are exactly the same described in 3.4.5 and following sections.

4.9 Probabilistic Turing Machines

Let M be a probabilistic Turing machine $M = (Q, q_0, F, \Sigma, \sqcup, \delta)$, where Q is the finite set of states of the machine; q_0 is the initial state; F is the set of final states of M; Σ is the finite alphabet of the tape; $\sqcup \in \Sigma$ is the symbol for empty string; $\delta \subseteq (Q \times \Sigma) \times (Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\})$ is the transition function of M. For each pair $(q, s) \in Q \times \Sigma$, there are exactly two triples

Example 4.2 *Let's see now an example about how the two machines $\Downarrow_{\mathsf{rf}}$ and $\Downarrow_{\mathsf{nf}}$ works. Suppose to have the following t term:*

$$(\lambda z : \blacksquare\mathbf{N}.\lambda h : \square\mathbf{N}.\mathtt{recursion}_{\mathbf{N}}\ z\, h\, (\lambda x : \square\mathbf{N}.(\lambda y : \blacksquare\mathbf{N}.\mathtt{case}_{\blacksquare\mathbf{N}\to\mathbf{N}}\ \mathtt{rand}\ \mathtt{zero}\ \mathtt{S}_1\ \mathtt{even}\ \mathtt{S}_1\ \mathtt{odd}\ \mathtt{S}_0)y))(10)(1110)$$

For simplify reading let define:

- Be $g \equiv (\mathtt{case}_{\blacksquare\mathbf{N}\to\mathbf{N}}\ \mathtt{rand}\ \mathtt{zero}\ \mathtt{S}_1\ \mathtt{even}\ \mathtt{S}_1\ \mathtt{odd}\ \mathtt{S}_0)$.
- Be $f \equiv \lambda x : \square\mathbf{N}.\lambda y : \blacksquare\mathbf{N}.(\mathtt{case}_{\blacksquare\mathbf{N}\to\mathbf{N}}\ \mathtt{rand}\ \mathtt{zero}\ \mathtt{S}_1\ \mathtt{even}\ \mathtt{S}_1\ \mathtt{odd}\ \mathtt{S}_0)y$.

$$\pi : \dfrac{\dfrac{\mathtt{S}_1 \Downarrow^1_{\mathsf{rf}} \mathtt{S}_1 \qquad \mathtt{rand} \Downarrow^1_{\mathsf{rf}} \mathtt{rand}}{\mathtt{S}_0 \Downarrow^0_{\mathsf{rf}} \mathtt{S}_0 \qquad \mathtt{S}_1 \Downarrow^1_{\mathsf{rf}} \mathtt{S}_1 \qquad y \Downarrow^1_{\mathsf{rf}} y} \quad \rho_0 : \dfrac{1110 \Downarrow^1_{\mathsf{rf}} 1110}{1110 \Downarrow^1_{\mathsf{nf}} 1110 \qquad \pi : \lambda y : \blacksquare\mathbf{N}.gy \Downarrow^1_{\mathsf{rf}} \lambda y : \blacksquare\mathbf{N}.gy}}{f1110 \Downarrow^1_{\mathsf{rf}} \lambda y : \blacksquare\mathbf{N}.gy}$$

$$\rho_1 : \dfrac{111 \Downarrow^1_{\mathsf{rf}} 111}{111 \Downarrow^1_{\mathsf{nf}} 111 \qquad \pi : \lambda y : \blacksquare\mathbf{N}.gy \Downarrow^1_{\mathsf{rf}} \lambda y : \blacksquare\mathbf{N}.gy}{f111 \Downarrow^1_{\mathsf{rf}} \lambda y : \blacksquare\mathbf{N}.gy} \qquad \rho_3 : \dfrac{11 \Downarrow^1_{\mathsf{rf}} 11}{11 \Downarrow^1_{\mathsf{nf}} 11 \qquad \pi : \lambda y : \blacksquare\mathbf{N}.gy \Downarrow^1_{\mathsf{rf}} \lambda y : \blacksquare\mathbf{N}.gy}{f11 \Downarrow^1_{\mathsf{rf}} \lambda y : \blacksquare\mathbf{N}.gy} \qquad \rho_4 : \dfrac{1 \Downarrow^1_{\mathsf{rf}} 1}{1 \Downarrow^1_{\mathsf{nf}} 1 \qquad \pi : \lambda y : \blacksquare\mathbf{N}.gy \Downarrow^1_{\mathsf{rf}} \lambda y : \blacksquare\mathbf{N}.gy}{f1 \Downarrow^1_{\mathsf{rf}} \lambda y : \blacksquare\mathbf{N}.gy}$$

$$\dfrac{\dfrac{1110 \Downarrow^1_{\mathsf{rf}} 1110 \qquad \dfrac{\rho_0 : f1110 \Downarrow^1_{\mathsf{rf}} \lambda y : \blacksquare\mathbf{N}.gy \qquad \rho_3 : f11 \Downarrow^1_{\mathsf{rf}} \lambda y : \blacksquare\mathbf{N}.gy \qquad \dfrac{h \Downarrow^1_{\mathsf{rf}} h}{1110 \Downarrow^1_{\mathsf{rf}} 1110}}{\rho_1 : f111 \Downarrow^1_{\mathsf{rf}} \lambda y : \blacksquare\mathbf{N}.gy \qquad \rho_4 : f1 \Downarrow^1_{\mathsf{rf}} \lambda y : \blacksquare\mathbf{N}.gy \qquad 1110 \Downarrow^1_{\mathsf{nf}} 1110}}{1110 \Downarrow^1_{\mathsf{nf}} 1110 \qquad \mathtt{recursion}_{\mathbf{N}}\ 1110\, h\, (\lambda x : \square\mathbf{N}.\lambda y : \blacksquare\mathbf{N}.(\mathtt{case}_{\blacksquare\mathbf{N}\to\mathbf{N}}\ \mathtt{rand}\ \mathtt{zero}\ \mathtt{S}_1\ \mathtt{even}\ \mathtt{S}_1\ \mathtt{odd}\ \mathtt{S}_0)y) \Downarrow^1_{\mathsf{rf}} (\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)z)))}{\lambda h : \square\mathbf{N}.\mathtt{recursion}_{\mathbf{N}}\ z\, h\, (\lambda x : \square\mathbf{N}.\lambda y : \blacksquare\mathbf{N}.(\mathtt{case}_{\blacksquare\mathbf{N}\to\mathbf{N}}\ \mathtt{rand}\ \mathtt{zero}\ \mathtt{S}_1\ \mathtt{even}\ \mathtt{S}_1\ \mathtt{odd}\ \mathtt{S}_0)y)(1110) \Downarrow^1_{\mathsf{rf}} ((\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)z))))}\ 10 \Downarrow^1_{\mathsf{nf}} 1}{\lambda h : \square\mathbf{N}.\mathtt{recursion}_{\mathbf{N}}\ z\, h\, (\lambda x : \square\mathbf{N}.\lambda y : \blacksquare\mathbf{N}.(\mathtt{case}_{\blacksquare\mathbf{N}\to\mathbf{N}}\ \mathtt{rand}\ \mathtt{zero}\ \mathtt{S}_1\ \mathtt{even}\ \mathtt{S}_1\ \mathtt{odd}\ \mathtt{S}_0)y)(10)(1110) \Downarrow^1_{\mathsf{rf}} \lambda z : \blacksquare\mathbf{N}.((\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)z))))(10)}\ 10 \Downarrow^1_{\mathsf{rf}} 1$$

Then, by applying the machine for $\Downarrow_{\mathsf{nf}}$ we could obtain the following derivation tree. Recall that, for the reason we have `rand` inside our term, there will be more than one possible derivation tree.

$$\dfrac{\dfrac{\dfrac{10 \Downarrow^1_{\mathsf{nf}} 10 \qquad \dfrac{\mathtt{rand} \Downarrow^{1/2}_{\mathsf{nf}} 1 \qquad \mathtt{S}_0 0 \Downarrow^1_{\mathsf{nf}} 100}{g(10) \Downarrow^{1/2}_{\mathsf{nf}} 100}}{(\lambda y : \blacksquare\mathbf{N}.gy)10 \Downarrow^{1/2}_{\mathsf{nf}} 100} \qquad \dfrac{\mathtt{rand} \Downarrow^{1/2}_{\mathsf{nf}} 0 \qquad \mathtt{S}_1 100 \Downarrow^1_{\mathsf{nf}} 1001}{g(100) \Downarrow^{1/2}_{\mathsf{nf}} 1001}}{(\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)10) \Downarrow^{1/4}_{\mathsf{nf}} 1001} \qquad \dfrac{\mathtt{rand} \Downarrow^{1/2}_{\mathsf{nf}} 0 \qquad \mathtt{S}_1 1001 \Downarrow^1_{\mathsf{nf}} 1001}{g(1001) \Downarrow^{1/2}_{\mathsf{nf}} 10011}}{(\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)10)) \Downarrow^{1/8}_{\mathsf{nf}} 10011} \qquad \dfrac{\mathtt{rand} \Downarrow^{1/2}_{\mathsf{nf}} 1 \qquad \mathtt{S}_0 10011 \Downarrow^1_{\mathsf{nf}} 100110}{g(10011) \Downarrow^{1/2}_{\mathsf{nf}} 100110}}{(\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)10))) \Downarrow^{1/16}_{\mathsf{nf}} 100110} \qquad 10 \Downarrow^1_{\mathsf{nf}} 10}{\lambda z : \blacksquare\mathbf{N}.((\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)((\lambda y : \blacksquare\mathbf{N}.gy)z))))(10) \Downarrow^{1/16}_{\mathsf{nf}} 100110}$$

□

(r_1, t_1, d_1) and (r_2, t_2, d_2) such that $((q, s), (r_1, t_1, d_1)) \in \delta$ and $((q, s), (r_1, t_1, d_1)) \in \delta$. Configurations of M can be encoded as follows:

$$\langle t_{left}, t, t_{right}, s\rangle : \mathbf{S}_\Sigma \times \mathbf{F}_\Sigma \times \mathbf{S}_\Sigma \times \mathbf{F}_Q,$$

where t_{left} represents the left part of the main tape, t is the symbol read from the head of M, t_{right} the right part of the main tape; s is the state of our Turing Machine. Let the type $\mathbf{C}_M$ be a shortcut for $\mathbf{S}_\Sigma \times \mathbf{F}_\Sigma \times \mathbf{S}_\Sigma \times \mathbf{F}_Q$.

Suppose that M on input x runs in time bounded by a polynomial $p : \mathbb{N} \to \mathbb{N}$. Then we can proceed as follows:

- encode the polynomial p by using function $\mathsf{encode}, \mathsf{add}, \mathsf{mult}, \mathsf{dec}$ so that at the end we will have a function $\underline{p} : \Box\mathbf{N} \to \mathbf{U}$;
- write a term $\underline{\delta} : \blacksquare\mathbf{C}_M \to \mathbf{C}_M$ which mimicks δ.
- write a term $\mathsf{init}_M : \blacksquare\mathbf{S}_\Sigma \to \mathbf{C}_M$ which returns the initial configuration for M corresponding to the input string.

The term of type $\Box\mathbf{N} \to \mathbf{N}$ which has exactly the same behavior as M is the following:

$$\lambda x : \Box\mathbf{N}.\mathsf{StoN}_\Sigma(\mathtt{recursion}_{\mathbf{C}_M}\ (\underline{p}\ x)\,(\mathsf{init}_M\ (\mathsf{NtoS}_\Sigma(x)))\,(\lambda y : \blacksquare\mathbf{N}.\lambda z : \blacksquare\mathbf{C}_M.\underline{\delta}\ z)).$$

The main difference with the encoding in 3.4.8 is the presence of `rand` operator in $\underline{\delta}$.

4.9.1 Encoding the transition function

Our $\underline{\delta}$ function will take a configuration and will produce in output a new configuration.

$\underline{\delta} \equiv \lambda x : \blacksquare\mathbf{C}.\mathbf{if}_{\mathbf{C}}\,(\mathtt{rand} = 0)$

then // here the value of rand is 0

$\mathsf{switch}_{\mathbf{C}}^{\{0,\ldots,3\}}\mathbf{BtoS}(\pi_2)$

$\mathsf{switch}_{\mathbf{C}}^{\{0,\ldots,n\}}(\pi_4 x)$ // here the value of π_2 is 0.

// Here is the case where π_4 is q_0; apply $\mathsf{shiftdx}$ or $\mathsf{shiftsx}$ or the identity on the tapes, with the relative new state, according to the original δ function.

... // here are the other cases.

$\langle \pi_1 x, \pi_2 x, \pi_3 x, \pi_4 x, \pi_5 x, \pi_6 x, q_{err}\rangle$ // default value

$\ldots$ // here are the other three cases.

$\langle \pi_1 x, \pi_2 x, \pi_3 x, \pi_4 x, \pi_5 x, \pi_6 x, q_{err} \rangle$ // default value

else // here the value of rand is 1

$\mathsf{switch}_{\mathbf{C}}^{\{0,\ldots,3\}} \mathbf{BtoS}(\pi_2)$

$\mathsf{switch}_{\mathbf{C}}^{\{0,\ldots,n\}}(\pi_4 x)$ // here the value of π_2 is 0.

// Here is the case where π_4 is q_0; apply $\mathsf{shiftdx}$ or $\mathsf{shiftsx}$ or the identity on the tapes, with the relative new state, according to the original δ function.

$\ldots$ // here are the other cases.

$\langle \pi_1 x, \pi_2 x, \pi_3 x, \pi_4 x, \pi_5 x, \pi_6 x, q_{err} \rangle$ // default value

$\ldots$ // here are the other three cases.

$\langle \pi_1 x, \pi_2 x, \pi_3 x, \pi_4 x, \pi_5 x, \pi_6 x, q_{err} \rangle$ // default value

default // default value

$\langle \pi_1 x, \pi_2 x, \pi_3 x, \pi_4 x, \pi_5 x, \pi_6 x, q_{err} \rangle$

$: \blacksquare\mathbf{C} \to \mathbf{C}$

Obviously, the definition of this function strictly depends on how the function δ of our Turing machine is made. We allow to flip a coin at each possible step. In the previous description of δ function we introduce some comments in order to make more readable the function.

We then get a faithful encoding of PPTM into RSLR, which will be useful in the forthcoming section:

Theorem 4.5 *Suppose M is a probabilistic Turing machine running in polynomial time such that for every n, $\mathscr{D}_n$ is the distribution of possible results obtained by running M on input n. Then there is a first order term t such that for every n, tn evaluates to $\mathscr{D}_n$.*

Example 4.3 *S*uppose that our probabilistic Turing machine has a δ relation such that $\delta(q_i, 1)$ is in relation between $(q_j, 0, -1)$ and $(q_k, 1, 0)$.

So, our $\underline{\delta}$ will be encoded in this way:

$\underline{\delta} \equiv \lambda x : \blacksquare\mathbf{C}.\mathbf{if}_{\mathbf{C}}\,(\mathtt{rand} = 0)$

then // here the value of rand is 0

$\mathsf{switch}_{\mathbf{C}}^{\{0,\ldots,3\}}\mathbf{BtoS}(\pi_2)$

... // three cases ahead

$\mathsf{switch}_{\mathbf{C}}^{\{0,\ldots,n\}}(\pi_4 x)$

... // k cases ahead

$\langle \pi_1 x, 1, \pi_3 x, \underline{q_k} \rangle$

...

$\langle \pi_1 x, \pi_2 x, \pi_3 x, \pi_4 x, \pi_5 x, \pi_6 x, q_{err} \rangle$

$\langle \pi_1 x, \pi_2 x, \pi_3 x, \pi_4 x, \pi_5 x, \pi_6 x, q_{err} \rangle$

else // here the value of rand is 1

$\mathsf{switch}_{\mathbf{C}}^{\{0,\ldots,3\}}\mathbf{BtoS}(\pi_2)$

... // three cases ahead

$\mathsf{switch}_{\mathbf{C}}^{\{0,\ldots,n\}}(\pi_4 x)$

... // i cases ahead

$\mathsf{shiftsx}\,\langle \pi_1 x, 0, \pi_3 x, \underline{q_j} \rangle$

...

$\langle \pi_1 x, \pi_2 x, \pi_3 x, \pi_4 x, \pi_5 x, \pi_6 x, q_{err} \rangle$

$\langle \pi_1 x, \pi_2 x, \pi_3 x, \pi_4 x, \pi_5 x, \pi_6 x, q_{err} \rangle$

default // default value

$\langle \pi_1 x, \pi_2 x, \pi_3 x, \pi_4 x, \pi_5 x, \pi_6 x, q_{err} \rangle$

□

4.10 Relations with Complexity Classes

The last two sections established a precise correspondence between RSLR and probabilistic polynomial time Turing machines. But how about probabilistic complexity *classes*, like **BPP** or **PP**? They are defined on top of probabilistic Turing machines, imposing constraints on the probability of error: in the case of **PP**, the error probability can be anywhere near $\frac{1}{2}$, but not equal to it, while in **BPP** it can be non-negligibly smaller than $\frac{1}{2}$. There are two ways RSLR can be put in correspondence with the complexity classes above, and these are explained in the following two sections.

4.10.1 Leaving the Error Probability Explicit

Of course, one possibility consists in leaving bounds on the error probability explicit *in the very definition* of what an RSLR term represents:

Definition 4.12 (Recognising a Language with Error ϵ) *A first-order term t of arity 1 recognizes a language $L \subseteq \mathbb{N}$ with probability error less than ϵ if, and only if, both:*

- *$x \in L$ and $tx \rightsquigarrow \mathscr{D}$ implies $\mathscr{D}(0) > 1 - \epsilon$.*
- *$x \notin L$ and $tx \rightsquigarrow \mathscr{D}$ implies $\sum_{s>0} \mathscr{D}(s) > 1 - \epsilon$.*

So, 0 encodes an accepting state of tx and $s > 0$ encodes a reject state of tx. Theorem 4.4, together with Theorem 4.5 allows us to conclude that:

Theorem 4.6 ($\frac{1}{2}$-Completeness for PP) *The set of languages which can be recognized with error ϵ in* RSLR *for some $0 < \epsilon \leq 1/2$ equals* **PP**.

But, interestingly, we can go beyond and capture a more interesting complexity class:

Theorem 4.7 ($\frac{1}{2}$-Completeness for BPP) *The set of languages which can be recognized with error ϵ in* RSLR *for some $0 < \epsilon < 1/2$ equals* **BPP**.

Observe how ϵ can be even equal to $\frac{1}{2}$ in Theorem 4.6, while it cannot in Theorem 4.7. This is the main difference between **PP** and **BPP**: in the first class, the error probability can very fast approach $\frac{1}{2}$ when the size of the input grows, while in the second it cannot.

The notion of recognizing a language with an error ϵ allows to capture complexity classes in RSLR, but it has an obvious drawback: the error probability remains explicit and external to the system; in other words, RSLR does not characterize *one* complexity class but many, depending on the allowed values for ϵ. Moreover, given an RSLR term t and an error ϵ, determining whether t recognizes *any* function with error ϵ is not decidable. As a consequence, theorems 4.6 and 4.7 do not suggest an enumeration of all languages in either **PP** or **BPP**. This in contrast to what happens with other ICC systems, e.g. SLR, in which all terms (of certain types) compute a function in **FP** (and, *viceversa*, all functions in **FP** are computed this way). As we have already mentioned in the Introduction, this discrepancy between **FP** and **BPP** has a name: the first is a *syntactic* class, while the second is a *semantic* class (see [3]).

4.10.2 Getting Rid of Error Probability

One may wonder whether a more implicit notion of representation can be somehow introduced, and which complexity class corresponds to RSLR this way. One possibility is taking representability by majority:

Definition 4.13 (Representability-by-Majority) *Let t be a first-order term of arity* 1. *Then t is said to* represent-by-majority *a language $L \subseteq \mathbb{N}$ iff:*

1. *If $n \in L$ and $tn \rightsquigarrow \mathscr{D}$, then $\mathscr{D}(0) \geq \sum_{m>0} \mathscr{D}(m)$;*
2. *If $n \notin L$ and $tn \rightsquigarrow \mathscr{D}$, then $\sum_{m>0} \mathscr{D}(m) > \mathscr{D}(0)$.*

There is a striking difference between Definition 4.13 and Definition 4.12: the latter is asymmetric, while the first is symmetric.

Please observe that any RSLR first order term t represents-by-majority *a* language, namely the language defined from t by Definition 4.13. It is well known that **PP** can be defined by majority itself [3], stipulating that the error probability should be *at most* $\frac{1}{2}$ when handling strings in the language and *strictly smaller than* $\frac{1}{2}$ when handling strings not in the language. As a consequence:

Theorem 4.8 (Completeness-by-Majority for PP) *The set of languages which can be represented-by-majority in* RSLR *equals* **PP**.

In other words, RSLR can indeed be considered as a tool to enumerate all functions in a complexity class, namely **PP**. It comes with no surprise, since the latter is a syntactic class.

Chapter 5

Static analyzer for complexity

In this Chapter we are going to "reverse" the main problem analysed by ICC; instead of focusing on creating a programming language with computational complexity properties, we focus on developing techniques sound and complete in order to determine if a program computes in Probabilistic Polytime Time or not.

One of the crucial problems in program analysis is to understand how much time it takes a program to complete its run. Having a bound on running time or on space consumption is really useful, specially in fields of information technology working with limited computing power. Solving this problem for every program is well known to be undecidable. The best we can do is to create an analyser for a particular complexity class able to say "yes", "no", or "don't know". Creating such an analyser can be quite easy: the one saying every time "don't know" is a static complexity analyser. The most important thing is to create one that answers "don't know" the minimum number of time as possible.

We try to combine this problem with techniques derived from Implicit Computational Complexity (ICC). ICC systems usually work by restricting the constructions allowed in a program. This *de facto* creates a small programming language whose programs all share a given complexity property (such as computing in polynomial time). ICC systems are normally **extensionally complete**: for each function computable within the given complexity bound, there exists one program in the system computing this function. They also aim at **intentional completeness**: each program computing within the bound should be recognised by the system. Full intentional completeness, however, is undecidable and ICC systems try to capture as many programs as possible (that is, answer "don't know" as little time as possible).

Having an ICC system characterising a complexity class $\mathcal{C}$ is a good starting point for developing a static complexity analyser.

There is a large literature on static analysers for complexity bounds. We develop an analysis recalling methods from [25, 8, 27]. Comparatively to these approaches our system works with a more concrete language of lists, where variables, constants and commands are defined; we are also sound and complete with respect to the Probabilistic Polynomial time complexity class (**PP**).

We introduce a probabilistic variation of the Loop language. Randomised computations are nowadays widely used and most of efficient algorithms are written using stochastic information. There are several probabilistic complexity classes and **BPP** (which stands for Bounded-error Probabilistic Polytime) is considered close to the informal notion of feasibility. Our work would be a first step into the direction of being able to capture real feasible programs solving problems in **BPP** (**BPP** $\subseteq$ **PP**).

In the following we are going to present fundamental papers on which we based our analysis. Notice also that we are no more interested in functional programming, as seen in the previous chapters. We are going to apply and study ICC techniques over imperative paradigm.

5.1 The complexity of loop programs

In 1967, Alber R.Meyer and Dennis M.Ritche published a paper called "The complexity of loop programs" [32]. In this paper the authors ask themselves if (and how) it is possible to determine an upper bound on the running time of a program. Of course theory says that this cannot be done automatically for all programs, but for particularly categories of interesting programs it can be done.

We are interested in the formalism presented by the authors. The class of programs proposed is called "loop programs" and consists of imperative programs built by using assignments and loop iteration.

Definition 5.1 *A Loop program is a finite sequence of instructions for changing non-negative integers stored in registers. Instructions are of five types:*

- $X_1 = X_2$

- $X_1 = X_1 + 1$
- $X_1 = 0$
- `loop` X_1
- `end`

where X_1, X_2 are names of some registers.

Formally, no semantics is defined but authors gives an intended meaning for each operation. The first three operations have the same meaning that they have in every imperative programming language. They are associations between a value of an expression on the right side and a register location on the left side. The previous value stored in the register is erased and a new value is associated. Notice that these are the only instructions that write values in registers.

Every sequence of instructions between a `loop` X_1 and an `end` is executed a number of times equal to the value in the register X_1. For every `loop` X_1 instruction, there is a matched `end`; otherwise, the program is not well formed. Notice also that if C is a list of commands well formed, then the algorithm 1

Algorithm 1 Scratch of a program

loop (X_1)
  C
EndLoop

iterates the list of commands C exactly X_1 number of times. Even if the commands in C alter the value of X_1, the number of iteration of the loop is not modified. So, every loop program cannot loop indefinitely and always terminates.

Example 5.1 *C*onsider the following program, where we modify the value in the register X_1 during a loop depending on X_1 itself. The program terminates.

loop (X_1)
  $X_1 = X_1 + 2$
EndLoop

If X_1 is 0 before the loop, then the value does not change after the execution of the program. If X_1 has a value greater than 0, then the execution of the program triplicates the value of X_1. Indeed, if at the beginning X_1 is 3, then at the end it has value 9. □

Meyer and Ritche present a way to give a bounding time for execution of the program. For our purpose these results are not important because the system is not working on a specific complexity class such as `P`, **L** or other classes. The system presented above is enough powerful; indeed, authors shows that there are a lot of possible functions representable with this scheme: the group of Primitive Recursive Functions.

In the following we are going to use the rules and the paradigm presented in this chapter.

5.2 Flow calculus of *MWP*-bounds

In 2009, Neil D.Jones and Lars Kristiansen [25] presented a method for certifying polynomiality of running time of imperative loop programs [32]. Authors define two relations. One is a semantic relation between a program C and a mwp-matrix $\mathbf{A}$, written $\vDash C : \mathbf{A}$. This relation says that if $\vDash C : \mathbf{A}$, then every value computed by the program C is bounded by a polynomial in the size of inputs. The latter relation is $\vdash C : \mathbf{A}$, where C is a program and $\mathbf{A}$ a mwp-matrix, holds if and only if there is a derivation, using the rules of the *MWP*-system, where the radix is $\vdash C : \mathbf{A}$.

Let's now define formally the syntax and operators in such system.

Definition 5.2 (Syntax) *Expressions and commands are defined by the following grammar:*

$$
\begin{aligned}
X \in \texttt{variable} &::= X_1 \mid X_2 \mid X_3 \ldots \\
b \in \texttt{boolean} &::= e = e \mid e \leq e \\
e \in \texttt{expression} &::= X \mid e + e \mid e \times e \\
C \in \texttt{command} &::= \texttt{skip} \mid X ::= e \mid \texttt{loop}\, X\, \{C\} \mid C; C \mid \texttt{If}\, b\, \texttt{Then}\, C\, \texttt{Else}\, C \mid \texttt{While}\, b\, \{C\}
\end{aligned}
$$

where variable X, appearing as the one controlling the loop, cannot appear inside the loop command C.

The semantics associated to every expressions and commands is the standard one, the one expected from its syntax. Variables are denoted as $X_1, X_2, X_3, \ldots$ and at any point of the program they hold a value, a natural number. The expressions are evaluated using the standard way and no side effect could appear during the execution of the program.

In the following we define a relation between commands and variables and a relation between expressions, variables and values.

Definition 5.3 *Let C_1 be a command whose variables are a subset of $\{X_1, \ldots, X_n\}$. The command execution relation*

$$||C_1||(x_1, \ldots, x_n \rightsquigarrow x'_1, \ldots, x'_n)$$

holds if the command terminates and the variables $X_1, \ldots, X_n$ having, respectively, $x_1, \ldots, x_n$ as value, take values $x'_1, \ldots, x'_n$ after the execution of the command.

This relation is quite similar to semantic definition for commands. Indeed, given a command C_1, a state of the system (the set of all the variables and their values) there is a relation between them and the output state, where each variable $X_1, \ldots, X_n$ take $x'_1, \ldots, x'_n$.

Similarly we can define a analogue relation with expressions.

Definition 5.4 *Let e_1 be an expression working with variables $X_1, \ldots, X_n$ having value, respectively, $x_1, \ldots, x_n$. The expression execution relation*

$$||e_1||(x_1, \ldots, x_n \rightsquigarrow a)$$

holds if the evaluation of the expression e_1 is a.

The reader could easily see that this definition is quite similar, as the previous one, to the definition of evaluation of an arithmetic expression in the standard semantics for imperative languages. Indeed, in a particular state, where variables $X_1, \ldots, X_n$ have value, respectively, $x_1, \ldots, x_n$, the result of the evaluation is a.

Let's see some example in order to understand the meaning of these relations.

Example 5.2 Consider the following program

$X_1 ::= 1$

$\texttt{loop}\, X_2\, \{X_1 ::= X_1 + X_1\}$

The number of iterations of the loop command depends on variable X_2. At every cycle the value of the variable X_1 is doubled. It's clear that this program requires exponential time respect to the value of the variable X_2.

We can easily say that:

$$||X_1 ::= X_1 + X_1||(x_1 \rightsquigarrow x_1')$$

implies $x_1' = 2 \cdot x_1$, that is ok because is polynomially correlated, but

$$||\texttt{loop}\, X_2\, \{X_1 ::= X_1 + X_1\}||(x_1, x_2 \rightsquigarrow x_1', x_2')$$

implies $x_1' = 2^{x_2} \cdot x_1$ that is exponentially correlated.

It is, therefore, important to keep track how flow's value from variable to variable may interfere with time computation. Should be clear that some particular patterns, as the one above, where we duplicate a value at every cycle, should not be accepted. There are a lot of patterns that should not be allowed and sometimes is not very easy to understand which of them leads to exponential blowup in time and space usage. For this reason is necessary to focus on the particular flows that occur between variables. □

5.2.1 *MWP* flows

Authors describe three kinds of flows. They are called m-flows, which stands for "maximum", w-flows, which stands for "weak" and p-flows, which stands for "polynomial".

They define a mwp-bound an expression of the following form:

$$\max(\overline{x} + p(\overline{y})) + q(\overline{z})$$

where $\overline{x}$,$\overline{y}$ and $\overline{z}$ are disjoint sets of variables and p, q two polynomials built up from positive constants and variables by applying just addition "+" and multiplication "$\cdot$". Variables in $\overline{x}$ are the ones tagged with m-flows, the one in $\overline{y}$ are the ones tagged with w-flows and finally the ones in $\overline{z}$ are the ones that express an p-flow.

Example 5.3 Here is an example on how this flow can be recognised. Consider the following program:

$\texttt{loop}\, X_3\, \{X_1 := X_1 + X_2\}$

If $||\texttt{loop}\, X_3\, \{X_1 + X_2\}||(x_1, x_2, x_3 \rightsquigarrow x'_1, x'_2, x'_3)$ holds, then we can derive some bounds for each single variable's value in output. The variables con be bound with the following inequalities:

$$x'_1 \leq x_1 + x_2 \cdot x_3$$
$$x'_2 \leq x_2$$
$$x'_3 \leq x_3$$

The bounds given above could be written in few different ways. the polynomial bound $x_1 + x_2 \cdot x_3$ could has been seen as the result of $\max\{x_1\} + (x_2 \cdot x_3)$ or $\max\{x_2 \cdot x_3\} + x_1$. The result does not change. □

In order to resolve the problem of the example, we need to introduce the algebra on which the type system is based. Type system will tell us exactly how to tag correctly variables.

5.2.2 Algebra

Definition 5.5 (Scalars) *A scalar is an element of* `Values` $= \{0, m, w, p\}$. *These elements are ordered as* $0 < m < w < p$.

We can, therefore, define the least upper bound between two elements of `Values`. As usual, let a, b be elements of `Values`: the least upper bound, written as $a + b$, is defined as a if $a \geq b$, otherwise is b.

Product of two elements of `Values`, written as $a \times b$ is defined as $a + b$ if both elements are different from 0, otherwise the result is 0. With these premises, it is easy to check that $(\texttt{Values}, +, \times)$ is a semiring.

Definition 5.6 (Vectors) *We use* $V', V'', V''', \ldots$ *to denote column vectors over* `Values`. *With* V'_i *we denote the* i*-th element of the vector.*

The least upper bound between two vectors of the same size, denoted as $V' \oplus V''$, is a new vector defined as the least upper bound componentwise. Formally $(V' \oplus V'')_i = V'_i + V''_i$ for every index i.

We can also define scalar product between a value a of `Values` and a vector V'. It is defined exactly as usual, that is, by multiplying every element of V' by a.

Definition 5.7 (Matrices) *We use* $\mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots$ *to denote squared matrices over* `Values`. *With* $\mathbf{A}_{i,j}$ *we indicate the element in the* i*-th row and* j*-th column.*

Least upper bound between two matrices of the same size is defined componentwise, as is for the vectors. It is possible also to define a partial order over this algebra of matrices. The usual symbols $\geq, \leq, >, <$ have their usual meaning and we could say that $\mathbf{A} \geq \mathbf{B}$ if and only if the relation $\geq$ is satisfied componentwise.

There are particular matrices such as the zero matrix $\mathbf{0}$, the one filled with 0 and the identity matrix $\mathbf{I}$. Finally we can define product between matrices. As usual the product is made with the row-column mode. $\mathbf{A} \otimes \mathbf{B}$ at position i, j has value $\sum_k (\mathbf{A}_{i,k} \times \mathbf{B}_{k,j})$.

Definition 5.8 (Closure Operator) *Given a matrix* $\mathbf{A}$*, an unary operator* $*$ *(called the closure operator), we define the closure operator as the infinite sum:*

$$\mathbf{I} \oplus \mathbf{A} \oplus \mathbf{A}^2 \oplus \mathbf{A}^3 \oplus \mathbf{A}^4 \oplus \ldots$$

The closure operator is well defined in every closed semiring and it is possible to prove that $\mathbf{A}^* = \mathbf{I} \oplus (\mathbf{A} \otimes \mathbf{A}^*)$.

Definition 5.9 (Substitution) *Let* $\mathbf{A}$ *be a matrix and* V' *a column vector, we indicate with* $\mathbf{A} \overset{k}{\leftarrow} V'$ *the matrix obtained by substituting the* k*-th column of* $\mathbf{A}$ *with* V'*. Formally, we can define it in the following way. Let* $\mathbf{B} = \mathbf{A} \overset{k}{\leftarrow} V'$*:*

$$\mathbf{B}_{i,j} = \begin{cases} \mathbf{A}_{i,j} & \text{if } j \neq k \\ V'_i & \text{if } i = k \end{cases}$$

5.2.3 Typing

The calculus *MWP* gives a set of rules for typing expressions and commands generated by the grammar (definition 5.2) of the system. Expressions are typed with vectors and commands are typed with matrices.

Definition 5.10 (Typing rules for expressions) *Typing rules for expressions are presented in figure 5.1. Notice that there are several ways to type an expression: there is not an unique assignment. Rule* (E1) *tells us that a variable can be labelled with value* m *but rule* (E2) *assures that we can label it also with bigger value* w. *In general, rule* (E2) *tells*

us that we could label every variable appearing in an expression with value w*. Finally, rules* (E3) *and* (E4) *state how to type a sum between two expressions. Notice that these two rules are quite similar.*

$$\frac{}{\vdash X_i : [\underbrace{0,\ldots,0}_{i-1},\mathsf{m},0,\ldots]^{\mathbf{T}}}\ (\text{E1}) \qquad \frac{}{\vdash e : V' \quad \text{s.t. } \forall X_i \in var(e_1).V'_i = \mathsf{w}}\ (\text{E2})$$

$$\frac{\vdash e_1 : V' \qquad \vdash e_2 : V''}{\vdash e_1 + e_2 : \mathsf{p}V' + V''}\ (\text{E3}) \qquad \frac{\vdash e_1 : V' \qquad \vdash e_2 : V''}{\vdash e_1 + e_2 : V' + \mathsf{p}V''}\ (\text{E4})$$

Figure 5.1: MWP typing rules for expressions

A rule for typing the multiplication between expression is not presented in the original paper, even if expressions multiplication appears in the grammar. Let's see some examples, in order to understand how typing rules work.

Example 5.4 [Typing an expression] We have seen from definition 5.10 how to type expressions. In the following we will see that it is possible to type same expressions with different types.

Consider the following expression $(X_1+X_2)+(X_3+X_4)$. One possible typing derivation is the following one:

$$\dfrac{\dfrac{\dfrac{}{\vdash X_1 : [\mathsf{m},0,0,0]^{\mathbf{T}}} \quad \dfrac{}{\vdash X_2 : [0,\mathsf{m},0,0]^{\mathbf{T}}}}{\vdash X_1 + X_2 : [\mathsf{p},\mathsf{m},0,0]^{\mathbf{T}}} \quad \dfrac{\dfrac{}{\vdash X_3 : [0,0,\mathsf{m},0]^{\mathbf{T}}} \quad \dfrac{}{\vdash X_4 : [0,0,0,\mathsf{m}]^{\mathbf{T}}}}{\vdash X_3 + X_4 : [0,0,\mathsf{p},\mathsf{m}]^{\mathbf{T}}}}{\vdash (X_1 + X_2) + (X_3 + X_4) : [\mathsf{p},\mathsf{p},\mathsf{p},\mathsf{m}]^{\mathbf{T}}}$$

but is not the only one correct. We could type the same expression in these following ways. One derivation could be:

$$\frac{}{(X_1 + X_2) + (X_3 + X_4) : [\mathsf{w},\mathsf{w},\mathsf{w},\mathsf{w}]^{\mathbf{T}}}$$

and another one could be:

$$\dfrac{\dfrac{}{\vdash X_1 + X_2 : [\mathsf{w},\mathsf{w},0,0]^{\mathbf{T}}} \quad \dfrac{\dfrac{}{\vdash X_3 : [0,0,\mathsf{m},0]^{\mathbf{T}}} \quad \dfrac{}{\vdash X_4 : [0,0,0,\mathsf{m}]^{\mathbf{T}}}}{\vdash X_3 + X_4 : [0,0,\mathsf{m},\mathsf{p}]^{\mathbf{T}}}}{\vdash (X_1 + X_2) + (X_3 + X_4) : [\mathsf{w},\mathsf{w},\mathsf{p},\mathsf{p}]^{\mathbf{T}}}$$

All of these three derivations are strongly different but they are typing the same expression. Notice also that by the rule (E2), (E3), (E4) we can easily understand that at most one variable can be typed with value m. Once one is labelled with m, all the other are labelled with p. □

We need now to introduce typing rules for commands. Every command is typed with a matrix. As the vector, for expression, was conveying information about the relation between the output value and variables appearing in the expression, each column of the matrix represents a relation between the specific variable and all the other ones.

Definition 5.11 (Typing rules for commands) *Typing rules for commands are shown in figure 5.2. Notice the side condition for the rule* (L) *expressing that the matrix associates to the command inside the loop does not have to present value greater than* m *on the main diagonal. It is a reasonable condition. Indeed, the presence of a value, in the main diagonal, greater than* m *could be a sign of a too-fast growth of some variable. The matrix* $\mathbf{A}^{\boxplus k}$ *represents an empty matrix where, for each column* j *we put a value* p *on row* k *if* $\exists i$ *such that* $\mathbf{A}^*_{i,j} = \mathsf{p}$*. This operation creates a perturbed matrix of initial matrix* $\mathbf{A}^*$*. We put a value* p *in position* k, j *to keep track that final values depends also on value of* X_k*.*

Condition on the matrix associated to the command inside the while loop tells us that the only thing we are allowed to do is a permutation of the variables' values.

$$\frac{}{\vdash \texttt{skip} : \mathbf{I}}\ (\mathrm{S}) \qquad \frac{\vdash e_1 : V'}{X_i ::= e_1 : \mathbf{I} \overset{i}{\leftarrow} V'}\ (\mathrm{A})$$

$$\frac{\vdash C_1 : \mathbf{A} \qquad \vdash C_2 : \mathbf{B}}{\vdash C_1; C_2 : \mathbf{A} \otimes \mathbf{B}}\ (\mathrm{C}) \qquad \frac{\vdash C_1 : \mathbf{A} \qquad \vdash C_2 : \mathbf{B}}{\texttt{If}\, b_1\, \texttt{Then}\, C_1\, \texttt{Else}\, C_2 : \mathbf{A} \oplus \mathbf{B}}\ (\mathrm{I})$$

$$\frac{\vdash C_1 : \mathbf{A} \qquad \forall i. \mathbf{A}^*_{i,i} = \mathsf{m}}{\texttt{loop}\, X_k\, \{C_1\} : \mathbf{A}^* \oplus \mathbf{A}^{\boxplus k}}\ (\mathrm{L})$$

$$\frac{\vdash C_1 : \mathbf{A} \qquad \forall i. \mathbf{A}^*_{i,i} = \mathsf{m} \wedge \forall ij. \mathbf{A}^*_{i,j} \neq \mathsf{p}}{\vdash \texttt{While}\, b_1\, \{C_1\} : \mathbf{A}^*}\ (\mathrm{W})$$

Figure 5.2: MWP typing rules for commands

5.2.4 Main Result

Authors provide a proof concerning polysize bound for variables' value. Notice that this does not mean that a well typed program terminates. The system does not give any bound on while-loop iteration. Authors focus to computed values during the execution and not after its termination.

Be an honest polynomial a polynomial build up from constants in **N** and variables by applying the operators + (addition) and x (multiplication). Please, note that any honest polynomial p is monotone in all its variables, i.e. we have $p(\overline{x}, y, \overline{z}) \leq p(\overline{x}, y+1, \overline{z})$ for all $\overline{x}, y, \overline{z}$.

Theorem 5.1 (Polynomial bound size of values) $\vdash C_1 : \mathbf{A}$ *implies* $\vDash C_1 : \mathbf{A}$.

The theorem says that if a command is typable in *MWP*-system, then every value computed by C_1 is bounded by a polynomial in the size of input. Let see the outline of the proof.

Proof:[Outline] Proof of soundness is proved by following these steps. We are going to avoid technical details and complete proof transcription that can be easily found in [25].

- First the following lemma is proved. If $\vdash e_1 : V'$ then $\vDash e_1 : V'$. It is proved by structural induction on the typing tree of e_1. The thesis should not be so surprising since expressions can be created by using variables, sum and multiplication.
- Then by structural induction on the command C_1, they analyse all the cases .
- For the rule (S), (A) the proof is trivial, while for rules (C), (I) by using induction hypothesis on the premises, they can create some correct *MWP*-bounds for the final command.
- The most important case is of course the one concerning the loop command. This particular thesis is proved by using a lemma, the following one.
  - Let ${C_1}^0 = \texttt{skip}$ and ${C_1}^{n+1} = {C_1}^n; C_1$. Assume that $\vDash C_1 : \mathbf{A}$ and that $\mathbf{A}^*_{i,i} = \mathsf{m}$ for all i. Then for any $j \in \{1, \ldots, m\}$ there exists a fixed number o and an honest polynomial p, q such that for any n we have:

$$||{C_1}^n||(x_1, \ldots, x_n \rightsquigarrow x'_1, \ldots, x'_n) \Rightarrow x_k \leq \max(\overline{y}, q(\overline{z})) + (n+1)^o p(\overline{h})$$

where $\overline{y} = \{x_i | \mathbf{A}^*_{i,j} = \mathsf{m}\}$ and $\overline{z} = \{x_i | \mathbf{A}^*_{i,j} = \mathsf{w}\}$ and $\overline{h} = \{x_i | \mathbf{A}^*_{i,j} = \mathsf{p}\}$. Moreover neither the polynomial p nor the polynomial q depends on n or o and if the list $\overline{h}$ is empty, then $p(\overline{h})$ is 0.

Notice how the lemma is talking about the relation $\vDash$ and not about the relation $\vdash$. However, it assumes $\vDash C_1 : \mathbf{A}$ but it adds also all the constraints needed by the typing rule (L). In this way we can easily talk about the following thesis:

$$\text{If} \vDash C_1 : \mathbf{A} \text{ and } \mathbf{A}^*_{i,i} \text{ for all } i, \text{ then } \vDash \mathtt{loop}\, X_k \, \{C_1\} : \mathbf{A}^* \oplus \mathbf{A}^{\boxplus k}$$

- A particular note has to be done for the while command. There is no upper bound on the number of times the while command loops its body. Anyhow, the side condition tells us that on the main diagonal there are just m flows and no p can appear inside the matrix. This condition means that the values of the variables inside the body of the while are iteration independent. All the datas are polynomially bounded even if the while continues indefinitely.

□

5.2.5 Indeterminacy of the calculus

In [25] the authors made some final digression about the complexity of the system. What follows are, so, the results concerned the complexity and proved by Kristiansen and Jones. As should be clear from typing rules in figure 5.1 and 5.2, the calculus is not deterministic. We can associate many different matrices to a single command. In this way, the number of possible typing tree for each single command is very big. It has been proved by the author that the problem of typing a command in *MWP*-system lays **NP**.

Theorem 5.2 (Complexity) *The derivability problem is in* **NP**.

Proof: Given a command C_1 and a matrix $\mathbf{A}$ we have to decide whether $\vdash C_1 : \mathbf{A}$ holds or not. Clearly, every single typing rule is **PTIME** decidable when the premises and the conclusion are satisfied and exists a polynomial bound on the depth and total size of any proof tree whose conclusion is $\vdash C_1 : \mathbf{A}$. Rules are compositional and so we can re-create the derivation tree by starting from the bottom and check whether the inference rules of the calculus are satisfied or not. As we have seen in figure 5.1, there are non deterministic choices about which rule we have to apply for expressions. This leads us to a blow-up on

the number of possible derivation tree; it's easy to notice that sometimes the number of possible trees grows exponentially in the size of the depth. □

In conclusion, the system presented in [25] checks whether the value of computed variables is polynomially bound respect to the value in input. We could say that *MWP*-system is sound for **PSPACE**, but this is not completely true seeing that it could type also programs that runs indefinitely. No theorem of soundness respect to a complexity class is presented in the paper.

Chapter 6

Imperative Static Analyzer for Probabilistic Polynomial Time

Our system is called **iSAPP**, which stands for *Imperative Static Analyser for Probabilistic Polynomial Time*. It works on a prototype of imperative programming language based on the LOOP language [32]. The reader will immediately notice the lack of a `While` command (thus, the programs in our language are restricted to compute Primitive Recursive functions even before we start our analyse). The main purpose of this paper is to present a minimal probabilistic polytime certifier for imperative programming languages. **iSAPP** can be extended with `While`, by putting constraints on the guard and on the body of `While`.

Following ideas from [25, 8] we "type" commands with matrices and expressions with vectors. The underlying idea is that these matrices express a series of polynomials bounding the size of variables, with respect to their input size. The algebra on which these matrices and vectors are based is a finite (more or less tropical) semi-ring.

iSAPP works on lists (or on stacks). Arithmetic is done using the size of lists. For example, $e + e$ actually stands for $\texttt{concat}(e, e)$. Thus, we bound the *size* of the lists handled by the program, even is these could represent larger numbers (*e.g.* using binary lists for integers). Similarly, iterations are controlled by the size of the list. Thus, our `loop` command is closer to a `foreach` (element in the list) than to a `for` (going up to the value represented by the list).

6.1 Syntax

The underlying system is based on lists of numbers. We denoted lists with $\langle\rangle$ (in this case it is an empty list) and terms of lists with letters t, s, r. **iSAPP** works independently with respect to list implementation.

Definition 6.1 *Terms, constant and commands are defined as follows:*

$$
\begin{aligned}
c \in \texttt{Constant} &::= \langle\rangle \mid \langle t\rangle \mid \langle ts\rangle \mid \dots \\
X \in \texttt{variable} &::= X_1 \mid X_2 \mid X_3 \dots \\
b \in \texttt{boolean} &::= e = e \mid e \leq e \mid \neg b \mid b \wedge b \mid b \vee b \mid \mathsf{true} \mid \mathsf{false} \mid \texttt{rand} \mid \texttt{testzero}\{e\} \\
e \in \texttt{expression} &::= c \mid X \mid \texttt{concat}(e, e) \mid \texttt{tail}(e) \mid \texttt{head}\{e\} \\
C \in \texttt{command} &::= \texttt{skip} \mid X ::= e \mid \texttt{loop}\, X\, \{C\} \mid C; C \mid \texttt{If}\, b\, \texttt{Then}\, C\, \texttt{Else}\, C
\end{aligned}
$$

Focusing on expressions, the reader will notice that no sum or subtraction is defined. While working on lists, $\texttt{concat}(e, e)$ is the formal way to express $e + e$ and $\texttt{tail}(e)$ stands for $e - 1$. Indeed when needed (*e.g.* to simulate TMs), we use the length of the lists to encode values. In the following we will use $+, -$ instead of $\texttt{concat}(,), \texttt{tail}()$ where the list-representation is not necessary.

Constants are so encoded in the following way: 0 is $\langle\rangle$, 1 is $\langle t\rangle$, 2 is $\langle ts\rangle$, 3 is $\langle tsr\rangle$ and so on. $\texttt{testzero}\{e\}$ tests whether e is a list starting with 0. Indeed, 0 may be part of language of letters.

iSAPP proposes the command `loop` $\{\}$, instead of well known `while` or `for` commands. The command loop is semantically quite similar to `for` command. Informally, it is a controlled loop, where the number of iteration cannot be changed during loop execution.

6.2 Algebra

Before going deeply in explaining our system, we need to present the algebra on which it is based. **iSAPP** is based on a finite algebra of values. Set of possible values is $\texttt{Values} = \{0, L, A, M\}$ and these are ordered in the following way $0 < L < A < M$. The idea behind these elements is to express how the value of variables influences the result

$\times$	0	L	A	M
0	0	0	0	0
L	0	L	A	M
A	0	A	A	M
M	0	M	M	M

$+$	0	L	A	M
0	0	L	A	M
L	L	A	A	M
A	A	A	A	M
M	M	M	M	M

$\cup$	0	L	A	M
0	0	L	A	M
L	L	L	A	M
A	A	A	A	M
M	M	M	M	M

Table 6.1: Multiplication, addition and union of values

of an expression. 0 expresses no-dependency between variable and result; L (stands for "Linear") expresses that the result linearly depends with coefficient 1 from this variable. A (stands for "Additive") expresses the idea of generic linear dependency. M (stands for "Multiplicative") expresses the idea of generic polynomial dependency.

We are using a slight different algebra respect to the one presented in [24]. This choice has been made in order to get more expressive informations from a "certificate". We pur more emphasis on how much each variable appears during the exectution. In this way we can (if exists) easily extract a polynomial bounding the time exectution and size of variables.

Let's see one example. Suppose we are working with six variables and we have the following expression: $X_1 + 3X_2 + 4X_3 + 5X_4^2X_5^3$. We associate 0 to variable X_6, L to X_1, A to X_2 and X_3 and finally M to X_4 and X_5. This abstracts the polynomial by only keeping the way its result depends on each variable.

We define sum, multiplication and union in our algebra as expressed in Table 6.1 and we can easily check that $(\texttt{Values}, +, \times)$ is a semi-ring. The reader will immediately notice that $L + L$ gives A, while $L \cup L$ gives L. This is the only difference between the two operations; The operator $\cup$ works as a maximum.

Over this semi-ring we create a module of matrices, where values are elements of `Values`. We define a partial order $\leq$ between matrices of the same size as component wise ordering. Particular matrices are $\mathbf{0}$, the one filled with all 0, and $\mathbf{I}$, the identity matrix, where elements of the main diagonal are L and all the others are 0.

Multiplication and addition between matrices work as usual[1] and we define point-wise union between matrices: $(\mathbf{A} \cup \mathbf{B})_{i,j} = \mathbf{A}_{i,j} \cup \mathbf{B}_{i,j}$. Notice that $\mathbf{A} \cup \mathbf{B} \leq \mathbf{A} + \mathbf{B}$. As usual,

[1] That is: $(\mathbf{A} + \mathbf{B})_{i,j} = \mathbf{A}_{i,j} + \mathbf{B}_{i,j}$ and $(\mathbf{A} \times \mathbf{B})_{i,j} = \sum \mathbf{A}_{i,k} \times \mathbf{B}_{k,j}$

multiplication between a value and a matrix corresponds to multiplying every element of the matrix by that value.

We can easily check associativity of matrix multiplication and distributivity with respect to sum. We define power of matrices as usual: $\mathbf{A}^n$ is $\mathbf{A}^{n-1} \times \mathbf{A}$, where $\mathbf{A}^0$ is $\mathbf{I}$.

We can now move on and present some new operators and properties of matrices. Given a column vector V of dimension n, a matrix $\mathbf{A}$ of dimension $n \times m$ an index i ($i \leq m$), we indicate with $\mathbf{A} \overset{i}{\leftarrow} V$ a substitution of the i-th column of the matrix $\mathbf{A}$ with the vector V.

Next, we need two closure operators. The "union closure" is the union of all powers of the matrix: $\mathbf{A}^{\cup} = \bigcup_{i \geq 0} \mathbf{A}^i$. It is always defined because the set of possible matrices is finite. The "multiplication closure" is the smallest fixed point in the chain of powers: $\mathbf{A}^*$ is $\mathbf{A}^n$ such that $\mathbf{A}^n \times \mathbf{A} = \mathbf{A}^n$ and n is the smallest integer verifying this property. Multiplication closure is not always defined as the chain of powers can loop over different values. Of course $\mathbf{A}^*$ is contained in $\bigcup_{i \geq 0} \mathbf{A}^i$, but the union closure is always defined. Indeed if $\mathbf{A}^*$ exists, then clearly $\mathbf{A}^{\cup}$ is defined. If $\mathbf{A}^*$ is not defined because the chain of powers loop, then $\mathbf{A}^{\cup}$ is still defined because it contains also the union of all of these chain of powers. So, the union exists.

Consider for example $\mathbf{A} = \begin{bmatrix} 0 & 0 & L \\ L & 0 & 0 \\ 0 & L & 0 \end{bmatrix}$. It is easy to check that $\mathbf{A}^*$ is not defined while $\mathbf{A}^{\cup} = \begin{bmatrix} L & L & L \\ L & L & L \\ L & L & L \end{bmatrix}$.

All of these operators are implemented in a program that can be found in [37].

Finally, we'll need a "merge down" operator. Its use is to propagate the influence of some variables to some other and it is used to correctly detect the influence of variables controlling loops onto variables modified within the loop (hence, we can also call it "loop correction"). The last row and column of the matrix is treated differently because it will be use to handle constants and not variables.

Merge down is applied independently on each column of the matrix, thus we will describe the merge down of a column-vector. To understand the idea behind this, consider a polynomial P over $n-1$ variables, the dependency of the result to these variables can be expressed as a n-large vector over our semi-ring (the last component being used for constants). The idea of merge down is to find the dependency of iterating this polynomial

X_k times (and putting the result into X_j each time (that is, iterating the assignment $X_j ::= P$).

- $(V^{\downarrow k,j})_n = V_n \qquad (V^{\downarrow k,j})_j = V_j \qquad (V^{\downarrow k,j})_i = V_i$ if $V_j = 0$
- $(V^{\downarrow k,j})_k = \begin{cases} M & \text{if } \exists k < n, k \neq k \text{ such that } V_k \neq 0 \\ A & \text{otherwise and } V_n \neq 0 \\ V_k & \text{otherwise} \end{cases}$
- $(V^{\downarrow k,j})_i = \begin{cases} M & \text{if } V_i \neq 0 \text{ and } V_j \neq 0 \\ V_i & \text{otherwise} \end{cases}$

The first cases say that the last line (corresponding to constant) and the j-th line ("diagonal") should not be modified ; similarly, if the j-th line ("diagonal") is empty, the vector should not be modified. The second case explains what happens to the line controlling the merge down: it is replaced by a M (resp. A) if there is another variable (resp. a constant) influencing the result. The last case "upgrades" any non-0 value to M (if the j-th line is also non-0).

Let's explain the cases with examples. Consider the polynomial $X_2 + X_3$. As stated earlier, we can associate 0 to X_1 and constants, and L to X_2 and X_3. This corresponds to the vector $V = \begin{bmatrix} 0 \\ L \\ L \\ 0 \end{bmatrix}$. Now, suppose we iterate X_1 times the assignment $X_2 ::= X_2 + X_3$. The resulting effect is $X_2 ::= X_2 + X_3 \times X_1$. This polynomial can be associated with the vector $\begin{bmatrix} M \\ L \\ M \\ 0 \end{bmatrix}$ which is exactly $V^{\downarrow 1,2}$. This shows why both the "merge down" and "other cases" lines must be replaced by M.

Next, if we iterate X_1 times the assignment $X_3 ::= X_3+2$, then the result is $X_3 ::= X_3+ 2 \times X_1$. Similarly, we have $\begin{bmatrix} 0 \\ 0 \\ L \\ A \end{bmatrix}^{\downarrow 1,3} = \begin{bmatrix} A \\ 0 \\ L \\ A \end{bmatrix}$.

Lastly, iterating $X_j ::= P$ when X does not appear in P does not change the result, thus if there is a 0 on the j-th line, we should not change the vector.

Note that the merge down will not correspond directly to iteration but to some correction applied after iteration. So the previous examples are only hints at why we need it that way.

For matrices, $\mathbf{A}^{\downarrow k}$ is obtained by replacing each column $\mathbf{A}_j$ by ${\mathbf{A}_j}^{\downarrow k,j}$.

6.3 Multipolynomials and abstraction

Following [31], we use *multipolynomials* to represents several bounds in one object. These multipolynomials are abstracted as matrices.

First we need to introduce the concept of abstraction of polynomial. Abstraction gives a vector representing the shape of our polynomial and how variables appear inside it.

Definition 6.2 (Abstraction of polynomial) *Let $p(\overline{X})$ a polynomial over n variables, $\lceil p(\overline{X}) \rceil$ is a column vector of size $n+1$ such that:*

- *If $p(\overline{X})$ is a constant c, then $\lceil p(\overline{X}) \rceil$ is $[0, \ldots, 0, L]^{\mathbf{T}}$*
- *Otherwise if $p(\overline{X})$ is X_i, then $\lceil p(\overline{X}) \rceil$ is $[\underbrace{0, \ldots, 0}_{i-1}, L, 0, \ldots, 0]^{\mathbf{T}}$.*
- *Otherwise if $p(\overline{X})$ is αX_i (for some constant $\alpha > 1$), then $\lceil p(\overline{X}) \rceil$ is $[\underbrace{0, \ldots, 0}_{i-1}, A, 0, \ldots, 0]^{\mathbf{T}}$.*
- *Otherwise if $p(\overline{X})$ is $q(\overline{X}) + r(\overline{X})$, then $\lceil p(\overline{X}) \rceil$ is $\lceil q(\overline{X}) \rceil + \lceil r(\overline{X}) \rceil$.*
- *Otherwise, $p(\overline{X})$ is $q(\overline{X}) \cdot r(\overline{X})$ (where none of the polynomials is a constant), then $\lceil p(\overline{X}) \rceil$ is $M \cdot \lceil q(\overline{X}) \rceil \cup M \cdot \lceil r(\overline{X}) \rceil$.*

Size of vectors is $n+1$ because n cells are needed for keeping track of n different variables and the last cell is the one associated to constants. We can now introduce multipolynomials and their abstraction.

Definition 6.3 *A multipolynomial is a tuple of polynomials. Formally $P = (p_1, \ldots, p_n)$, where each p_i is a polynomial.*

Abstracting a multipolynomial naturally gives a matrix where each column is the abstraction of one of the polynomials.

Definition 6.4 *Let $P = (p_1, \ldots, p_n)$ be a multipolynomial, its abstraction $\lceil P \rceil$ is a matrix where the i-th column is the vector $\lceil p_i \rceil$.*

In the following, we use polynomials to bound size of single variables. Since handling polynomials is too hard (*i.e.* undecidable), we only keep their abstraction. Similarly, we use multipolynomials to bound the size of all the variables of a program at once. Again, rather than handling the multipolynomials, we only work with their abstractions.

Definition 6.5 (Extraction of set from polynomial) *Given polynomials p, q, we define $\lceil(p)$ in the following way. $\lceil(n)$ is $\{n\}$; $\lceil(X)$ is $\{X\}$; $\lceil(p+q)$ is $\lceil(p) \cup \lceil(q)$; $\lceil(p \cdot q)$ is $\{p \cdot q\}$*

Main purpose of the operator $\lceil(p)$ is to create a set representation of the polynomial, erasing some duplication. The next operator implements the inverse operation, creating a polynomial from a set of terms. Of course, these terms have to be positive polynomials.

Example 6.1 *L*et's see how the extraction operator works. $\lceil(X+X)$ is $\{X\}$, while $\lceil(2X)$ is exactly $\{X\}$. This is correct and it is deliberate. This operation is needed just for capture the shape of a polynomial and it is strictly connected with the union between values of our algebra. □

Definition 6.6 (From set to polynomial) *Be $\downarrow(\cdot)$ the operator such that $\downarrow(M)$ is M and $\downarrow(M_1, \ldots, M_n)$ is $M_1 + \ldots + M_n$.*

Should be clear that if we combine both operator we obtain a new polynomial where some equal terms are erased. This is exactly what we need in order to create a approximation of concept of union of matrices.

Definition 6.7 (Union of polynomial) *We define the operator $(p \oplus q)$ over polynomials in the following way. Be $\overline{p}$ the canonical form of the polynomial p. We have $(p \oplus q) = \overline{\downarrow(\lceil(\overline{p}) \cup \lceil(\overline{q}))}$*

So, first we take the normal form of inputs, then we extract the set representing the monomials. Union of sets will erase duplication of equal monomials. We recreate a polynomial from the set and finally we calculate its canonical form. Let's see some example. Suppose we have these two polynomials: $X_1 + 2X_2 + 3X_4^2X_5$ and $X_1 + 3X_2 + 3X_4^2X_5 + X_6$. Call them, respectively p and q. We have that $(p \oplus q)$ is $X_1 + 5X_2 + 3X_4^2X_5 + X_6$.

We can now introduce the composition and sum between abstractions of polynomials.

Definition 6.8 *Given two multipolynomials P and Q over the same set of variables, we define addition in the following way: $(P+Q)_i = P_i + Q_i$.*

Definition 6.9 *Given two multipolynomials P and Q over the same set of variables, we define composition in the following way: $(P \odot Q)(X_1, \ldots, X_n) = Q(P_1(\overline{X}), \ldots, P_n(\overline{X}))$.*

6.4 Semantics

The semantics of **iSAPP** is similar to the standard one of imperative programming languages. We use the following notation $\sigma[n/x]$ to intend a function that on input x returns n, otherwise it behaves as σ.

All the cases are defined in figure 6.1, 6.2, 6.3 where $\rightarrow_c^{\alpha}, \rightarrow_a, \rightarrow_b^{\alpha}$ express, respectively, semantics of commands, semantics of arithmetic expressions and semantics of boolean expressions.

$$\frac{}{\langle X, \sigma\rangle \rightarrow_a \sigma(X)} \qquad \frac{}{\langle l_1, \sigma\rangle \rightarrow_a l_1}$$

$$\frac{\langle e_1, \sigma\rangle \rightarrow_a \langle \bar{t}\rangle \qquad \langle e_2, \sigma\rangle \rightarrow_a \langle \bar{s}\rangle}{\langle \texttt{concat}(e_1, e_2), \sigma\rangle \rightarrow_a \langle \bar{t}\bar{s}\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \rightarrow_a \langle \bar{t}s\rangle}{\langle \texttt{tail}(e_1), \sigma\rangle \rightarrow_a \langle \bar{t}\rangle} \qquad \frac{\langle e_1, \sigma\rangle \rightarrow_a \langle\rangle}{\langle \texttt{tail}(e_1), \sigma\rangle \rightarrow_a \langle\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \rightarrow_a \langle \bar{t}s\rangle}{\langle \texttt{head}\{e_1\}, \sigma\rangle \rightarrow_a \langle s\rangle} \qquad \frac{\langle e_1, \sigma\rangle \rightarrow_a \langle\rangle}{\langle \texttt{head}\{e_1\}, \sigma\rangle \rightarrow_a \langle\rangle}$$

Figure 6.1: Semantics of arithmetic expressions

Semantics for boolean value is labelled with probability. As expected, most of boolean operator have probability 1, while operator `rand` reduced to true or false with probability $\frac{1}{2}$. For this reason, also semantics of commands is labelled with a probability. It tells us the probability to reach a particularly final state after having executed a command from an initial state.

The most interesting semantics is the one for command $\texttt{loop}\, X_k\, \{C\}$. Semantics tells us that if the value of the variable controlling the loop is zero (a shortcut for empty list), then the loop is not executed. Otherwise we execute the loop a number of times equal to the length of the list representing the value inside the variable X_k. Of course, the final probability associated to the loop is the product of all the probabilities associated to each command C iteration.

$$\frac{s \text{ is } 0}{\texttt{testzero}\{\langle \bar{t}s \rangle\} \to_b^1 \mathsf{true}} \qquad \frac{s \text{ is not } 0}{\texttt{testzero}\{\langle \bar{t}s \rangle\} \to_b^1 \mathsf{false}} \qquad \frac{}{\texttt{testzero}\{\langle\rangle\} \to_b^1 \mathsf{false}}$$

$$\frac{}{\langle \texttt{rand}, \sigma \rangle \to_b^{\frac{1}{2}} \mathsf{true}} \qquad \frac{}{\langle \texttt{rand}, \sigma \rangle \to_b^{\frac{1}{2}} \mathsf{false}}$$

$$\frac{}{\langle \mathsf{true}, \sigma \rangle \to_b^1 \mathsf{true}} \qquad \frac{}{\langle \mathsf{false}, \sigma \rangle \to_b^1 \mathsf{false}}$$

$$\frac{\langle b_1, \sigma \rangle \to_b^\alpha \mathsf{true}}{\langle \neg b_1, \sigma \rangle \to_b^\alpha \mathsf{false}} \qquad \frac{\langle b_1, \sigma \rangle \to_b^\alpha \mathsf{false}}{\langle \neg b_1, \sigma \rangle \to_b^\alpha \mathsf{true}}$$

$$\frac{\langle e_1, \sigma \rangle \to_a l_1 \qquad \langle e_2, \sigma \rangle \to_a l_2}{\langle e_1 = e_2, \sigma \rangle \to_b^1 \mathsf{true}} \text{ if } |l_1| = |l_2|$$

$$\frac{\langle e_1, \sigma \rangle \to_a l_1 \qquad \langle e_2, \sigma \rangle \to_a l_2}{\langle e_1 = e_2, \sigma \rangle \to_b^1 \mathsf{false}} \text{ if } |l_1| \neq |l_2|$$

$$\frac{\langle e_1, \sigma \rangle \to_a l_1 \qquad \langle e_2, \sigma \rangle \to_a l_2}{\langle e_1 \leq e_2, \sigma \rangle \to_b^1 \mathsf{true}} \text{ if } |l_1| \leq |l_2|$$

$$\frac{\langle e_1, \sigma \rangle \to_a l_1 \qquad \langle e_2, \sigma \rangle \to_a l_2}{\langle e_1 \leq e_2, \sigma \rangle \to_b^1 \mathsf{false}} \text{ if } |l_1| > |l_2|$$

$$\frac{\langle b_1, \sigma \rangle \to_b^\alpha \mathsf{false} \qquad \langle b_2, \sigma \rangle \to_b^\beta \mathsf{false}}{\langle b_1 \wedge b_2, \sigma \rangle \to_b^{\alpha+\beta-\alpha\beta} \mathsf{false}} \qquad \frac{\langle b_1, \sigma \rangle \to_b^\alpha \mathsf{true} \qquad \langle b_2, \sigma \rangle \to_b^\beta \mathsf{true}}{\langle b_1 \wedge b_2, \sigma \rangle \to_b^{\alpha\beta} \mathsf{true}}$$

$$\frac{\langle b_1, \sigma \rangle \to_b^\alpha \mathsf{true} \qquad \langle b_2, \sigma \rangle \to_b^\beta \mathsf{false}}{\langle b_1 \wedge b_2, \sigma \rangle \to_b^{1-\alpha-\alpha\beta} \mathsf{false}} \qquad \frac{\langle b_1, \sigma \rangle \to_b^\alpha \mathsf{false} \qquad \langle b_2, \sigma \rangle \to_b^\beta \mathsf{true}}{\langle b_1 \wedge b_2, \sigma \rangle \to_b^{1-\beta-\alpha\beta} \mathsf{false}}$$

$$\frac{\langle b_1, \sigma \rangle \to_b^\alpha \mathsf{false} \qquad \langle b_2, \sigma \rangle \to_b^\beta \mathsf{false}}{\langle b_1 \vee b_2, \sigma \rangle \to_b^{\alpha\beta} \mathsf{false}} \qquad \frac{\langle b_1, \sigma \rangle \to_b^\alpha \mathsf{true} \qquad \langle b_2, \sigma \rangle \to_b^\beta \mathsf{true}}{\langle b_1 \vee b_2, \sigma \rangle \to_b^{\alpha+\beta-\alpha\beta} \mathsf{true}}$$

$$\frac{\langle b_1, \sigma \rangle \to_b^\alpha \mathsf{true} \qquad \langle b_2, \sigma \rangle \to_b^\beta \mathsf{false}}{\langle b_1 \vee b_2, \sigma \rangle \to_b^{1-\beta-\alpha\beta} \mathsf{true}} \qquad \frac{\langle b_1, \sigma \rangle \to_b^\alpha \mathsf{false} \qquad \langle b_2, \sigma \rangle \to_b^\beta \mathsf{true}}{\langle b_1 \vee b_2, \sigma \rangle \to_b^{1-\alpha-\alpha\beta} \mathsf{true}}$$

Figure 6.2: Semantics of boolean expressions

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \to_c^1 \sigma}$$

$$\frac{\langle e_1, \sigma\rangle \to_a n}{\langle x ::= e_1, \sigma\rangle \to_c^1 \sigma[n/x]}$$

$$\frac{\langle C_1, \sigma_1\rangle \to_c^\alpha \sigma_2 \qquad \langle C_2, \sigma_2\rangle \to_c^\beta \sigma_3}{\langle C_1; C_2, \sigma\rangle \to_c^{\alpha\beta} \sigma_3}$$

$$\frac{\langle b_1, \sigma\rangle \to_b^\alpha \mathsf{true} \qquad \langle C_1, \sigma\rangle \to_c^\beta \sigma_1}{\langle \texttt{If}\, b_1\, \texttt{Then}\, C_1\, \texttt{Else}\, C_2, \sigma\rangle \to_c^{\alpha\beta} \sigma_1}$$

$$\frac{\langle b_1, \sigma\rangle \to_b^\alpha \mathsf{false} \qquad \langle C_2, \sigma\rangle \to_c^\beta \sigma_1}{\langle \texttt{If}\, b_1\, \texttt{Then}\, C_1\, \texttt{Else}\, C_2, \sigma\rangle \to_c^{\alpha\beta} \sigma_1}$$

$$\frac{\langle X_k, \sigma\rangle \to_a 0}{\langle \texttt{loop}\, X_k\, \{C_1\}, \sigma\rangle \to_c^1 \sigma}$$

$$\frac{\begin{array}{ccc} & & \langle C_1, \sigma\rangle \to_c^{\alpha_1} \sigma_1 \\ & |l_1| = n & \langle C_1, \sigma_1\rangle \to_c^{\alpha_2} \sigma_2 \\ & & \cdots \\ \langle X_k, \sigma\rangle \to_a l_1 & n > 0 & \langle C_1, \sigma_{n-1}\rangle \to_c^{\alpha_n} \sigma_n \end{array}}{\langle \texttt{loop}\, X_k\, \{C_1\}, \sigma\rangle \to_c^{\Pi\alpha_i} \sigma_n}$$

Figure 6.3: Semantics of commands

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \to_{\mathscr{D}} \{\sigma^1\}} \qquad \frac{\langle e, \sigma\rangle \to_a n}{\langle x ::= e, \sigma\rangle \to_{\mathscr{D}} \{\sigma[n/x]^1\}} \qquad \frac{\langle C_1, \sigma\rangle \to_{\mathscr{D}} \mathscr{D} \qquad \forall \sigma_i \in \mathscr{D}.\langle C_2, \sigma_i\rangle \to_{\mathscr{D}} \mathscr{E}_i}{\langle C_1; C_2, \sigma\rangle \to_{\mathscr{D}} \bigcup_i \mathscr{D}(\sigma_i) \cdot \mathscr{E}_i}$$

$$\frac{\langle X_k, \sigma\rangle \to_a 0}{\langle \texttt{loop}\, X_k\, \{C\}, \sigma\rangle \to_{\mathscr{D}} \{\sigma^1\}} \qquad \frac{\begin{array}{ccc} & |l_1| = n & \\ \langle X_k, \sigma\rangle \to_a l_1 & n > 0 & \langle \overbrace{C; C; \ldots; C}^{n}, \sigma\rangle \to_{\mathscr{D}} \mathscr{E} \end{array}}{\langle \texttt{loop}\, X_k\, \{C\}, \sigma\rangle \to_{\mathscr{D}} \mathscr{E}}$$

$$\frac{\langle b, \sigma\rangle \to_b^\alpha \mathsf{true} \qquad \langle C_1, \sigma\rangle \to_{\mathscr{D}} \mathscr{D} \qquad \langle C_2, \sigma\rangle \to_{\mathscr{D}} \mathscr{E}}{\langle \texttt{If}\, b\, \texttt{Then}\, C_1\, \texttt{Else}\, C_2, \sigma\rangle \to_{\mathscr{D}} (\alpha \cdot \mathscr{D}) \cup ((1-\alpha) \cdot \mathscr{E})}$$

Figure 6.4: Distributions of output states

6.5 Distributions

iSAPP is working on stochastic computations. In order to reach soundness and completeness respect to **PP**, we need to define a semantics for distribution of final states. We need to introduce some more definitions. Let $\mathscr{D}$ be a distribution of probabilities over states. Formally, $\mathscr{D}$ is a function whose type is $(\texttt{variable} \to \texttt{Values}) \to [0,1]$. Sometimes we will use the following notation $\mathscr{D} = \{\sigma_1^{\alpha_1}, \ldots, \sigma_n^{\alpha_n}\}$ indicating that probability of σ_i is α_i.

We say that a distribution $\mathscr{D} = \{\sigma_1^{\alpha_1}, \ldots, \sigma_n^{\alpha_n}.\}$ is normalised when $\sum_i \alpha_i = 1$. Semantics for distribution of final states is shown in Figure 6.4 and we can easily check that rule creates a normalised distribution in output. Unions of distributions and multiplication between real number and a distribution have the natural meaning.

Here we can present our first result.

Theorem 6.1 *A command C in a state σ_1 reduce to another state σ_2 with probability equal to $\mathscr{D}(\sigma_2)$, where $\mathscr{D}$ is the distribution of probabilities over states such that $\langle C, \sigma_1\rangle \to_{\mathscr{D}} \mathscr{D}$.*

Proof is done by structural induction on derivation tree. It is quite easy to check that this property holds, as the rules in Figure 6.4 are showing us exactly this statement. The reader should also not be surprised by this property. Indeed, we are not considering just one possible derivation from $\langle C_1, \sigma_1\rangle$ to σ_2, but all the ones going from the first to the latter.

6.6 Typing and certification

We presented all the ingredients of **iSAPP** and we are ready to introduce typing rules. Typing rules, in figure 6.5, associate at every expression a column vector and at every command a matrix.

These vectors (matrices) tell us about the behaviour of an expression (command). We can think about them as a certificate. Certificates for expressions tell us about the bound for the result of the expression, while certificates for commands tell us about the correlation between input and output variables. Each column gives the bound of one output variable while each row corresponds to one input variable. Last row and column handle constants.

Most rules are quite obvious. When sequencing two instructions, the actual bounds are composed, and one can check that the abstraction of the composition of multipolynomials

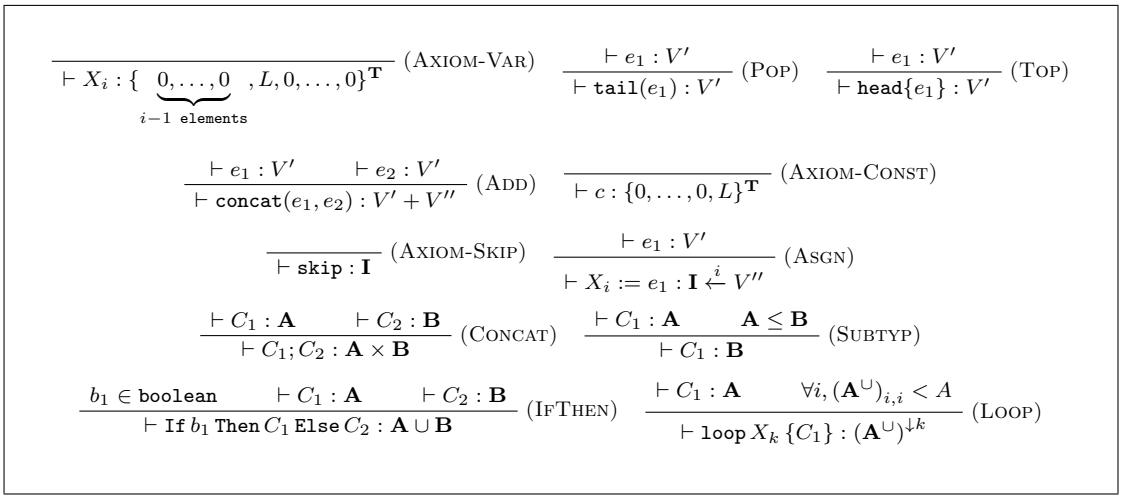


Figure 6.5: Typing rules for expressions and commands

is indeed the product of the abstractions. When there is a test, taking the union of the abstractions means taking the worst possible case between the two branches.

The most interesting type rule is the one concerning the (Loop) command. The right premise acts as a guard: an A on the diagonal means that there is a variable X such that iterating the loop a certain number of time results in X depending affinely of itself, *e.g.* $X = 2 \times X$. Obviously, iterating this loop may create an exponential, so we stop the analysis immediately. Next, the union closure used as a certificate corresponds to a worst case scenario. We can't know if the loop will be executed 0, 1, 2, ... times each corresponding to certificates $\mathbf{A}^0, \mathbf{A}^1, \mathbf{A}^2, \ldots$ Thus we assume the worst and take the union of these, that is the union closure. Finally, the loop correction (merge down) is here to take into account the fact that the result will also depends on the size of the variable controlling the loop.

6.7 Extra operators

We are going to show how to type multiplication and subtraction in **iSAPP**. The grammar of our system does not provide multiplication and a real subtraction as basic operands. While the subtraction is not a dangerous operation, multiplication can lead to exponential blow up in size of variables if iterated. In the following we will focus on three possible implementation of multiplication and subtraction.

Even if we haven't yet introduced the semantics of our system, the reader will understand immediately the associated semantics of these programs: it is the standard one. Recall that the command $\texttt{loop}\, X_i\, \{C\}$ execute X_i times the command C.

Definition 6.10 (Multiplication between two variables) *Suppose we have the following variables: X_1, X_2, X_3 and we want to compute the expression $X_1 \times X_2$. We will use the variable X_3 as the variable that will take the result. Here is the program.*

$X_3 = 0$

$\texttt{loop}\, X_2\, \{X_3 = X_3 + X_1\}$

The command inside the loop is typed with $\mathbf{A} = \begin{bmatrix} L & 0 & L & 0 \\ 0 & L & 0 & 0 \\ 0 & 0 & L & 0 \\ 0 & 0 & 0 & L \end{bmatrix}$ *and so* $(\mathbf{A}^{\cup})^{\downarrow 2} = \begin{bmatrix} L & 0 & M & 0 \\ 0 & L & M & 0 \\ 0 & 0 & L & 0 \\ 0 & 0 & 0 & L \end{bmatrix}$. *First command is typed with* $\begin{bmatrix} L & 0 & 0 & 0 \\ 0 & L & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & L & L \end{bmatrix}$ *and so* **iSAPP** *types the whole program with matrix* $\begin{bmatrix} L & 0 & M & 0 \\ 0 & L & M & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & L & L \end{bmatrix}$, *that is a good bound for multiplication. The informal meaning of our certificate tells us that the result on X_3 is bounded by a multiplication between X_1 and X_2 plus some possible constant. Indeed, the certificate would have been the same if we would have associate another constant to X_3 instead of* 0.

We have shown how to type multiplication in our system. The result obtained is something that we were expecting. Let's see another kind of multiplication, one between a constant and a variable.

Definition 6.11 (Multiplication between constant and variable) *Suppose we have variable X_1, X_2 and we want to calculate $n \cdot X_1$. We use the variable X_2 as a temporary variable. The following program calculates our multiplication.*

$X_2 = 0$

$\texttt{loop}\, X_1\, \{X_2 = X_2 + n\}$

The loop is typed as $\begin{bmatrix} L & A & 0 \\ 0 & L & 0 \\ 0 & A & L \end{bmatrix}$ *and therefore the whole program is typed with* $\begin{bmatrix} L & A & 0 \\ 0 & 0 & 0 \\ 0 & A & L \end{bmatrix}$.

The result obtained tells us that the final value of X_2 depends linearly from X_1 and a constant. This is what we were expecting; indeed, the final result is $0 + n \cdot X_1$. Let's now introduce the subtraction typing. We encode the subtraction as an iterated minus one operation.

Definition 6.12 (Subtraction) *Suppose we have the following variables:* X_1, X_2, X_3 *and we want to compute the expression* $X_1 - X_2$*. We use the variable* X_3 *as the variable that is keeping the result. Here is the program.*

$X_3 = X_1$

$\texttt{loop}\, X_2\, \{X_3 = X_3 - 1\}$

It is easy to check that the loop is typed with: $\begin{bmatrix} L & 0 & 0 & 0 \\ 0 & L & 0 & 0 \\ 0 & 0 & L & 0 \\ 0 & 0 & 0 & L \end{bmatrix}$ *and so, the typing for the subtraction is, as expected,* $\begin{bmatrix} L & 0 & L & 0 \\ 0 & L & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & L \end{bmatrix} \begin{bmatrix} L & 0 & 0 & 0 \\ 0 & L & 0 & 0 \\ 0 & 0 & L & 0 \\ 0 & 0 & 0 & L \end{bmatrix} = \begin{bmatrix} L & 0 & L & 0 \\ 0 & L & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & L \end{bmatrix}$*. Notice that the bound for a real subtraction* **iSAPP** *is the original value.*

6.8 Soundness

The language recognised by **iSAPP** is an imperative language where iteration is bounded and arithmetical expressions are built only with addition and subtraction. These are ingredients of a lot of well known ICC polytime systems. It is no surprise that every program written with the language recognised by **iSAPP** runs in polytime.

First we will focus on multipolynomial properties in order to show that the behaviour of these algebraic constructor is similar to the behaviour of matrices in our system. Finally we will link these things together to get polytime bound for **iSAPP**. Here are two fundamental lemmas.

Lemma 6.1 *Let* p *and* q *two positive multipolynomials, then it holds that* $\lceil (p \oplus q) \rceil = \lceil p \rceil \cup \lceil q \rceil$.

Recalling definition in section 6.3, it is easy to check that property holds. The other important operator with matrices in our system is the multiplication that corresponds to concatenation of commands.

Lemma 6.2 *Let* Q *a positive multipolynomial and let* p *a polynomial, both in canonical form, then it holds that* $\lceil p(q_1, \ldots, q_n) \rceil \leq \lceil Q \rceil \times \lceil p \rceil$.

Proof:by structural induction on the polynomial p_k

- If p is X_i, then $\lceil p(q_1, \ldots, q_n) \rceil$ is $\lceil q_i \rceil$. property holds.

- If p is c then $\lceil p(q_1,\ldots,q_n)\rceil$ is $\lceil c\rceil$ that is less than $\lceil Q\rceil \times \lceil c\rceil$. Property holds.
- If p is αX_i $(\alpha > 1)$ then $\lceil p(q_1,\ldots,q_n)\rceil$ is $\lceil \alpha q_i\rceil$. property holds.
- If p is $\alpha X_i \cdot r(\overline{X})$ then $\lceil p(q_1,\ldots,q_n)\rceil$ is $\lceil \alpha q_i r(\overline{X})\rceil$. property holds.
- If p is $r(\overline{X})+s(\overline{X})$, then $\lceil r(q_1,\ldots,p_n)+s(q_1,\ldots,p_n)\rceil$ is $\lceil r(q_1,\ldots,p_n)\rceil+\lceil s(q_1,\ldots,p_n)\rceil$ and by induction we get $\lceil Q\rceil \times \lceil r\rceil + \lceil Q\rceil \times \lceil s\rceil$ that is $\lceil Q\rceil \times (\lceil r\rceil + \lceil s\rceil)$. Property holds.
- If p is $r(\overline{X}) \cdot s(\overline{X})$ then $\lceil p(q_1,\ldots,q_n)\rceil$ is $\lceil r(q_1,\ldots,q_n) \cdot s(q_1,\ldots,q_n)\rceil$ and by induction it easy to check that the property holds.

This concludes the proof. □

Lemma 6.3 *Let P and Q two positive multipolynomials in canonical form, then it holds that $\lceil P \odot Q\rceil \leq \lceil P\rceil \times \lceil Q\rceil$*

Proof: Proof of this property could be a little bit tricky. we show how to proceed.

$$
\begin{aligned}
\lceil P \odot Q\rceil_{i,j} &= \lceil q_j(p_1,\ldots,p_n)\rceil_i && \text{by definition of composition} && (6.1)\\
&\leq \{\lceil P\rceil \times \lceil q_j\rceil\}_i && \text{by theorem 6.2} && (6.2)\\
&= \sum_k \lceil P\rceil_{i,k} \cdot \lceil q_j\rceil_k && \text{by definition of matrix multiplication} && (6.3)\\
&= \sum_k \lceil P\rceil_{i,k} \cdot \lceil Q\rceil_{k,j} && \text{by definition of multipolynomial} && (6.4)\\
&= (\lceil P\rceil \times \lceil Q\rceil)_{i,j} && \text{by definition of matrix multiplication} && (6.5)
\end{aligned}
$$

□

Lemma 6.4 *Let e be an expression on variables $X_1,\ldots,X_n$ typed with V'; Let σ be a state function; Let $\langle e,\sigma\rangle$ reduce to a value a. Then there exists a polynomial p on input $\sigma(X_1),\ldots,\sigma(X_n)$ such that for every expression e appearing in π s.t. $\langle e,\sigma\rangle \to_a a$ we have $|e| \leq p$.*

Proof: By structural induction on semantic derivation tree.

- If e is a variable or a constant the proof is trivial.

- Otherwise, by induction on the premises we can easily conclude the thesis.

□

We get a polynomial bound for the size of an expression, and this is enough for having polynomial bound on time execution. We can easily prove the following lemma:

Lemma 6.5 *Let e be an expression well typed with V'; Let σ be a state function and let $\langle e, \sigma\rangle$ reduce to a value a. We have that $|\pi : \langle e, \sigma\rangle \to_a a|$ is polynomially bounded respect to the size of e.*

We can easily prove the previous lemma by structural induction on the semantics derivation tree. Having a polynomial bound for every expressions in our system is quite easy because with just addition and a kind of subtraction there is no way to get an exponential bound. However this lemmas are fundamental in order to prove polynomial bound on size and time for commands in **iSAPP**.

We can now move on and investigating the polynomiality of command execution time and expression size. The following theorem tell us that at each step of execution of a program, size of variables are polynomially correlated with size of variables in input.

Theorem 6.2 *Given a command C well typed in* **iSAPP** *with matrix* **A**, *such that $\langle C, \sigma_1\rangle \to_c^\alpha \sigma_2$ we get that exists a multipolynomial P such that for all variables X_i we have that $|\sigma_2(X_i)| \leq P_i(|\sigma_1(X_1)|, \ldots, |\sigma_1(X_n)|)$ and $\lceil P \rceil$ is* **A**.

Proof: By structural induction on typing tree.

- If last rule is (Axiom-Skip) or (Subtyp) the thesis is trivial.
- If last rule is (IfThen), by applying induction on hypothesis and by lemma 6.1 we can easily conclude the thesis.
- If last rule is (Asgn) then by lemma 6.4 we are done.
- If last rule is (Concat), by applying induction on hypothesis and get two multipolynomials Q, R and by lemma 6.1 we can easily conclude the thesis.

- Finally we are in the case where last rule is (Loop) and so $\mathbf{A}$ is $(\mathbf{B}^{\cup})^{\downarrow k}$, for some matrix $\mathbf{B}$ and index k. The derivation tree for typing has the following shape

$$\frac{\rho :\vdash C_1 : \mathbf{B} \qquad \forall i, (\mathbf{B}^{\cup})_{i,i} < A}{\vdash \texttt{loop}\, X_k \,\{C_1\} : (\mathbf{B}^{\cup})^{\downarrow k}}\ (\textsc{Loop})$$

and the associate semantics is

$$\frac{\mu : \langle X_k, \sigma\rangle \to_a 0}{\langle \texttt{loop}\, X_k\,\{C_1\}, \sigma\rangle \to_c^1 \sigma}$$

$$\frac{\begin{array}{ll}\mu : \langle X_k, \sigma\rangle \to_a l_1 & \nu_1 : \langle C_1, \sigma\rangle \to_c^{\alpha_1} \sigma_1 \\ |l_1| = n & \nu_2 : \langle C_1, \sigma_1\rangle \to_c^{\alpha_2} \sigma_2 \\ & \cdots \\ n > 0 & \nu_n : \langle C_1, \sigma_{n-1}\rangle \to_c^{\alpha_n} \sigma_n\end{array}}{\langle \texttt{loop}\, X_k\,\{C_1\}, \sigma\rangle \to_c^{\Pi\alpha_i} \sigma_n}$$

Semantics of command `loop {}` tells us to compute first the value of the guard X_k; suppose $\langle X_k, \sigma\rangle \to_a n$, then we have to apply n times the command C_1.

First rule of semantics of command (Loop) tells us that if the variable X_k reduces to 0 the final state is not changed. In this particularly case, it is not so difficult to create a multipolynomial such that its abstraction is $(\mathbf{B}^{\cup})^{\downarrow k}$. Whatever is, the thesis is of course proved because values of variables are not changed.

Let's focus on the second rule of semantics for (Loop), the case where the loop is performed at least once. By induction on the premise we have a multipolynomial P bound for command C_1 such that its abstraction is $\mathbf{B}$.

If P is a bound for C_1, then $P \odot P$ is a bound for $C_1; C_1$ and $(P \odot P) \odot P$ is a bound for $C_1; C_1; C_1$ and so on. All of these are multipolynomial because we are composing multipolynomials with multipolynomials.

By lemma 6.3 and knowing that $\lceil P \rceil$ is $\mathbf{B}$ we can easily deduce to have a multipolynomial bound for every iteration of command C_1. In particularly by lemma 6.1 we can easily sum up everything and find out a multipolynomial Q such that $\lceil Q \rceil$ is $\mathbf{B}^{\cup}$. This means that further iterations of sum of powers of P will not change the abstraction of the result.

So, for every iteration of command C_1 we have a multipolynomial bound whose abstraction cannot be greater than $\mathbf{B}^{\cup}$. We study the worst case, analysing the matrix $\mathbf{B}^{\cup}$.

Side condition on (Loop) rule tells us to check elements on the main diagonal. Recall that by definition of union closure, elements on the main diagonal are supposed to be greater than 0. We required also to be less than A. Let's analyse all the possibilities of an element in position i, i:

- Value cannot be 0. If value is L it means that Q_i bound for such column has shape $X_i + r(\overline{X})$, where X_i does not appear in $r(\overline{X})$. It is easily to check that concatenation of C_1 cannot create exponential blow up because at most, at every iteration, we copy the value (recall that Q is a bound for all the iterations).
- If value is A could means that Q_i bound for such column has shape $\alpha X_i + r(\overline{X})$ (for some $\alpha > 1$), where X_i does not appear in $r(\overline{X})$ and so there could be a way to duplicate the value after some steps of iteration.
- Otherwise value is M and we cannot make assumptions. We might have exponential bound.

The abstract bound $\mathbf{B}$ is still not a correct abstract bound for the loop because loop iteration depends on some variable X_k. We need to adjust our bound in order to keep track of the influence of variable X_k on loop iteration.

We take multipolynomial Q because we know that further iterations of the algorithm explained before will not change $\lceil Q \rceil$. Looking at i-th polynomial of multipolynomial Q we could have three different cases. We behave in the following way:

- The polynomial has shape $X_i + p(\overline{X})$. In this case we multiply the polynomial p by X_k because this is the result of iteration. We substitute the i-th polynomial with the canonical form of polynomial $X_i + p(\overline{X}) \cdot X_k$.
- The polynomial has shape $X_i + \alpha$, for some constant α. In this case we substitute with $X_i + \alpha \cdot X_k$.
- The polynomial has shape X_i. We leave as is.

In this way we generate a new multipolynomial, call it R. The reader should easily check that these new multipolynomial expresses a good bound of iterating Q a number of times equal to X_k. Should also be quite easy to check that $\lceil R \rceil$ is exactly

$(\mathbf{B}^{\cup})^{\downarrow k}$. We are so allowed to type the loop with this new matrix and the thesis is proved.

□

Polynomial bound on size of variables is not enough; we need to prove also polynomiality of number of steps. In this case the number of steps is equal to size of the semantic tree generated by the system. We can proceed and demonstrate the following theorem.

Theorem 6.3 *Let C be a command well typed in* **iSAPP** *and σ_1, σ_n state functions. If $\pi : \langle C_1, \sigma_0\rangle \to_c^{\alpha} \sigma_n$, then there is a polynomial p such that $|\pi|$ is bounded by $p(\sum_i |\sigma_0(X_i)|)$.*

Proof: By structural induction of command C.

- If C is `skip`, the proof is trivial.
- If C is $X_i = e$, then by lemma 6.5 we have the thesis.
- If C is $C_1; C_2$, then by induction we get polynomials q, r. th evaluation of C takes $q + r$
- If C is $\texttt{If}\, b_1\, \texttt{Then}\, C_1\, \texttt{Else}\, C_2$, then by induction we can easily get a polynomial bound.
- If C is $\texttt{loop}\, X_i\, \{C_1\}$ we are in the following case:

$$\frac{\begin{array}{cc}\mu : \langle X_k, \sigma_0\rangle \to_a l_1 & \nu_1 : \langle C_1, \sigma_0\rangle \to_c^{\alpha_1} \sigma_1 \\ |l_1| = n & \nu_2 : \langle C_1, \sigma_1\rangle \to_c^{\alpha_2} \sigma_2 \\ & \cdots \\ n > 0 & \nu_n : \langle C_1, \sigma_{n-1}\rangle \to_c^{\alpha_n} \sigma_n\end{array}}{\langle \texttt{loop}\, X_k\, \{C_1\}, \sigma_0\rangle \to_c^{\Pi\alpha_i} \sigma_n}$$

By lemma 6.5 we have polynomial bound for $|\mu|$. Thanks to theorem 6.2 we get a polynomial bound for n.

We have now to perform n iterations of C_1. By induction we get polynomial r_i bound for each iteration. Formally, $|\nu_i|$ is bounded by $r_i(\sum_j |\sigma_{i-1}(X_j)|)$. We can easily rewrite all the polynomials r in terms of size of variables in σ_0 thanks to theorem 6.2.

Therefore we can conclude that there exists a polynomial in size of $\sum_i |\sigma_0(X_i)|$ bounding the size of derivation of this case.

This concludes the proof. □

Nothing has been said about probabilistic polynomial soundness. Theorems 6.2 and 6.3 tell us just about polytime soundness. Probabilistic part is now introduced. We will prove probabilistic polynomial soundness following idea in [14], by using "representability by majority".

Definition 6.13 (Representability by majority) *Let C be a program and let σ_0 the state function define as $\forall X, \sigma_0(X) = 0$. Let $\sigma_0[\overline{X}/n]$ define as $\forall X, \sigma_0(X) = n$. Then C is said to* represent-by-majority *a language $L \subseteq \mathbb{N}$ iff:*

1. *If $n \in L$ and $\langle C, \sigma_0[\overline{X}/n]\rangle \rightarrow_{\mathscr{D}} \mathscr{D}$, then $\mathscr{D}(\sigma_0) \geq \sum_{m>0} \mathscr{D}(\sigma_m)$;*
2. *If $n \notin L$ and $\langle C, \sigma_0[\overline{X}/n]\rangle \rightarrow_{\mathscr{D}} \mathscr{D}$, then $\sum_{m>0} \mathscr{D}(\sigma_m) > \mathscr{D}(\sigma_0)$.*

That is, if $n \in L$ then starting with every variable set to n, the result is σ_0 (every variable to 0, "accepting state") with a probability more than 0.5: the majority of the executions "vote" that $n \in L$.

Observe that every command C in **iSAPP** represents by majority a language as defined in 6.13. In literature [3] is well known that we can define **PP** by majority itself. We say that the probability error should be at most $\frac{1}{2}$ when we are considering string in the language and strictly smaller than $\frac{1}{2}$ when the string is not in the language. So we can conclude that **iSAPP** is sound also respect to Probabilistic Polynomial Time.

6.9 Probabilistic Polynomial Completeness

There are several ways to demonstrate completeness with respect to some complexity class. We show how to encode Probabilistic Turing Machines (PTM) solving a problem in **PP**. Not all the possible PTMs are codable in the language recognised by **iSAPP**, but all the ones with particularly shape. This lead us to reach extensional completeness and not intentional completeness. For every problem in **PP** there is at least an algorithm solving that problem that is recognised by **iSAPP**.

A Probabilistic Turing Machine can be seen as a non deterministic TM with one tape where at each iteration is able to flip a coin and choose between two possible transition functions to apply.

The language recognised by **iSAPP** gives all the ingredients. In order to encode Probabilistic Turing Machines we will proceed with the following steps:

- We encode the polynomial representing the number of steps performed by our PTM.
- We encode the input tape of the machine.
- We encode the transition function δ.
- We put all together and we have an encoding of a PTM running in polytime.

It should be quite obvious that we can encode polynomials in **iSAPP**. Grammar and examples 6.10, 6.11, 6.12 give us all the tools for encoding polynomials. Next, we need to encode the tape of our PTMs. We subdivide our tape in three sub-tapes. The left part $\mathbf{tape}_l$, the head $\mathbf{tape}_h$ and the right part $\mathbf{tape}_r$. $\mathbf{tape}_r$ is encoded right to left, while the left part is encoded as usual left to right. If $\bar{t}$ represents the original binary input of our PTM, we set $\mathbf{tape}_l = \langle \bar{t} \rangle$, $\mathbf{tape}_h = \langle \rangle$ and $\mathbf{tape}_r = \langle \rangle$. Extra variable called $\mathbf{M_{state}}$ keeps track of the state of the machine.

The reader should notice that we are encoding the tape using list-representation of our system. We are going to keep track of all tape information inside lists; we are no more interested to see lists as encoding of natural numbers. In this part we are going to use extra operators introduced in the grammar: `testzero`$\{e\}$ and `head`$\{e\}$. Thanks to these operators and their semantics we are able to break some dependencies that lead our system to fail on encoding a PTM.

In the following we present the algorithm for moving the head to the right. Similar algorithm can be written for moving the head to the left. It is really important to pay attention on how we encode these operations. Recall that a PTM loops the δ function and our system requires that the matrix certifying/typing the loop needs to have values of the diagonal less than A. The trivial encoding will not be typable by **iSAPP**. Notice also that the following procedure works because we are assuming that our Probabilistic Turing Machine is working on binary alphabet.

Definition 6.14 (Move head to right) *Moving head to right means to concatenate the bit pointed by the head to the left part of the tape; therefore we need to retrieve the first bit of the right part of the tape and associate it to the head. Procedure is presented as*

algorithm number 2. It is easy to check that **iSAPP** *types the algorithm number 2 with the matrix* $\begin{bmatrix} L & 0 & 0 & 0 & 0 \\ L & 0 & 0 & 0 & 0 \\ 0 & 0 & L & 0 & 0 \\ 0 & 0 & 0 & L & 0 \\ 0 & L & 0 & 0 & L \end{bmatrix}$*, where the first column of the matrix represents dependencies for variables* $\mathbf{tape}_l$*, the second represents* $\mathbf{tape}_h$*, third is* $\mathbf{tape}_r$*, forth is* $\mathbf{M_{state}}$ *and finally recall that last column is for constants.*

Algorithm 2 Move head to right

$\mathbf{tape}_l ::= \texttt{concat}(\mathbf{tape}_l, \mathbf{tape}_h)$
if $\texttt{testzero}\{\texttt{head}\{\mathbf{tape}_r\}\}$ **then**
  $\mathbf{tape}_h ::= \langle 0 \rangle$
else
  if $\texttt{head}\{\mathbf{tape}_r\} = \langle\rangle$ **then**
    $\mathbf{tape}_h ::= \langle\rangle$
  else
    $\mathbf{tape}_h ::= \langle 1 \rangle$
  end if
end if
$\mathbf{tape}_r ::= \texttt{tail}(\mathbf{tape}_r)$

The reader should focus on the second column. It tells us that variable $\mathbf{tape}_h$ depends just from some constant. This is the key point: knowing that our PTM is working with binary elements, we can just perform some nested `If-Then-Else` and retrieve the correct value without showing explicitly the dependency between $\mathbf{tape}_h$ and $\mathbf{tape}_r$.

We can now move on and show how to encode the procedure to move left the head of the tape.

Definition 6.15 (Move head to left) *Moving head to left means to concatenate the bit pointed by the head to right part of the tape; therefore we need to retrieve the rightmost bit of tape left and associate it to the head. Procedure is presented as algorithm 3; call it* MoveToLeft().

iSAPP types/certificates the procedure in 3 with the following matrix: $\begin{bmatrix} L & 0 & 0 & 0 & 0 \\ 0 & 0 & L & 0 & 0 \\ 0 & 0 & L & 0 & 0 \\ 0 & 0 & 0 & L & 0 \\ 0 & L & 0 & 0 & L \end{bmatrix}$. Reader should again focus on the second column. Exactely as for procedure in 2, here

Algorithm 3 Move head to left

```
tape_r ::= concat(tape_r, tape_h)
if testzero{head{tape_l}} then
    tape_h ::= ⟨0⟩
else
    if head{tape_l} = ⟨⟩ then
        tape_h ::= ⟨⟩
    else
        tape_h ::= ⟨1⟩
    end if
end if
tape_l ::= tail(tape_l)
```

it tells us that variable $\mathbf{tape}_h$ depends just from some constants. Finally we need to introduce the procedure that does anything.

Definition 6.16 (Not moving head) *Our PTM could also not perform any movment of the head. This means that no operation is executed. This is* `skip` *command, whose type is the identity matrix* $\mathbf{I}$*. Call this procedure* NOP().

Moving head on the tape is not enough for having an encoding of transition function. We need to show how to perform the coin flip, changing the state and writing on tape. We introduce a new command, as a shorter notation for nested `if-then-else`. Program shown in algorithm 5 is rewritten with shorter notation as the one in 4.

The reader should not be surprised. This is the standard definition of SWITCH command. We are now ready to show how to encode our δ function. Prototype of delta function encoding is presented in algorithm 6.9. A PTMs is a finite state machine and we suppose that the number of states is n.

In order to encode δ function we can proceed by nesting `If-Then-Else` commands, checking `rand` value and variable containing the state value. The first command is an `If-Then-Else` testing the value of `rand`. in both cases, then we will execute commands. Then a `Switch` is performed on the state of the machine and finally an operation of reading

Algorithm 4 Switch command

Switch (X_i)
  Case c_1 **:** C_1
    EndCase
  Case c_2 **:** C_2
    EndCase
  Case c_3 **:** C_3
    EndCase
  Case ... **:**
    EndCase
  Default: C_j
    EndDefault
EndSwitch

Algorithm 5 Nested if-then-else

if $X_i = c_1$ **then**
  C_1
else
  if $X_i = c_2$ **then**
    C_2
  else
    if $X_i = c_3$ **then**
      C_3
    else
      ...
      else C_j
    end if
  end if
end if

Algorithm 6 Prototype of encoded δ function

```
if rand then
    Switch (M_state)
        Case 0 :
            if testzero{head{tape_h}} then
                // tape_h takes ⟨0⟩ or ⟨1⟩ or ⟨⟩
                // MoveToLeft or MoveToRight or Nop
                M_state ::= c_1
            else
                if head{tape_h} = ⟨⟩ then
                    // tape_h takes ⟨0⟩ or ⟨1⟩ or ⟨⟩
                    // MoveToLeft or MoveToRight or Nop
                    M_state ::= c_2
                else
                    // tape_h takes ⟨0⟩ or ⟨1⟩ or ⟨⟩
                    // MoveToLeft or MoveToRight or Nop
                    M_state ::= c_3
                end if
            end if
          EndCase
        Case ... :
          EndCase
        Case n − 1 :
            if testzero{head{tape_h}} then
                // tape_h takes ⟨0⟩ or ⟨1⟩ or ⟨⟩
                // MoveToLeft or MoveToRight or Nop
                M_state ::= d_1
            else
                if head{tape_h} = ⟨⟩ then
                    // tape_h takes ⟨0⟩ or ⟨1⟩ or ⟨⟩
                    // MoveToLeft or MoveToRight or Nop
                    M_state ::= d_2
                else
                    // tape_h takes ⟨0⟩ or ⟨1⟩ or ⟨⟩
                    // MoveToLeft or MoveToRight or Nop
                    M_state ::= d_3
                end if
            end if
          EndCase
```

and moving the tape is performed. The algorithm presented in 6.9 has just the purpose to show a scratch of how a delta function could be encoded. The reader can easily check that there is an encoding of δ function typable in **iSAPP** such that **iSAPP** assigns the matrix $\begin{bmatrix} L & 0 & 0 & 0 & 0 \\ L & L & L & 0 & 0 \\ 0 & 0 & L & 0 & 0 \\ 0 & 0 & 0 & L & 0 \\ 0 & L & 0 & L & L \end{bmatrix}$, whose union closure is $\mathbf{A}^{\cup} = \begin{bmatrix} L & 0 & 0 & 0 & 0 \\ A & L & A & 0 & 0 \\ 0 & 0 & L & 0 & 0 \\ 0 & 0 & 0 & L & 0 \\ A & A & A & A & L \end{bmatrix}$. Indeed, once we have the encoded δ function, we need to put it in a loop and iterate it a number of times equal to the polynomial representing the number of steps required by our PTM. The union closure of the matrix is correct with respect to the typing rules because the main diagonal is filled with value L.

We have now all the ingredients to show our encoding of Probabilistic Turing Machines. Let X_k be the number of steps required to be performed for our PTM. We encode everything as shown in algorithm number 7.

Algorithm 7 Encoding of a probabilistic Turing machine

$\mathbf{M_{state}} ::= 0$

$\mathbf{tape}_l ::= \langle\rangle$

$\mathbf{tape}_h ::= \texttt{head}\{\mathbf{tape}_r\}$

$\mathbf{tape}_r ::= \texttt{tail}(\mathbf{tape}_r)$

loop (X_k)

  DeltaFunction

EndLoop

Let $\overline{t}$ be the list representing the input tape of our machine. The initial state σ_0 of our program would be the one where $\sigma_0(\mathbf{tape}_r)$ is $\langle\overline{t}\rangle$.

6.10 Benchmarks and polynomiality

One of the most interesting feature of **iSAPP** is that it is able to give or not a certificate of polynomiality in polytime, with respect to the number of variables used. The key problem lays on typing rule for iteration; it is not trivial to understand how much it costs performing the union closure. Given a matrix $\mathbf{A}$, we can start by calculating $\mathbf{A}^2$, then $\mathbf{A}^3$ and so on, till we reach a matrix $\mathbf{A}^n$ such that exists a $\mathbf{A}^m = \mathbf{A}^n$ where $m < n$.

Theorem 6.4 (Polytime) *Given a squared matrix* $\mathbf{A}$ *of dimension* n, $\mathbf{A}^{\cup}$ *can be computed in* $2n^2 + n$ *steps.*

Proof: The proof is a little bit tricky. Let see how to prove this property. First, we need to see $\mathbf{A}$ as an adjacency matrix of a graph G, where every edge is labelled with L, A or M. If $\mathbf{A}_{i,j}$ is 0, then there is no edge between node i and j.

Let's define the weight of a path as the maximum edge encountered in the path. Notice that if $(\mathbf{A}^{\cup})_{i,j}$ has some value a greater than 0, then it means that there is some $\mathbf{A}^n$ such that $(\mathbf{A}^n)_{i,j}$ is a. Graphically speaking it means that the sum of the weight of all path whose length is n between nodes i and j is a.

Notice that if $\mathbf{A}$ defines a graph G, then $\mathbf{A}^2$ defines a graph where edged are connected if and only if there is a path of length 2 between them in G; edged are labelled as the sum of the weight of the paths between them.

Given a squared matrix $\mathbf{A}$ of dimension n, for every node i, j we proceed in the following way:

- Knowing if there is a path between i and j can be done in n steps of iteration of $\mathbf{A}$. If all the matrices power of $\mathbf{A}$ till $\mathbf{A}^n$ (representing paths potentially passing through all the nodes) have 0 in position (i, j), then there is no path between this two nodes.

- Suppose that there is a path of weight M between i, j, if so, there should be an edge labelled with M. Iterate $\mathbf{A}$ till $\mathbf{A}^n$; if there is such path, then there is a $\mathbf{A}^m$ $(m \leq n)$ such that $(\mathbf{A}^m)_{i,j}$ is M.

- Suppose that there is a path of weight A between i, j, where one edge of this path is A. For the same reason of the previous point, we can iterate n times and if there is such path, we should have find it.

- Suppose that all the paths between i, j have weight L. It means that all the edges encountered are labelled with L. For the reason that $L + L$ is A we need to check the presence of at least two path of same length inside the graph. In this case we can proceed and create a graph G' where vertex are labelled with $\mathbf{V} \times \mathbf{V} \times \{0, 1\}$. In G' there is an edge between (k, k, e) and (k', k', e') if and only if there are edges (k, k') and (k, k') in G. If (k, k') and (k, k') are different edges then e' is 1, otherwise e' is e. Third value is set to 1 if it encodes distinct paths.

  So, we start to build the graph starting from vertex $(i, i, 0)$ to reach $(j, j, 1)$. If this happens, then we have found two paths of same length. The algorithm takes $O(V^4)$

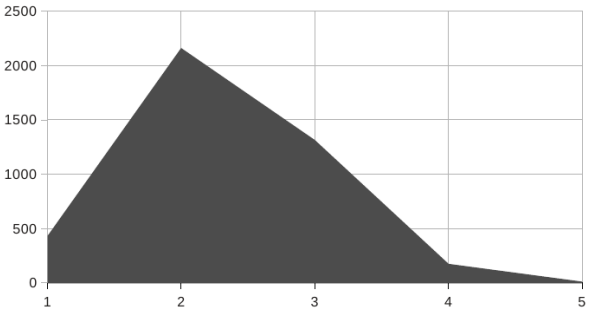


Figure 6.6: Distributions of matrices of size 3 over number of steps

to be performed and the length of the path in G' is at most $2|V|^2$. So, if we iterate $\mathbf{A}$ till $\mathbf{A}^{2n^2}$, then there is a $\mathbf{A}^m$ ($m \leq 2n^2$) such that $(\mathbf{A}^m)_{i,j}$ is A.

- If none of the previous cases occur, then, knowing that there is at least one path, the worst case is L and $(\mathbf{A}^{\cup})_{i,j}$ is L.

Finally, we need to unify all the matrices found from the previous passages with the identity matrix (that is $\mathbf{A}^0$).

Let's sum up everything; Starting with $\mathbf{A}^0$ we iterate by calculating $\mathbf{A}^1$, $\mathbf{A}^2$, $\mathbf{A}^3$, ... till we reach $\mathbf{A}^{2n^2+n}$. After that we can stop because we have enumerated all the possible "interesting" matrix and union of all of them gives $\mathbf{A}^{\cup}$.

□

Theorem 6.4 gives us theoretical bound that is clearly much more than what is really needed. Here are exhaustive benchmarks for matrices of size 3 (figure 6.6) 4 (figure 6.7) 5 (figure 6.8). On the axis x is shown the number of steps required to calculate the union closure and on the other axis there is the distribution among all the matrices of the same size. The reader should notice that the avarage number of steps required is linear respect to the size of the matrix.

Given a program C, **iSAPP** validates in polytime with respect to the size of the program and the number of variables used. Formally:

Theorem 6.5 (Certificate in Polytime) *Given a program C,* **iSAPP** *is able to find, if exists or not, a matrix $\mathbf{A}$ such that $\vdash C : \mathbf{A}$ in polytime respect to the number of variables appearing in C.*

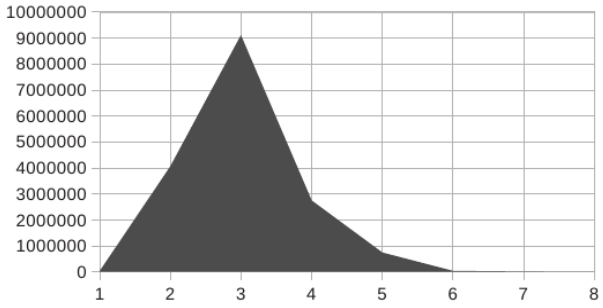


Figure 6.7: Distributions of matrices of size 4 over number of steps

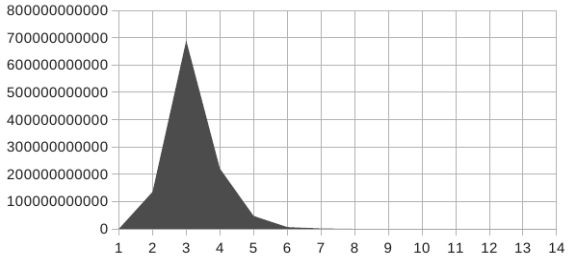


Figure 6.8: Distributions of matrices of size 5 over number of steps

Proof: This proof is quite trivial. Consider that the typing rule for loop command, by theorem 6.4, can be calculated in polytime and all the other rules takes constant time or quadratic/cubic (such as matrix multiplication) time respect to the number of variables in the program. □

Chapter 7

Conclusions

We have seen what are the difficulties that come out when Implicit Computational Complexity meets Probabilistic Polynomial classes. We were able to give some answers to this problem and we showed that some probabilistic classes have intrinsic semantic values that seem impossible to capture with only syntactical restrictions.

In the first part of the thesis we presented an implicit characterisation of Probabilistic Polynomial Time, a higher order system with subject reduction. We also investigate how to characterise all subclasses of **PP**, such as **RP** or **ZPP**, but we were not able to give a syntactical characterisation of them; instead, we give a parametric characterisation. An interesting point is also the proof, a syntactical and constructive proof instead of a semantic one. This lead the system to be able to show how reduction is performed step by step in polynomial time. In the second part we focused more on the reverse problem, still correlated with ICC. Instead of finding out a system sound and complete for a particular class, we tried to build a static analyser for complexity able to say "no" when the program doesn't compute in probabilistic polynomial time. It is a hard job to understand how variable's value flows during the execution of a program. For this reason we decided to focus more on the imperative paradigm in order to understand better and try to solve this problem. We were able to create a complexity static analyser enough powerful to capture for every function in **PP** at least one algorithm. Our system is so able to say "no" if the program doesn't run in probabilistic polynomial time but is not able to say the viceversa. Our completeness respect to the class **PP** is extensional and not intentional. However, it is well known that is not possible to get an intentional complete static analyser for a certain complexity class, without loosing other aspects of the language recognized, because the

problem is not decidable. We showed also that our method could be implementable and really usable because of its performances.

The original contributions of this thesis could be summarise in the following points:

- Extension of ICC techniques to probabilistic classes.
- RSLR is sound and complete respect to Probabilistic Polynomial Time.
- RSLR allows recursion with higher order terms.
- RSLR has subject reduction and confluence of terms is proved.
- We give parametric characterisation of other probabilistic classes.
- **iSAPP** is sound and complete respect to Probabilistic Polynomial Time.
- Analysis is made over a concrete language and also takes account of constants.
- **iSAPP** works in polynomial time with a very low exponent in the worst case.

This thesis would characterise itself as one of the first step for Implicit Computational Complexity over probabilistic classes. There are still open hard problem to investigate and try to solve. There are a lot of theoretical aspects strongly connected with these topics and we expect that in the future there will be wide attention to ICC and probabilistic classes. Some problems such as syntactical characterisation of **BPP** and **ZPP** seem really complex but, exactly for this reason, also very challenging.

Some readers could think that all of these problems have no practical meaning and that, probably, are not so really interesting. Apart from the theoretical and foundational interest, sometimes theoretical studies precede practical one. In this thesis we tried to show one well known immediate practical aspect of this research branch, such as the static complexity analysers. Other application could go in the direction of making proof in the field of security and cryptography. Usually they have to deal with attackers working in Probabilistic Polynomial Time. Our system RSLR could be used to describe easily this kind of entities without focusing explicitly on complexity properties. These are automatically given at no charge. No one knows which benefits could give this research path in the future. The only way to know it is to follow it.

"Computer science is not really about computers; and it's not about computers in the same sense that physics is not really about particle accelerators, and biology is not about microscopes and Petri dishes and geometry isn't really about using surveying instruments. Now the reason that we think computer science is about computers is pretty much the same reason that the Egyptians thought geometry was about surveying instruments: when some field is just getting started and you don't really understand it very well, it's very easy to confuse the essence of what you're doing with the tools that you use." - Hal Abelson

References

[1] S. Aaronson, G. Kuperberg, and C. Granade. The complexity zoo. `http://qwiki.stanford.edu/index.php/Complexity_Zoo`, 2005.

[2] L. M. Adleman and M.-D. A. Huang. Recognizing primes in random polynomial time. In *STOC*, pages 462–469, 1987.

[3] S. Arora and B. Barak. *Computational Complexity, A Modern Approach*. Cambridge University Press, 2009.

[4] A. Asperti. Light affine logic. In *Logic in Computer Science, 1998. Proceedings. Thirteenth Annual IEEE Symposium on*, pages 300–308. IEEE, 1998.

[5] S. Bellantoni. Predicative recursion and the polytime hierarchy. In P. Clote and J. Remmel, editors, *Feasible Mathematics II*, pages 15–29. Birkhauser, 1995.

[6] S. Bellantoni and S. A. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

[7] S. Bellantoni, K. Niggl, and H. Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104(1-3):17–30, 2000.

[8] A. M. Ben-Amram, N. D. Jones, and L. Kristiansen. Linear, polynomial or exponential? complexity inference in polynomial time. In *Proceedings of the 4th conference on Computability in Europe: Logic and Theory of Algorithms*, CiE '08, pages 67–76, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] G. Bonfante, R. Kahle, J.-Y. Marion, and I. Oitavem. Recursion schemata for NC^k. In M. Kaminski and S. Martini, editors, *Computer Science Logic, 22nd International Workshop, Proceedings*, volume 5213 of *LNCS*, pages 49–63, 2008.

[10] A. Church. A note on the entscheidungsproblem. *J. Symb. Log.*, 1(1):40–41, 1936.

[11] A. Cobham. The intrinsic computational difficulty of functions. *Proceedings of the International Conference on Logic, ...*, page 24, 1965.

[12] U. Dal Lago, S. Martini, and D. Sangiorgi. Light logics and higher-order processes. In S. B. Fröschle and F. D. Valencia, editors, *Expressiveness in Concurrency, 17th International Workshop, Proceedings*, volume 41 of *EPTCS*, 2010.

[13] U. Dal Lago, A. Masini, and M. Zorzi. Quantum implicit computational complexity. *Theoretical Computer Science*, 411(2):377–409, 2010.

[14] U. Dal Lago and P. Parisen Toldin. A higher-order characterization of probabilistic polynomial time. In R. Peña, M. van Eekelen, and O. Shkaravska, editors, *Proceedings of* 2^{nd} *International Workshop on Foundational and Practical Aspects of Resource Analysis, FOPARA 2011*, volume 7177 of *LNCS*. Springer, 2011. To be appeared in.

[15] L. Fortnow and R. Santhanam. Hierarchy theorems for probabilistic polynomial time. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '04, pages 316–324, Washington, DC, USA, 2004. IEEE Computer Society.

[16] M. Gaboardi, J. Marion, and S. Della Rocca. An implicit characterization of PSPACE. *Arxiv preprint arXiv:1006.0030*, 2010.

[17] J. T. Gill, III. Computational complexity of probabilistic turing machines. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, STOC '74, pages 91–95, New York, NY, USA, 1974. ACM.

[18] J. Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[19] J. Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.

[20] J. Hartmanis. New developments in structural complexity theory. *Theor. Comput. Sci.*, 71(1):79–93, 1990.

[21] M. Hofmann. A mixed modal/linear lambda calculus with applications to Bellantoni-Cook safe recursion. In M. Nielsen and W. Thomas, editors, *Computer Science Logic,*

11th International Workshop, Proceedings, volume 1414 of *LNCS*, pages 275–294, 1997.

[22] M. Hofmann. Programming languages capturing complexity classes. *ACM SIGACT News*, 31(1):31–42, 2000.

[23] M. John C., M. Mark, and S. Andre. A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In *Foundations of Computer Science, 39th Annual Symposium, Proceedings*, pages 725–733. IEEE Computer Society, 1998.

[24] N. D. Jones. Logspace and ptime characterized by programming languages. *Theoretical Computer Science*, 228:151–174, October 1999.

[25] N. D. Jones and L. Kristiansen. A flow calculus of mwp-bounds for complexity analysis. *ACM Trans. Comput. Logic*, 10(4):28:1–28:41, 2009.

[26] K.-I. Ko. Some observations on the probabilistic algorithms and np-hard problems. *Inf. Process. Lett.*, 14(1):39–43, 1982.

[27] L. Kristiansen and N. D. Jones. The flow of data and the complexity of algorithms. In *Proceedings of the First international conference on Computability in Europe: new Computational Paradigms*, CiE'05, pages 263–274, Berlin, Heidelberg, 2005. Springer-Verlag.

[28] Y. Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318(1-2):163–180, 2004.

[29] D. Leivant. Stratified functional programs and computational complexity. In *Principles of Programming Languages, 20th International Symposium, Proceedings*, pages 325–333. ACM, 1993.

[30] D. Leivant and J.-Y. Marion. Ramified recurrence and computational complexity II: Substitution and poly-space. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic, 9th International Workshop, Proceedings*, volume 933 of *LNCS*, pages 486–500. 1995.

[31] R. Metnani and J.-Y. Moyen. Equivalence between the *mwp* and Quasi-Interpretations analysis. In J.-Y. Marion, editor, *DICE'11*, Apr. 2011.

[32] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. In *Proceedings of the 1967 22nd national conference*, ACM '67, pages 465–469, New York, NY, USA, 1967. ACM.

[33] C. M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[34] H. E. Rose. *Subrecursion: Functions and Hierarchies*. Oxford University Press, 1984.

[35] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177 – 192, 1970.

[36] H. Schwichtenberg and S. Bellantoni. Feasible computation with higher types. In *Proof and System-Reliability*, pages 399–415. Kluwer Academic Publisher, 2001.

[37] P. P. Toldin. Algebra implementation. `http://www.cs.unibo.it/~parisent/programs/MatrixCalculator.zip`.

[38] G. J. Tourlakis. *Computability*. Reston, 1984.

[39] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[40] Y. Zhang. The computational SLR: a logic for reasoning about computational indistinguishability. *Mathematical Structures in Computer Science*, 20(5):951–975, 2010.