UNIVERSITÉ PARIS 13

L'Université Paris 13 est membre de : USPC Université Sorbonne Paris Cité

CAMPUS CONDORCET Paris-Aubervilliers

Mémoire d'habilitation à diriger des recherches
Spécialité: informatique

Polyadic Approximations in Logic and Computation

Damiano Mazza

17 novembre 2017

Rapporteurs:

Robert Harper	Carnegie Mellon University
Martin Hyland	King's College Cambrigde
Paul-André Melliès	CNRS – Université Paris 7

Jury:

Patrick Baillot	CNRS – ENS Lyon
Pierre-Louis Curien	CNRS – Université Paris 7
Christophe Fouqueré	Université Paris 13
Stefano Guerrini	Université Paris 13
Robert Harper	Carnegie Mellon University
Martin Hyland	King's College Cambrigde
Paul-André Melliès	CNRS – Université Paris 7
Simona Ronchi Della Rocca	Università di Torino

CNRS, UMR 7030
Laboratoire d'Informatique de Paris Nord
Institut Galilée, Université Paris 13, SPC

To Plumpy and Pudgy

Preface

This document presents some aspects of the research I carried out after my Ph.D. thesis, which I defended nearly eleven years ago. In fact, for the sake of uniformity, I chose to focus on a single subject, that of polyadic approximations and their applications to quantitative program analysis and computational complexity, which means that the present document covers only research developed in the past five or six years.

My research on other topics, such as the denotational semantics of interaction nets and light subsystems of linear logic, graded comonads and quantitative coeffects, linear explicit substitutions and abstract machines, as well as logical approaches to process calculi, was intentionally left out. Aside from uniformity, the choice is justified by the fact that the interaction between logic, programming languages and computational complexity is my main research interest at present and I feel much more motivated in making the effort of organizing this material in a coherent form, rather than collecting a series of disparate papers.

Precisely because I did not wish to make of this document a mere compilation of previously published papers, I strove to present each result in the light of my most recent advances, which often led to a quite different presentation with respect to the published form. It is the case of Chapter 1, which takes the ideas contained in [Maz12, Maz13, Maz17] and puts them in a higher multicategorical context which is completely absent from those papers. I took a similar approach with Chapter 4, in which I use the tools of Chapter 2 to remold my paper [Maz16] and bring it closer to the world of "standard" computational complexity. The closing section of that chapter, which is quite speculative, is a sort of window into what I currently think of my future research.

Chapter 3 is the closest to its originally published form, containing sections taken verbatim from [Maz15], as well as results from a joint work with Kazushige Terui [MT15]. However, I do make the techniques of Chapter 2 appear explicitly in the exposition, whereas these were only implicit in the published presentation.

Finally, the above-mentioned Chapter 2 contains material which I developed with my Ph.D. students Luc Pellissier and Pierre Vial, and which is going to appear in the Proceedings of the ACM issue associated with POPL 2018.

Acknowledgments

Let me start by expressing my gratitude to Robert Harper, Martin Hyland and Paul-André Melliès for having accepted to review this work, a gratitude which extends to Patrick Baillot, Pierre-Louis Curien, Christophe Fouqueré, Stefano Guerrini and Simona Ronchi Della Rocca, for being part of my *habilitation* committee.

I also wish to thank my fellow researchers, coauthors and friends who shared with me this past decade of learning, discovering, proving and disproving, giving me much appreciated inspiration and support. An exhaustive list would be too long here, so I will restrict it to those who are somehow connected with the work presented here: Beniamino Accattoli, Flavien Breuvart, Ugo Dal Lago, Thomas Ehrhard, Marco Gaboardi, Yves Guiraud, Tom Hirschowitz, Michele Pagani, Andrew Polonsky, Kazushige Terui, Lorenzo Tortora de Falco and, last but not least, my students Luc Pellissier and Pierre Vial.

I am sincerely thankful to my colleagues at LIPN, particularly to all members of the logic group, for providing an amicable working environment. I also want to thank the administrative staff for their precious assistance, especially Aimé Bayonga, Marie Fontanillas and Brigitte Guéveneux.

I guess this is a good place to give credit to all the institutions who funded my research after my Ph.D.: the *Agence Nationale de la Recherche*, the *Fondation Sciences Mathématiques de Paris* and, of course, the *Centre Nationale de la Recherche Scientifique*, my employee for the past nine years. I am particularly grateful to the members of the 2007 hiring committee who believed in my potential and granted me the amazing opportunity of a life-long career at CNRS. Although it is unlikely that any of them will ever read this, the present work is hopefully a testimony to their judgment not being entirely misled.

Let me also give my special thanks to my sister Elena and brother-in-law Giampaolo, as well as my mom and dad, who provided invaluable practical help during the final stages of writing this thesis (no daycare for my daughter in August...).

Finally, it is hopeless to convey how grateful I am to my wife Kristin and her inexhaustible support and understanding. I may be unsure about the value of this work but, thanks to her, I am sure there is something valuable in my life.

Villetaneuse, 17 October 2017

Contents

Introduction

The notion of polyadic approximation originates in the very first paper on linear logic [Gir87], where Girard proves a result, deemed Approximation Theorem, supporting the "equation"

$$!A = \lim_{n\to\infty} \overbrace{(A \,\&\, 1) \otimes \cdots \otimes (A \,\&\, 1)}^{n}. \tag{1}$$

We recall that the $!(-)$ modality is the non-linear part of linear logic, *i.e.*, it mediates the introduction of the structural rules weakening and contraction. From the Curry-Howard perspective, weakening and contraction correspond to erasing and duplication, respectively, which are responsible (especially the latter) for the presence of infinity in computation: without duplication, every computation is bound to take place within a progressively shrinking space, making it finite in a very strong sense.

The formulas on the right hand side of Equation 1, instead, are !-free as long as A is. In other words, such formulas live in the "purely linear" fragment of linear logic. Girard's Approximation Theorem therefore hints at the fact that the non-linear part of linear logic is the limit of its purely linear part. We say that the result "hints" at such a relationship because, in fact, Equation 1 is never given a technical status in Girard's paper, in that there is no topology or other structure in which the limit may be taken in a formal sense.

The starting point of this thesis is the idea of taking Girard's Approximation Theorem seriously, thus giving a fully formal content to Equation 1. We will see that the consequences of such an endeavor are surprisingly rich, with applications ranging from the construction of intersection type systems to the development of a new viewpoint on the quantitative analysis of functional programming languages, with some implications reaching as far as structural complexity theory. We now proceed to give a quick overview of such results.

Polyadic approximations

Chapter 1 contains our formalization of the computational contents of Girard's Approximation Theorem. The idea is that Equation 1, which is given at the level of formulas/types, should actually be read first and foremost at the level of cut-elimination/execution.

The categorical view of the Curry-Howard correspondence sees formulas/types as objects and proofs/programs as morphisms. It is very natural

to extend this up the dimensional ladder and see cut-elimination/execution paths as 2-arrows. We take this approach, which is well-established in the literature on term rewriting [See87, Hil96, CG01, Hir13, Mel17], as the basis of our syntax for term calculi. This will allow us to define in a precise way what it means for a reduction sequence to approximate another one.

The only twist that we add with respect to the common two-dimensional approaches to term rewriting is the use of a higher *multi*categorical perspective rather than a 2-categorical one, *i.e.*, we follow a mixed operadic/globular approach rather than a fully globular one. This is inspired by Hyland's recent presentation of the λ-calculus in the framework of cartesian operads [Hyl17]. Although we take it in a rather different direction than Hyland's (in particular, we will never consider questions of denotational semantics, leaving them for future work), we will find this viewpoint extremely fruitful for our exposition.

After introducing several variants of polyadic calculi (with a varying degree of freedom as to the use of structural rules), we introduce the approximation order and show, via a general categorical construction, that the term calculus of intuitionistic linear logic is isomorphic to a quotient of the ideal completion of the affine polyadic calculus. In proof-theoretic terms, a linear logic proof using the modality $!(-)$ may be seen (modulo a quotient) as an ideal of proofs in the purely linear fragment of linear logic.

More evocatively, we may say that the ideal completion is a process similar to the completion of a metric space, which adds the "missing limits" of Cauchy sequences. This is one of the standard ways, due to Cantor, of defining the real numbers $\mathbb{R}$ from the rational numbers $\mathbb{Q}$. In our case, the space of purely linear proofs is, like $\mathbb{Q}$, incomplete; among the "missing limits" added by the completion we find the proofs of usual linear logic. Moreover, these may be characterized by a uniformity condition, which underlies the quotient mentioned above.

The most important consequence of our computational version of Girard's Approximation Theorem is that it induces the notion of *polyadic approximation*: linear logic reductions may be approximated arbitrarily well by polyadic reductions which, by nature, are much finer and better suited for quantitative analyses. Between terms, the approximation relation is written

$$t \sqsubset M,$$

where t is a polyadic term and M a linear logic term (or a λ-term, depending on the context).

We also observe that, although Girard's Approximation Theorem is about *affine* polyadic approximations, the notion actually makes sense in all the other variants of polyadic calculi, with any combination of structural rules (linear, relevant, cartesian). This is why, instead of speaking of "affine approximations", we chose the broader terminology "polyadic approximation": the key aspect is that the constructions associated with the $!(-)$ modality are approximated by greater and greater polyadic constructions; affinity is just one (particularly useful) possibility.

Intersection types

In Chapter 2, we show how every well-known intersection type system, as well as new potentially interesting examples, arises from a very general construction involving polyadic approximations at its heart.

The starting point is an observation concerning simple types in polyadic calculi (which are essentially fragments of propositional linear logic) and intersection types. Without being precise about what particular system of intersection types we are talking about (we will give the details in Sect. 2.1), the empirical remark is that:

- polyadic simple types and intersection types are isomorphic (so we identify them in what follows);
- if we take a derivation δ of

  $$\Gamma \vdash M : A$$

  in intersection types (where M is a pure λ-term), then δ is isomorphic to a Church-style (*i.e.*, with type decorations) simply-typed polyadic term $\delta : A$ with free variables of type Γ;
- moreover, if we denote by δ^- the underlying polyadic term (*i.e.*, the pure term without type decorations), then $\delta^- \sqsubset M$.

In order to explain the above observation, we are led to seek an abstract definition of what an intersection type system is. A surprisingly simple, but extremely general proposal was recently given by Melliès and Zeilberger [MZ15], who argue that a type system (hence, in particular, an intersection type system) is nothing but a functor

$$\begin{array}{c} \mathcal{E} \\ {\scriptstyle p}\downarrow \\ \mathcal{B} \end{array}$$

where the source category $\mathcal{E}$ is seen as a (generalized) category of derivations and the target category $\mathcal{B}$ as a (generalized) category of programs. This viewpoint of course works just as well in our 2-operadic framework, so we take it to be our working definition of type system. Accordingly, we seek a general way of constructing morphisms of 2-operads

$$\begin{array}{c} \mathcal{E} \\ {\scriptstyle p}\downarrow \\ \Lambda \end{array}$$

where Λ is the 2-operad presenting the pure λ-calculus.

To this effect, we show how, for any choice of suboperad $\mathcal{D}$ of polyadic simple type derivations, the polyadic approximations as introduced in Chapter 1 induce a morphism

$$\mathrm{Apx}[\mathcal{D}] : \Lambda_! \longrightarrow \mathfrak{Rel}$$

where $\Lambda_!$ is the 2-operad presenting the term calculus of linear logic and $\mathfrak{Rel}$ is a certain bioperad based on relational distributors (*i.e.*, functors $\mathcal{A} \times \mathcal{B}^{\mathrm{op}} \to \mathbf{Rel}$, where **Rel** is the category of sets and relations). We call this the *approximation presheaf* (relative to $\mathcal{D}$).

Now, the pure λ-calculus is well-known to embed in linear logic via Girard's translation

$$\mathbf{G}_0 : \Lambda \longrightarrow \Lambda_!.$$

Composing Girard's translation with the approximation presheaf gives us another presheaf

$$\mathrm{Apx}[\mathcal{D}] \circ \mathbf{G}_0 : \Lambda \longrightarrow \mathfrak{Rel}.$$

We then show how, using an operadic version of the Grothendieck construction (a standard categorical construction), this induces a type system (in the sense of Melliès and Zeilberger)

$$\begin{array}{c} \mathcal{E}[\mathcal{D}, \mathbf{G}_0] \\ {\scriptstyle \mathbf{p}[\mathcal{D},\mathbf{G}_0]}\big\downarrow \\ \Lambda \end{array}$$

It turns out that, by suitably varying $\mathcal{D}$ (and $\mathbf{G}_0$), we obtain in this way all the major variants of intersection type systems, from the original one of Coppo and Dezani [CDC80], to the later variant with intersections only to the left of arrows [CDCV81]; from the various non-idempotent versions [Gar94, dC09] to the several variants for characterizing different normalization properties (head normalization, weak normalization, etc. [Kri93]), as well as any combination of these.

This general construction has several interesting aspects:

- not only $\mathcal{D}$, but also the morphism $\mathbf{G}_0$ is just a parameter; the construction applies to any calculus $\mathcal{L}$ admitting an encoding

  $$\mathbf{G} : \mathcal{L} \longrightarrow \Lambda_!$$

  in linear logic. For instance, we will see examples where $\mathcal{L}$ is the call-by-value λ-calculus or the $\lambda\mu$-calculus.

- There are sufficient conditions on $\mathrm{Apx}[\mathcal{D}] \circ \mathbf{G}$ which, if met, automatically guarantee that the induced type system ensures certain normalization properties. In this way, not only the usual systems of intersection types, but also their properties may be recovered from the framework.

- In general, the construction shows how the structural properties of intersection types come from the structural rules of polyadic approximations: contraction gives idempotency, weakening gives (basic) subtyping.

This leads to a framework for synthesizing, in a nearly automatic way, systems of intersection types for *a priori* arbitrary calculi. It will prove its usefulness in both of the subsequent chapters.

Parsimony

In Chapter 3 we turn our attention to quantitative aspects of reduction and, in particular, find applications of polyadic approximations to implicit computational complexity. As in Chapter 1, we will focus on affine polyadic approximations.

One of the most important (and useful) properties of affine polyadic approximations is that reduction enjoys a continuity property with respect to them: let

$$M \to^* N;$$

then, for all

$$u \sqsubset N,$$

there exists

$$t \sqsubset M$$

such that

$$t \to^* u.$$

In other words, if we are interested in an approximation of the result of a linear logic/λ-calculus computation, it is enough to look at an affine computation starting from a sufficiently big approximation of the initial term.

This immediately leads to the question: how big must t be? Supposing that the size of u does not matter, for instance because N is a Boolean value and we only need a constant amount of information from it (the head variable), it is legitimate to expect that $|t|$ (the size of t) is a function of the length of the reduction $M \to^* N$. This is indeed the case; however, the dependency, which we informally call "modulus of continuity", is quite bad: it may be exponential in general.

We are therefore led to introduce the *parsimonious λ-calculus*, a calculus enjoying a polynomial modulus of continuity. The parsimonious paradigm is quite robust: apart from the calculus, there is an underlying logical system, a variant of linear logic called *parsimonious logic*, admitting a simple categorical semantics, with everything fitting together nicely as in the prototypical "trinity" of the λ-calculus, intuitionistic logic and cartesian closed categories. The parsimonious paradigm may also be extended along the standard directions: polymorphism (parsimonious system F) or simply-typed recursive definitions (parsimonious PCF). In each case, the fact that parsimony is centered upon the polynomial modulus of continuity has interesting consequences, which we discuss in some depth.

The viewpoint of computational complexity is especially important for us. In this respect, we show that parsimony is a fruitful context for the development of both explicit and implicit computational complexity. In particular, concerning the latter, parsimony brings forth a novel approach to linear-logic-based characterization of complexity classes, radically different than those obtained with the more established "light logics" [Gir98, Laf04].

Church meets Cook and Levin

In Chapter 4, we venture still farther into computational complexity territory, and show how the techniques developed in the previous chapters may be used to present in a novel way the proof of the Cook-Levin theorem, one of the central results of structural complexity theory.

The Cook-Levin theorem, which is the famous statement that SAT (satisfiability of propositional formulas) is NP-complete, is usually proved by showing how a formula or, better, a Boolean circuit may simulate the computation of a Turing machine with only a polynomial overhead. It is implicit in the way circuits are constructed in the proof that some notion of approximation is actually being used, and complexity theorists are well aware of this intuition.

We are able to give a fully formal content to this intuition, once again thanks to the notion of polyadic approximation and its connection with intersection types. The novelty with respect to the previous chapters is that, this time, we apply our machinery to a first-order language, not too distant from Turing machines. Instead of being approximated by polyadic terms, the programs of this language are approximated by actual Boolean circuits. Applying the construction of Chapter 2, we obtain an intersection type system for "Turing machines" whose derivations are isomorphic to Boolean circuits approximating them. The continuity of reduction and its polynomial modulus of continuity become, in this setting, a quantitative subject expansion property of the type system. From there, the proof of the Cook-Levin theorem unfolds quite smoothly.

Having found a type-theoretic presentation of the proof of a major theorem of computational complexity theory may seem like an interesting achievement, not because of the proof itself (it is essentially the same proof, presented in a different language) but because it confronts the language of our community with the needs of a different community, which is always an enriching experience. However, one quickly realizes that the "confrontation" offered by the Cook-Levin theorem is, in reality, still extremely limited and far from being of any consequence. We therefore conclude our exposition with some suggestions for further research in this direction, which is intended to stir up the interaction between logic, programming languages and computational complexity, hopefully bearing fruits for one of these domains.

Chapter 1

Polyadic Approximations

1.1 An Operadic Take on Syntax

1.1.1 Reduction terms

In the course of this thesis, we will often need to have a concise way of denoting reductions in term calculi. In the term rewriting community, such notations are known as "proof terms" and are well developed for the first-order case [Ter03]. In this section, we introduce a term notation for reductions of λ-terms which may be easily generalized to other term calculi. We prefer calling such notations *reduction terms*, because "proof terms" may be a source of ambiguity in the Curry-Howard context.

The syntax of λ-terms is the usual one:

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

Given $n \in \mathbb{N}$, a *context with n holes* is a λ-term C with n special free variables $\{\cdot\}_1, \ldots, \{\cdot\}_n$, such that each $\{\cdot\}_i$ appears exactly once in C. The case $n = 0$, in which a context is just a term, is allowed. Substitution of n terms $M_1, \ldots, M_n$ to the holes in a context C is denoted by $\mathsf{C}\{M_1, \ldots, M_n\}$, where it is assumed that M_i is substituted to $\{\cdot\}_i$. As usual, the difference with respect to usual substitution (of a term in a term) is that this substitution is *not* capture-free.

Definition 1 (reduction term) Reduction terms *are defined as follows:*

$$\begin{array}{rll} \beta ::= M(\!|x \leftarrow N|\!) & & \textit{basic steps;} \\ \rho, \tau ::= \mathsf{C}\{\beta_1, \ldots, \beta_n\} \mid \rho; \tau & & \textit{reduction terms.} \end{array}$$

In $M(\!|x \leftarrow N|\!)$, the variable x is bound in M. In $\mathsf{C}\{\beta_1, \ldots, \beta_n\}$, $n = 0$ is allowed, so terms are also reduction terms.

Each basic step is assigned a source term *and a* target term*, as follows:*

$$M(\!|x \leftarrow N|\!) : (\lambda x.M)N \to^* M\{N/x\}.$$

The definition of source and target is extended to reduction terms by means of the

$$\frac{\alpha_1 : M_1 \to^* M_1' \quad \dots \quad \alpha_n : M_n \to^* M_n'}{\mathsf{C}\{\alpha_1, \dots, \alpha_n\} : \mathsf{C}\{M_1, \dots, M_n\} \to^* \mathsf{C}\{M_1', \dots, M_n'\}}\ \mathsf{ctxt}$$

$$\frac{\rho : M \to^* P \qquad \tau : P \to^* N}{\rho;\tau : M \to^* N}\ \mathsf{comp}$$

Figure 1.1: Source and target for reduction terms.

$$\frac{\rho : M \to^* N \quad \tau : N \to^* P \quad \varphi : P \to^* Q}{(\rho;\tau);\varphi \equiv \rho;(\tau;\varphi)}\ \mathsf{assoc}$$

$$\frac{\rho : M \to^* M'}{M;\rho \equiv \rho}\ \mathsf{lunit} \qquad \frac{\rho : M' \to^* M}{\rho;M \equiv \rho}\ \mathsf{runit}$$

Figure 1.2: Structural equivalence on reduction terms.

rules of Fig. 1.1. Reduction terms with the same source (resp. target) are called coinitial *(resp.* cofinal*). A reduction term is* valid *only if it may be assigned a source and a target; we will implicitly consider only valid reduction terms.*

Structural equivalence *is the least congruence on reduction terms containing the equations of Fig. 1.2.*

The operation of (capture-free) substitution of a term N for a free variable x in a reduction term $\rho : M \to^* M'$, denoted by $\rho\{N/x\}$, is defined as expected. For what concerns substitution of a general reduction term $\tau : N \to^* N'$ in ρ, we first define it in case $\rho = M$, which is done by induction on τ:

- $M\{\mathsf{C}\{\beta_1, \dots, \beta_n\}/x\}$ is defined as expected;
- $M\{\tau';\tau''/x\} := M\{\tau'/x\};M\{\tau''/x\}$.

Then, we define

$$\rho\{\tau/x\} \ := \ \rho\{N/x\};M'\{\tau/x\}.$$

Substitution behaves in the expected way with respect to sources and targets:

Lemma 1 *If $\rho : M \to^* M'$ and $\tau : N \to^* N'$, then*

$$\rho\{\tau/x\} : M\{N/x\} \to^* M'\{N'/x\}.$$

Proof. A straightforward induction. □

Definition 2 (permutation equivalence) *We define a relation $\sim$, called* permutation equivalence*, on pairs of coinitial and cofinal reduction terms, as the least congruence containing structural equivalence (Fig. 1.2) and the equations of Fig. 1.3, where α, β are basic steps and, in the* par *rule, C is a two-hole context. Lemma 1 guarantees that the reduction terms related by Fig. 1.3 are indeed coinitial and cofi-*

$$\frac{\alpha : M \to^* M' \qquad \beta : N \to^* N'}{\mathsf{C}\{M,\beta\};\mathsf{C}\{\alpha,N'\} \sim \mathsf{C}\{\alpha,\beta\} \sim \mathsf{C}\{\alpha,N\};\mathsf{C}\{M',\beta\}}\ \textsf{par}$$

$$\frac{\alpha : \mathsf{C}\{x\} \to^* M' \qquad \beta : N \to^* N'}{\mathsf{C}\{\beta\};\alpha\{N'/x\} \sim \alpha\{N/x\};M'\{\beta/x\}}\ \textsf{unnest}$$

Figure 1.3: Permutation equivalence.

nal.

We stress that the above definition is simply a rephrasing of the standard definition of permutation equivalence. The three rules of Fig. 1.2 just take care of formalizing the fact that reduction terms are sequences of atomic reductions, with terms being identities; in Fig. 1.3, the par rule states the equivalence of two reductions reducing two independent ("parallel") redexes in two different orders, and the unnest rule states the equivalence of the two ways of reducing two nested redexes: first the inner one and then the outer one, or first the outer one and then all the potential copies of the inner one.

The most important role of permutation equivalence is allowing Proposition 3 below, *i.e.*, the *interchange law* between substitution and composition of reductions. We start by generalizing the unnest rule:

Lemma 2 *For all $\rho : M \to^* M'$ and $\tau : N \to^* N'$, we have $M\{\tau/x\};\rho\{N'/x\} \sim \rho\{N/x\};M'\{\tau/x\}$.*

Proof. By diagram chasing (technically, a double induction on ρ and τ). □

Proposition 3 (interchange law) *For all $\rho : M \to^* M'$, $\varphi : M' \to^* M''$, $\tau : N' \to^* N'$ and $\psi : N' \to^* N''$, we have $\rho\{\tau/x\};\varphi\{\psi/x\} \sim (\rho;\varphi)\{\tau;\psi/x\}$.*

Proof. We have

$$\begin{aligned}
\rho\{\tau/x\};\varphi\{\psi/x\} &= \rho\{N/x\};M'\{\tau/x\};\varphi\{N'/x\};M''\{\psi/x\} && \text{(def.)}\\
&\sim \rho\{N/x\};\varphi\{N/x\};M''\{\tau/x\};M''\{\psi/x\} && \text{(Lemma 2)}\\
&= (\rho;\varphi)\{N/x\};M''\{\tau/x\};M''\{\psi/x\} && \text{(def.)}\\
&= (\rho;\varphi)\{N/x\};M''\{\tau;\psi/x\} && \text{(def.)}\\
&= (\rho;\varphi)\{\tau;\psi/x\}, && \text{(def.)}
\end{aligned}$$

as claimed. □

The following lemma, which states that permutation equivalence and substitution are compatible, is technically important but unsurprising. It will be tacitly used in the sequel.

Lemma 4 *If $\rho \sim \rho'$ and $\tau \sim \tau'$, then $\rho\{\tau/x\} \sim \rho'\{\tau'/x\}$.*

PROOF. By induction on the number of elementary transformations (of Fig. 1.2 and Fig. 1.3) used to obtain $\rho \sim \rho'$ and $\tau \sim \tau'$. □

It is worth mentioning that what we gave above is an alternative presentation of the theory that Hilken calls 2-λ [Hil96], restricted to β-reduction (Hilken considers also η-expansion). It is also covered by Hirschowitz's general theory [Hir13], although our syntax is different (it is specialized to our purposes).

1.1.2 A strict 2-category of lambda-terms

Consider the set Λ_1 of λ-terms with at most one free variable x. Let us write $M \bullet N$ for $M\{N/x\}$; obviously, $M, N \in \Lambda_1$ implies $M \bullet N \in \Lambda_1$. Moreover, substitution is associative ($M\{N\{P/x\}/x\} = M\{N/x\}\{P/x\}$) and there is a neutral element, namely x, so $(\Lambda_1, \bullet, x)$ is a monoid or, more verbosely, a category with only one object.

The set Λ_1 may be equipped with another natural categorical structure, namely it may be taken as the set of objects of the free category generated by the graph of one-step β-reductions between λ-terms. The arrows of this category are exactly the reduction terms introduced above (modulo structural equivalence, which is always assumed implicitly): composition is the operation we denote by ; and identities are terms.

It is natural to wonder whether the monoid structure $(\Lambda_1, \bullet, x)$ may be extended to arrows. This means endowing the category of reductions with a (strict) monoidal structure or, equivalently, promoting the one-object category $(\Lambda_1, \bullet, x)$ to a one-object (strict) 2-category. We already introduced general substitution, so the obvious definition is $\rho \bullet \tau := \rho\{\tau/x\}$. Here is where we see the importance of permutation equivalence: this definition does not work unless we take the quotient under permutation equivalence.

So the right way to combine the two structures is as follows: Λ_1 is the strict 2-category with only one object whose

- 1-morphisms are λ-terms with at most one free variable x; composition is given by $\bullet$, the identity is the term x;
- 2-morphisms from M to N are reduction terms $\rho : M \to^* N$ modulo permutation equivalence; vertical composition is given by ; and its identities are the terms; horizontal composition is given by $\bullet$, with identity x (seen as a reduction term).

The fact that this defines a strict 2-category is ensured by Proposition 3 which, with our current notations, becomes $(\rho \bullet \tau); (\varphi \bullet \psi) = (\rho; \varphi) \bullet (\tau; \psi)$.

The above 2-categorical presentation of the λ-calculus was first advocated by Seely [See87]. The observation that permutation equivalence is necessary is implicit in his Definition/"Lemma" 2.5.[1] The category of pure λ-terms and reductions modulo permutation equivalence is also described in some detail

[1] The quote marks in "Lemma" are Seely's. What he meant is probably that, with more precise definitions than those he considers in his extended abstract, such as the ones we gave here, his definition should really be a lemma, indeed our Lemma 2.

by Melliès in his *habilitation* thesis [Mel17]; for instance, it is shown therein that it has pushouts, which is a strong form of confluence.

Although satisfactory in spirit, the 2-categorical framework is a bit artificial in practice, for the simple reason that λ-terms in general may have more than one free variable. In this respect, it feels much more natural to use multicategories, which is what we do in this thesis.

1.1.3 Operads

We give here a brief survey of the categorical definitions used in the sequel. We refer the reader to [Lei04] for more complete definitions.

Caveat on terminology: the definition of "operad" adopted in this thesis is not the standard one; it differs in the following points:

- our "operads" are colored by default, *i.e.*, seen as multicategories, they may have more than one object; we will occasionally call *monochromatic* an operad (in our sense) with only one object;
- our "operads" do not necessarily have identity operations; there is no standard terminology for these structures, which are however useful from the viewpoint of programming languages; an operad (in our sense) with identities will be called *unital*.

Summarizing, we may give the following "dictionary":

our terminology	**standard terminology**
operad	–
unital operad	colored operad/symmetric multicategory
unital monochromatic operad	operad

The reason why we chose to depart from standard terminology is simple: the concepts in the above table are listed, from top to bottom, in decreasing order of importance for our work; had we followed the standard terminology, we would have ended up calling our most used structures something like "non-unital colored operad" or "symmetric semimulticategory", quite a mouthful in both cases.

In what follows, $\mathbb{B}(n)$ denotes the group of permutations on n elements.

Definition 3 ($\mathcal{V}$-operad) *Let $(\mathcal{V}, \otimes, 1)$ be a symmetric monoidal category. A $\mathcal{V}$-*operad *$\mathcal{C}$ is given by the following:*

- *a set*[2] *$\mathcal{C}_0$ of* objects*, also called* colors*;*
- *for every sequence of colors $C_1, \ldots, C_n, A$, an object $\mathcal{C}(C_1, \ldots, C_n; A)$ of $\mathcal{V}$;*
- *for every color A, sequence of colors $\Delta = B_1, \ldots, B_n$ and sequence of sequences of colors $\mathbf{\Gamma} = \Gamma_1, \ldots, \Gamma_n$, a morphism of $\mathcal{V}$*

$$\circ_{\mathbf{\Gamma};\Delta;A} : \mathcal{C}(\Delta; A) \otimes \mathcal{C}(\Gamma_1; B_1) \otimes \cdots \otimes \mathcal{C}(\Gamma_n; B_n) \longrightarrow \mathcal{C}(\mathbf{\Gamma}; A),$$

called operadic composition*;*

- *for each $n \in \mathbb{N}$, $\sigma \in \mathbb{B}(n)$ and colors $A, \Gamma = C_1, \ldots, C_n$, a morphism of $\mathcal{V}$*

$$\mathsf{exch}_\sigma^{\Gamma;A} : \mathcal{C}(\Gamma; A) \longrightarrow \mathcal{C}(\sigma\Gamma; A)$$

where $\sigma\Gamma = C_{\sigma(1)}, \ldots, C_{\sigma(n)}$, satisfying $\mathsf{exch}_{\sigma'}^{\sigma\Gamma;A} \circ \mathsf{exch}_\sigma^{\Gamma;A} = \mathsf{exch}_{\sigma'\circ\sigma}^{\Gamma;A}$ and $\mathsf{exch}_{\mathrm{id}}^{\Gamma;A} = \mathrm{id}_{\mathcal{C}(\Gamma;A)}$;

such that the operadic composition morphisms satisfy the obvious associativity laws and compatibility laws with the exchange morphisms.

A unital $\mathcal{V}$-operad *is an operad equipped, for each color A, of a morphism of $\mathcal{V}$*

$$\mathrm{id}_A : 1 \longrightarrow \mathcal{C}(A; A)$$

such that the obvious neutrality laws with respect to operadic composition hold.

An operad whose set of colors is a singleton will be called monochromatic*. In that case, if $*$ is the only color, we write $\mathcal{C}(n)$ for $\mathcal{C}(\underbrace{*, \ldots, *}_{n}; *)$.*

Note that, in the unital case, operadic composition may also be defined "pointwise", *i.e.*, one may instead consider morphisms

$$\circ^i_{\Gamma;\Delta;A} : \mathcal{C}(\Delta; A) \otimes \mathcal{C}(\Gamma; B_i) \longrightarrow \mathcal{C}(B_1, \ldots, B_{i-1}, \Gamma, B_{i+1}, \ldots, B_n; A)$$

where $\Delta = B_1, \ldots, B_n$, such that the suitable compatibility conditions hold. We will sometimes resort to this equivalent definition. In any case, the subscripts are omitted in practice because usually clear from the context.

Definition 4 (morphism of $\mathcal{V}$-operads) *A* morphism $F : \mathcal{C} \to \mathcal{D}$ *of $\mathcal{V}$-operads is given by:*

- *a function $F_0 : \mathcal{C}_0 \to \mathcal{D}_0$;*
- *for each color A and sequence of colors Γ of $\mathcal{C}$, a morphism of $\mathcal{V}$*

$$F_{\Gamma;A} : \mathcal{C}(\Gamma; A) \to \mathcal{D}(F_0\Gamma; F_0 A);$$

such that, for every color A, sequence $\Delta = B_1, \ldots, B_n$ of colors and sequence of sequences $\mathbf{\Gamma} = \Gamma_1, \ldots, \Gamma_n$ of colors of $\mathcal{C}$, the following diagrams commute in $\mathcal{V}$:

$$\begin{array}{ccc}
\mathcal{C}(\Delta; A) \otimes \bigotimes_{i=1}^n \mathcal{C}(\Gamma_i; B_i) & \xrightarrow{\quad\circ_{\mathbf{\Gamma};\Delta;A}\quad} & \mathcal{C}(\mathbf{\Gamma}; A) \\
\Big\downarrow{\scriptstyle F_{\Delta;A} \otimes \bigotimes_{i=1}^n F_{\Gamma_i;B_i}} & & \Big\downarrow{\scriptstyle F_{\mathbf{\Gamma};A}} \\
\mathcal{D}(F_0\Delta; F_0 A) \otimes \bigotimes_{i=1}^n \mathcal{D}(F_0\Gamma_i; F_0 B_i) & \xrightarrow[\quad\circ_{F_0\mathbf{\Gamma};F_0\Delta;F_0 A}\quad]{} & \mathcal{D}(F_0\mathbf{\Gamma}; F_0 A)
\end{array}$$

[2]We only need *small* operads for managing the syntax. We will occasionally use large operads, silently glossing over size issues.

$$\begin{array}{ccc} \mathcal{C}(\Delta; A) & \xrightarrow{\mathsf{exch}_\sigma^{\Delta;A}} & \mathcal{C}(\sigma\Delta; A) \\ \downarrow{\scriptstyle F_{\Delta;A}} & & \downarrow{\scriptstyle F_{\sigma\Delta;A}} \\ \mathcal{D}(F_0\Delta; F_0A) & \xrightarrow[\mathsf{exch}_\sigma^{F_0\Delta;F_0A}]{} & \mathcal{D}(\sigma F_0\Delta; F_0A) \end{array}$$

In practice, one denotes F_0 and $F_{\Gamma;A}$ simply by F.

A unital morphism *of unital $\mathcal{V}$-operads is as above but further satisfies, for every color A of $\mathcal{C}$, $F_{A;A} \circ \mathrm{id}_A = \mathrm{id}_{F_0A}$.*

We denote by $\mathcal{V}$-**Op** the category of $\mathcal{V}$-operads and their morphisms.

The plain term "operad" is usually understood to mean **Set**-operad (the monoidal structure on **Set** is given by the cartesian product). For us, the most important case of $\mathcal{V}$-operad is when $\mathcal{V} = \mathbf{Cat}$, the category of categories, with the monoidal structure given by the product of categories.

Definition 5 (2-operad, bioperad) *A (strict)* 2-operad *is a* **Cat***-operad. A bit more explicitly, a 2-operad $\mathcal{C}$ consists of a set of colors and a family of categories $\mathcal{C}(\Gamma; A)$, whose objects are called* multimorphisms *and whose arrows are called* 2-arrows*. Operadic composition is implemented by a family of functors between these categories. Accordingly, a morphism of 2-operads is a family of functors suitably commuting with the composition functors.*

A weak 2-operad*, or* bioperad*, is defined similarly to a 2-operad, but associativity and compatibility with exchange are only required to be satisfied up to coherent natural isomorphisms.*

We will occasionally use other hom-categories than **Set** and **Cat**, which is why we gave the general definition. On the other hand, we will use the notion of bioperad only in a very specific case, which is why the definition is not given in detail. The details, which may be found in [CO17], are obtained by following the standard yoga of weak enrichment: bioperads are a special case of operads weakly enriched over a symmetric monoidal 2-category, namely $(\mathbf{Cat}, \times, \mathbf{1})$.

Every unital $\mathcal{V}$-operad $\mathcal{C}$ induces a $\mathcal{V}$-enriched category $\mathcal{C}_1$ by restricting to multimorphisms with only one source. For instance, if $\mathcal{C}$ is a unital 2-operad, $\mathcal{C}_1$ is the strict 2-category whose objects are the colors of $\mathcal{C}$, whose 1-morphisms $A \to B$ are the objects of $\mathcal{C}(A; B)$, and whose 2-morphisms are the arrows of $\mathcal{C}(A; B)$; vertical composition is composition of $\mathcal{C}(A; B)$, whereas horizontal composition is given by operadic composition restricted to $\mathcal{C}(B; A) \times \mathcal{C}(B; C) \to \mathcal{C}(A; C)$.

In fact, unital (**Set**-)operads generalize symmetric monoidal categories: every such category $(\mathcal{D}, \otimes, 1)$ induces a unital operad $\mathbf{M}\mathcal{D}$ by letting

$$\mathbf{M}\mathcal{D}(C_1, \ldots, C_n; A) := \mathcal{D}(C_1 \otimes \cdots \otimes C_1, A).$$

Not every unital operad arises in this way (example: the unital operad $\mathcal{T}wo$ with two objects a, b such that $\mathcal{T}wo(a; a) = \mathcal{T}wo(b; b)$ is a singleton and every other homset is empty). Those that do are called *representable*. It is a theorem of Hermida [Her00] that a unital operad $\mathcal{C}$ is representable iff there is an object

1 and, for every pair of objects A, B, there is an object $A \otimes B$ such that, for all Γ, C, there are bijections

$$\mathcal{C}(\Gamma, 1; C) \cong \mathcal{C}(\Gamma; C) \qquad \text{and} \qquad \mathcal{C}(\Gamma, A, B; C) \cong \mathcal{C}(\Gamma, A \otimes B; C),$$

natural in Γ, C. Such bijections exist by definition if $\mathcal{C}$ is representable. For the converse, the proof shows that the existence of such a structure is enough to make $\mathcal{C}_1$ (the induced category introduced above) symmetric monoidal, and then checks that $\mathbf{M}\mathcal{C}_1$ is equivalent to $\mathcal{C}$. (Now we may prove why $\mathcal{T}wo$ is not representable: if it were, $\mathcal{T}wo(a, b; a)$ would have to be isomorphic to $\mathcal{T}wo(c; a)$ for some c; since $\mathcal{T}wo(a, b; a) = \emptyset$ and $\mathcal{T}wo(a; a) \neq \emptyset$, we must have $c = b$; but then $\mathcal{T}wo(a, b; b) \cong \mathcal{T}wo(b; b)$, which is false).

Similarly, the notion of 2-operad is a generalization of the notion of strict symmetric monoidal 2-category. In this respect, we find 2-operads to be slightly lighter to present: the associativity/exchange laws for operadic composition are in our opinion more intuitive than the coherence laws for symmetric monoidal 2-categories. This becomes even more conspicuous in the weak case: although laborious (so much so that we preferred not to give it!), the definition of bioperad seems to us less imposing than that of symmetric monoidal bicategory, a fully detailed definition of which may be found in [Sta16], where it occupies roughly 14 pages... But this may only be a matter of taste of course.

Usually, one introduces (**Set**-)operads to study their algebras. Given an operad $\mathcal{C}$ and a symmetric monoidal category $(\mathcal{D}, \otimes, 1)$, a *$\mathcal{C}$-algebra in $\mathcal{D}$* is a morphism of operads

$$\mathbf{A} : \mathcal{C} \longrightarrow \mathbf{M}\mathcal{D}.$$

A motivating example: let $\mathbf{1}$ be the terminal unital monochromatic operad, such that $\mathbf{1}(n)$ is a singleton for all $n \in \mathbb{N}$; then, a $\mathbf{1}$-algebra in a symmetric monoidal category $\mathcal{D}$ is exactly a commutative monoid in $\mathcal{D}$. Sometimes (*e.g.* from the perspective of Lawvere theories) one is interested only in algebras in a *cartesian* category. In that case, it is enough to consider algebras in $(\mathbf{Set}, \times, 1)$: indeed, the copresheaf $\mathcal{D}(1, -) : \mathcal{D} \to \mathbf{Set}$ preserves products and thus induces a morphism $\mathbf{M}\mathcal{D} \to \mathbf{M}\,\mathbf{Set}$, so an algebra in $\mathcal{D}$ induces an algebra in $\mathbf{Set}$.

In this thesis, we will never study the algebras of the operads we introduce.[3] This does not mean that they are not interesting: on the contrary, they correspond to *denotational semantics*, as we will explain momentarily.

1.1.4 Term calculi as 2-operads

From the point of view of a programming languages theorist, 2-operads are probably best understood as an abstract way of speaking of (typed) term calculi:

- colors are types;

[3] This is another reason why our use of the terminology "operad" is a bit at odds with tradition; but we already explained that it is a choice dictated mostly by conciseness.

- a multimorphism $t : C_1, \ldots, C_n \to A$ is a term in context

$$x_1 : C_1, \ldots, x_n : C_n \vdash t : A;$$

operadic composition is substitution, *i.e.*, $t \circ^i u = t\{u/x_i\}$, and the action of the symmetric group corresponds to the possibility of injectively renaming free variables or, equivalently, to the presence of an exchange rule on typing contexts;

- a 2-arrow $t \Rightarrow t'$ is a type-preserving computation starting from t and leading to t' (*e.g.* term-rewriting, cut-elimination, etc.). Computations are required to be compatible with substitution, in the sense that, for any two computations $t \Rightarrow t'$ and $u \Rightarrow u'$, there is a computation $t\{u/x\} \Rightarrow t'\{u'/x\}$ and such a computation is equal to both sides of the diagram

$$\begin{array}{ccc} t\{u/x\} & \Longrightarrow & t'\{u/x\} \\ \Downarrow & & \Downarrow \\ t\{u'/x\} & \Longrightarrow & t'\{u'/x\} \end{array}$$

where the intermediate computations are obtained by considering the identity computations on t and u. If we see variables as parameters, so that terms are parametric programs, then the above is saying that computations too are parametric.

In this perspective, a morphism $\mathbf{f} : \mathcal{D} \to \mathcal{C}$ of 2-operads is just a modular, semantic-preserving encoding/translation/compilation of term languages:

- a type A of $\mathcal{D}$ is encoded by $\mathbf{f}(A)$ in $\mathcal{C}$;
- a term $x_1 : C_1, \ldots, x_n : C_n \vdash t : A$ of $\mathcal{D}$ is encoded by $x_1 : \mathbf{f}(C_1), \ldots, x_n : \mathbf{f}(C_n) \vdash \mathbf{f}(t) : \mathbf{f}(A)$;
- we have a "substitution lemma" $\mathbf{f}(t\{u/x\}) = \mathbf{f}(t)\{\mathbf{f}(u)/x\}$ (modularity);
- if $t \Rightarrow t'$, then $\mathbf{f}(t) \Rightarrow \mathbf{f}(t')$ (preservation of operational semantics) and modularity holds for computations too.

Following the above intuition, it is natural to consider unital operads: identities are just variables. However, unital morphisms are less anodyne: they force the encoding of a variable to be necessarily a variable. We will see very natural examples of translations of programming languages in which this requirement is not met, leading us to consider 2-operads to be non-unital in general. Indeed, although naturally occurring 2-operads are unital, including the syntactic ones coming from programming languages, the category of unital operads and non-unital morphisms is not well behaved (for instance, it lacks pullbacks), justifying the relaxation of the notion of operad itself.

At this point, the reader will likely be able to guess how the 2-categorical presentation of the pure λ-calculus of Sect. 1.1.2 may be promoted to a 2-operadic one. We will do so in a slightly more general setting, that of simple

types (which is actually the one originally considered by Seely [See87]). By simple types we mean of course those generated by

$$A, B ::= \alpha \mid A \to B;$$

it is immediate to adapt Definition 1 to the Church-style simply-typed λ-calculus: just add type decorations as appropriate.

Definition 6 (2-operads of λ-terms) *Let Λ_{ST} be the following (unital) 2-operad:*

- *its colors are the simple types;*
- *its multimorphisms $C_1, \dots, C_n \to A$ are simply-typed Church-style λ-terms $M : A$ such that $\mathrm{fv}(M) \subseteq \{x^1 : C_1, \dots, x^n : C_n\}$; by Curry-Howard, this is the same as saying that multimorphisms are type derivations of*

  $$x^1 : C_1, \dots, x^n : C_n \vdash M : A,$$

  with M a pure term;
- *its 2-arrows are Church-style reduction terms modulo permutation equivalence;*
- *its operadic composition functors are defined by means of substitution: on 2-arrows, we let*

  $$\rho \circ (\tau_1, \dots, \tau_n) := \rho\{\tau_1, \dots, \tau_n / x^1, \dots, x^n\},$$

  whenever this makes sense with respect to sources and targets. The same definition applies to multimorphisms (seen as identity reductions). The identity multimorphism on A is $x^1 : A$.

The (unital) 2-operad Λ of pure λ-terms is obtained by forgetting the colors: $\Lambda(n)$ is the category whose objects are λ-terms with free variables contained in $\{x^1, \dots, x^n\}$ and whose arrows are reduction terms modulo permutation equivalence.

Note that the 2-category Λ_1 of Sect. 1.1.2 arises from the 2-operad Λ precisely by restricting to $\Lambda(1)$, as described at the end of Sect. 1.1.3.

It is fairly evident that the λ-calculus, simply-typed or pure, is little more than a parameter in the above definition. Indeed, suppose we are given:

- a set of types;
- rules for generating terms in typing contexts $\Gamma \vdash_{\mathcal{C}} t : A$;
- a set of basic steps with source and target $\alpha : t \to^* u$, preserving types.

Multi-hole contexts may be defined from the term syntax in the obvious way; from that, Definition 1 and Figures 1.1, 1.2 and 1.3 may be applied verbatim, guaranteeing the development of Sect. 1.1.1 (we encourage the reader to check that no result in there depends on the λ-calculus). Then, Definition 6 may be adapted to define a 2-operad $\mathcal{C}$ whose colors, multimorphisms and 2-arrows

are types, typed terms and reduction terms modulo permutation equivalence, respectively, with operadic composition given by substitution. The calculi we use in the sequel will be presented following the above specification.

We should stress that the above definition does not *always* yield the 2-operad that one may want to associate with a given term calculus. Typically, one may want to add more equations to Fig. 1.3 for permutation equivalence, because some critical pairs may not be covered by that (very basic) definition. However, for our purposes, the above definition works just fine.

It is important at this point to highlight that the operadic approach to programming languages has non-trivial restrictions: presenting calculi whose evaluation contexts do not coincide with every possible "program with a hole" may be problematic and may require some tricky workaround. This is because 2-arrows in a 2-operad must not only compose "vertically" (this is concatenation of reductions, which is usually unproblematic) but also "horizontally", *i.e.*, given $\rho : t \to^* t'$ and $\tau : u \to^* u'$, one must be able to define $\rho\{\tau/x\} : t\{u/x\} \to^* t'\{u'/x\}$. This may bring trouble: if $\rho = \mathsf{C}\{x\}$ is an identity reduction with only one free occurrence of x such that C is *not* an evaluation context, then $\rho\{\tau/x\} = \mathsf{C}\{\tau\}$ will in general not be a "legal" reduction. For instance, restricting Λ to most typical reduction strategies (head reduction, weak head reduction, leftmost reduction, non-erasing reduction, Böhm reduction...) does *not* yield a 2-operad (example for head reduction: $\tau := x(\!|x \leftarrow I|\!) : II \to I$ is a head reduction but $(zy)\{\tau/y\} : z(II) \to zI$ is not). So, although operads are undoubtedly useful (as the coming chapter will hopefully show), they have their limitations and are by no means the ultimate answer to the question of modeling programming languages.

The (**Set**-enriched) operadic approach to term calculi of course is not new. It was brought forth (at least) by Fiore, Plotkin and Turi [FPT99] and more recently adopted by Hyland [Hyl17], whose work is our main source of inspiration. We use a **Cat**-enriched setting because we want to be able to have a direct account of computation.

Let $(\mathcal{D}, \otimes, 1)$ be a symmetric monoidal category; we may see it as a strict symmetric monoidal 2-category by promoting its hom-sets to discrete hom-categories. Then, if $\mathcal{C}$ is a 2-operad, a $\mathcal{C}$-algebra in $\mathcal{D}$ is a morphism

$$[\![\cdot]\!] : \mathcal{C} \longrightarrow \mathbf{M}\mathcal{D}.$$

Spelling this out with the perspective of term calculi in mind, we have:

- for each type A, an object $[\![A]\!]$;
- for each term $x_1 : C_1, \dots, x_n : C_n \vdash t : A$, a morphism
$$[\![t]\!] : [\![C_1]\!] \otimes \cdots \otimes [\![C_1]\!] \to [\![A]\!];$$
- such that $[\![t\{u/x\}]\!] = [\![t]\!] \circ [\![u]\!]$;
- and such that, whenever $t \to^* t'$, $[\![t]\!] = [\![t']\!]$.

The latter condition results from the fact that each hom-category of $\mathcal{D}$ (and thus of $\mathbf{M}\mathcal{D}$) is discrete: every 2-arrow of $\mathcal{C}$ is mapped to an identity. So, if $\mathcal{C}$

is a term calculus, a $\mathcal{C}$-algebra in $\mathcal{D}$ is precisely a denotational model of $\mathcal{C}$ in $\mathcal{D}$. In [Hyl17], Hyland studies in detail Λ-algebras in **Set**, proving that these are equivalent to certain operadic structures he calls *λ-theories*, among which Λ is initial. In the following, we give a quick overview of the **Cat**-enriched version of such a structure.

1.1.5 Digression: logical structure

Most of the 2-operads we will consider in this thesis present a programming language of some sort. In every case, such 2-operads do not "fall from the sky" but originate from a precise logical structure, which generates them as a whole, from the objects, to the multimorphisms, to the 2-arrows.

While we are unable at present to give a general definition of what we mean by "logical structure", we may give a first sketch of a definition, which covers some very basic cases and gives a rough idea of what we are after. In the following, we write

$$\Gamma \vdash_{\mathcal{C}} A$$

for the hom-category $\mathcal{C}(\Gamma; A)$ of a 2-operad $\mathcal{C}$.

Definition 7 (naive propositional connective) *Let $\mathcal{C}$ be a 2-operad. Given $n \in \mathbb{N}$, $p_1, \dots, p_h > 0$ such that $\sum p_i = n$, a* naive positive connective *of arity n on $\mathcal{C}$ is given by the following data:*

- *a function $\Phi : \mathcal{C}_0^n \to \mathcal{C}_0$;*
- *for every object C and sequences of objects Γ and*

  $$\Xi = \Delta_1, \dots, \Delta_h,$$

  such that Δ_i is of length p_i, an adjunction

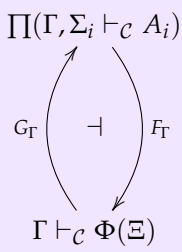
  $$\begin{array}{ccc} \prod(\Gamma, \Delta_i \vdash_{\mathcal{C}} C) & & \\ \uparrow & & \\ G_{\Gamma;C} & \dashv & F_{\Gamma;C} \\ \downarrow & & \\ \Gamma, \Phi(\Xi) \vdash_{\mathcal{C}} C & & \end{array}$$

  natural in Γ and C (in the obvious sense) and compatible with operadic composition.

Given $n \in \mathbb{N}$, $q_1, \dots, q_k > 0$ such that $\sum(q_i + 1) = n$, a negative connective *of arity n on $\mathcal{C}$ is given by the following data:*

- *a function $\Phi : \mathcal{C}_0^n \to \mathcal{C}_0$;*
- *for every sequences of objects Γ and $\Xi = \Sigma_1, A_1, \dots, \Sigma_k, A_k$, such that Σ_i is*

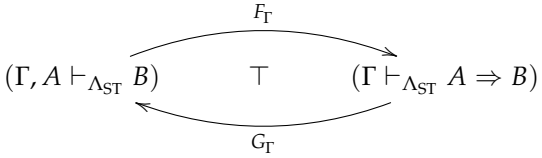
of length q_i, an adjunction



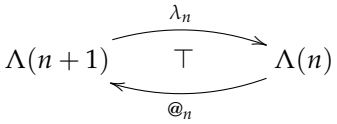
natural in Γ and compatible with operadic composition.

The family F of functors gives the reversible rule for the connective Φ in sequent calculus: it is a left rule if Φ is positive, a right rule if Φ is negative. The family G gives the irreversible rule. The counit $\beta : GF \Rightarrow \mathrm{Id}$ of the adjunction is the cut-elimination step associated with Φ; the unit $\eta : \mathrm{Id} \Rightarrow FG$ is the corresponding η-expansion step.[4]

All propositional intuitionistic logical connectives that the author is aware of, including those of linear logic, are captured by the above definition. For instance, implication is a negative connective, whose corresponding adjunction is



For simplicity, let us look at the details in the case of the pure λ-calculus. The above adjunction becomes



where (for simplicity of notations, we will write x instead of x_{n+1}):

- λ_n maps M to $\lambda x.M$ and a reduction $M \to^* M'$ to the obviously induced reduction $\lambda x.M \to^* \lambda x.M'$;
- $@_n$ maps M to Mx and a reduction $M \to^* M'$ to $Mx \to^* M'x$ in the obvious way;
- the component β_M of the counit must be an arrow $(\lambda x.M)x \to^* M$, which may obviously be taken as a single β-reduction step;
- the component η_M of the unit must be an arrow $M \to^* \lambda x.Mx$, which may obviously be taken as a single step of η-expansion (remember that $x = x_{n+1}$, which is not free in M because $M \in \Lambda(n)$, hence $\mathrm{fv}(M) \subseteq \{x_1, \ldots, x_n\}$).

[4] A lucky coincidence of notations! It would have been unfortunate if the unit of the adjunction, usually denoted by η, turned out to correspond to β-reduction…

The usual β-reduction step $(\lambda x.M)N \to M\{N/x\}$ is obtained from

$$\beta_M \circ^{n+1} \mathrm{id}_N.$$

Compatibility of the adjunction with operadic composition ensures, for instance, that $(\lambda x.M)\{N/y\} = \lambda x.M\{N/y\}$.

The adjunction forces the 2-arrows

$$Mx \to_\eta (\lambda x.Mx)x \to_\beta Mx$$

and

$$\lambda x.M \to_\eta \lambda x.(\lambda x.M)x \to_\beta \lambda x.M$$

to be identities. Albeit natural, this does not seem anodyne from a purely syntactic standpoint. Of course, one usually does not even want to deal with η-expansion; this is possible by stipulating that F and G form, instead of an adjunction, a lax retraction, with the same naturality and compatibility properties.

What Hyland calls a λ-theory in [Hyl17] is precisely the above structure, but in a **Set**-enriched setting, so the retraction is a strict one (it cannot be lax anymore).

Inductive datatypes also follow the above pattern. For instance, the type Nat of unary integers is characterized by the following adjunction:

$$(\Gamma \vdash_{\mathcal{C}} A) \times (\Gamma, \mathsf{Nat} \vdash_{\mathcal{C}} A) \quad \underset{G_{\Gamma,A}}{\overset{F_{\Gamma,A}}{\top}} \quad (\Gamma, \mathsf{Nat} \vdash_{\mathcal{C}} A)$$

Albeit admittedly superficial, we thought it was worthwhile to record these observations here, because we think it will be interesting to develop them more deeply. In particular, we made no attempt at defining our calculi as initial 2-operads having a certain structure of the above form, but we think this should be done at some point.

In a wider perspective, understanding logical structure at this level is understanding, in some sense, what the Curry-Howard isomorphism is an isomorphism of. It is definitely not an isomorphism of cartesian closed categories, as some may be tempted to say, for at least two reasons:

- the **Set**-enriched approach obviously looses a great deal of dynamic information;
- forcing λ-abstraction to be defined only in presence of pairing does not seem to do justice to the syntax; Hyland's approach is much more natural in this respect.

Understanding logical structure is also understanding what defines a connective, its logical rules and its cut-elimination rules so that they form a cohesive whole. This is especially relevant in type theory, where this kind of triple (formation rules, introduction/elimination rules, computational rules) is an

Colors: l, c (ranged over by C below).

Terms:

$$\frac{}{a : \mathsf{l} \vdash a : \mathsf{l}}\ \mathsf{lvar} \qquad \frac{\Gamma, a : \mathsf{l} \vdash T : \mathsf{l}}{\Gamma \vdash \lambda a.T : \mathsf{l}}\ \mathsf{lam} \qquad \frac{\Gamma \vdash T : \mathsf{l} \quad \Delta \vdash U : \mathsf{l}}{\Gamma, \Delta \vdash TU : \mathsf{l}}\ \mathsf{app}$$

$$\frac{}{x : \mathsf{c} \vdash x : \mathsf{l}}\ \mathsf{cvar} \qquad \frac{\overline{x} : \mathsf{c} \vdash T : \mathsf{l}}{\overline{x} : \mathsf{c} \vdash\ !T : \mathsf{c}}\ \mathsf{box} \qquad \frac{\Delta \vdash U : \mathsf{l} \quad \Gamma, x : \mathsf{c} \vdash T : \mathsf{l}}{\Gamma, \Delta \vdash T[!x := U] : \mathsf{l}}\ \mathsf{let}$$

$$\frac{\Gamma \vdash T : \mathsf{c}}{\Gamma \vdash T : \mathsf{l}}\ \mathsf{lin} \qquad \frac{\Gamma \vdash T : C}{\Gamma, x : \mathsf{c} \vdash T : C}\ \mathsf{weak} \qquad \frac{\Gamma, x : \mathsf{c}, y : \mathsf{c} \vdash T : C}{\Gamma, x : \mathsf{c} \vdash T\{x/y\} : C}\ \mathsf{cntr}$$

Basic steps:

$$\begin{array}{rrcl} T(\!|a \leftarrow U|\!)[-] : & (\lambda a.T)[-]U & \to^* & T\{U/a\}[-] \\ T(\!|x \leftarrow U|\!)_![-] : & T[!x := !U[-]] & \to^* & T\{U/x\}[-] \end{array}$$

The notation $[-]$ is a metavariable for *substitution contexts*, defined as follows:

$$[-] ::= \{\cdot\} \mid [-][!x := U].$$

We write $T[-]$ instead of $[-]\{T\}$.

Figure 1.4: The term calculus $\Lambda_!$ for intuitionistic linear logic.

essential building block which is currently lacking a precise mathematical formulation. So there is perhaps something important to be uncovered here, and further investigation seems necessary, although it will not be the subject of this thesis.

1.2 Polyadic Variants of Linear Logic

1.2.1 A term calculus for intuitionistic linear logic

We start by presenting a calculus corresponding, via Curry-Howard, to the proofs of the implicational fragment of intuitionistic multiplicative exponential linear logic. Apart from being a main actor in the sequel, it will be the occasion to introduce our contextual treatment of rewriting in presence of let binders, which we borrow from Accattoli [Acc12].

The 2-operad $\Lambda_!$ is defined in Fig. 1.4. There are two colors: l, for *linear*, and c, for *cartesian*.To make the distinction between linear and cartesian variables even more apparent, we use a, b to range over the former and x, y to range over the latter. True to their name, cartesian variables may be weakened and contracted, linear variables may not.

Note that the only terms of color c are of the form $!T$; these are called *boxes*. The lin rule coerces boxes to also have color l. Operadic composition for the color c is defined as follows:

$$T' \circ_{\mathsf{c}}^{i} !T := T'\{T/x^i\},$$

i.e., the $!(-)$ is removed when the substitution is performed. On the other hand, operadic composition on l is plain substitution; therefore, when a box $!T$ is seen as a linear term, the $!(-)$ is not removed:

$$T' \circ_{\mathsf{l}}^{i} !T = T'\{!T/a^i\}.$$

The operad is unital: a^1 (resp. $!x^1$) is the identity of color l (resp. c).

The term $T[!x := U]$ would perhaps more traditionally be written $\mathsf{let}\, !x = U\, \mathsf{in}\, T$. We use the explicit substitution notation because we define reduction following the methodology that Accattoli developed for the *linear substitution calculus* [Acc12]. This is essentially a way of importing the behavior of graph rewriting (as exemplified, for instance, by proof nets) in term rewriting. Let us look at an example. Let

$$T := (\lambda a.(\lambda b.xb)[!x := a])UW.$$

By reducing the redex $(\lambda a.-)U$, we get

$$T \to (\lambda b.xb)[!x := U]W =: T_1.$$

In proof nets, there is now another cut which may be reduced immediately, corresponding to the λb applied to W. However, in the above term, there is an explicit substitution between abstraction and argument which prevents us from seeing a β-redex, at least not one of the usual form $(\lambda b.-)W$. The traditional approach to explicit substitutions (or to let binders) is to introduce commutation rules thanks to which such "hidden" redexes may be unveiled. Accattoli's approach is entirely different: his idea is that terms should behave like proof nets, *i.e.*, there simply *ought* to be a β-redex in T_1, without any further rewriting being necessary. He found an extremely simple, consistent way of doing this, which is to generalize the notion of β-redex to terms of the form

$$(\lambda b.-)[-]W,$$

where $[-]$ is an arbitrary sequence of explicit substitutions/let binders, *i.e.*, a substitution context as defined in Fig. 1.4. So we have

$$T_1 \to xW[!x := U] =: T_2.$$

Suppose now that

$$U := (\lambda c.!y[!y := c])V.$$

We have

$$T_2 \to xW[!x := !y[!y := V]] =: T_3.$$

Again, regardless of the shape of V, proof nets are able to "see" that the value of x is the same as that of y, *i.e.*, there is an immediately reducible

cut performing this identification. And yet, T_3 contains no let-redex, at least not of the usual form $-[!x := !Z]$ (we stress that $!y[!y := V]$ is implicitly parenthesized as $(!y)[!y := V]$, *i.e.*, it is *not* of the form $!Z$). So let-redexes too must be generalized, namely to

$$-[!y := !Z[-]],$$

allowing us to further reduce, without commutation steps, as

$$T_3 \to yW[!y := V].$$

Accattoli's formulation of reduction actually includes a further technical adjustment (introduced in his work with Kesner [AK12]): the "global" reduction step

$$T[!x := !U[-]] \to_! T\{U/x\}[-]$$

is decomposed into

$$\begin{aligned} \mathsf{C}\{x\}[!x := !U[-]] &\to_{\mathsf{ls}} \mathsf{C}\{U\}[!x := !U][-], \\ T[!x := !U[-]] &\to_{\mathsf{gc}} T[-] \qquad\qquad x \notin \mathrm{fv}(T), \end{aligned}$$

where C is an arbitrary context not capturing x. In other words, the copies of U are created "on demand", until U is no longer needed and is therefore "garbage-collected". This is of course semantically sound: $\to_{\mathsf{gc}}$ is just a special case of $\to_!$ and $\to_{\mathsf{ls}}$ relates two terms which are equivalent with respect to $\to_!$ (namely, $T \to_{\mathsf{ls}} T'$ implies $T \to_! {}_!^*\!\leftarrow T'$). This decomposition enables a finer-grained study of normalization and allows Accattoli's linear substitution calculus to neatly encode abstract machines (we will mention this again below). It is also useful in terms of complexity (see Chapter 3), but for the moment we stick to the traditional formulation, on the grounds that it suffices for our purposes and that Fig. 1.3 is enough to define permutation equivalence (further equations would be needed for $\to_{\mathsf{ls}}$).

Although we will never use it, for the sake of completeness we give the standard simple type assignment for $\Lambda_!$ in Fig. 1.5. The types are

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

Typing judgments are of the form $\Theta; \Gamma \vdash T : A$ where, for typographic convenience, all type declarations for cartesian (resp. linear) variables are collected in Θ (resp. Γ). In those judgments, we always have $T : \mathsf{l}$. When $T : \mathsf{c}$, we use judgments of the form $\Theta \vdash_{\mathsf{c}} T : A$, in which all variables declared in Θ are cartesian. We invite the reader to check that, after erasing the term annotations, one obtains a natural deduction formulation of the implicational fragment of propositional intuitionistic multiplicative exponential linear logic.

Accattoli's rewriting methodology is sometimes called "rewriting at a distance", because proximity in graphs (the truly relevant notion) is not reflected in the corresponding terms, and subterms may form a redex even if they seem far apart. It is grounded in work by Accattoli and Kesner [AK12] and was foreshadowed, although incompletely, by Milner [Mil07]. In spite of its apparent

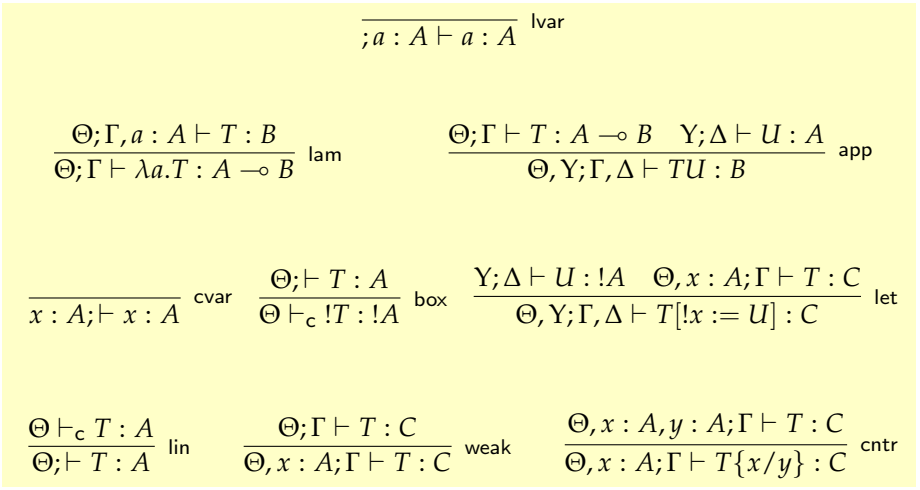


Figure 1.5: Term assignment for intuitionistic linear logic.

naiveness, this viewpoint has strikingly deep applications in rewriting theory [Acc12, ABKL14], the complexity of λ-calculus evaluation [AL16] as well as in programming languages theory, where it induces an extremely precise connection with abstract machines [ABM14, ABM15]. It is also the basis of the cleanest and most concise formulation of call-by-need that the author is aware of [ABM14].

Although not strictly necessary to the work presented here, rewriting at a distance still brings great simplifications, similar to the simplifications induced on cut-elimination by moving from sequent calculus to proof nets: the rewriting rules closely reflect the logical structure, without inessential syntactic detours. In the case of $\Lambda_!$, there are two connectives ($\multimap$ and !), so we have 2 rewriting rules, instead of the 6 required by the traditional approach. Adding the remaining connectives of intuitionistic linear logic ($\otimes$, 1, & and $\oplus$) would bring the number of rewriting rules to 6, whereas the traditional approach would require several dozens (66, if we did not miscount).

1.2.2 Girard's translations

Let us give a couple of (important) examples of encodings of term calculi seen as morphisms of 2-operads. We start with inductively defining the following map from λ-terms to terms of $\Lambda_!$:

$$\begin{aligned}\mathbf{G}_0(x) &:= x\\ \mathbf{G}_0(\lambda x.M) &:= \lambda a.\mathbf{G}_0(M)[!x := a]\\ \mathbf{G}_0(MN) &:= \mathbf{G}_0(M)!\mathbf{G}_0(N).\end{aligned}$$

An immediate induction shows that $\mathbf{G}_0(M\{N/x\}) = \mathbf{G}_0(M)\{\mathbf{G}_0(N)/x\}$. We may extend this map on reduction terms by letting

$$\begin{aligned}\mathbf{G}_0(M(\!|x \leftarrow N|\!)) := \mathbf{G}_0(M)[!x := a](\!|a \leftarrow !\mathbf{G}_0(N)|\!)\\ ;\mathbf{G}_0(M)(\!|x \leftarrow \mathbf{G}_0(N)|\!)_!.\end{aligned}$$

Note that

$$\begin{aligned}\mathbf{G}_0(M(\!|x \leftarrow N|\!)) : \mathbf{G}_0((\lambda x.M)N) &= (\lambda a.\mathbf{G}_0(M)[!x := a])!\mathbf{G}_0(N)\\ &\to \mathbf{G}_0(M)[!x := !\mathbf{G}_0(N)]\\ &\to \mathbf{G}_0(M)\{\mathbf{G}_0(N)/x\} = \mathbf{G}_0(M\{N/x\}),\end{aligned}$$

as expected. Since the reduction terms of the λ-calculus are generated by $M(\!|x \leftarrow N|\!)$, this is enough to define $\mathbf{G}_0$ everywhere. And yet, the above does *not* define a morphism of 2-operads $\Lambda \to \Lambda_!$, because of a subtle but important point: Λ, as we defined it, is monochromatic, whereas $\Lambda_!$ has two colors. Now, it is enough to inspect the definition to see that, for all M, $\mathrm{fv}(\mathbf{G}_0(M))$ contains only cartesian variables but is *never* of the form $!T$, so it is a morphism of $\Lambda_!(\mathsf{c}^n;\mathsf{l})$: source and target colors do not match, so no matter what we choose as the image of $*$ (the only color of Λ), we are in trouble. The connoisseur will have understood that, in terms of linear logic types, we have $\mathsf{c} = !\mathsf{l}$ and we are replacing the equation $* = * \to *$ (intuitionistic arrow) with Girard's translation $\mathsf{l} = !\mathsf{l} \multimap \mathsf{l}$, so the asymmetry is inevitable.

There are two ways out of this apparent dead-end. The first is to coerce encodings to be of the form $!T$: we simply define

$$\mathbf{G}_!(\varphi) := !\mathbf{G}_0(\varphi),$$

for all reduction terms, including terms themselves. In this way, everything becomes consistent with the choice $\mathbf{G}_!(*) := \mathsf{c}$. Indeed,

$$\begin{aligned}\mathbf{G}_!(M \circ^i N) &= \mathbf{G}_!(M\{N/x^i\}) = !\mathbf{G}_0(M\{N/x^i\})\\ &= !(\mathbf{G}_0(M)\{\mathbf{G}_0(N)/x^i\}) = !\mathbf{G}_0(M)\{\mathbf{G}_0(N)/x^i\}\\ &= \mathbf{G}_!(M) \circ^i_{\mathsf{c}} \mathbf{G}_!(N)\end{aligned}$$

(remember that, in $\Lambda_!$, $T \circ^i_{\mathsf{c}} !U = T\{U/x^i\}$). We have thus defined a morphism of 2-operads

$$\mathbf{G}_! : \Lambda \longrightarrow \Lambda_!,$$

which amounts to using $\mathbf{G}_0$ with a useless $!(-)$ around everything.

The other solution is to change the presentation of the λ-calculus, making it bichromatic too. This amounts to taking seriously the idea that the λ-calculus is the Kleisli category of the comonad $!(-)$ of linear logic. So we have the color of terms t and the color of values v (the terminology is chosen in view of the call-by-value case, see below). Only variables of color v exists, whereas terms are of color t by default. However, since we are in call-by-name, there is a coercion rule saying that everything is (also) a value. The corresponding operad Λ_k ("k" is for "Kleisli") is presented in Fig. 1.6.

Colors: t, v (ranged over by C below).

Terms:

$$\frac{}{x : \mathsf{v} \vdash x : \mathsf{t}}\ \mathsf{var} \qquad \frac{\Gamma, x : \mathsf{v} \vdash M : \mathsf{t}}{\Gamma \vdash \lambda x.M : \mathsf{t}}\ \mathsf{lam} \qquad \frac{\Gamma \vdash M : \mathsf{t} \quad \Delta \vdash N : \mathsf{t}}{\Gamma, \Delta \vdash MN : \mathsf{t}}\ \mathsf{app}$$

$$\frac{\Gamma \vdash M : \mathsf{t}}{\Gamma \vdash M : \mathsf{v}}\ \mathsf{val} \qquad \frac{\Gamma \vdash M : C}{\Gamma, x : \mathsf{v} \vdash M : C}\ \mathsf{weak} \qquad \frac{\Gamma, x : \mathsf{v}, y : \mathsf{v} \vdash M : C}{\Gamma, x : \mathsf{v} \vdash M\{x/y\} : C}\ \mathsf{cntr}$$

Basic steps: as in the usual λ-calculus.

Figure 1.6: Bichromatic presentation of the (call-by-name) λ-calculus Λ_k.

It is clear that Λ_k is just a different presentation of the usual λ-calculus, just with two copies of every λ-term, one of type t and one of type v. The latter is the one we (silently, because there is no syntactic difference) use when we substitute a λ-term for a variable.

Now it becomes possible to use $\mathbf{G}_0$ to define a morphism

$$\mathbf{G}_k : \Lambda_k \longrightarrow \Lambda_!.$$

It is enough to set $\mathbf{G}_k(\mathsf{t}) := \mathsf{l}$, $\mathbf{G}_k(\mathsf{v}) := \mathsf{c}$, $\mathbf{G}_k = \mathbf{G}_0$ on λ-terms of type t and, on λ-terms of type v, we define $\mathbf{G}_k(M : \mathsf{v}) := !\mathbf{G}_0(M)$ (in fact, $\mathbf{G}_k(M : \mathsf{v}) = \mathbf{G}_!(M)$). Note that, since only composition of color v exists in Λ_k (by the way, this is a genuine example of non-unital operad: there is no identity of type t), we have

$$\begin{aligned}\mathbf{G}_k(M \circ^i_{\mathsf{v}} (N : \mathsf{v})) &= \mathbf{G}_0(M\{N/x_i\}) = \mathbf{G}_0(M)\{\mathbf{G}_0(N)/x_i\} \\ &= \mathbf{G}_0(M) \circ^i_{\mathsf{c}} !\mathbf{G}_0(N) = \mathbf{G}_k(M) \circ^i_{\mathsf{c}} \mathbf{G}_k(N : \mathsf{v}),\end{aligned}$$

where we assume that a term has type t unless otherwise stated. The case in which $M : \mathsf{v}$ works just as for the definition of $\mathbf{G}_!$. So we have, indeed, a morphism of operads.

The bichromatic presentation of the λ-calculus is more general than the monochromatic one: we have $i : \Lambda \hookrightarrow \Lambda_k$ by setting $i(*) := \mathsf{v}$, and $\mathbf{G}_! = \mathbf{G}_k \circ i$. In fact, there is also a suboperad $\Lambda_0 \hookrightarrow \Lambda_k$ obtained by keeping only λ-terms of type t. This 2-operad is particularly degenerate because composition is void; $\mathbf{G}_0$ becomes a well-defined morphism on it, which will have a remarkable role when we will discuss intersection types (Chapter 2).

Before moving on, we should check that the above definition is compatible with permutation equivalence: technically speaking, what we did above is assigning a reduction term $\mathbf{G}_0(\rho)$ of $\Lambda_!$ to a reduction term ρ of Λ_k (or Λ); it is possible in principle that $\rho \sim \tau$ and yet $\mathbf{G}_0(\rho) \not\sim \mathbf{G}_0(\tau)$, in which case the definition would be unsound. Fortunately, it is easy to check that this is not

the case. For example, the diagram

$$\begin{array}{ccc} (\lambda x.M)N & \xrightarrow{M(\!|x \leftarrow N|\!)} & M\{N/x\} \\ {\scriptstyle (\lambda x.M)\tau}\downarrow & \sim & \downarrow{\scriptstyle M\{\tau/x\}} \\ (\lambda x.M)N' & \xrightarrow[M(\!|x \leftarrow N'|\!)]{} & M\{N'/x\} \end{array}$$

(which is essentially the unnest rule of Fig. 1.3) becomes

$$\begin{array}{ccccc} (\lambda x.M)_g !N_g & \xrightarrow{M_g[!x:=a](\!|a \leftarrow !N_g|\!)} & M_g[!x := !N_g] & \xrightarrow{M_g(\!|x \leftarrow N_g|\!)_!} & M_g\{N_g/x\} \\ {\scriptstyle (\lambda x.M)_g !\tau}\downarrow & \sim & {\scriptstyle M_g[!x:=!\tau_g]}\downarrow & \sim & \downarrow{\scriptstyle M_g\{\tau_g/x\}} \\ (\lambda x.M)_g !N'_g & \xrightarrow[M_g[!x:=a](\!|a \leftarrow !N'_g|\!)]{} & M_g[!x := !N'_g] & \xrightarrow[M_g(\!|x \leftarrow N'_g|\!)_!]{} & M_g\{N'_g/x\} \end{array}$$

(we abbreviated $\mathbf{G}_0(-)$ by $(-)_g$).

The above is usually referred to as the *call-by-name* translation of the λ-calculus in linear logic. It was introduced of course by Girard in his seminal paper [Gir87]. Anachronistically, we immediately see in it the relevance of linear logic with respect to explicit substitutions: if we take $\lambda x.M$ and MN merely as syntactic sugar for $\lambda a.M[!x := a]$ and $M!N$, respectively, we see that linear logic decomposes the usual β-reduction step

$$(\lambda x.M)N \to M\{N/x\}$$

into

$$(\lambda x.M)N \to M[!x := !N] \to M\{N/x\}.$$

The intermediate step "escapes" λ-calculus syntax, but it is captured as soon as we introduce the explicit substitution notation $M[x \leftarrow N]$ as syntactic sugar for $M[!x := !N]$. People of course became aware of this soon after the introduction of linear logic. What took a long time to realize (basically not until around 2010, with Accattoli and Kesner's contribution [AK12]) is that the behavior of proof nets could be directly imported into explicit substitutions, yielding a great benefit to that theory.

In the same paper [Gir87], Girard also introduced another translation, which, for reasons that escape us, he deemed "boring", and which is usually (and more suitably) dubbed the *call-by-value* translation. In order to define it, we must first present the call-by-value λ-calculus as a 2-operad, which is done in Fig. 1.7. The translation is then a morphism

$$\mathbf{G}_v : \Lambda_v \longrightarrow \Lambda_!$$

defined as follows:

- on colors, $\mathbf{G}_v(\mathsf{t}) := \mathsf{l}$ and $\mathbf{G}_v(\mathsf{v}) := \mathsf{c}$;

Colors: t, v (ranged over by C below).

Terms:

$$\frac{}{x : \mathsf{v} \vdash x : \mathsf{v}}\ \mathsf{var} \qquad \frac{\Gamma, x : \mathsf{v} \vdash M : \mathsf{t}}{\Gamma \vdash \lambda x.M : \mathsf{v}}\ \mathsf{lam} \qquad \frac{\Gamma \vdash M : \mathsf{t} \quad \Delta \vdash N : \mathsf{t}}{\Gamma, \Delta \vdash MN : \mathsf{t}}\ \mathsf{app}$$

$$\frac{\Gamma \vdash V : \mathsf{v}}{\Gamma \vdash V : \mathsf{t}}\ \mathsf{val} \qquad \frac{\Gamma \vdash M : C}{\Gamma, x : \mathsf{v} \vdash M : C}\ \mathsf{weak} \qquad \frac{\Gamma, x : \mathsf{v}, y : \mathsf{v} \vdash M : C}{\Gamma, x : \mathsf{v} \vdash M\{x/y\} : C}\ \mathsf{cntr}$$

Basic steps:

$$M(\!|x \leftarrow V|\!) : (\lambda x.M)V \to^* M\{V/x\} \qquad V : \mathsf{v}$$

Figure 1.7: The call-by-value λ-calculus Λ_v.

- on terms, we must distinguish whether the λ-term being encoded is of type t or v:

$$\begin{aligned} \mathbf{G}_v(x : \mathsf{v}) &:= !x \\ \mathbf{G}_v(\lambda x.M : \mathsf{v}) &:= !(\lambda a.\mathbf{G}_v(M : \mathsf{t})[!x := a]) \\ \mathbf{G}_v(MN : \mathsf{t}) &:= \xi[!\xi := \mathbf{G}_v(M : \mathsf{t})]\mathbf{G}_v(N : \mathsf{t}) \\ \mathbf{G}_v(V : \mathsf{t}) &:= \mathbf{G}_v(V : \mathsf{v}). \end{aligned}$$

  In the last line, what we mean is that $\mathbf{G}_v(V : \mathsf{t})$ is defined by taking $\mathbf{G}_v(V : \mathsf{v})$, which is of type c, and coercing it to the type l using the lin rule of Fig. 1.4. It is easy to check that $\mathbf{G}_v(M\{V/x\} : \mathsf{t}) = \mathbf{G}_v(M : \mathsf{t})\{\mathbf{G}_v(V : \mathsf{v})/x\}$;

- on reductions, we set

$$\begin{aligned} \mathbf{G}_v(M(\!|x \leftarrow V|\!)) := \xi(\!|\xi \leftarrow \mathbf{G}_v(\lambda x.M)|\!)_!!\mathbf{G}_v(V) \\ ; \mathbf{G}_v(M)[!x := a](\!|a \leftarrow !\mathbf{G}_v(V)|\!) \\ ; \mathbf{G}_v(M)(\!|x \leftarrow \mathbf{G}_v(V)|\!)_!, \end{aligned}$$

  which corresponds to the reduction

$$\begin{aligned} \mathbf{G}_v((\lambda x.M)V : \mathsf{t}) &= \xi[!\xi := !(\lambda a.\mathbf{G}_v(M : \mathsf{t})[!x := a])]!\mathbf{G}_v(V : \mathsf{v}) \\ &\to (\lambda a.\mathbf{G}_v(M : \mathsf{t})[!x := a])!\mathbf{G}_v(V : \mathsf{v}) \\ &\to \mathbf{G}_v(M : \mathsf{t})[!x := !\mathbf{G}_v(V : \mathsf{v})] \\ &\to \mathbf{G}_v(M : \mathsf{t})\{\mathbf{G}_v(V : \mathsf{v})/x\} = \mathbf{G}_v(M\{V/x\} : \mathsf{t}), \end{aligned}$$

  as desired.

As for the call-by-name translation, verifying that the definition on 2-arrows is consistent with permutation equivalence poses no difficulty.

The definition of Λ_v is quite natural as it mimics the usual, mutually inductive definition ($V ::= x \mid \lambda x.M$ and $M, N ::= V \mid MN$). Nevertheless, one

may wonder whether a monochromatic presentation of call-by-value exists; after all, the calculus is still untyped. For instance, what happens if we just take the monochromatic 2-operad Λ and restrict it to 2-arrows corresponding to call-by-value reductions? The answer is that such a restriction does not define a sub-operad of Λ, for the reasons discussed at the end of Sect. 1.1.4. To see why, take

$$\rho := x(\!|x \leftarrow y|\!) : Iy \to y,$$
$$\tau := z(x(\!|x \leftarrow I|\!)) : z(II) \to zI.$$

These are both valid call-by-value reductions; and yet,

$$\rho\{\tau/x\} = x(\!|x \leftarrow zII|\!); z(x(\!|x \leftarrow I|\!)) \sim I(z(x(\!|x \leftarrow I|\!))); x(\!|x \leftarrow zI|\!)$$

is not, because it fires a redex whose argument is either $z(II)$ or zI, neither of which is a value. Using a color v of values and declaring all variables to be of color v allows us to state that only values may be substituted to variables, thus preventing the above problem.

1.2.3 Polyadic calculi and simple polyadic types

We now introduce the main actors of the thesis, the *polyadic calculi*. The operads presenting them are defined in Fig. 1.8. They have two colors and are actually all suboperads of $\Lambda^{\mathsf{p}}_{\mathsf{c}}$: they are obtained from it by removing none, one or both of the two structural rules (weak and cntr), yielding the *cartesian* ($\Lambda^{\mathsf{p}}_{\mathsf{c}}$), *relevant* ($\Lambda^{\mathsf{p}}_{\mathsf{r}}$, no weakening), *affine* ($\Lambda^{\mathsf{p}}_{\mathsf{a}}$, no contraction) or *linear* ($\Lambda^{\mathsf{p}}_{\mathsf{l}}$) polyadic calculus, respectively. As in $\Lambda_!$, there are linear variables (of type l), which we distinguish by denoting them a, b, and *polyadic* variables (of type $\overline{\mathsf{p}}$), which may be linear, affine, relevant or cartesian. Similarly to $\Lambda_!$, the only terms of type $\overline{\mathsf{p}}$ are of the form $\langle t_1, \dots, t_n \rangle$; these are called *boxes*.

For reasons which will become clear when we will introduce approximations, the definition of the hom-categories of $\Lambda^{\mathsf{p}}_{\mathsf{p}}$ is slightly different from what one would expect. First, to each cartesian variable x^i of $\Lambda_!$ we assign a countably infinite sequence of polyadic variables $x^i_1, x^i_2, x^i_3, \dots$, denoted by $\overline{x}^i$ and called *supervariable*. In the terms of the form $t[\langle x_1, \dots, x_n \rangle := u]$ (and $t(\!|x_1, \dots, x_n \leftarrow u|\!)[-]$) we use α-equivalence to assume that $x_1, \dots, x_n$ are always an initial segment of a supervariable. We then define the hom-categories of $\Lambda^{\mathsf{p}}_{\mathsf{p}}$ as follows:

- the multimorphisms of $\Lambda^{\mathsf{p}}_{\mathsf{p}}(\overline{\mathsf{p}}^m, \mathsf{l}^n; C)$ are polyadic terms $t : C$ such that the free *linear* variables of t are exactly $\{a^1, \dots, a^n\}$ and the free *polyadic* variables are contained in $\overline{x}^1, \dots, \overline{x}^m$ (and, from Fig. 1.8, we see that $C = \overline{\mathsf{p}}$ implies $n = 0$). Furthermore, in case $\mathsf{p} \in \{\mathsf{l}, \mathsf{r}\}$ (*i.e.*, when we do not have weakening), we require that, for all $1 \leq i \leq m$ and for all $j < k \in \mathbb{N}$, if x^i_k is free in t, then so is x^i_j.
- The 2-arrows are defined similarly.

Colors: I and $\overline{\mathsf{p}}$, where p is one of $\mathsf{I}, \mathsf{a}, \mathsf{r}, \mathsf{c}$ (below, we use C for a generic color).

Terms:

$$\frac{}{x : \overline{\mathsf{p}} \vdash x : \mathsf{I}}\ \mathsf{var} \qquad \frac{\Gamma, a : \mathsf{I} \vdash t : \mathsf{I}}{\Gamma \vdash \lambda a.t : \mathsf{I}}\ \mathsf{lam} \qquad \frac{\Gamma \vdash t \quad \Delta \vdash u : \mathsf{I}}{\Gamma, \Delta \vdash tu : \mathsf{I}}\ \mathsf{app}$$

$$\frac{\overline{x}_1 : \overline{\mathsf{p}} \vdash t_1 : \mathsf{I} \quad \dots \quad \overline{x}_n : \overline{\mathsf{p}} \vdash t_n : \mathsf{I}}{\overline{x}_1 : \overline{\mathsf{p}}, \dots, \overline{x}_n : \overline{\mathsf{p}} \vdash \langle t_1, \dots, t_n \rangle : \overline{\mathsf{p}}}\ \mathsf{box} \qquad \frac{\Gamma \vdash t : \overline{\mathsf{p}}}{\Gamma \vdash t : \mathsf{I}}\ \mathsf{lin}$$

$$\frac{\Delta \vdash u : \mathsf{I} \quad \Gamma, x_1 : \overline{\mathsf{p}}, \dots, x_n : \overline{\mathsf{p}} \vdash t : \mathsf{I}}{\Gamma, \Delta \vdash t[\langle x_1, \dots, x_n \rangle := u] : \mathsf{I}}\ \mathsf{let} \qquad \frac{}{\vdash \bot : \mathsf{I}}\ \mathsf{undef}$$

$$\frac{\Gamma \vdash t : C}{\Gamma, x : \overline{\mathsf{p}} \vdash t : C}\ \mathsf{weak}, \mathsf{p} \in \{\mathsf{a}, \mathsf{c}\} \qquad \frac{\Gamma, x : \overline{\mathsf{p}}, y : \overline{\mathsf{p}} \vdash t : C}{\Gamma, x : \overline{\mathsf{p}} \vdash t\{x/y\} : C}\ \mathsf{cntr}, \mathsf{p} \in \{\mathsf{r}, \mathsf{c}\}$$

Basic steps:

$$\begin{array}{lrcl} t(\!|a \leftarrow u|\!)[-] : & (\lambda a.t)[-]u & \to^* & t\{u/a\}[-] \\ t(\!|\overline{x} \leftarrow \overline{u}|\!)[-] : & t[\langle \overline{x} \rangle := \langle \overline{u} \rangle[-]] & \to^* & t\{\overline{u}/\overline{x}\}[-] \end{array}$$

In the second basic step, $\overline{x} = x_1, \dots, x_m$, $\overline{u} = u_1, \dots, u_n$ and $\mathsf{p} \in \{\mathsf{I}, \mathsf{r}\}$ implies $m \geq n$, whereas any value of m, n is allowed if $\mathsf{p} \in \{\mathsf{a}, \mathsf{c}\}$.
The notation $[-]$ is a metavariable for *substitution contexts*, defined as follows:

$$[-] ::= \{\cdot\} \mid [-][\langle \overline{x} \rangle := u].$$

We write $t[-]$ instead of $[-]\{t\}$.

Figure 1.8: Polyadic calculi $\Lambda^{\mathsf{p}}_{\mathsf{p}}$.

Operadic composition in $\Lambda^{\mathsf{p}}_{\mathsf{p}}$ is defined in a similar way as in Λ_{I}: on the color I, it is plain substitution; on the color $\overline{\mathsf{p}}$, we set

$$t \circ^i_{\overline{\mathsf{p}}} \langle u_1, \dots, u_n \rangle := t\{u_1, \dots, u_n / \overline{x}^i\}$$

where the latter term is defined by simultaneously substituting, for $1 \leq j \leq n$, each u_j to x^i_j and, if there is $j > n$ such that x^i_j is free in t, then it is replaced with $\bot$. This is the notion of substitution used in the second basic step of Fig. 1.8. We repeat the observation made for Λ_{I}: when terms of the form $\langle u_1, \dots, u_n \rangle$ are seen as of type I, composition does *not* remove the box:

$$t \circ^i_{\mathsf{I}} \langle u_1, \dots, u_n \rangle = t\{\langle u_1, \dots, u_n \rangle / a^i\}.$$

Polyadic calculi may be naturally endowed with a discipline of simple types, coming from linear logic via Curry-Howard. *Polyadic types* are defined inductively as follows:

$$A, B ::= \alpha \mid A \multimap B \mid \langle A_1, \dots, A_n \rangle,$$

Colors: as explained in the text.

Terms:

$$\frac{}{;a : A \vdash a : A}\ \mathsf{lvar} \qquad \frac{}{x : A;\vdash x : A}\ \mathsf{pvar}$$

$$\frac{\Theta;\Gamma, a : A \vdash t : B}{\Theta;\Gamma \vdash \lambda a.t : A \multimap B}\ \mathsf{lam} \qquad \frac{\Theta;\Gamma \vdash t : A \multimap B \quad \Upsilon;\Delta \vdash u : A}{\Theta,\Upsilon;\Gamma,\Delta \vdash tu : B}\ \mathsf{app}$$

$$\frac{\Theta_1;\vdash t_1 : A_1 \quad \dots \quad \Theta_n;\vdash t_n : A_n}{\Theta_1,\dots,\Theta_n \vdash_{\mathsf{p}} \langle t_1,\dots,t_n\rangle : \langle A_1,\dots,A_n\rangle}\ \mathsf{box} \qquad \frac{\Theta \vdash_{\mathsf{p}} t : A}{\Theta;\vdash t : A}\ \mathsf{lin}$$

$$\frac{\Upsilon;\Delta \vdash u : \langle A_1,\dots,A_n\rangle \quad \Theta, x_1 : A_1,\dots,x_n : A_n;\Gamma \vdash t : C}{\Theta,\Upsilon;\Gamma,\Delta \vdash t[\langle x_1,\dots,x_n\rangle := u] : C}\ \mathsf{let}$$

$$\frac{\Theta;\Gamma \vdash t : C}{\Theta, x : A;\Gamma \vdash t : C}\ \mathsf{weak}\ \mathsf{p}\in\{\mathsf{a},\mathsf{c}\} \qquad \frac{\Theta, x : A, y : A;\Gamma \vdash t : C}{\Theta, x : A;\Gamma \vdash t\{x/y\} : C}\ \mathsf{cntr}\ \mathsf{p}\in\{\mathsf{r},\mathsf{c}\}$$

Basic steps: The same as in Fig. 1.8, with the obvious typing.

Figure 1.9: The 2-operad $\mathbf{Poly}_{\mathsf{p}}$ of simply-typed polyadic terms.

where $n = 0$ is allowed. The type derivations are defined in Fig. 1.9, which induces a 2-operad of (Church-style) simply-typed polyadic terms. The colors are the polyadic types, plus a color $\langle A_1,\dots,A_n\rangle_{\mathsf{p}}$ for each sequence of types $A_1,\dots,A_n$. We use the following notations for typing judgments:

- $\Theta;\Gamma \vdash t : A$, in which all polyadic (resp. linear) variables are typed in Θ (resp. Γ) and $t : \mathsf{I}$;
- $\Theta \vdash_{\mathsf{p}} t : A$, in which $t : \overline{\mathsf{p}}$, so in fact $t = \langle u_1,\dots,u_n\rangle$ and this is just a notation for $\Theta;\vdash \langle u_1,\dots,u_n\rangle : \langle B_1,\dots,B_n\rangle_{\mathsf{p}}$, with $A = \langle B_1,\dots,B_n\rangle$.

The multimorphisms of the hom-category

$$\mathbf{Poly}_{\mathsf{p}}(\langle\overline{B}^1\rangle_{\mathsf{p}},\dots,\langle\overline{B}^m\rangle_{\mathsf{p}}, C_1,\dots,C_n; A),$$

where C_i is not of the form $\langle\overline{D}\rangle_{\mathsf{p}}$, are defined in a similar way to the hom-categories of $\Lambda^{\mathsf{p}}_{\mathsf{p}}$: they are Church-style terms/derivations of

$$x^1_1 : B^1_1 \dots, x^1_{k_1} : B^1_{k_1},\dots, x^m_1 : B^m_1,\dots, x^m_{k_m} : B^m_{k_m}; a^1 : C_1,\dots, a^n : C_n \vdash t : A$$

with $\overline{x}^i$ a supervariable as in the case of $\Lambda^{\mathsf{p}}_{\mathsf{p}}$ (and, in case $A = \langle D_1,\dots,D_k\rangle_{\mathsf{p}}$, then $n = 0$ and $t = \langle u_1,\dots,u_k\rangle$). Again, for every p, $\mathbf{Poly}_{\mathsf{p}}$ is a sub-operad of $\mathbf{Poly}_{\mathsf{c}}$.

Note that $\bot$ is not typable. As a consequence, a basic step like $t(\!|x \leftarrow|\!)$ is not typed unless $x \notin \mathrm{fv}(t)$, which means we must have $\mathsf{p} \in \{\mathsf{a},\mathsf{c}\}$, *i.e.*, we must have weakening. The presence of $\bot$ is technically necessary to achieve

monotonicity of reduction, in the sense of Proposition 12. However, $\bot$ is not a "valid" term, which is why it is discarded when types are introduced.

The type system is a direct import from linear logic, in the following sense: if we erase term annotations from Fig. 1.9, we obtain a system of natural deduction such that:

- if $\mathsf{p} = \mathsf{l}$, every rule is derivable in intuitionistic multiplicative linear logic by setting $\langle A_1, \ldots, A_n \rangle := A_1 \otimes \cdots \otimes A_n$;
- if $\mathsf{p} = \mathsf{a}$, every rule is derivable in intuitionistic multiplicative additive linear logic by setting $\langle A_1, \ldots, A_n \rangle := (A_1 \,\&\, 1) \otimes \cdots \otimes (A_n \,\&\, 1)$;
- if $\mathsf{p} = \mathsf{c}$, every rule is derivable in intuitionistic multiplicative exponential linear logic (the system of Fig. 1.5) by setting $\langle A_1, \ldots, A_n \rangle := !A_1 \otimes \cdots \otimes !A_n$.

Moreover, the reduction rules also reflect the cut-elimination rules of linear logic, via the above encodings of $\langle - \rangle$. So the simply-typed polyadic calculi are nothing really new: they are just fragments of propositional intuitionistic linear logic. Indeed, linear polyadic calculi similar to our own have already appeared here and there in the literature [Kfo00, Mel04].

The reason why we are interested in simple types is the following:

Proposition 5 (strong normalization) *Simply-typed polyadic terms are strongly normalizing.*

Proof. By the above encoding of $\mathbf{Poly}_{\mathsf{c}}$ in linear logic, this is an immediate consequence of strong normalization of propositional linear logic, which is well known [Gir87]. □

It is worth observing that in the linear and affine cases (*i.e.*, $\mathsf{p} \in \{\mathsf{l}, \mathsf{a}\}$), strong normalization actually holds even in the untyped calculus (*i.e.*, in $\Lambda^{\mathsf{p}}_{\mathsf{l}}$ and $\Lambda^{\mathsf{p}}_{\mathsf{a}}$), for trivial reasons: the size of terms strictly decreases under reduction.

1.3 The Approximation Theorem

1.3.1 Girard's cue

In the concluding sections of his paper introducing linear logic [Gir87], Girard observes that the formulas of propositional linear logic may be approximated arbitrarily well by formulas of its "purely linear" fragment (multiplicative additive linear logic, **MALL**). Let us quote directly from Sect. 5.6 of his paper:

> [T]he approximation theorem [...] is just the mathematical contents of our slogan: *usual logic is obtained from linear logic* (without modalities) *by a passage to the limit*.

5.1. Definition (*approximants*). The connectives ! and ? are approximated by the connectives $!_n$ and $?_n$ ($n \neq 0$):

$$!_n A = (1 \,\&\, A) \otimes \cdots \otimes (1 \,\&\, A) \qquad (n \text{ times})$$
$$?_n A = (\bot \oplus A) \parr \cdots \parr (\bot \oplus A) \qquad (n \text{ times})$$

5.2. Theorem (*Approximation Theorem*). *Let A be a theorem of linear logic; with each occurrence of ! in A, assign an integer $\neq 0$; then it is possible to assign integers $\neq 0$ to all occurrences of ? in such a way that if B denotes the result of replacing each occurrence of ! (respectively ?) by $!_n$ (respectively $?_n$) where n is the integer assigned to it, then B is still a theorem of linear logic.*

Without entering into the technical details, the intuition is that $!_n A$ means "A is available at most n times", which is indeed an approximation of $!A$, whose intuitive meaning is "A is available at will". (The dual approximation $?_n A$ means "$A^\perp$ will be needed at most n times"). Let us observe that the approximations $!_n A$ and $?_n A$ are formulas of **MALL** as long as A is. Therefore, the formula B of the Approximation Theorem is a **MALL** formula (*i.e.*, it has no modalities) and, by cut-elimination, its proof is also in **MALL**.

The proof of Girard's Approximation Theorem is of little interest: it is a straightforward induction on the cut-free proof of A, using the key property that $!_m A \multimap !_n A$ with $m \geq n$. In fact, from the computational viewpoint, invoking cut-elimination sort of obscures the actual depth of the theorem. In what follows we will take, so to speak, a complementary approach: we will forget types and concentrate solely on the computational aspect.

Of course, ours is not the first attempt at providing a formal content to the idea underlying Girard's Approximation Theorem (*i.e.*, Equation 1). For instance, Melliès, Tabareau and Tasson interpreted it as a categorical limit, giving sufficient conditions for the formula expressed by Equation 1 to yield the free exponential modality in a model of linear logic [MTT09]. Our approach is similar to that of Ehrhard and Regnier [ER08], who also use a term calculus (the resource λ-calculus) to approximate λ-terms. We will discuss a bit more thoroughly the comparison with that work in Sect. 2.4.3, after having developed a sufficient amount of technical material.

1.3.2 Affine approximations

Consider the system $\mathbf{Poly}_!$ defined in Sect. 1.2.3. As a logical system, this is just intuitionistic multiplicative linear logic (**IMLL**) plus affine tensors of arbitrary arity, *i.e.*, it is exactly **IMLL** plus the approximation modalities $!_n$ defined by Girard. Therefore, $\Lambda^{\mathrm{p}}_!$ (the polyadic calculus underlying $\mathbf{Poly}_!$) is the natural setting for formulating a computational version of the Approximation Theorem.

Definition 8 (approximation order) *We define the* approximation order $\sqsubseteq$ *on reduction terms of $\Lambda^{\mathrm{p}}_!$ by means of the rules of Fig. 1.10. The order also applies on terms, by considering them as identity reductions. It also extends to contexts, by*

$$\frac{}{x \sqsubseteq x}\ \mathsf{var} \qquad \frac{\rho \sqsubseteq \rho'}{\lambda a.\rho \sqsubseteq \lambda a.\rho'}\ \mathsf{lam} \qquad \frac{\rho \sqsubseteq \rho' \quad \tau \sqsubseteq \tau'}{\rho\tau \sqsubseteq \rho'\tau'}\ \mathsf{app}$$

$$\frac{\rho_1 \sqsubseteq \rho'_1 \quad \dots \quad \rho_n \sqsubseteq \rho'_n \quad n \leq m}{\langle \rho_1, \dots, \rho_n \rangle \sqsubseteq \langle \rho'_1, \dots, \rho'_m \rangle}\ \mathsf{box} \qquad \frac{}{\bot \sqsubseteq \rho}\ \mathsf{undef}$$

$$\frac{\rho \sqsubseteq \rho' \quad \tau \sqsubseteq \tau' \quad n \leq m \quad x_{n+1}, \dots, x_m \notin \mathrm{fv}(\rho)}{\rho[\langle x_1, \dots, x_n \rangle := \tau] \sqsubseteq \rho'[\langle x_1, \dots, x_m \rangle := \tau']}\ \mathsf{let}$$

$$\frac{(\lambda a.t)[-]u \sqsubseteq (\lambda a.t')[-]'u'}{t(\!|a \leftarrow u|\!)[-] \sqsubseteq t'(\!|a \leftarrow u'|\!)[-]'}\ \mathsf{beta}$$

$$\frac{t[\langle \overline{x} \rangle := \langle \overline{u} \rangle[-]] \sqsubseteq t'[\langle \overline{x}' \rangle := \langle \overline{u}' \rangle[-]']}{t(\!|\overline{x} \leftarrow \overline{u}|\!)[-] \sqsubseteq t'(\!|\overline{x}' \leftarrow \overline{u}'|\!)[-]}\ \mathsf{theta}$$

$$\frac{\rho \sqsubseteq \rho' \qquad \tau \sqsubseteq \tau'}{\rho;\tau \sqsubseteq \rho';\tau'}\ \mathsf{comp} \qquad \frac{\rho \sim \rho' \quad \rho' \sqsubseteq \tau' \quad \tau' \sim \tau}{\rho \mathrel{\underset{\sim}{\sqsubseteq}} \tau}\ \mathsf{permeq}$$

Figure 1.10: Approximation order on affine polyadic reductions.

treating the hole just like the term $\bot$.

It should be clear that the relation $\sqsubseteq$ of Fig. 1.10 is defined on reduction sequences in the strict syntactic sense, *i.e.*, we are not taking *any* quotient, not even under structural equivalence (contrarily to what we usually do). The relation $\mathrel{\underset{\sim}{\sqsubseteq}}$ takes permutation equivalence (and, therefore, structural equivalence) into account. For instance, if $\rho : t \to^* t'$, $\tau : t' \to^* t''$, $t \sqsubseteq \rho$ and $\tau \sqsubseteq \tau'$, then $t;\tau \sqsubseteq \rho;\tau'$, from which we may deduce $\tau \mathrel{\underset{\sim}{\sqsubseteq}} \rho;\tau'$ but *not* $\tau \sqsubseteq \rho;\tau'$.

Let us make two immediate observations, which will be tacitly used in the sequel:

- every rule of Fig. 1.10 is valid with $\mathrel{\underset{\sim}{\sqsubseteq}}$ in place of $\sqsubseteq$;
- on terms, $\mathrel{\underset{\sim}{\sqsubseteq}}$ coincides with $\sqsubseteq$.

We also list a few properties which will be quite useful in the sequel (especially points 3 and 6):

Lemma 6 *1. For every context* C *and terms* r *and* t', $\mathsf{C}\{r\} \sqsubseteq t'$ *implies* $t' = \mathsf{C}'\{r'\}$ *for some* $\mathsf{C} \sqsubseteq \mathsf{C}'$ *and* $r \sqsubseteq r'$;

2. if $t \sqsubseteq t'$ *and* $u \sqsubseteq u'$, *then* $t\{u/x\} \sqsubseteq t'\{u'/x\}$;

3. if $\rho : t \to^* t'$, $\tau : u \to^* u'$ *and* $\rho \mathrel{\underset{\sim}{\sqsubseteq}} \tau$, *then* $t \sqsubseteq u$ *and* $t' \sqsubseteq u'$;

4. if $\rho \sqsubseteq \rho'$ *and* $u \sqsubseteq u'$, *then* $\rho\{u/x\} \sqsubseteq \rho'\{u'/x\}$;

5. *if* $v \sqsubseteq v'$ *and* $\tau \sqsubseteq \tau'$ *then* $v\{\tau/x\} \sqsubseteq v'\{\tau'/x\}$;

6. *if* $\rho \sqsubseteq \rho'$ *and* $\tau \sqsubseteq \tau'$, *then* $\rho\{\tau/x\} \sqsubseteq \rho'\{\tau'/x\}$; *therefore, if* $\rho \mathrel{\underline{\sqsubseteq}\!\!\sim} \rho'$ *and* $\tau \mathrel{\underline{\sqsubseteq}\!\!\sim} \tau'$, *then* $\rho\{\tau/x\} \mathrel{\underline{\sqsubseteq}\!\!\sim} \rho'\{\tau'/x\}$.

Proof. Points 1 and 2 are obtained by straightforward inductions, on C and t, respectively. For point 3, $\rho \mathrel{\underline{\sqsubseteq}\!\!\sim} \tau$ means $\rho \sim \rho' \sqsubseteq \tau' \sim \tau$; the proof is then by induction on the derivation of $\rho' \sqsubseteq \tau'$, using point 2 in the beta and theta cases. Point 4 is by induction on ρ (point 2 is one of the inductive cases). For point 5, we observe that, by affinity, there is at most one occurrence of x is v; if $x \notin \mathrm{fv}(v)$, the result is trivial, so we may suppose $v = \mathsf{C}\{x\}$ with x not appearing in C. At this point, the proof proceeds straightforwardly by induction on τ. Finally, by definition of general substitution for reduction terms, point 6 is an immediate consequence of points 4 and 5. □

It may not be obvious that $\mathrel{\underline{\sqsubseteq}\!\!\sim}$ is, indeed, a partial order. In fact, this is the case if we are willing to work modulo permutation equivalence, which is what our operadic approach forces us to do anyway.

Lemma 7 *For all reduction terms* ρ, ρ, τ, τ':

1. $\rho \sqsubseteq \tau \sim \tau'$ *implies* $\rho \sim \rho' \sqsubseteq \tau'$ *for some* ρ';

2. $\rho \sqsubseteq \tau \sim \tau' \sqsubseteq \rho$ *implies* $\rho = \tau = \tau'$.

Proof. Both points follow immediately by induction once they are established in the particular case in which $\sim$ is replaced by $\sim_1$, where by $\sim_1$ we mean that exactly one rule of Fig. 1.2 or Fig. 1.3 is applied (inside an arbitrary context). For point 1, this is proved by an induction on the derivation of $\rho \sqsubseteq \tau$ joint with a case analysis on $\sim_1$. Albeit tedious, this is not particularly problematic. For point 2, we define the *syntactic length* of a reduction term ψ to be $\#\psi := n + 1$ where n is the number of semicolons in ψ. Then, we observe that $\varphi \sqsubseteq \psi$ implies $\#\varphi = \#\psi$, so under our hypotheses we have $\#\rho = \#\tau = \#\tau'$. This puts a heavy constraint on the step $\sim_1$: there are only two possibilities, which are, again, tedious but unproblematic. □

Proposition 8 *The relation* $\mathrel{\underline{\sqsubseteq}\!\!\sim}$ *is a preorder such that* $\rho \mathrel{\underline{\sqsubseteq}\!\!\sim} \tau$ *and* $\tau \mathrel{\underline{\sqsubseteq}\!\!\sim} \rho$ *imply* $\rho \sim \tau$.

Proof. That $\sqsubseteq$ is a partial order is immediate from the rules of Fig. 1.10. Reflexivity of $\mathrel{\underline{\sqsubseteq}\!\!\sim}$ trivially follows. Transitivity and antisymmetry modulo $\sim$ follow from Lemma 7. For the former, $\rho \mathrel{\underline{\sqsubseteq}\!\!\sim} \tau$ and $\tau \mathrel{\underline{\sqsubseteq}\!\!\sim} \psi$ imply the existence of $\rho', \tau', \tau'', \psi'$ such that $\rho \sim \rho' \sqsubseteq \tau' \sim \tau \sim \tau'' \sqsubseteq \psi' \sim \psi$, and we conclude $\rho \mathrel{\underline{\sqsubseteq}\!\!\sim} \psi$ by point 1 and transitivity of $\sqsubseteq$ and $\sim$. For the latter, $\rho \mathrel{\underline{\sqsubseteq}\!\!\sim} \tau$ and $\tau \mathrel{\underline{\sqsubseteq}\!\!\sim} \rho$ imply the existence of $\rho', \rho'', \tau', \tau''$ such that $\rho' \sqsubseteq \tau' \sim \tau \sim \tau'' \sqsubseteq \rho'' \sim \rho \sim \rho'$, so we apply point 1 and obtain some τ_1 such that $\rho' \sqsubseteq \tau' \sim \tau_1 \sqsubseteq \rho'$, and we conclude $\rho \sim \tau$ by point 2. □

The whole point of course is that $\sqsubseteq\!\!\!\sim$ immediately lifts to a partial order on $\sim$-equivalence classes: if R, S are two such classes, just define $R \lesssim S$ iff there exist $\rho \in R$, $\tau \in S$ such that $\rho \sqsubseteq\!\!\!\sim \tau$. If we denote by $[\rho]$ the $\sim$-equivalence class of ρ, we have $[\rho] \lesssim [\tau]$ iff $\rho \sqsubseteq\!\!\!\sim \tau$, so $\lesssim$ is indeed a partial order by Proposition 8, and we may transparently use representatives instead of equivalence classes, which is what we will always do in the sequel.

Before moving on, let us comment on the technical necessity of affinity and of the term $\bot$, which was mentioned in Sect. 1.2.3. Both are motivated by the desire (which, as we will see, is actually also a technical necessity) of making computation monotonic (Proposition 12): in a nutshell, if $t \to^* u$ and $t \sqsubseteq t'$, then we must have $t' \to^* u'$ such that $u \sqsubseteq u'$. For affinity, consider

$$t := u[\langle\rangle := \langle\rangle], \qquad t' := u[\langle\rangle := \langle v\rangle].$$

We have $t \to u$ and $t \sqsubseteq t'$, but, in absence of weakening, v cannot be erased and t' is "stuck". So rule theta of Fig. 1.10 uses weakening.

For what concerns $\bot$, let t be as above, and let

$$t' := u'[\langle x\rangle := \langle\rangle],$$

such that $u \sqsubseteq u'$ and $x \notin \mathrm{fv}(u)$. Again, we have $t \to u$ and $t \sqsubseteq t'$ and yet, if we want to reduce t', we do not know what to substitute for x, which *may* appear in u'. This leads to the introduction of an "undefined" term in the reduction $t' \to u'\{\bot/x\}$. Note that this is monotonic because $u \sqsubseteq u'\{\bot/x\}$ by a direct application of point 2 of Lemma 6.

1.3.3 Ideal completion in posetal double categories

The approximation order may be formulated very naturally in the language of double categories. Instead of considering 2-operads, we will consider **DblPos**-operads, *i.e.*, operads enriched in posetal double categories. Let us give a very quick account of these notions.

Definition 9 (posetal double category) *A* double category *is a category internal to* **Cat**. *We will mostly need the special case of* posetal double categories, *which are categories internal to* **Pos**, *the category of posets and monotonic functions. More explicitly, a posetal double category $\mathcal{D}$ consists of:*

- *a poset $(\mathcal{D}_0, \leq_0)$, whose elements are called* objects*;*
- *a poset of $(\mathcal{D}_1, \leq_1)$ of* arrows *between objects, whose composition, denoted by $\cdot$, is strictly associative and has strict neutral elements id_a, such that*
  - *if $f : a \to a'$, $g : b \to b'$ and $f \leq_1 g$, then $a \leq_0 b$ and $a' \leq_0 b'$;*
  - *if $a \leq_0 b$, then $\mathrm{id}_a \leq_1 \mathrm{id}_b$;*
  - *if $f : a \to b$, $g : b \to c$, $f' : a' \to b'$, $g' : b' \to c'$ and $f \leq_1 f'$, $g \leq_1 g'$, then $g \cdot f \leq_1 g' \cdot f'$;*

A posetal double category $\mathcal{D}$ has two opposites: a vertical opposite $\mathcal{D}^{\mathrm{op}}$, *in which the order on the posets $\mathcal{D}_0$ and $\mathcal{D}_1$ is reversed, and a* horizontal opposite

$\mathcal{D}^{\mathrm{co}}$, in which the direction of the arrows is reversed. A general double category $\mathcal{D}$ also has a transpose *$\mathcal{D}^{\mathrm{t}}$, in which the role of vertical and horizontal arrows is exchanged. The transpose of a posetal double category is not posetal in general but exists as a double category, and we will use it. Of course, all of these three involutions may be combined, which we denote by $\mathcal{D}^{\mathrm{tcoop}}$.*

A morphism of posetal double categories $F : \mathcal{D} \to \mathcal{D}'$ is a pair $(F_0 : \mathcal{D}_0 \to \mathcal{D}'_0, F_1 : \mathcal{D}_1 \to \mathcal{D}'_1)$ of monotonic functions satisfying the obvious functoriality conditions:

- *if $f : a \to b$, then $F_1(f) : F_0(a) \to F_0(b)$;*
- *$F_1(g \cdot f) = F_1(g) \cdot F_1(f)$ and $F_1(\mathrm{id}_a) = \mathrm{id}_{F_0(a)}$.*

We denote by **DblPos** *the category of posetal double categories and their morphisms. This category has products, which are inherited from* **Pos**.

The forgetful functor **Pos** $\to$ **Set** *immediately lifts to a functor*

$$\mathrm{Hor} : \mathbf{DblPos} \to \mathbf{Cat}$$

(our categories are small, hence they are categories internal to **Set***).*[5] *This functor is easily seen to preserve products.*

The idea now is to promote $\Lambda^{\mathrm{p}}_{\mathsf{a}}$ to a **DblPos**-operad by considering cells of the form

$$\begin{array}{ccc} t & \xrightarrow{\ \rho\ } & {}^{*}u \\ \rotatebox{90}{$\sqsubset$} & \swarrow\!\sqcap & \rotatebox{90}{$\sqsubset$} \\ t' & \xrightarrow[\ \rho'\ ]{} & {}^{*}u' \end{array}$$

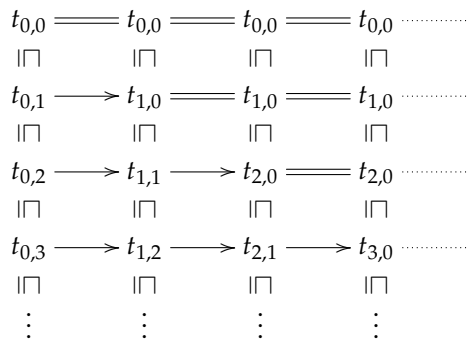
We will give the exact definition momentarily; for the time being, let us look at a motivating example. Let $I := \lambda x.x$, and let

$$t_{m,n} := \langle \overbrace{I, \ldots, I}^{m}, \overbrace{II, \ldots, II}^{n} \rangle,$$
$$\rho_{m,n} := \langle \overbrace{I, \ldots, I}^{m}, x \langle\!| x \leftarrow I |\!\rangle, \overbrace{II, \ldots, II}^{n} \rangle : t_{m,n+1} \to t_{m+1,n}.$$

We invite the reader to check that $\rho_{m,n} \sqsubseteq \rho_{m,n+1}$ and that $t_{m,0} \sqsubseteq \rho_{m,0}$. Then,

[5] The reason behind the notation Hor is that this is a special case of the so-called *horizontal edge category* of a double category.

we may construct the following "patchwork" of cells:



This "patchwork" may be extended arbitrarily, both vertically and horizontally. We are tempted to say that the increasing chain $t_{0,n}$ tends to the infinitary term $t_{0,\infty} := \langle II, II, II, \ldots \rangle$ (with infinitely many copies of II), whereas $t_{n,0}$ tends to $t_{\infty,0} := \langle I, I, I, \ldots \rangle$, and that the longer and longer reductions $t_{0,n} \to^* t_{n,0}$ tend to a reduction $t_{0,\infty} \to t_{\infty,0}$. This idea may be formalized by adapting to double categories the well-known notion of ideal completion.

We remind that a subset $\Delta \subseteq P$ of a poset $(P, \leq)$ is *directed* if it is nonempty and if $x, y \in \Delta$ implies that there exists $z \in \Delta$ such that $x, y \leq z$. A *directed-complete poset* (dcpo) is a poset in which every directed set has a supremum. A monotonic function between dcpo's is (Scott-)*continuous* if it preserves suprema of directed sets. We denote by **Dcpo** the category of dcpo's and continuous functions, and by **DblDcpo** the category of categories internal to **Dcpo**.

An *ideal* of a poset P is a downward-closed directed subset of P. We denote by $\mathrm{Ide}(P)$ the set of all ideals of P. When ordered by inclusion, this may be seen to be a dcpo, called the *ideal completion* of P [AC98]. In fact, this is the object part of a functor $\mathrm{Ide} : \mathbf{Pos} \to \mathbf{Dcpo}$ which is left adjoint to the forgetful functor $\mathbf{Dcpo} \to \mathbf{Pos}$. The action on morphisms is as follows: given a monotonic $f : P \to Q$ and $I \in \mathrm{Ide}(P)$, $\mathrm{Ide}(f)(I) := f(I)\!\downarrow \in \mathrm{Ide}(Q)$, where $(\cdot)\!\downarrow$ denotes downward closure. We will use the fact that Ide is (lax) monoidal, via the maps $(I, I') \mapsto I \times I'$ (whereas $\mathrm{Ide}(1) \cong 1$).

Unfortunately, the above construction does not lift to posetal double categories: the forgetful functor $\mathbf{DblDcpo} \to \mathbf{DblPos}$ does not have a left adjoint. To see this, consider the following posetal double category $\mathcal{D}$:

- $(\mathcal{D}_0, \leq_0) = \mathbb{N} + \mathbb{N} + \mathbb{N} \cong \mathbb{N} \times \{0, 1, 2\}$, *i.e.*, three disjoint copies of $\mathbb{N}$ with its usual order
- $(\mathcal{D}_1, \leq_1)$ is isomorphic to five disjoint copies of $\mathbb{N}$: three consisting of the identity arrows, and two consisting of the following families of arrows:

$$\begin{aligned} f_i : (i, 0) \to (2i, 1) \qquad & f_i \leq_1 f_j \text{ whenever } i \leq j \\ g_i : (2i + 1, 1) \to (i, 2) \qquad & g_i \leq_1 g_j \text{ whenever } i \leq j \end{aligned}$$

- composition is trivial (no pair of non-identity arrows is composable).

Consider now the category $\mathcal{E}$ whose objects are $\{a, b, b', c\}$ and with two non-identity arrows $f : a \to b$ and $g : b' \to c$. This may be seen as a category internal to **Dcpo** by endowing objects and arrows with the discrete order. Let now $F : \mathcal{D} \to \mathcal{E}$ be the morphism such that, for all $i \in \mathbb{N}$,

- on objects, $F(i,0) = a$ and $F(i,2) = c$, whereas $F(i,1) = b$ if i is even, otherwise $F(i,1) = b'$;
- on morphisms, $F(f_i) = f$ and $F(g_i) = g$.

Suppose that we have $\mathcal{D}' \in \mathbf{DblDcpo}$ with $\eta : \mathcal{D} \to \mathcal{D}'$. By completeness, $\mathcal{D}'$ must contain, for each $j \in \{0,1,2\}$, an object $(\omega, j) = \sup_{i\in\mathbb{N}} \eta(i,j)$, as well as two arrows $f_\omega : x \to y$ and $g_\omega : y' \to z$ such that $f_\omega = \sup_{i\in\mathbb{N}} \eta(f_i)$ and $g = \sup_{i\in\mathbb{N}} \eta(g_i)$. Since the source and target functions of $\mathcal{D}'$ are continuous, we must have $x = (\omega, 0)$, $y = y' = (\omega, 1)$ and $z = (\omega, 2)$. Therefore, $\mathcal{D}'$ must also contain an arrow $g_\omega \cdot f_\omega : (\omega,0) \to (\omega,2)$. But there is no way to map this arrow to $\mathcal{E}$ continuously and consistently with F. Indeed, let $G \in \mathbf{DblDcpo}(\mathcal{D}', \mathcal{E})$ be such that $F = G \circ \eta$. Since $F(i,0) = a$ and $F(i,2) = c$ for all $i \in \mathbb{N}$, by continuity we must have $G(\omega,0) = a$ and $G(\omega,2) = c$, so $G(g_\omega \cdot f_\omega)$ must have source a and target c; but there is no such arrow in $\mathcal{E}$.

We will now give sufficient conditions for the left adjoint to exist. Recall that, as a category internal to **Pos**, a posetal double category $\mathcal{D}$ is given by the following data:

$$\mathcal{D}_0 \underset{t}{\overset{e}{\underset{\overset{s}{\leftarrow}}{\rightleftarrows}}} \mathcal{D}_1 \qquad\qquad \mathcal{D}_1 \times_{s,t} \mathcal{D}_1 \xrightarrow{c} \mathcal{D}_1$$

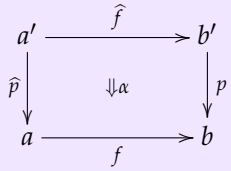
where everything lives in **Pos** and $\mathcal{D}_1 \times_{s,t} \mathcal{D}_1$ denotes the pullback of the cospan formed by s and t (of course, certain diagrams involving the above data are required to commute). The naive idea to obtain an "ideal completion" of $\mathcal{D}$ is to apply the Ide functor to the above morphisms. While this is fine for s, t and e (we obtain a "quiver with pointed loops"), it does not work for c (composition) because Ide does not preserve pullbacks (it is cocontinuous, as every left adjoint of a forgetful functor, but not continuous). However, we may hope to define a monotonic function $\widehat{c} : \mathrm{Ide}(\mathcal{D}_1) \times_{\mathrm{Ide}(s),\mathrm{Ide}(t)} \mathrm{Ide}(\mathcal{D}_1) \to \mathrm{Ide}(\mathcal{D}_1)$ such that

$$\begin{array}{ccc} \mathrm{Ide}(\mathcal{D}_1 \times_{s,t} \mathcal{D}_1) & \xrightarrow{\mathrm{Ide}(c)} & \mathrm{Ide}(\mathcal{D}_1) \\ \downarrow{\scriptstyle u} & \nearrow{\scriptstyle \widehat{c}} & \\ \mathrm{Ide}(\mathcal{D}_1) \times_{\mathrm{Ide}(s),\mathrm{Ide}(t)} \mathrm{Ide}(\mathcal{D}_1) & & \end{array} \tag{1.1}$$

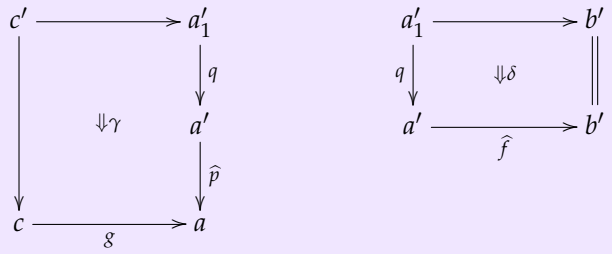
commutes, where u is given by the universal property of the pullback $\mathrm{Ide}(\mathcal{D}_1) \times_{\mathrm{Ide}(s),\mathrm{Ide}(t)} \mathrm{Ide}(\mathcal{D}_1)$. This is what will be ensured by our next definition.

In the following, we consider general double categories and we write $p : a \nrightarrow b$ to denote vertical arrows (which in the posetal case are just order relations $a \leq_0 b$) and we denote by $\cdot$ and $*$ horizontal and vertical composition of cells, respectively.

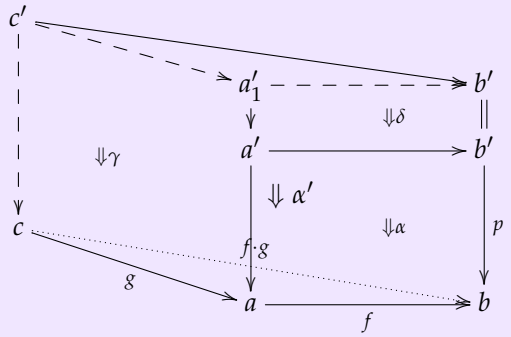
Definition 10 (monotonic posetal double category) *Let $\mathcal{D}$ be a double category and let $f : a \to b$ and $p : b' \nrightarrow b$. A* cartesian lifting *of f with respect to p is a cell α*



such that, if α' is any cell whose vertical target is p and whose horizontal target is $f \cdot g$ for some g, then there exist unique cells



such that $\alpha' = (\alpha * \delta) \cdot \gamma$. Pictorially:



A weak cartesian lifting *is as above but γ, δ are not required to be unique. A double category is (weakly)* fibrational *if every horizontal arrow has a (weak) cartesian lifting with respect to every coterminal vertical arrow.*

A posetal double category $\mathcal{D}$ is monotonic *if $\mathcal{D}^{\mathrm{tcoop}}$ is weakly fibrational (as a double category). We denote by* **MntDblPos** *(resp.* **MntDblDcpo***) the full subcategory of* **DblPos** *(resp.* **DblDcpo***) whose objects are monotonic.*

The terminology "fibrational" is justified as follows.[6] Let $p : \mathcal{A} \to \mathcal{B}$ be a functor; one may define a double category $\mathcal{D}(p)$ such that:

- its objects are (the disjoint union of) the objects of $\mathcal{A}$ and $\mathcal{B}$;
- its horizontal arrows are (the disjoint union of) the arrows of $\mathcal{A}$ and $\mathcal{B}$;

[6]Note that this is unrelated to the notion of *fibrant* double category [Shu10], which is also known as a framed bicategory [Shu08] and is essentially the same a proarrow equipment on a bicategory.

- there is a non-identity vertical arrow $a \nrightarrow b$ exactly when $a \in \mathcal{A}$, $b \in \mathcal{B}$ and $b = p(a)$;
- there is a non-identity cell of horizontal source $f : a \to a'$ and target $g : b \to b'$ exactly when $g = p(f)$;
- composition of horizontal arrows is as in $\mathcal{A}$ and $\mathcal{B}$; from this, horizontal composition of cells is defined in the obvious way, using the functoriality of p; vertical composition is void, both for arrows and cells.

Then, $\mathcal{D}(p)$ is fibrational exactly when p is a fibration. In this particular case, the cell δ is always the identity because $\mathcal{D}(p)$ is vertically degenerate (non-identity vertical arrows are never composable).

Let us spell out what it means for a posetal double category $\mathcal{D}$ to be monotonic. Remember that the horizontal arrows of $\mathcal{D}$ are "actual" arrows, whereas the vertical arrows (and cells) are order relations $\leq_0$ (and $\leq_1$), and remember that in $\mathcal{D}^{\text{tcoop}}$ horizontal and vertical arrows are swapped, and their directions reversed. So, when looking at the diagrams of Definition 10, horizontal arrows $\to$ are to be read as $\geq_0$, vertical arrows $\downarrow$ are to be read as arrows $\uparrow$, and cells $\Downarrow$ are to be read as $\geq_1$, yielding the following: for all $b \leq_0 a$ and $p : b \to b'$, there exists $\widehat{p} : a \to a'$ with $p \leq_1 \widehat{p}$ such that, for all $r : c \to c'$ such that $p \leq_1 r$ and $a \leq_0 c$, there exists $q : a' \to a'_1$ such that $\mathrm{id}_{b'} \leq_1 q$ and $q \cdot \widehat{p} \leq_1 r$ (and, therefore, $p \leq_1 \widehat{p} \cdot q \leq_1 r$). This is saying that "horizontal arrows are monotonic" and, moreover, each arrow my be "overapproximated" in a minimal way.

The property becomes especially conspicuous if we see objects as programs and arrows as computations: if a program b may perform a computation p, then an overapproximation a of b may perform an overapproximation $\widehat{p}$ of p (computation is monotonic!), which is minimal in the sense that every overapproximation of p starting from an overapproximation of b "bigger" than a is in fact an overapproximation of an extension of $\widehat{p}$.

The greater part of the rest of this section is devoted to proving the following:

Theorem 9 *The forgetful functor* **MntDblDcpo** $\to$ **MntDblPos** *has a left adjoint* $\widehat{(\cdot)}$.

Let $\mathcal{D}$ be a monotonic posetal double category, and let $(P, \leq_P)$ be the pullback $\mathrm{Ide}(\mathcal{D}_1) \times_{\mathrm{Ide}(s),\mathrm{Ide}(t)} \mathrm{Ide}(\mathcal{D}_1)$ of the cospan formed by $\mathrm{Ide}(s)$ and $\mathrm{Ide}(t)$. This may be described as

$$P = \{(I, J) \in \mathrm{Ide}(\mathcal{D}_1) \times \mathrm{Ide}(\mathcal{D}_1) \mid t(I)\!\downarrow = s(J)\!\downarrow\}$$

with $(I, J) \leq_P (I', J')$ iff $I \subseteq I'$ and $J \subseteq J'$. If we see $\mathrm{Ide}(\mathcal{D}_1)$ as containing the "limits" of all arrows of $\mathcal{D}$, P is the poset of such limits which are composable. It is easy to see that, given $K \in \mathrm{Ide}(\mathcal{D}_1 \times_{s,t} \mathcal{D}_1)$ (*i.e.*, an ideal of pairs of composable arrows), the monotonic function $u : \mathrm{Ide}(\mathcal{D}_1 \times_{s,t} \mathcal{D}_1) \to P$ of diagram 1.1 is defined by

$$u(K) := (\pi_1(K), \pi_2(K)).$$

Now, we want to define a (monotonic) composition function

$$\widehat{c} : P \longrightarrow \mathrm{Ide}(\mathcal{D}_1)$$

making diagram 1.1 commute. The obvious definition is

$$\widehat{c}(I, J) := \{g \cdot f \mid f \in I, g \in J,\ t(f) = s(g)\}{\downarrow},$$

which is easily seen to guarantee the commutation of diagram 1.1. However, while certainly downward closed, the above set may fail to be directed; in fact, it may very well be empty in general (the double category used above to show that the ideal completion fails gives an example of such phenomenon). This is where we need the fibrational hypothesis, which allows to prove the following key technical result:

Lemma 10 *Let $\mathcal{D}$ be monotonic and let $(I, J) \in P$ (with P defined as above). Then, for all $f_0 \in I$ and $g_0 \in J$, there exist $f \in I$ and $g \in J$ which are composable and such that $f_0 \leq_1 f$, $g_0 \leq_1 g$.*

Proof. Let $f_0 \in I$ and $g_0 \in J$, with $f_0 : a_0 \to b_0$ and $g_0 : b_0' \to c_0$. Since $t(I){\downarrow} = s(J){\downarrow}$, we have $f \in I$ with $f : a \to b$ such that $b_0' \leq_0 b$. By directedness of I, we may suppose $f_0 \leq_1 f$ (if this is not the case, replace f with $f' \in I$ such that $f_0, f \leq_1 f'$). By the same reasoning, we must have $g' \in J$ such that $g' : b' \to c'$, $b \leq_0 b'$ and $g_0 \leq_1 g'$. Now, if we call $\widehat{g} : b \to \widehat{c}$ the arrow obtained by invoking the cartesian lifting of $b_0' \leq b$ with respect to g_0, by minimality we have $g_1 \cdot \widehat{g} \leq_1 g'$ for some $g_1 : \widehat{c} \to c$. If we set $g := g_1 \cdot \widehat{g}$, we have $g : b \to c$ (so it is composable with f) and $g \in J$ by downward closure. □

Lemma 10 immediately implies that the set $\widehat{c}(I, J)$ defined above is an ideal: first, it cannot be empty because $I \neq \emptyset$ and $J \neq \emptyset$; second, given $g_1 \cdot f_1, g_2 \cdot f_2 \in \widehat{c}(I, J)$, by directedness of I and J we have $f_0 \in I$ and $g_0 \in J$ such that $f_1, f_2 \leq_1 f_0$ and $g_1, g_2 \leq_1 g_0$, so we conclude by applying Lemma 10 to f_0 and g_0.

By way of similar applications of Lemma 10, it poses no particular problem to check that, taking $\widehat{c}$ as composition, one indeed obtains a monotonic posetal double category $\widehat{\mathcal{D}}$ on the quiver $\mathrm{Ide}(\mathcal{D}_0) \leftleftarrows \mathrm{Ide}(\mathcal{D}_1)$ resulting from $\mathrm{Ide}(s), \mathrm{Ide}(t)$, with identities $\mathrm{Ide}(e)$. The action of $\widehat{(\cdot)}$ on morphisms is $\widehat{(F_0, F_1)} := (\mathrm{Ide}(F_0), \mathrm{Ide}(F_1))$; that this works is again an easy consequence of Lemma 10. The fact that this is the left adjoint of the forgetful functor **MntDblDcpo** $\to$ **MntDblPos** follows from the left adjointness of Ide. In a sense, the definition of $\widehat{(\cdot)}$ is guaranteed to be universal, as long as the double category actually exists, which is what the fibrational condition ensures.

We conclude this section by observing that $\widehat{(\cdot)}$ lifts to a functor **MntDblPos-Op** $\to$ **MntDblDcpo-Op**. First, one uses the monoidality of Ide to endow $\widehat{(\cdot)}$ with a lax monoidal structure in the obvious way. Then, given a **MntDblPos**-operad $\mathcal{C}$, we define $\widehat{\mathcal{C}}$ by keeping the same colors and setting $\widehat{\mathcal{C}}(\Gamma; A) := \widehat{\mathcal{C}(\Gamma; A)}$. Horizontal composition (and its identities) is defined by precomposing the horizontal composition (and identities) with the lax monoidal structure of $\widehat{(\cdot)}$.

1.3.4 Infinitary affine polyadic terms

Let us prove that, indeed, computation in $\Lambda^{\mathrm{p}}_{\mathsf{a}}$ is monotonic. We start by defining the notion of lift with respect to an overapproximation:

Definition 11 (lift) *Let $\rho : t \to^* u$ and $t \sqsubseteq t'$. We define a reduction term $\rho \uparrow t'$ of source t' by induction on ρ:*

- *$\rho = \mathsf{C}\{\beta\}$, with β a basic step: then, $t = \mathsf{C}\{r\}$ with r a redex, and point 1 of Lemma 6 gives us $t' = \mathsf{C}'\{r'\}$ with $\mathsf{C} \sqsubseteq \mathsf{C}'$ and $r \sqsubseteq r'$; by rule* beta *or* theta *of Fig. 1.10 (according to the shape of r), there is a basic step β' of source r' such that $\beta \sqsubseteq \beta'$, so we define $\rho \uparrow t' := \mathsf{C}'\{\beta'\}$;*
- *$\rho = t$ (a term): we take $\rho \uparrow t' := t'$;*
- *$\rho = \rho_1; \rho_2$: if $\rho_1 \uparrow t' : t' \to^* v'$, we let $\rho \uparrow t' := (\rho_1 \uparrow t'); (\rho_2 \uparrow v')$.*

It is immediate from the definition that $\rho \sqsubseteq \rho \uparrow t'$. It is easy to check that $\rho \sim \rho'$ implies $\rho \uparrow t' \sim \rho' \uparrow t'$, *i.e.*, the definition is consistent with permutation equivalence.

Lemma 11 *Let $\rho' : u' \to^* u''$ be such that $u \sqsubseteq\!\!\!\sim \rho'$, with u a term (i.e., ρ' overapproximates an identity reduction). Then, for all $\tau : u \to^* v$, there exists $\tau' : u' \to^* v'$ such that $v \sqsubseteq \tau'$ and $\rho'; (\tau \uparrow u'') \sim (\tau \uparrow u'); \tau'$.*

Proof. First, we observe that the result follows from the special case in which $u \sqsubseteq \rho'$. Indeed, the general case is $u \sim \varphi \sqsubseteq \psi \sim \rho'$; now, since u is a term, we must have $\varphi = u; \cdots; u$ with $n > 0$ occurrences of u (and some choice of parentheses, which we omit), hence $\psi = \rho'_1; \cdots; \rho'_n$ with $\rho'_i : u'_{i-1} \to^* u'_i$, $u'_0 = u'$ and $u'_n = u''$, and such that $u \sqsubseteq \rho'_i$ for all $1 \leq i \leq n$; then, if we admit the (apparently) weaker version of the lemma, we have that, for all $1 \leq i \leq n$, there exists τ'_i such that $v \sqsubseteq \tau'_i$ and $\rho'_i; (\tau \uparrow u'_{i+1}) \sim (\tau \uparrow u'_i); \tau'_i$; therefore, $\rho'; (\tau \uparrow u'') \sim \rho'_1; \cdots; \rho'_n; (\tau \uparrow u'') \sim \rho_1; \cdots; (\tau \uparrow u'_{n-1}); \tau'_n \sim \cdots \sim (\tau \uparrow u'); \tau'_1; \cdots; \tau'_n$, so we conclude by letting $\tau' := \tau'_1; \ldots; \tau'_n$.

We may therefore assume $u \sqsubseteq \rho'$. In that case, by definition of $\sqsubseteq$ (Fig. 1.10), either $\rho' = u'$ is also a term, in which case the result is trivial; or $\rho' = \mathsf{C}\{\beta\}$ such that the source of the redex β is *not* a subterm of u (*e.g.*, $u = \langle I \rangle$ and $\rho' = \langle I, x(\!|x \leftarrow I|\!)\rangle : \langle I, II\rangle \to \langle I, I\rangle$). Therefore, the reduction τ cannot possibly interfere with ρ' (it reduces other redexes), and permutation equivalence applies. This is formalized by a straightforward induction on u. □

Proposition 12 (monotonicity) *Let $\rho : t \to^* u$, let $t \sqsubseteq t'$ and let v' be the target of $\rho \uparrow t'$. Then, for all $\rho'' : t'' \to^* u''$ such that $t' \sqsubseteq t''$ and $\rho \sqsubseteq\!\!\!\sim \rho''$, there exists $\rho' : u' \to^* u'_0$ such that $u \sqsubseteq\!\!\!\sim \rho'$ and $(\rho \uparrow t'); \rho' \sqsubseteq\!\!\!\sim \rho''$.*

Proof. Again, it is enough to consider the special case $\rho \sqsubseteq \rho''$. In fact, the general case is $\rho \sim \psi \sqsubseteq \psi'' \sim \rho''$; if we admit the (apparently) weaker statement, we get ψ' such that $u \sqsubseteq\!\!\!\sim \psi'$ and $(\psi \uparrow t'); \psi' \sqsubseteq\!\!\!\sim \psi''$; therefore, recalling that $\rho \sim \psi$ implies $\rho \uparrow t' \sim \psi \uparrow t'$, we have $(\rho \uparrow t'); \psi' \sim (\psi \uparrow t'); \psi' \sqsubseteq\!\!\!\sim \psi'' \sim \rho''$,

so we conclude by letting $\rho' := \psi'$.

We now proceed by induction on the derivation of $\rho \sqsubseteq \rho''$. The key cases are when the last rule is box or comp. In the first case, let us assume for convenience of notations that $t = \langle t_1 \rangle$, $t' = \langle t'_1, t'_2 \rangle$ and $t'' = \langle t''_1, t''_2, t''_3 \rangle$. By definition, we have $t \uparrow t' = t'$. Since $t \sqsubseteq \rho''$, we must be in one of the following mutually exclusive cases:

1. $\rho'' = \langle \alpha''_1, t''_2, t''_3 \rangle$,
2. $\rho'' = \langle t''_1, \alpha''_2, t''_3 \rangle$,
3. $\rho'' = \langle t''_1, t''_2, \alpha''_3 \rangle$,

with $\alpha''_i : t''_i \to^* u''_i$ an atomic reduction. Let us deal with each one of them:

1. we know that $t_1 \sqsubseteq \alpha''_1$, so we apply the induction hypothesis, which gives us ρ'_1 such that $t_1 \sqsubseteq\!\!\!\!\sim\, \rho'_1$ and $t'_1; \rho'_1 \sqsubseteq\!\!\!\!\sim\, \alpha''_1$; we let the reader check that $\rho' := \langle \rho'_1, t'_2 \rangle$ satisfies the desired properties;
2. we know that $t'_2 \sqsubseteq t''_2$, the latter term being the source of α''_2. There are two possibilities:
   - α''_2 reduces a redex *not* present in t'_2; in that case, letting $\rho' := \langle t'_1, t'_2 \rangle$ meets the requirements;
   - α''_2 reduces a redex which is also in t'_2; in that case, we have a reduction $\alpha'_2 : t'_2 \to^* u'_2$ reducing that redex, so that $\alpha'_2 \sqsubseteq \alpha''_2$, and we conclude by letting $\rho' := \langle t'_1, \alpha'_2 \rangle$.
3. Letting $\rho' := \langle t'_1, t'_2 \rangle$ (*i.e.*, t' itself) meets the requirements.

Suppose now that the derivation of $\rho \sqsubseteq \rho''$ ends with a comp rule. We have $\rho = \rho_1; \rho_2$, with $\rho_1 : t \to^* v$, and $\rho'' = \rho''_1; \rho''_2$, with $\rho''_1 : t'' \to^* v''$, such that $\rho_1 \sqsubseteq \rho''_2$ and $\rho_2 \sqsubseteq \rho''_2$. Let v' be the target of $\rho_1 \uparrow t'$. We apply the induction hypothesis to $\rho_1 \sqsubseteq \rho''_1$, obtaining $\rho'_1 : v' \to^* v'_0$ such that $v \sqsubseteq\!\!\!\!\sim\, \rho'_1$ and $(\rho_1 \uparrow t'); \rho'_1 \sqsubseteq\!\!\!\!\sim\, \rho''_1$. This latter implies (by point 3 of Lemma 6) $v'_0 \sqsubseteq v''$; hence, if u' is the target of $\rho_2 \uparrow v'_0$, we apply the induction hypothesis to $\rho_2 \sqsubseteq \rho''_2$ and obtain $\rho'_2 : u' \to^* u'_0$ such that $u \sqsubseteq\!\!\!\!\sim\, \rho'_2$ and $(\rho_2 \uparrow v'_0); \rho'_2 \sqsubseteq\!\!\!\!\sim\, \rho''_2$. The fact that $v \sqsubseteq\!\!\!\!\sim\, \rho'_1$ allows us to apply Lemma 11: we get τ' such that $u \sqsubseteq \tau'$ and $\rho'_1; (\rho_2 \uparrow v'_0) \sim (\rho_2 \uparrow v'); \tau'$. We may then conclude by letting $\rho' := \tau'; \rho'_2$. Indeed, $u \sim u; u \sqsubseteq\!\!\!\!\sim\, \tau'; \rho'_2$; and $(\rho_1; \rho_2 \uparrow t'); \tau'; \rho'_2 = (\rho_1 \uparrow t'); (\rho_2 \uparrow v'); \tau'; \rho'_2 \sim (\rho_1 \uparrow t'); \rho'_1; (\rho_2 \uparrow v'_0); \rho'_2 \sim \rho''_1; \rho''_2$. □

It is straightforward to turn $\Lambda^{\mathsf{p}}_{\mathsf{a}}$ into a **MntDblPos**-operad: simply promote each hom-category $\Lambda^{\mathsf{p}}_{\mathsf{a}}(\Gamma; A)$ to a posetal double category using the approximation order; point 6 of Lemma 6 guarantees compatibility with operadic composition. We denote by $\Lambda^{\sqsubseteq}_{\mathsf{a}}$ the **MntDblPos**-operad thus obtained.

It is now possible to explain the reason behind the slightly quirky definition of $\Lambda^{\mathsf{p}}_{\mathsf{a}}$ (and, in general, $\Lambda^{\mathsf{p}}_{\mathsf{p}}$): in view of applying ideal completion, we need to be able to consider ideals as multimorphisms/2-arrows, and these may have infinitely many free affine variables. Consider for instance the as-

cending chain

$$x_1\langle\rangle \sqsubseteq x_1\langle x_2\rangle \sqsubseteq x_1\langle x_2, x_3\rangle \sqsubseteq x_1\langle x_2, x_3, x_4\rangle \sqsubseteq \cdots \tag{1.2}$$

Had we defined $\Lambda_{\mathsf{a}}^{\mathrm{p}}$ in a more customary way, *i.e.*, with each free affine variable counting as an input of color $\bar{\mathsf{a}}$, the chain (1.2) would not belong to $\Lambda_{\mathsf{a}}^{\mathrm{p}}(\bar{\mathsf{a}}^n; \mathsf{I})$ for any $n \in \mathbb{N}$ and, therefore, the ideal completion would not introduce its limit because it would simply fail to "see" the chain. With the definition we adopted, the chain (1.2) is entirely in $\Lambda_{\mathsf{a}}^{\sqsubseteq}(\bar{\mathsf{a}}; \mathsf{I})$, as long as we make sure that all x_i belong to the same supervariable (so, formally, we should write x_i^1).

Of course, Proposition 12 immediately gives us

Corollary 13 *The operad $\Lambda_{\mathsf{a}}^{\sqsubseteq}$ is in* **MntDblPos-Op**, *i.e., every $\Lambda_{\mathsf{a}}^{\sqsubseteq}(\Gamma; C)$ is monotonic.*

We may therefore apply Theorem 9 (or, rather, its operadic version mentioned at the end of the previous section) and define

$$\Lambda_{\mathsf{a}}^{\infty} := \mathrm{Hor}(\widehat{\Lambda_{\mathsf{a}}^{\sqsubseteq}}),$$

i.e., we take the ideal completion of $\Lambda_{\mathsf{a}}^{\sqsubseteq}$ and then forget the order, obtaining a 2-operad. Let us give some examples of how $\Lambda_{\mathsf{a}}^{\infty}$ may be seen as an infinitary affine calculus. Let

$$\begin{aligned} \Delta_n &:= \lambda y.x_0\langle x_1, \ldots, x_n\rangle[\langle x_0, x_1, \ldots, x_n\rangle := y], \\ \Omega_n &:= \Delta_n\langle \Delta_{n-1}, \ldots, \Delta_0, \Delta_0\rangle. \end{aligned}$$

If we denote by ρ_n the obvious reduction $\Omega_{n+1} \to^* \Omega_n$, we have $\rho_n \sqsubseteq \rho_{n+1}$ for all $n \in \mathbb{N}$. The ideal $\{\rho_n \mid n \in \mathbb{N}\}\downarrow$ has source and target equal to the ideal $\{\Omega_n \mid n \in \mathbb{N}\}\downarrow$; clearly we are describing the reduction of the infinitary term Ω_∞ defined by

$$\begin{aligned} \Delta_\infty &:= \lambda y.x_0\langle x_1, x_2, \ldots\rangle[\langle x_0, x_1, x_2, \ldots\rangle := y] \\ \Omega_\infty &:= \Delta_\infty\langle \Delta_\infty, \Delta_\infty, \ldots\rangle. \end{aligned}$$

The term Δ_∞ takes as input an infinite list, extracts the head and applies it to the rest of the list. If we feed to Δ_∞ an infinite list of copies of itself, we get an infinite loop. Note that no duplication is performed during the reduction; the calculus is still affine, it achieves non-termination thanks to its infinitary character. Essentially, we have replaced a potential, non-linear infinity with an actual, linear infinity.

The above is an example of term which is infinite "in width". These are a novelty in the context of infinitary term rewriting and are the most interesting to us. Nevertheless, more traditional infinite terms (as in [KKSdV95]), which are infinite "in height", also exist in $\Lambda_{\mathsf{a}}^{\infty}$. For instance, the ideal $\{\langle\rangle, \langle\langle\rangle\rangle, \langle\langle\langle\rangle\rangle\rangle, \ldots\}$ corresponds to a term V satisfying the equation $V = \langle V\rangle$. These two kinds of "infinities" may of course be combined, obtaining terms satisfying equations such as $U = \langle U, U, U, \ldots\rangle$.

Let us look at a slightly more involved example: fix a sequence $x_0, x_1, x_2, \ldots$ of affine variables, and let u^{++} denote a term u in which each free x_i is replaced by x_{i+1}. Then, if we define

$$\begin{aligned} u_0 &:= x_0\langle\rangle \\ u_{n+1} &:= x_0\langle u_n^{++}\rangle \\ t_n &:= \lambda f.u_n[\langle x_0, \ldots, x_{n-1}\rangle := f] \end{aligned}$$

we have that the ideal $\{t_n \mid n \in \mathbb{N}\}\downarrow$ corresponds to the infinite term $\lambda f.x_0\langle x_1\langle x_2\langle\ldots\rangle\rangle\rangle[\langle x_0, x_1, x_2, \ldots\rangle := f]$. This is essentially the Böhm tree of a linear fixpoint combinator, linear in the sense that the function f takes as argument a list with exactly one element. Note that, by contrast, the (perhaps more natural) infinitary term $\lambda f.x_0(x_1(x_2(\ldots)))[\langle x_0, x_1, x_2, \ldots\rangle := f]$ does not exist in $\Lambda_{\mathsf{a}}^{\infty}$. The Böhm tree of the usual fixpoint combinator may also be represented, although it is a bit more challenging from the notational point of view. In fact, we will se that the infinitary λ-calculus Λ^{001} of Kennaway et al. [KKSdV97] may be embedded in $\Lambda_{\mathsf{a}}^{\infty}$. Along with the usual pure λ-calculus, this calculus contains also all Böhm trees, seen as infinitary λ-terms.

It is possible to give an explicit description of $\Lambda_{\mathsf{a}}^{\infty}$ as a calculus of infinitary affine terms, as we did in [Maz12]. This amounts to allowing infinite terms of the form $\langle t_1, t_2, t_3, \ldots\rangle$, in which the t_i are furthermore allowed to be defined coinductively. We will not indulge in the details but rather go directly to the heart of the matter, which is recovering linear logic from $\Lambda_{\mathsf{a}}^{\infty}$, thus achieving the computational reformulation of Girard's Approximation Theorem promised at the beginning of this section.

1.3.5 Recovering linear logic

Following Girard's cue, it is intuitively clear how to construct an embedding $[\![-]\!]$ of $\Lambda_!$ into $\Lambda_{\mathsf{a}}^{\infty}$, the key case being the encoding of $!T$: if one supposes the ideal $[\![T]\!]$ to be already defined, then $[\![!T]\!]$ should be an ideal of the form

$$\{\langle t_1, \ldots, t_n\rangle \mid t_i \in [\![T]\!]\}.$$

Although morally correct, the above definition is technically wrong because of affinity: we must make sure that $t_1, \ldots, t_n$ share no free variable. This is obviously problematic, for instance, if $T = x$. It is clear that we should allow $[\![x]\!]$ to be equal to $\{x_i\}$ for any $i \in \mathbb{N}$ (*i.e.*, x_i belongs to the supervariable associated with x), so that $[\![!x]\!]$ may be any ideal of the form $\{\langle x_{i_1}, \ldots, x_{i_n}\rangle \mid n \in \mathbb{N}, i_n \text{ arbitrary sequence}\}$. So the embedding is not going to be uniquely defined; but we may hope for it to be defined modulo an equivalence relation, which is what we will aim at.

Before we continue, let us recall our convention that, for every term or reduction term of $\Lambda_{\mathsf{a}}^{\mathsf{p}}$ and for any of its subterms of the form $\rho[\langle x_1, \ldots, x_n\rangle := \tau]$ or $\rho(\!|x_1, \ldots, x_n \leftarrow \tau|\!)[-]$, we assume, using α-equivalence, that $x_1, \ldots, x_n$ is the initial segment of a supervariable.

We start by defining a map $(-)^\bullet$ from $\Lambda^{\mathrm{p}}_{\mathsf{a}}$ to $\Lambda^{\mathrm{p}}_{\mathsf{c}}$:

$$\begin{array}{rlrl} a^\bullet &:= a & x_i^\bullet &:= x \\ (\lambda a.\rho)^\bullet &:= \lambda a.\rho^\bullet & \langle \rho_1, \dots, \rho_n \rangle^\bullet &:= \langle \rho_1^\bullet, \dots, \rho_n^\bullet \rangle \\ (\rho\tau)^\bullet &:= \rho^\bullet \tau^\bullet & (\rho[\langle x_1, \dots, x_n \rangle := \tau])^\bullet &:= \rho^\bullet[\langle x \rangle := \tau^\bullet] \\ (\rho;\tau)^\bullet &:= \rho^\bullet;\tau^\bullet & (t(\!|a \leftarrow u|\!)[-])^\bullet &:= t^\bullet(\!|a \leftarrow u^\bullet|\!)[-]^\bullet \end{array}$$

$$(t(\!|x_1, \dots, x_n \leftarrow u_1, \dots, u_m|\!)[-])^\bullet := t^\bullet(\!|x \leftarrow u_1^\bullet, \dots, u_m^\bullet|\!)[-]^\bullet$$

Note that this does *not* yield a morphism of operads: if $\rho : t \to^* u$, we do not have in general that $\rho^\bullet : t^\bullet \to^* u^\bullet$ (because of the last line of the definition). In fact, saying that the target of $(-)^\bullet$ is $\Lambda^{\mathrm{p}}_{\mathsf{c}}$ is just a convenient way to say where the images of $(-)^\bullet$ belong, but such images will be used in a quite trivial way: we just check them for equality.

Definition 12 (renaming equivalence) *Let R, R' be two ideals of $\Lambda^\infty_{\mathsf{a}}$ (multimorphisms or 2-arrows). We write $R \approx R'$ just if there is an isomorphism of posets $\iota : R \to R'$ such that, for all $\rho \in R$, $\rho^\bullet = \iota(\rho)^\bullet$.*

In other words, two ideals are renaming equivalent when they are isomorphic and the isomorphism relates terms which are equal once we forget the indices of the affine variables, free or bound.

It is immediate that $(\rho\{\tau/x\})^\bullet = \rho^\bullet\{\tau^\bullet/x\}$, so $R \approx R'$ and $S \approx S'$ implies $R\{S/x\} \approx R'\{S'/x\}$; we also obviously have $R;S \approx R';S'$ (whenever this makes sense), so we may form a 2-operad $\Lambda^\infty_{\mathsf{a}}/\!\approx$ by taking the quotient under $\approx$ in $\Lambda^\infty_{\mathsf{a}}$. It is in this 2-operad that $\Lambda_!$ will be embedded, as follows:

$$\begin{aligned} [\![a]\!] &:= \{\{a\}\downarrow\} \\ [\![\lambda a.\varphi]\!] &:= \{\{\lambda a.\rho \mid \rho \in R\} \mid R \in [\![\varphi]\!]\} \\ [\![\varphi\psi]\!] &:= \{\{\rho\tau \mid \rho \in R,\ \tau \in S\} \mid R \in [\![\varphi]\!],\ S \in [\![\psi]\!],\ R \# S\} \\ [\![x]\!] &:= \{\{x_i\}\downarrow \mid i \in \mathbb{N}\} \\ [\![!\varphi]\!] &:= \{\{\langle \rho_1, \dots, \rho_n \rangle \mid \rho_i \in R_i,\ n \in \mathbb{N}\} \mid (R_i)_{i\in\mathbb{N}} \in [\![\varphi]\!], \\ &\qquad\qquad R_i \# R_j\ \forall i, j \in \mathbb{N}\} \\ [\![\varphi[!x := \psi]]\!] &:= \{\{\rho[\langle x_1, \dots, x_m \rangle := \tau] \mid \rho \in R,\ \tau \in S, \\ &\qquad\qquad m \geq \max i \text{ s.t. } x_i \in \mathrm{fv}(\rho)\} \\ &\qquad \mid R \in [\![\varphi]\!],\ S \in [\![\psi]\!],\ R \# S\} \\ [\![T(\!|a \leftarrow U|\!)[-]]\!] &:= \{\{t(\!|a \leftarrow u|\!)[-]' \mid t \in I,\ u \in J,\ [-]' \in K\} \\ &\qquad \mid I \in [\![T]\!],\ J \in [\![U]\!],\ K \in [\![[-]]\!],\ I \# J \# K \text{ pairwise}\} \\ [\![T(\!|x \leftarrow U|\!)[-]]\!] &:= \{\{t(\!|x_1, \dots, x_m \leftarrow u_1, \dots, u_n|\!)[-]' \mid t \in I,\ u_i \in J_i,\ [-]' \in K, \\ &\qquad\qquad n \in \mathbb{N}, \\ &\qquad\qquad m \geq \max i \text{ s.t. } x_i \in \mathrm{fv}(t)\} \\ &\qquad \mid I \in [\![T]\!],\ (J_i)_{i\in\mathbb{N}} \in [\![U]\!],\ K \in [\![[-]]\!] \\ &\qquad\quad I \# J_i \# J_j \# K \text{ pairwise},\ \forall i, j \in \mathbb{N}\} \\ [\![\varphi;\psi]\!] &:= \{\{\rho;\tau \mid \rho \in R,\ \tau \in S\} \mid R \in [\![\varphi]\!],\ S \in [\![\psi]\!],\ R \# S\} \end{aligned}$$

where the notation $R \# S$ means $\mathrm{fv}(R) \cap \mathrm{fv}(S) = \emptyset$, and each set in $[\![\varphi]\!]$ is understood to be closed under permutation equivalence.

By a straightforward induction on φ one may prove that, if $\varphi : T \to^* U$ is a 2-arrow of $\Lambda_!(\mathsf{I}^m, \mathsf{c}^n; \mathsf{I})$, then for all $R \in [\![\varphi]\!]$, there exist $I \in [\![T]\!]$, $J \in [\![U]\!]$ such that $R : I \to^* J$ is a 2-arrow of $\Lambda_{\mathsf{a}}^{\infty}(\mathsf{I}^m, \overline{\mathsf{a}}^n; \mathsf{I})$. In particular, each $R \in [\![\varphi]\!]$ is an ideal. The approximation theorem will follow from a characterization of such ideals.

The main point of a $\underset{\sim}{\sqsubset}$-ideal of terms is that it has an underlying syntactic tree. Indeed, as pointed out in the previous section, ideals may be seen as infinitary terms. In particular, an ideal R has a "kind", which is the syntactic constructor of its root (lvar, lam, app, etc.), and, when appropriate, it has immediate "sub-ideals" corresponding to subterms: *e.g.*, if R is of kind lam, then there will be an ideal R_1 such that $R = \{\lambda a.\rho_1 \mid \rho_1 \in R_1\}$; R_1 is the immediate sub-ideal of R. Of course, ideals of kind lvar, avar and undef do not have immediate sub-ideals. Of special interest to us is the case of ideals of kind box; such an ideal R has, for each $i \in \mathbb{N}$, an immediate sub-ideal $\pi_i R := \{\rho_i \mid \langle \rho_1, \ldots, \rho_n \rangle \in R, n \geq i\}{\downarrow}$. Note that $\pi_i R$ is defined for all i; at worst, it will be equal to $\{\bot\}$. The same applies to ideals of kind theta.

Definition 13 (uniform, finitary ideal) *We define the class of* uniform *ideals to be the largest not containing $\{\bot\}$, closed under immediate sub-ideals and such that, if R is uniform of kind* box *or* theta*, then $\pi_i R \approx \pi_j R$ for all $i, j \in \mathbb{N}$.*

An ideal R is finitary *if there is $h \in \mathbb{N}$ such that, for all $\rho \in R$, the height of ρ (as a syntactic tree) is bounded by h.*

Lemma 14 *For every reduction term φ of $\Lambda_!$, $[\![\varphi]\!]$ is a $\approx$-equivalence class of finitary uniform ideals.*

Proof. A straightforward induction on φ, from the definition of $[\![-]\!]$. □

Lemma 15 *For every finitary uniform ideal R of $\Lambda_{\mathsf{a}}^{\infty}$, there exists a unique reduction term φ of $\Lambda_!$ such that $R \in [\![\varphi]\!]$.*

Proof. Note that finitary ideals have a well-defined notion of *height*: it is the maximum height of their terms. We may therefore proceed by induction on the height of R. The most interesting cases are the ones in which R is of kind box or theta. We illustrate the first, the second is analogous. Being finitary of course is stable under sub-ideals, so $\pi_i R$ is finitary uniform for all $i \in \mathbb{N}$, so by induction there is a unique φ_i such that $R_i \in [\![\varphi_i]\!]$ for all $i \in \mathbb{N}$. But, by uniformity, $R_i \approx R_j$ for all $i, j \in \mathbb{N}$, so by Lemma 14 $\varphi_i = \varphi_j =: \varphi$ for all $i, j \in \mathbb{N}$, and this is obviously unique such that $R = [\![!\varphi]\!]$. □

Uniformity and being finitary are obviously preserved by substitution and composition, and are stable under renaming equivalence. Therefore, we may define a 2-operad $\Lambda_{\mathsf{a}}^{\infty\mathsf{fu}}/\approx$ which is a suboperad of $\Lambda_{\mathsf{a}}^{\infty}/\approx$ in which we only consider finitary uniform ideals. This is exactly linear logic:

Theorem 16 (Girard's approximation theorem, computational version) *There is an isomorphism of 2-operads*

$$\Lambda_{\mathsf{a}}^{\infty\mathsf{fu}}/\approx \quad \cong \quad \Lambda_{!}.$$

Proof. We have basically already defined a morphism

$$[\![-]\!] : \Lambda_{!} \longrightarrow \Lambda_{\mathsf{a}}^{\infty\mathsf{fu}}/\approx .$$

Its behavior on colors has been specified implicitly: $[\![\mathsf{l}]\!] := \mathsf{l}$ and $[\![\mathsf{c}]\!] := \bar{\mathsf{a}}$. The fact that this is indeed a morphism into $\Lambda_{\mathsf{a}}^{\infty\mathsf{fu}}/\approx$ follows from Lemma 14. That such a morphism is invertible is a consequence of Lemma 15. □

So a λ-term is an equivalence class of *uniform* infinitary affine terms. The idea of uniformity is far from new: it is a staple of the denotational semantics of linear logic, especially games semantics [Gir01, Mel04].

In [Maz12], some applications of Theorem 16 to the theory of the λ-calculus are explored, most notably a proof of confluence via strong confluence of $\Lambda_{\mathsf{a}}^{\mathsf{p}}$ and a simple proof by "passage to the limit" of Wadsworth's result that head reduction terminates for solvable terms [Wad71].

We also observe that, if we drop the constraint on height, equivalence classes of uniform ideals correspond to terms of an infinitary calculus in which $!(-)$ is coinductive, *i.e.*, terms satisfying equations such as $T = I!T$ are allowed. The infinitary λ-calculus Λ^{001} of Kennaway et al. [KKSdV97] may be embedded in such an infinitary version of linear logic, allowing one to recover their whole theory on an affine basis.

For us, the most noteworthy consequence of Theorem 16 is that it gives rise to a notion of affine approximation for (reduction) terms of $\Lambda_{!}$ and, therefore, of the λ-calculus (via Girard's embeddings). In fact, such a notion of approximation makes sense in *all* polyadic calculi, not just the affine one. We give it in Fig. 1.11, where the relation is defined by means of judgments of the form

$$\Xi \vdash \rho \sqsubset \varphi,$$

where:

- ρ is a reduction term of $\Lambda_{\mathsf{p}}^{\mathsf{p}}$ (with $\mathsf{p} \in \{\mathsf{l}, \mathsf{a}, \mathsf{r}, \mathsf{c}\}$);
- φ is a reduction term of $\Lambda_{!}$;
- Ξ is a sequence (permutable at will) of pairs $y \sqsubset x$, with y a variable of type p of $\Lambda_{\mathsf{p}}^{\mathsf{p}}$ and x a cartesian variable of $\Lambda_{!}$, such that $y \sqsubset x, y' \sqsubset x'$ in Ξ implies $y \neq y'$; instead, $x = x'$ is allowed, it means that both y and y' approximate x.

In the following chapters we will explore some interesting applications of these polyadic approximations.

$$\frac{}{\vdash a \sqsubset a}\ \textsf{lvar} \qquad \frac{}{y \sqsubset x \vdash y \sqsubset x}\ \textsf{cvar}$$

$$\frac{\Xi \vdash \rho \sqsubset \varphi}{\Xi \vdash \lambda a.\rho \sqsubset \lambda a.\varphi}\ \textsf{lam} \qquad \frac{\Xi \vdash \rho \sqsubset \varphi \qquad \Upsilon \vdash \tau \sqsubset \psi}{\Xi, \Upsilon \vdash \rho\tau \sqsubset \varphi\psi}\ \textsf{app}$$

$$\frac{\Xi_1 \vdash \rho_1 \sqsubset \varphi \quad \dots \quad \Xi_n \vdash \rho_n \sqsubset \varphi}{\Xi_1, \dots, \Xi_n \vdash \langle \rho_1, \dots, \rho_n \rangle \sqsubset\ !\varphi}\ \textsf{box}$$

$$\frac{\Upsilon \vdash \tau \sqsubset \psi \qquad \Xi, y_1 \sqsubset x, \dots, y_n \sqsubset x \vdash \rho \sqsubset \varphi}{\Xi, \Upsilon \vdash \rho[\langle y_1, \dots, y_n \rangle := \tau] \sqsubset \varphi[!x := \psi]}\ \textsf{let} \quad x \notin \Xi$$

$$\frac{\Xi \vdash (\lambda a.t)[-]'u \sqsubset (\lambda a.T)[-]U}{\Xi \vdash t(\!|a \leftarrow u|\!)[-]' \sqsubset T(\!|a \leftarrow U|\!)[-]}\ \textsf{beta}$$

$$\frac{\Xi \vdash t[\langle y_1, \dots, y_m \rangle := \langle u_1, \dots, u_n \rangle[-]'] \sqsubset T[!x := !U[-]]}{\Xi \vdash t(\!|y_1, \dots, y_m \leftarrow u_1, \dots, u_n|\!)[-]' \sqsubset T(\!|x \leftarrow U|\!)[-]}\ \textsf{theta}\ (*)$$

$$\frac{\Xi \vdash \rho \sqsubset \varphi \quad \Upsilon \vdash \tau \sqsubset \psi}{\Xi, \Upsilon \vdash \rho; \tau \sqsubset \varphi; \psi}\ \textsf{comp}$$

$$\frac{\Xi \vdash \rho \sqsubset \varphi}{\Xi, y \sqsubset x \vdash \rho \sqsubset \varphi}\ \textsf{weak, p}\in\{\textsf{a,c}\} \qquad \frac{\Xi, y \sqsubset x, y' \sqsubset x \vdash \rho \sqsubset \varphi}{\Xi, y \sqsubset x \vdash \rho\{y/y'\} \sqsubset \varphi}\ \textsf{cntr, p}\in\{\textsf{r,c}\}$$

Figure 1.11: Polyadic approximations. The side condition $(*)$ in the theta rule is that, if $\mathsf{p} \in \{\mathsf{l}, \mathsf{r}\}$, then $m \geq n$.

Chapter 2

Intersection Types

2.1 A Surprising Correspondence

Intersection types, originally introduced by Coppo and Dezani [CDC80] with semantic motivations, are a well-known type-theoretic approach to expressing and capturing dynamic properties of programs. Through the years, the theory of intersection types has ramified along a host of different directions and taken a number of different forms. One of these is the so-called *non-idempotent* variant of intersection types. Intuitively, intersection is non-idempotent when $A \to A \wedge A$ does not hold. So, for instance, the typing judgment

$$f : A \to A \to B \vdash \lambda x.fxx : A \to B$$

is not derivable with a non-idempotent intersection; instead

$$f : A \to A \to B \vdash \lambda x.fxx : A \wedge A \to B$$

is derivable.

The first (and simplest) example of non-idempotent intersection type system was introduced by Gardner [Gar94] and, independently and with a slightly different formulation, by de Carvalho [dC09]. Types are defined by

$$\begin{array}{ll} A ::= \alpha \mid \Theta \multimap A & \text{types,} \\ \Theta ::= A_1 \wedge \cdots \wedge A_n & \text{intersections.} \end{array}$$

Nullary intersections are authorized and denoted by $\top$. Intersections should be seen as elements of the free monoid over types: $\Theta \wedge \Theta'$ is defined by concatenation and $\top$ is the neutral element. Typing judgments are of the form $\Gamma \vdash M : A$, where M is a pure λ-term, A a type and Γ a list (permutable at will) of type declarations of the form $x : \Theta$. Gardner's version of the typing rules is recalled in Fig. 2.1. In rule app, the context $\Gamma \cdot \Delta_1 \cdots \Delta_n$ is obtained by concatenating the type declarations in Γ, Δ_i, assuming that each variable appearing in any of Γ, Δ_i actually appears in all of them, by adding the fictitious declaration $x : \top$ when it does not.

The behavior of non-idempotent intersection of course hints to a connection with linear logic. Indeed, after Gardner, non-idempotent intersection

$$\frac{}{x : A \vdash x : A}\ \mathsf{var} \qquad \frac{\Gamma, x : A_1 \wedge \cdots \wedge A_n \vdash M : C}{\Gamma, x : A_{\sigma(1)} \wedge \cdots \wedge A_{\sigma(n)} \vdash M : C}\ \mathsf{perm}$$

$$\frac{\Gamma \vdash M : B}{\Gamma \vdash \lambda x.M : \top \multimap B}\ \mathsf{lam}_0 \quad x \notin \Gamma \qquad \frac{\Gamma, x : A_1 \wedge \cdots \wedge A_n \vdash M : B}{\Gamma \vdash \lambda x.M : A_1 \wedge \cdots \wedge A_n \multimap B}\ \mathsf{lam}$$

$$\frac{\Gamma \vdash M : A_1 \wedge \cdots \wedge A_n \multimap B \quad \Delta_1 \vdash N : A_1 \quad \ldots \quad \Delta_n \vdash N : A_n}{\Gamma \cdot \Delta_1 \cdots \Delta_n \vdash MN : B}\ \mathsf{app}$$

Figure 2.1: Gardner/de Carvalho's non-idempotent intersection type system, as presented in [Gar94].

types were considered again by Carlier et al. [CPWK04] and their relationship with linear logic, and comparison with the idempotent case, expounded by Neergaard and Mairson [NM04]. In his independent work, de Carvalho unveiled the strong link between non-idempotent intersection types and the relational semantics of linear logic [dC09], a line of work which was subsequently extended by Bernadet and Lengrand [BL13] and by de Carvalho himself with Pagani and Tortora [dCPTdF11]. We will see that there is an even tighter correspondence between intersection types and linear logic (in its polyadic form) which, in fact, goes well beyond the non-idempotent case.

Let us start by adapting the linear polyadic calculus to make it match more closely the pure λ-calculus. One way of doing this is to internalize Girard's translation in the calculus, obtaining the following terms and reduction rule (indeed, this is the syntax we used in [Maz12]):

$$t, u ::= x \mid \lambda\langle x_1, \ldots, x_n\rangle.t \mid t\langle u_1, \ldots, u_n\rangle,$$

$$(\lambda\langle x_1, \ldots, x_m\rangle.t)\langle u_1, \ldots, u_n\rangle \quad \to \quad t\{u_i/x_i\},$$

where x ranges over polyadic linear variables (of color $\bar{\mathrm{l}}$) and the reduction rule requires $m = n$ (otherwise the term is "stuck"). This, of course, is just a subcalculus of $\Lambda^{\mathrm{p}}_{\mathrm{l}}$: $\lambda\langle x_1, \ldots, x_n\rangle.t$ is merely syntactic sugar for $\lambda a.t[\langle x_1, \ldots, x_n\rangle := a]$, the term $\bot$ is excluded, and the reduction rule is consistent with all this. We may therefore adapt the approximation relation (Fig. 1.11) to it, obtaining the following rules (we consider only terms):

$$\frac{}{x_0 \sqsubset x \vdash x_0 \sqsubset x}\ \mathsf{var} \qquad \frac{\Gamma, x_1 \sqsubset x, \ldots, x_n \sqsubset x \vdash t \sqsubset M}{\Gamma \vdash \lambda\langle x_1, \ldots, x_n\rangle.t \sqsubset \lambda x.M}\ \mathsf{lam}$$

$$\frac{\Gamma \vdash t \sqsubset M \quad \Delta_1 \vdash u_1 \sqsubset N \quad \ldots \quad \Delta_n \vdash u_n \sqsubset N}{\Gamma, \Delta_1, \ldots, \Delta_n \vdash t\langle u_1, \ldots, u_n\rangle \sqsubset MN}\ \mathsf{app}$$

where, in rule lam, x does not appear in Γ.

We said that the above subcalculus of $\Lambda^{\mathrm{p}}_{\mathrm{l}}$ was obtained by internalizing Girard's translation; of course, this is above all a logical translation (intuitionistic to linear), so it is valid at the level of types. In fact, the subcalculus results

from considering the fragment of $\mathbf{Poly}_!$ whose types are restricted to

$$A, B ::= \alpha \mid \langle A_1, \ldots, A_n \rangle \multimap B.$$

The induced typing rules (from those of Fig. 1.9) are the following:

$$\frac{}{x_0 : A \vdash x_0 : A}\ \mathsf{var} \qquad \frac{\Gamma, x_1 : A_1 \ldots x_n : A_n \vdash t : B}{\Gamma \vdash \lambda\langle x_1, \ldots, x_n \rangle.t : \langle A_1, \ldots, A_n \rangle \multimap B}\ \mathsf{lam}$$

$$\frac{\Gamma \vdash t : \langle A_1, \ldots, A_n \rangle \multimap B \quad \Delta_1 \vdash u_1 : A_1 \quad \ldots \quad \Delta_n \vdash u_n : A_n}{\Gamma, \Delta_1, \ldots, \Delta_n \vdash t\langle u_1, \ldots, u_n \rangle : B}\ \mathsf{app}$$

Now, the typing rules are so similar to the approximation rules that it is tempting to superpose them:

$$\frac{}{x_0 \sqsubset x : A \vdash x_0 \sqsubset x : A}\ \mathsf{var}$$

$$\frac{\Gamma, x_1 \sqsubset x : A_1, \ldots, x_n \sqsubset x : A_n \vdash t \sqsubset M : B}{\Gamma \vdash \lambda\langle x_1, \ldots, x_n \rangle.t \sqsubset \lambda x.M : \langle A_1, \ldots, A_n \rangle \multimap B}\ \mathsf{lam}$$

$$\frac{\Gamma \vdash t \sqsubset M : \langle A_1, \ldots, A_n \rangle \multimap B \quad \Delta_1 \vdash u_1 \sqsubset N : A_1 \quad \ldots \quad \Delta_n \vdash u_n \sqsubset N : A_n}{\Gamma, \Delta_1, \ldots, \Delta_n \vdash t\langle u_1, \ldots, u_n \rangle \sqsubset MN : B}\ \mathsf{app}$$

where, in rule lam, x does not appear in Γ. If, in the above superposition, we keep only the types and the purple decorations, we obtain a type system for the λ-calculus, in which typing contexts may contain more than one declaration for each variable. If we adopt the following changes of notation

$$\begin{array}{rcl} \langle A_1, \ldots, A_n \rangle & \rightsquigarrow & A_1 \wedge \cdots \wedge A_n \\ \Gamma, x : A_1, \ldots, x : A_n \vdash M : B & \rightsquigarrow & \Gamma, x : A_1 \wedge \cdots \wedge A_n \vdash M : B \qquad x \notin \Gamma \end{array}$$

we see that this is exactly Gardner's system:

- the rules var and app are identical;
- Gardner's lam_0 rule is just the case $n = 0$ of our lam rule;
- Gardner's perm rule is given by the (implicit) exchange rule on contexts of polyadic derivations.

If we repeat the above syntactic game with the other variants of our polyadic calculi, we find other intersection type systems. For instance, using affine approximations gives us a relaxation of Gardner's system with a form of *subtyping* (*e.g.*, the identity may be given type $A \wedge B \to A$); with cartesian approximation we obtain a reformulation of Coppo, Dezani and Venneri's system [CDCV81] (which uses what are sometimes called "strict" intersection types [vB95], *i.e.*, with intersections only on the left of arrows, as opposed to

Coppo and Dezani's original system [CDC80]), in its incarnation characterizing solvability/head normalization, because of the unrestricted presence of the type Ω, represented here by the type $\langle\rangle$ (the "non-strict" version of such a system is what Krivine calls $D\Omega$ in [Kri93]). We therefore have what seems to be a completely general correspondence between polyadic simple types and intersection types:

$$\delta :: \Gamma \vdash_{\mathbf{IT}} M : A \qquad \text{iff} \qquad \delta^- \sqsubset M.$$

That is, an intersection type derivation δ for M is isomorphic to a (Church-style) simply-typed polyadic term, such that the underlying pure term δ^- is a polyadic approximation of M. More snappily: *a λ-term is typable in intersection types iff it admits a simply-typable polyadic approximation*. Such a sharp correspondence deserves to be treated more abstractly.

2.2 Fibrations

2.2.1 Type systems as morphisms of 2-operads

What is a type system? Melliès and Zeilberger [MZ15] recently suggested an amazingly simple answer to this question: a type system is a functor, mapping a category $\mathcal{D}$ of derivations to a monoid $\mathcal{C}$ of programs. This is of course in the simple case in which programs are untyped; otherwise, $\mathcal{C}$ is also a category, and we speak more generally of a type *refinement* system.

Melliès and Zeilberger's idea naturally applies to our operadic approach: for us, a type system is a morphism of 2-operads. Let us see how this works in the case of the simply-typed λ-calculus, whose operadic presentation we gave in Sect. 1.1.4.

Given a Church-style simply-typed λ-term δ, we denote by δ^- its "erasure", *i.e.*, the underlying pure λ-term with all typing annotations removed. Of course, a reduction sequence $\rho : \delta \to^* \delta'$ in Λ_{ST} induces a unique reduction sequence $\rho^- : \delta^- \to^* \delta'^-$ in Λ; moreover, $(\delta\{\varepsilon/x\})^- = \delta^-\{\varepsilon^-/x\}$, so we have a morphism of 2-operads, and thus a type system

$$\begin{array}{c} \Lambda_{\mathrm{ST}} \\ {\scriptstyle (\cdot)^-}\big\downarrow \\ \Lambda \end{array}$$

Although basic, this example showcases the essential idea behind Melliès and Zeilberger's view of type systems: Λ_{ST} is a 2-operad of type derivations "over" the monochromatic 2-operad Λ; the object part of each forgetful functor $(\cdot)^-_{\Gamma;A} : \Lambda_{\mathrm{ST}}(\Gamma; A) \to \Lambda(n)$ (where n is the length of Γ) takes a type derivation, *i.e.*, a proof of a judgment of the form $\Gamma \vdash M : A$, and gives its subject, *i.e.*, M.

Note that, while Church-style typing is used in the definition of Λ_{ST}, the type system $(\cdot)^-$ expresses simple typing in *Curry style*: a pure λ-term M is

simply-typable (rather than *simply-typed*) if it is in the image of $(\cdot)^-$. More conspicuously, consider the 2-operad **NJ** presenting propositional minimal natural deduction. In our language, the Curry-Howard correspondence reduces to the statement that there is an isomorphism of 2-operads $\mathbf{CH} : \mathbf{NJ} \cong \Lambda_{\mathrm{ST}}$, yielding

$$\begin{array}{c} \mathbf{NJ} \\ {\scriptstyle (\cdot)^- \circ \mathbf{CH}} \big\downarrow \\ \Lambda \end{array}$$

which is a reformulation of the above type system making no *a priori* use of Church-style typing.

2.2.2 Subject reduction/expansion and opfibrations/fibrations

It is a classical result that the (Curry-style) system of simple types for λ-terms enjoys subject reduction, *i.e.*, typing is stable under reduction. Rephrased in the above language, this means that if there is $\delta \in \Lambda_{\mathrm{ST}}(\Gamma; A)$ such that $\delta^- = M$ and a reduction $\psi : M \to^* M'$, then there exists $\delta' \in \Lambda_{\mathrm{ST}}(\Gamma; A)$ such that $\delta'^- = M'$. Of course, nothing in the definition of morphism of 2-operads forces this to hold. However, in the case of simple types, by inspecting the proof one notices that we actually have something stronger: there is a reduction $\tau : \delta \to^* \delta'$ such that $\tau^- = \psi$. Indeed, the typing derivation δ' is constructed from δ following the reduction sequence ψ. In categorical jargon, one says that the morphism $(\cdot)^-$ enjoys a form of *oplifting property*.

Dually, one may ask for a *lifting property*: if $M = \delta^-$ and $\psi : M' \to^* M$, then there exists $\tau : \delta' \to^* \delta$ such that $\tau^- = \psi$. The reader will have recognized here what is usually called *subject expansion*. Although it fails in simple types, many intersection type systems do enjoy it.

Lifting (resp. oplifting) properties are at the heart of the categorical notion of *fibration* (resp. *opfibration*). However, one usually asks liftings and opliftings to be canonical in some sense, *e.g. cartesian* for Grothendieck (op)fibrations, or even unique for discrete (op)fibrations. In our case, type systems do not, except in extremely rare cases, ensure any form of canonicity. Therefore, to obtain the high level of generality we are aiming at, we are forced to consider a weaker notion of (op)fibration.

Naively, we would be led to ask mere existence of (op)liftings. For instance, we would define a weak fibration as a functor $p : \mathcal{E} \to \mathcal{B}$ such that, for all $f : b \to p(e')$ in $\mathcal{B}$, there exists $g : e \to e'$ in $\mathcal{E}$ such that $p(g) = f$ (in a discrete fibration, g is required to be unique; in a Grothendieck fibration, g is required to satisfy a minimality property). However, this naive approach appears to be inadequate. In fact, there does not seem to be a structure on the fibers $p^{-1}(b)$ that encodes mere existence of liftings. For instance, if $p : \mathcal{E} \to \mathcal{B}$ is a discrete fibration, then the map

$$\begin{array}{rccc} \partial p : & \mathcal{B} & \to & \mathbf{Set} \\ & b & \mapsto & p^{-1}(b) \end{array}$$

actually yields a contravariant functor (*i.e.*, a presheaf). In the case of a Grothendieck fibration, the same map becomes a contravariant pseudofunctor in **Cat**. Mere existence of liftings does not seem to yield any structure of this sort. We do not claim that there is none; after all, plain functors, without any lifting property, do induce normal lax 2-functors in the category **Dist** of distributors (this is an old observation of Bénabou [Bén00]). So maybe there is some subcategory of **Dist** which corresponds to mere existence of liftings... we simply do not know. Luckily, there is an alternative approach offering a very robust solution.

2.2.3 Type systems as Niefield fibrations

There is a notion of functor which encompasses both fibrations and opfibrations: it is called a Conduché functor or, in the terminology of Johnstone [Joh99], a *Conduché fibration*. These are functors enjoying a certain *factorization lifting property*, *i.e.*, they lift compositions of arrows (and identities), rather than arrows themselves. Let us look at the discrete case, which was first studied by Johnstone [Joh99].

A *discrete Conduché fibration* is a functor $p : \mathcal{E} \to \mathcal{B}$ such that, for every arrow k of $\mathcal{E}$:

identity lifting: if $p(k)$ is an identity, then so is k;

factorization lifting: if $p(k) = f' \circ f$ for some arrows f, f' of $\mathcal{B}$, then there exists a unique pair g, g' of arrows of $\mathcal{E}$ such that $k = g' \circ g$, $p(g) = f$ and $p(g') = f'$.

It is immediate to check that both discrete fibrations and discrete opfibrations are particular cases of discrete Conduché fibrations. It turns out that, when p is a discrete Conduché fibration, the fiber map ∂p has the structure of a pseudofunctor

$$\partial p : \mathcal{B} \longrightarrow \mathbf{DiscDist},$$

where **DiscDist** is the bicategory of sets and discrete distributors. A discrete distributor $f : A \nrightarrow B$ is just a map assigning an arbitrary set $f(a, b)$ to every pair $(a, b) \in A \times B$. Composition of discrete distributors $f : A \nrightarrow B$, $g : B \nrightarrow C$ is defined by

$$(g \circ f)(a, c) := \bigcup_{b \in B} f(a, b) \times g(b, c).$$

Because of the presence of the cartesian product, composition is only associative modulo a (natural) bijection, which is why **DiscDist** is a bicategory, not a plain category. There is a faithful embedding $j : \mathbf{Set}^{\mathrm{op}} \to \mathbf{DiscDist}$ (resp. $i : \mathbf{Set} \to \mathbf{DiscDist}$) mapping every function to its opgraph (resp. graph). It turns out that a discrete Conduché fibration p is a discrete (op)fibration precisely when ∂p factors through j (resp. i).[1]

[1] A similar situation holds for Grothendieck fibrations, replacing **Set** and **DiscDist** with **Cat** and **Dist**, respectively.

What happens if we weaken the factorization lifting property to mere existence? It turns out that this has been studied by Niefield [Nie04], albeit in a slightly different context, and with a different focus, than the one concerning us: Niefield worked in a bicategorical framework and was interested in *lax* functors, whereas we will concentrate on plain functors. Nevertheless, there is a well-behaved notion of "weak discrete Conduché fibration", which we call more concisely *Niefield fibration* (even though the following definition is not explicitly given in [Nie04]):

Definition 14 (Niefield fibration) *Let $\mathcal{B}$ be a small category. A* Niefield fibration *on $\mathcal{B}$ is a functor $p : \mathcal{E} \to \mathcal{B}$, with $\mathcal{E}$ an arbitrary small category, verifying:*

faithfulness: *p is faithful;*

identity lifting: *if $p(k)$ is an identity, then so is k;*

factorization lifting: *if $p(k) = f' \circ f$ for some arrows f, f' of $\mathcal{B}$, then there exists a (not necessarily unique!) pair g, g' of arrows of $\mathcal{E}$ such that $k = g' \circ g$, $p(g) = f$ and $p(g') = f'$.*

Let $p_1 : \mathcal{E}_1 \to \mathcal{B}$ and $p_2 : \mathcal{E}_2 \to \mathcal{B}$ be Niefield fibrations. A relational morphism *from p_1 to p_2 is a relation $R \subseteq \mathcal{E}_1 \times \mathcal{E}_2$ on the objects of $\mathcal{E}_1, \mathcal{E}_2$ such that:*

- *$(e_1, e_2) \in R$ implies $p_1(e_1) = p_2(e_2)$;*
- *for every arrow $f : b \to b'$ of $\mathcal{B}$ and every $e_1 \in \mathcal{E}_1$ such that $p_1(e_1) = b$ and $e_2' \in \mathcal{E}_2$ such that $p_2(e_2') = b'$, the following conditions are equivalent:*
  - *there exists $g_2 : e_2 \to e_2'$ such that $p_2(g_2) = f$ and $(e_1, e_2) \in R$;*
  - *there exists $g_1 : e_1 \to e_1'$ such that $p_1(g_1) = f$ and $(e_1', e_2') \in R$.*

Note that discrete Conduché fibrations are necessarily faithful. Here, the lack of uniqueness forces us to require faithfulness explicitly.

If p is a Niefield fibration on $\mathcal{B}$, then the fiber map of p may be turned into a functor

$$\partial p : \mathcal{B} \longrightarrow \mathbf{Rel},$$

where **Rel** is the category of sets and relations. It is important to stress that ∂p is a plain 1-functor, not a lax functor into **Rel** seen as a 2-poset, which is the situation considered more commonly in the literature, *e.g.* in [Nie04].

Niefield fibrations are a robust framework for speaking of mere existence of liftings and oplifitings. Consider the subcategory **EntRel** of **Rel** whose morphisms are *entire relations*, *i.e.*, relations $R \subseteq A \times B$ such that, for all $a \in A$, there is $b \in B$ such that $(a, b) \in R$. Then, (op)liftings exist (merely, not uniquely or in an otherwise canonical way) for a Niefield fibration p precisely when ∂p factors via the inclusion $\mathbf{EntRel}^{\mathrm{op}} \hookrightarrow \mathbf{Rel}$ (resp. $\mathbf{EntRel} \hookrightarrow \mathbf{Rel}$).

Summing up, just as (discrete) fibrations are (discrete) Conduché fibrations satisfying a lifting property, the fibrations we were looking for, which we may call *weak discrete fibrations*, or *wd-fibrations*, are Niefield fibrations with an additional lifting property, except that, this time, nothing is asked of liftings other than their existence. Curiously, the lifting properties of (discrete) fibrations are

so strong that they *imply* the factorization lifting property of (discrete) Conduché fibration, whereas mere existence is in some sense too weak to "have a life of its own", and must be asked together with a weak factorization lifting property (and faithfulness).

Although satisfactory (and the next section will give more reasons to be satisfied), the notion of Niefield fibration is of no use *per se* in our approach: for us, type systems are morphisms of 2-operads, not functors. So we need to boost everything up to the 2-operadic level, obtaining the formal definition of "type system" adopted in this thesis:

Definition 15 (type system) *We say that a morphism of 2-operads $\mathbf{p} : \mathcal{E} \to \mathcal{B}$ is a* type system *if, for all objects $e_1, \dots, e_n, e$ of $\mathcal{E}$, the functor $\mathbf{p}_{e_1,\dots,e_n;e}$ is a Niefield fibration.*

A relational morphism *R between two type systems $p_1 : \mathcal{E}_1 \to \mathcal{B}$ and $p_2 : \mathcal{E}_2 \to \mathcal{B}$ is*

- *a relation $R_0 \subseteq \mathcal{E}_1 \times \mathcal{E}_2$ between the objects of $\mathcal{E}_1$ and the objects of $\mathcal{E}_2$;*
- *for all objects $\Gamma_1 = e_1^1, \dots, e_1^n$ and e_1 of $\mathcal{E}_1$ and for all objects $\Gamma_2 = e_2^1, \dots, e_2^n$ and e_2 of $\mathcal{E}_2$, such that for all $1 \leq i \leq n, (e_1^i, e_2^i) \in R_0$ and $(e_1, e_2) \in R_0$, a relational morphism $R^{\Gamma_1;e_1}_{\Gamma_2;e_2}$ between the Niefield fibrations $(\mathbf{p}_1)_{\Gamma_1;e_1}$ and $(\mathbf{p}_2)_{\Gamma_2;e_2}$.*

Type systems over a 2-operad $\mathcal{B}$ and relational morphisms between them form a category, which we denote $\mathbf{TypeSys}(\mathcal{B})$.

In terms of programming languages, faithfulness, identity lifting and factorization lifting correspond to the following properties:

- given a reduction $\rho : M \to^* M'$ and type derivations δ of M and δ' of M', there is at most one typed reduction $\psi : \delta \to^* \delta'$ typing ρ (there may exist one such reduction for every couple of type derivations of M, M');
- empty reductions are never typed by non-empty reductions;
- reductions are typed "modularly": if a decomposable reduction is typed, then so are its components.

These seem to be reasonable requirements to ask of a type system. By the way, most common type systems do not even come with an explicit notion of "typing a reduction", so it does not make sense to ask whether they comply with the above restrictions.

Before we move on, a side remark. The above notion of type system may look somehow wrong from the operadic viewpoint: it is a naive pointwise notion, the weak factorization lifting property is asked of horizontal composition of 2-arrows only, nothing is asked of operadic composition. There is a well-behaved (in the sense of the next section) notion of "operadic" Niefield fibration in which one also asks

- the weak factorization lifting property for vertical (*i.e.* operadic) composition of 2-arrows;

- the *unique* factorization lifting property (*i.e.*, that of discrete Conduché fibrations) for composition of multimorphisms.

However, this notion (which corresponds to asking that the morphism of Theorem 17 below be pseudo, rather than lax) turns out to be too strong for practical purposes: the factorization lifting property at the level of multimorphisms, be it unique or not, implies that every type derivation of $t\{u/x\}$ decomposes (perhaps uniquely) in a type derivation for t and a type derivation for u; in particular, derivations necessarily type every subterm of their subject. While true in certain type systems (*e.g.* the systems of the λ-cube), this is definitely false in some intersection types systems, and for good reasons: in a "strict" (intersections only on the left of arrows) system characterizing head normalization, one must be able to type $x\Omega$ without typing Ω.

2.2.4 An operadic Grothendieck construction

We motivated our search for an adequate notion of fibration by claiming that any such notion should come with a way of endowing the fiber map with suitable structure. For instance, we said that a discrete fibration p on $\mathcal{B}$ induces a presheaf $\partial p : \mathcal{B}^{\mathrm{op}} \to \mathbf{Set}$. Actually, we were telling only part of the story; the full story is that the correspondence should also go the other way around: *any* presheaf $F : \mathcal{B}^{\mathrm{op}} \to \mathbf{Set}$ induces a discrete fibration $\int F : \mathcal{E}(F) \to \mathcal{B}$, where $\mathcal{E}(F)$ is the so-called *category of elements* of the presheaf F. This is known as (the discrete case of) the *Grothendieck construction*. Moreover, the two functors $\int(-)$ and $\partial(-)$ form an equivalence between the category of discrete fibrations on $\mathcal{B}$ (with suitable morphisms) and the presheaf category $\mathbf{Set}^{\mathcal{B}^{\mathrm{op}}}$.

When we said that the notion of Niefield fibration is robust, we meant in particular that a version of the Grothendieck construction is available for it: a Niefield fibration on $\mathcal{B}$ is the same thing as a presheaf $\mathcal{B} \to \mathbf{Rel}$. We will present directly the construction at the 2-operadic level.

It is known that the equivalence between discrete fibrations and presheaves described above may be seen as a particular case of the "yoga" of classifying objects. In a given category $\mathcal{C}$, one says that arrows of a certain kind $\mathcal{K}$ are *classified* by an object c endowed with a special arrow $u : c_* \to c$, if every arrow $p : e \to b$ of kind $\mathcal{K}$ arises as the pullback of u along an arrow $f : b \to c$ (usually referred to as "change of base"):

$$\begin{array}{ccc} e & \longrightarrow & c_* \\ {\scriptstyle p}\downarrow & \lrcorner & \downarrow{\scriptstyle u} \\ b & \xrightarrow[f]{} & c \end{array}$$

Let us list a few examples:

- the most famous case is perhaps that of a *subobject classifier*, in which $\mathcal{K}$ is the class of monomorphisms. When $\mathcal{C} = \mathbf{Set}$, $\mathcal{K}$ becomes the class of injections, $c = \{0,1\}$ and $u : \{*\} \to \{0,1\}$ is the map picking the truth value 1.

- If $\mathcal{K}$ = discrete fibrations, then $\mathcal{C} = \mathbf{Cat}$, $c = \mathbf{Set}^{\mathrm{op}}$ and $u : \mathbf{Set}_*^{\mathrm{op}} \to \mathbf{Set}^{\mathrm{op}}$ is the forgetful functor from the category of pointed sets. In particular, given a presheaf $f : \mathcal{B}^{\mathrm{op}} \to \mathbf{Set}$, the discrete Grothendieck construction builds a discrete fibration $\int f$ simply by computing the pullback of $u : \mathbf{Set}_*^{\mathrm{op}} \to \mathbf{Set}^{\mathrm{op}}$ along f^{op}, so the category of elements of f is $\mathcal{E}(f) = \mathcal{B} \times_{f^{\mathrm{op}}, u} \mathbf{Set}_*^{\mathrm{op}}$.

- For $\mathcal{K}$ = Grothendieck fibrations, a similar picture holds with **Cat** in place of **Set**, and $\mathcal{C}$ is the category of 2-categories and pseudofunctors.

- In the case $\mathcal{K}$ = Niefield fibrations, we are back to $\mathcal{C} = \mathbf{Cat}$, the classifying object is **Rel**, and $u : \mathbf{Rel}_* \to \mathbf{Rel}$ is the forgetful functor from pointed sets and pointed relations.

What we do below essentially amounts to finding the classifying object for type systems, which turns out to live in **BiOp**, the category of bioperads.

Let $\mathfrak{Rel}$ be the following (large) bioperad:

- objects are small categories;

- multimorphisms $X_1 \dots X_n \nrightarrow Y$ are relational distributors, that is functors
$$X_1 \times \cdots \times X_n \times Y^{\mathrm{op}} \to \mathbf{Rel}.$$

- composition of multimorphisms is defined as the composition of distributors: given
$$G : Y_1 \dots Y_m \nrightarrow Z \quad \text{and} \quad F : X_1 \dots X_n \nrightarrow Y_i,$$
their composite $G \circ^i F$ is defined as the functor $Y_1 \times \cdots \times Y_{i-1} \times (X_1 \times \cdots \times X_n) \times Y_{i+1} \times \cdots \times Y_m \times Z^{\mathrm{op}} \to \mathbf{Rel}$
$$(y_1, \dots, y_{i-1}, x_1, \dots, x_n, y_{i+1}, \dots, y_m; z) \mapsto$$
$$\int^{y_i \in Y_i} G(y_1, \dots, y_i, \dots, y_m; z) \times F(x_1, \dots, x_n; y_i)$$
where the integral sign is the standard notation for a coend (it has nothing to do with the Grothendieck construction). Composition in $\mathfrak{Rel}$ is associative only modulo isomorphism.

- 2-arrows $\theta : F \Rightarrow G : X_1 \dots X_n \to Y$ are natural transformations of the underlying functors: a family of relations indexed by $X_1 \times \cdots \times X_n \times Y^{\mathrm{op}}$:
$$\forall (x_1, \dots, x_n, y) \in X_1 \times \cdots \times X_n \times Y^{\mathrm{op}},$$
$$\theta_{x_1, \dots, x_n, y} \subseteq F(x_1, \dots, x_n, y) \times G(x_1, \dots, x_n, y).$$
satisfying naturality conditions.

Let now $\mathfrak{Rel}_*$ be the following (large) bioperad:

- objects are small pointed categories, that is couples (X, x) where X is a small category and $x \in X$;

- multimorphisms $(X_1, x_1) \dots (X_n, x_n) \nrightarrow (Y, y)$ are pointed relational distributors (F, f), that is functors

$$F : X_1 \times \cdots \times X_n \times Y \to \mathbf{Rel}.$$

together with a point $\xi \in F(x_1, \dots, x_n, y)$.

- composition of multimorphisms is defined as the composition of distributors: given

$$(G, v) : (Y_1, y_1) \dots (Y_m, y_m) \nrightarrow (Z, z),$$
$$(F, \xi) : (X_1, x_1) \dots (X_n, x_n) \nrightarrow (Y_i, y_i),$$

the composite $(G, v) \circ^i (F, \xi)$ is defined as the pair $(G \circ_i F, \zeta)$, where ζ is the element of $(G \circ^i F)(y_1, \dots, y_{i-1}, x_1, \dots, x_n, y_{i+1}, \dots, y_m; z)$ canonically corresponding to (v, ξ);

- 2-arrows $\theta : (F, \xi) \Rightarrow (G, v) : (X_1, x_1) \dots (X_n, x_n) \to (Y, y)$ are natural transformations of the underlying functors such that, moreover,

$$(\xi, v) \in \theta_{x_1, \dots, x_n, y}.$$

There is an obvious forgetful functor $U : \mathfrak{Rel}_* \to \mathfrak{Rel}$.

A *lax natural transformation* between two lax morphisms $\theta : F \Rightarrow G : \mathcal{B} \to \mathfrak{Rel}$ is defined as follows:

- for each b in $\mathcal{B}$ a distributor $\theta_b : Fb \nrightarrow Gb$;
- for each $f : b_1 \dots b_n \to b$ in $\mathcal{B}$, a 2-arrow $\theta_f : Gf \circ (\theta_{b_1}, \dots, \theta_{b_n}) \Rightarrow \theta_b \circ Ff$, that is a family of relations indexed by the objects of $Fb_1 \times \cdots \times Fb_n \times Gb^{\mathrm{op}}$,

$$\forall (x_1, \dots, x_n, y) \in Fb_1 \times \cdots \times Fb_n \times Gb^{\mathrm{op}},$$
$$(\theta_f)_{x_1, \dots, x_n, y} \subseteq (Gf \circ (\theta_{b_1}, \dots, \theta_{b_n}))(x_1, \dots, x_n; y) \times (\theta_b \circ Ff)(x_1, \dots, x_n; y),$$

that satisfy naturality conditions.

We say that a lax natural transformation is *relational* if, for every object b, the distributor θ_b is a relation, that is, it is valued in a subsingleton.

Theorem 17 *Let $\mathcal{B}$ be a small 2-operad. The category* **TypeSys**$(\mathcal{B})$ *is equivalent to the category $\mathfrak{Rel}^{\mathcal{B}}$ of lax morphisms $\mathcal{B} \to \mathfrak{Rel}$ and relational lax natural transformations.*

Proof. $\int : \mathfrak{Rel}^{\mathcal{B}} \to$ **TypeSys**$(\mathcal{B})$ is defined by:

- given a lax functor $F : \mathcal{B} \to \mathfrak{Rel}$, $\int F$ is the pullback of the forgetful functor $U : \mathfrak{Rel}_* \to \mathfrak{Rel}$ along F. More explicitly, we denote by $\mathcal{E}(F)$ the pullback category:
  - the objects are pairs (b, x), where b is an object of $\mathcal{B}$ and $x \in Fb$;
  - a multimorphism $(b_1, x_1) \dots (b_n, x_n) \to (b', x')$ is a pair (f, p) where $f : b_1 \dots b_n \to b'$ is a multimorphism in $\mathcal{B}$ and $p \in Ff(x_1, \dots, x_n; x')$;

– given

$$(g,q) : (b_1,y_1)\dots(b_m,y_m) \to (c,z),$$
$$(f,p) : (a_1,x_1)\dots(a_n,x_n) \to (b_i,y_i),$$

the composite $(g,q) \circ^i (f,p)$ is the pair $(g \circ^i f, (q,p))$.

– a 2-arrow $\theta : (f,p) \Rightarrow (g,q) : (a_1,x_1)\dots(a_n,x_n) \to (b,y)$ is a family of relations indexed by $Fa_1 \times \cdots \times Fa_n \times Fb$:

$$\forall(\alpha_1,\dots,\alpha_n,\beta) \in Fa_1 \times \cdots \times Fa_n \times Fb,$$

$$\theta_{\alpha_1,\dots,\alpha_n,\beta} \subseteq Ff(\alpha_1,\dots,\alpha_n,\beta) \times Fg(\alpha_1,\dots,\alpha_n,\beta).$$

such that

$$(p,q) \in \theta_{x_1,\dots,x_n,y}.$$

$\int F$ is just the first projection. It is not hard to check that it is a type system.

- Given a relational lax natural transformation $\theta : F \Rightarrow G$, we define the relation $\int \theta$ from the objects of $\mathcal{E}(F)$ to the objects of $\mathcal{E}(G)$ by $((b,x),(b',y)) \in \int \theta$ iff $b = b'$ and $(x,y) \in \theta_b$. For every list $\Gamma = (a_1,x_1),\dots,(a_n,x_n)$ of objects of $\mathcal{E}(F)$ and object (b,y) of $\mathcal{E}(F)$, and list $\Gamma' = (a'_1,x'_1),\dots,(a'_n,x'_n)$ of objects of $\mathcal{E}(G)$ and object (b',y') of $\mathcal{E}(G)$, the relation $(\int \theta)^{\Gamma;(b,y)}_{\Gamma';(b',y')}$ is empty unless $a_1 = a'_1,\dots,a'_n = a_n$ and $b = b'$, in which case, given

$$(f,p) : (a_1,x_1),\dots,(a_n,x_n) \to (b,y)$$
$$(g,q) : (a_1,x'_1),\dots,(a_n,x'_n) \to (b,y')$$

we have $((f,p),(g,q)) \in (\int \theta)^{\Gamma;(b,y)}_{\Gamma';(b',y')}$ just if $(x_1,x'_1) \in \theta_{a_1},\dots,(x_n,x'_n) \in \theta_{a_n}, (y,y') \in \theta_b$. One may check that $\int \theta$ is a relational morphism.

$\partial : \mathbf{TypeSys}(\mathcal{B}) \to \mathfrak{Rel}^{\mathcal{B}}$ is defined by:

- given a type system $\mathbf{p} : \mathcal{E} \to \mathcal{B}$, we set, for b an object of $\mathcal{B}$,
  – $\partial\mathbf{p}(b) := p^{-1}(b)$, *i.e.*, the subcategory of $\mathcal{E}_1$ whose objects are sent to b and whose morphisms are sent to id_b;
  – for $f : b_1 \dots b_n \to b$ in $\mathcal{B}$, $\partial\mathbf{p}(f)$ is the distributor defined by:

$$\forall(e_1,\dots,e_n,e) \in \mathbf{p}^{-1}(b_1) \times \cdots \times \mathbf{p}^{-1}(b_n) \times \mathbf{p}^{-1}(b)^{\mathrm{op}},$$

$$\partial\mathbf{p}(f)(e_1,\dots,e_n;e) := \{g : e_1,\dots,e_n \to e \mid \mathbf{p}(g) = f\}$$

and, for all $1 \leq i \leq n$, given $k_i : e_i \to e_i'$ arrows of $\mathbf{p}^{-1}(b_i)$ and $k : e' \to e$ an arrow of $\mathbf{p}^{-1}(b)$, respectively,

$$\partial\mathbf{p}(f)(k_1, \ldots, k_n; k) := \{(g, g') \mid g = k \circ g' \circ (k_1, \ldots, k_n)\};$$

– for $\theta : f \Rightarrow f' : b_1 \ldots b_n \to b$ in $\mathcal{B}$, $\partial\mathbf{p}(\theta)_{e_1,\ldots,e_n;e}$ is the relation

$$\{(g, g') \in \partial\mathbf{p}(f)(\bar{e}; e) \times \partial\mathbf{p}(f')(\bar{e}; e) \mid \exists \rho : g \Rightarrow g', \mathbf{p}(\rho) = \theta\}.$$

This defines a lax functor $\partial p : \mathcal{B} \to \mathfrak{Rel}$.

- Given a relational morphism R between type systems $\mathbf{p}_1 : \mathcal{E}_1 \to \mathcal{B}$ and $\mathbf{p}_2 : \mathcal{E}_2 \to \mathcal{B}$,
  - for b an object of $\mathcal{B}$, ∂R_b is R restricted to $\mathbf{p}_1^{-1}(b) \times \mathbf{p}_2^{-1}(b)$, *i.e.*, the relations R_0 and $R^{e_1,e_1'}_{e_2,e_2'}$ (with $e_1, e_1' \in \mathbf{p}_1^{-1}(b)$ and $e_2, e_2' \in \mathbf{p}_2^{-1}(b)$) induce a distributor $\partial\mathbf{p}_1(b) \nrightarrow \partial\mathbf{p}_2(b)$ which is a relation (*i.e.*, valued in a subsingleton), and we take this to be ∂R_b.
  - with the above definition, given $f : b_1, \ldots, b_n \to b$ in $\mathcal{B}$, we have

$$\partial R_b \circ \partial\mathbf{p}_1(f) \cong \{g_1 : e_1^1, \ldots, e_1^n \to e_1 \mid \mathbf{p}_1(g_1) = f\},$$
$$\partial\mathbf{p}_2(f) \circ (\partial R_{b_1}, \ldots, \partial R_{b_n}) \cong \{g_2' : e_2^1, \ldots, e_2^n \to e_2 \mid \mathbf{p}_2(g_2') = f\}.$$

  The family of relations $(\partial R_f)_{e_1^1,\ldots,e_1^n;e_2}$ is then defined to contain all pairs (g_2', g_1) such that there exists g_1' and a 2-arrow $g_1 \Rightarrow g_1'$ such that $(g_1', g_2') \in R^{\bar{e}_1;e_1}_{\bar{e}_2,e_2}$ (which, by definition of relational morphism of Niefield fibrations, is equivalent to the existence of g_2 and a two arrow $g_2 \Rightarrow g_2'$ such that $(g_1, g_2) \in R^{\bar{e}_1;e_1}_{\bar{e}_2,e_2}$).

This defines ∂R as a relational lax natural transformation between $\partial\mathbf{p}_1$ and $\partial\mathbf{p}_2$.

The fact that $\int(-)$ and $\partial(-)$ form an equivalence of categories follows from elementary calculations, applying the above definitions. □

2.3 Intersection Types from Polyadic Approximations

2.3.1 The approximation presheaf

Technicalities aside, Theorem 17 is important because it gives us an alternative viewpoint on what a type system is. Fix a monochromatic 2-operad $\mathcal{L}$ presenting an untyped programming language (the restriction to one color is not necessary, it is just a simplifying assumption). As long as we accept Definition 15, a type system on $\mathcal{L}$ is:

- a morphism $\mathbf{p} : \mathcal{E} \to \mathcal{L}$ such that each functor $\mathbf{p}_{\Gamma;A}$ is a Niefield fibration (this is Melliès and Zeilberger's original point of view);

- a lax morphism $\mathbf{F} : \mathcal{L} \to \mathfrak{Rel}$.

The Grothendieck construction of Theorem 17 tells us that the two viewpoints are equivalent. The first viewpoint is the most natural:

- the types are the colors of $\mathcal{E}$;
- a subtyping relation $A <: B$ holds if the identity program id_* is in the image of $\mathbf{p}_{A;B}$;
- given types Γ, A, the multimorphisms of $\mathcal{E}(\Gamma, A)$ are derivations of judgments of the form $\Gamma \vdash t : A$, and the 2-arrows are typed computations relating them;
- $\mathbf{p}$ sends a derivation to its subject (t in the above notation) and a typed computation to the underlying computation;
- a program t is typable when it is in the image of $\mathbf{p}_{\Gamma;A}$ for some types Γ, A;
- subject reduction/expansion is oplifting/lifting of computations.

By contrast, in the presheaf viewpoint:

- the types are the objects of $\mathbf{F}(*)$, where $*$ is the only color of $\mathcal{L}$;
- a subtyping relation $A <: B$ holds if there is an arrow $A \to B$ in $\mathbf{F}(*)$;
- given a program t and types Γ, A, $\mathbf{F}(t)(\Gamma; A)$ is the set of derivations of $\Gamma \vdash t : A$;
- given a computation $\rho : t \to^* t'$ and types Γ, A, $\mathbf{F}(\rho)_{\Gamma;A} \subseteq \mathbf{F}(t)(\Gamma; A) \times \mathbf{F}(t')(\Gamma; A)$ relates derivations typing t with derivations typing t' along ρ, *i.e.*, it tells all possible ways in which ρ acts on a derivation typing t to yield a derivation typing t';
- a program t is typable when $\mathbf{F}(t)(\Gamma; A) \neq \emptyset$ for some types Γ, A;
- subject reduction (resp. expansion) corresponds to the fact that, for every computation ρ and types Γ, A, the relation $\mathbf{F}(\rho)_{\Gamma;A}$ is entire (resp. op-entire, *i.e.*, surjective).

It is thanks to this alternative view of type systems as (operadic, relational, lax) presheaves that polyadic approximations come back to center stage.

In what follows, we fix an arbitrary suboperad $\mathcal{D} \hookrightarrow \mathbf{Poly}_{\mathsf{c}}$, the 2-operad of simply-typed polyadic cartesian terms, introduced in Fig. 1.9.

Definition 16 (category of types) *We define the following sets, seen as discrete categories:*

$$\mathcal{T}_{\mathsf{l}}[\mathcal{D}] := \{A \mid \textit{the polyadic type } A \textit{ is a color of } \mathcal{D}\};$$
$$\mathcal{T}_{\mathsf{c}}[\mathcal{D}] := \{\langle A_1, \ldots, A_n \rangle \mid A_i \in \mathcal{T}_{\mathsf{l}}[\mathcal{D}]\}.$$

Definition 17 (approximation presheaf) *We define a lax morphism of bioperads*

$$\mathrm{Apx}[\mathcal{D}] : \Lambda_! \longrightarrow \mathfrak{Rel}$$

as follows:

- *on objects,* $\mathrm{Apx}[\mathcal{D}](\mathsf{I}) := \mathcal{T}_{\mathsf{I}}[\mathcal{D}]$ *and* $\mathrm{Apx}[\mathcal{D}](\mathsf{c}) := \mathcal{T}_{\mathsf{c}}[\mathcal{D}]$*;*
- *given* $T \in \Lambda_!(\mathsf{c}^m, \mathsf{I}^n; \mathsf{s})$ *with* $\mathsf{s} \in \{\mathsf{I}, \mathsf{c}\}$*, we must define a functor* $\mathrm{Apx}[\mathcal{D}](T) : \mathcal{T}_{\mathsf{c}}[\mathcal{D}]^m \times \mathcal{T}_{\mathsf{I}}[\mathcal{D}]^n \times \mathcal{T}_{\mathsf{s}}[\mathcal{D}]^{\mathrm{op}} \to \mathbf{Rel}$*; since the source categories are discrete, this is just a map assigning a set to each element of* $\mathcal{T}_{\mathsf{c}}[\mathcal{D}]^m \times \mathcal{T}_{\mathsf{I}}[\mathcal{D}]^n \times \mathcal{T}_{\mathsf{s}}[\mathcal{D}]$*. Let* $\Theta \in \mathcal{T}_{\mathsf{c}}[\mathcal{D}]^m$*, i.e.,*

  $$\Theta = \langle B^1_1, \dots, B^1_{k_1} \rangle, \dots, \langle B^m_1, \dots, B^m_{k_m} \rangle;$$

  we define $\overline{\Theta}$ *to be the polyadic context containing exactly the judgments* $x^i_j : B^i_j$ *for all* $1 \le i \le m$*,* $1 \le j \le k_i$*. Then, for* $\Gamma \in \mathcal{T}_{\mathsf{I}}[\mathcal{D}]^n$ *and* $A \in \mathcal{T}_{\mathsf{s}}[\mathcal{D}]$*, we set*

  $$\mathrm{Apx}[\mathcal{D}](T)(\Theta, \Gamma; A) := \left\{ \delta \in \mathcal{D}(\overline{\Theta}, \Gamma; A) \mid \Xi \vdash \delta^- \sqsubset T \right\},$$

  where Ξ *consists of exactly* $x^i_j \sqsubset x^i$ *for all* $1 \le i \le m$*,* $1 \le j \le k_i$*.*

- *given* $T, T' \in \Lambda_!(\mathsf{c}^m, \mathsf{I}^n; \mathsf{s})$ *with* $\mathsf{s} \in \{\mathsf{I}, \mathsf{c}\}$ *and* $\varphi : T \to^* T'$*,* $\mathrm{Apx}[\mathcal{D}](\varphi)$ *must be a natural transformation from* $\mathrm{Apx}[\mathcal{D}](T)$ *to* $\mathrm{Apx}[\mathcal{D}](T')$*; again, since the source categories of these distributors are discrete, this is just a family of relations indexed by* $\mathcal{T}_{\mathsf{c}}[\mathcal{D}]^m \times \mathcal{T}_{\mathsf{I}}[\mathcal{D}]^n \times \mathcal{T}_{\mathsf{s}}[\mathcal{D}]$*; we define it as follows:*

  $$\mathrm{Apx}[\mathcal{D}](\varphi)_{\Theta,\Gamma;A} := \left\{ (\delta, \delta') \in \mathrm{Apx}[\mathcal{D}](T)(\overline{\Theta}, \Gamma, A) \times \mathrm{Apx}[\mathcal{D}](T')(\overline{\Theta}, \Gamma, A) \mid \exists \tau : \delta \to^* \delta' \text{ in } \mathcal{D}(\overline{\Theta}, \Gamma; A) \text{ s.t. } \Xi \vdash \tau^- \sqsubset \varphi \right\},$$

  with $\overline{\Theta}$ *and* Ξ *defined as above. So,* $(\delta, \delta') \in \mathrm{Apx}[\mathcal{D}](\varphi)_{\Theta,\Gamma;A}$ *if these are related by a typed reduction approximating* φ*.*

Observe that $\mathrm{Apx}[\mathcal{D}]$ is only a lax morphism (not pseudo): consider a closed term $!T : \mathsf{c}$ and a term $T' : \mathsf{I}$ such that $\mathrm{fv}(T') = \{x\}$ (so $T' \circ^1 !T = T'\{!T/x\}$), and fix a type A. We have

$$\mathrm{Apx}[\mathcal{D}](T' \circ^1 T)(; A) = \left\{ \varepsilon \in (; \vdash_{\mathcal{D}} A) \mid \varepsilon^- \sqsubset T'\{!T/x\} \right\} =: E,$$

whereas, once composition of distributors is spelled out, we have

$$\Big(\mathrm{Apx}[\mathcal{D}](T') \circ^1 \mathrm{Apx}[\mathcal{D}](T) \Big)(; A) = \bigcup_{\langle \overline{B} \rangle \in \mathcal{T}_{\mathsf{c}}[\mathcal{D}]} D'_{\overline{B}} \times D_{\overline{B}} \quad =: E',$$

where

$$D'_{\overline{B}} := \left\{ \delta' \in (\overline{x} : \overline{B}; \vdash_{\mathcal{D}} A) \mid \overline{x} \sqsubset x \vdash \delta'^- \sqsubset T' \right\},$$
$$D_{\overline{B}} := \left\{ \langle \overline{\delta} \rangle \in (; \vdash_{\mathcal{D}} \langle \overline{B} \rangle) \mid \langle \overline{\delta} \rangle^- \sqsubset !T \right\}.$$

Now, for each $(\delta', \langle \overline{\delta} \rangle) \in E'$, we have $\delta'\{\langle \overline{\delta} \rangle / \overline{x}\}^- = \delta'^-\{\langle \overline{\delta} \rangle^- / \overline{x}\} \sqsubset T'\{!T/x\}$, so $E' \hookrightarrow E$. However, such an injection is not reversible in general because

$\varepsilon^- \sqsubset T\{!T'/x\}$ does not always induce a typed approximation of $!T$: x may be in a box inside T' which is approximated by $\langle\rangle$, meaning that the typing derivation ε does not type T at all (think of $T' = !(xI)$ and $\varepsilon^- = \langle\rangle$).

Note however that the definition of $\mathrm{Apx}[\mathcal{D}]$ does not exploit the full generality of Theorem 17: the categories of types (*i.e.*, the images of the colors of $\Lambda_!$) are always discrete. Introducing non-trivial arrows among polyadic types would lead to non-trivial subtyping relations on intersection types. We leave this possibility for future developments.

Nevertheless, the approximation presheaf is the cornerstone of a very general framework for constructing intersection types systems. Indeed, suppose we have a morphism of operads $\mathbf{G} : \mathcal{L} \to \Lambda_!$, *i.e.*, $\mathcal{L}$ is a programming language admitting a semantic-preserving embedding in intuitionistic linear logic. Then, we have a lax morphism

$$\mathcal{L} \xrightarrow{\mathbf{G}} \Lambda_! \xrightarrow{\mathrm{Apx}[\mathcal{D}]} \mathfrak{Rel}$$

to which we may apply the Grothendieck construction of Theorem 17, which gives a type system for $\mathcal{L}$ in the sense of Melliès and Zeilberger

$$\begin{array}{c} \mathcal{E}(\mathrm{Apx}[\mathcal{D}] \circ \mathbf{G}) \\ \int(\mathrm{Apx}[\mathcal{D}] \circ \mathbf{G}) \Big\downarrow \\ \mathcal{L} \end{array}$$

We claim that this is an intersection types system for $\mathcal{L}$ in the spirit of the correspondence of Sect. 2.1 (the attentive reader will have noticed that the approximation presheaf was defined exactly with that correspondence in mind). The suboperad $\mathcal{D}$ must be seen as an operad of "valid" derivations, *i.e.*, in certain intersection types systems we may want to exclude certain polyadic approximations (for instance the empty approximation $\langle\rangle$). It is a parameter of the construction, along with $\mathbf{G}$. Actually, we will see later that $\Lambda_!$ (and $\mathbf{Poly}_\mathsf{c}$) is also, in a sense, a parameter, *i.e.*, it is not the most general 2-operad for which polyadic approximation presheaves may be defined.

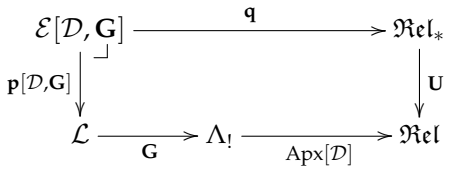
Definition 18 ($(\mathcal{D}, \mathbf{G})$-type system) *Let $\mathcal{D} \hookrightarrow \mathbf{Poly}_\mathsf{c}$ and $\mathbf{G} : \mathcal{L} \to \Lambda_!$. We define the abbreviations*

$$\mathbf{p}[\mathcal{D}, \mathbf{G}] := \textstyle\int(\mathrm{Apx}[\mathcal{D}] \circ \mathbf{G}),$$
$$\mathcal{E}[\mathcal{D}, \mathbf{G}] := \mathcal{E}(\mathrm{Apx}[\mathcal{D}] \circ \mathbf{G}),$$

and say that a multimorphism M of $\mathcal{L}$ is $(\mathcal{D}, \mathbf{G})$-typable if it is in the image of $\mathbf{p}[\mathcal{D}, \mathbf{G}]$ or, equivalently, there are Γ, A such that $(\mathrm{Apx}[\mathcal{D}] \circ \mathbf{G})(M)(\Gamma; A) \neq \emptyset$.

Let us give an idea of what the 2-operad $\mathcal{E}[\mathcal{D}, \mathbf{G}]$ looks like when $\mathcal{L}$ is a monochromatic 2-operad of terms of an untyped calculus (*e.g.* $\mathcal{L} = \Lambda$). By definition of the Grothendieck construction, we know this results from

computing the following strict pullback in **BiOp**:



Strict pullbacks in **BiOp** are computed much in the same way as in **Set**, *i.e.*, the objects, multimorphisms and 2-arrows of $\mathcal{E}[\mathcal{D},\mathbf{G}]$ are pairs of objects, multimorphisms and 2-arrows of $\mathcal{L} \times \mathfrak{Rel}_*$ which are equalized by the above diagram, where $\mathbf{p}[\mathcal{D},\mathbf{G}]$ and $\mathbf{q}$ are just the projections. Therefore:

- an object of $\mathcal{E}[\mathcal{D},\mathbf{G}]$ is a pair (a,b) with $a \in \mathcal{L}$ and $b \in \mathfrak{Rel}_*$, such that $\mathrm{Apx}[\mathcal{D}](\mathbf{G}(a)) = \mathbf{U}(b)$. Since $\mathcal{L}$ is monochromatic, $a = *$. On the other hand, by definition, $b = (X, A)$ where X is a set and $A \in X$. By the pullback condition, we must have $X = \mathbf{U}(b) = \mathrm{Apx}[\mathcal{D}](\mathbf{G}(*)) = \mathcal{T}[\mathcal{D}]$, where $\mathcal{T}[\mathcal{D}]$ is one of $\mathcal{T}_{\mathsf{I}}[\mathcal{D}]$ or $\mathcal{T}_{\mathsf{c}}[\mathcal{D}]$, according to whether $\mathbf{G}(*)$ is I or c. Summing up, an object of $\mathcal{E}[\mathcal{D},\mathbf{G}]$ has the form $(*,(\mathcal{T}[\mathcal{D}],A))$, of which only the type $A \in \mathcal{T}[\mathcal{D}]$ is non-constant, so we conclude that the set of objects of $\mathcal{E}[\mathcal{D},\mathbf{G}]$ is isomorphic to $\mathcal{T}[\mathcal{D}]$, *i.e.*, the set of polyadic simple types declared "valid" by $\mathcal{D}$.

- Given types $A, \Gamma = C_1, \ldots, C_n$, a multimorphism of $\mathcal{E}[\mathcal{D},\mathbf{G}](\Gamma; A)$ is a pair (M, g) of a term $M \in \mathcal{L}(n)$ and a multimorphism $g \in \mathfrak{Rel}_*((\mathcal{T}[\mathcal{D}],C_1),\ldots,(\mathcal{T}[\mathcal{D}],C_n);(\mathcal{T}[\mathcal{D}],A))$, such that $\mathrm{Apx}[\mathcal{D}](\mathbf{G}(M)) = \mathbf{U}(g)$. By definition (*cf.* Sect. 2.2.4), $g = (F,\delta)$, with $F : \mathcal{T}[\mathcal{D}]^n \times \mathcal{T}[\mathcal{D}] \to \mathbf{Rel}$ a functor and $\delta \in F(\Gamma, A)$. By the pullback condition, $F = \mathbf{U}(g) = \mathrm{Apx}[\mathcal{D}](\mathbf{G}(M))$, which means that $\delta \in \mathcal{D}(\Gamma; A)$ such that $\delta^- \sqsubset \mathbf{G}(M)$. Since F is constant, the multimorphisms of $\mathcal{E}[\mathcal{D},\mathbf{G}](\Gamma; A)$ are actually pairs (M,δ) such that $\delta^- \sqsubset \mathbf{G}(M)$; since $\mathbf{p}[\mathcal{D},\mathbf{G}](M,\delta) = M$, the pair (M,δ) must be seen as a derivation typing M, which consists of a "valid" polyadic simply-typed derivation of $\Gamma \vdash t : A$ such that $t \sqsubset \mathbf{G}(M)$.

- A similar analysis shows that, given types A, Γ as above and $(M,\delta),(M',\delta') \in \mathcal{E}[\mathcal{D},\mathbf{G}](\Gamma; A)$, a 2-arrow $(M,\delta) \to (M',\delta')$ is a pair (ψ,τ) such that $\psi : M \to^* M'$ in $\mathcal{L}(n)$, $\tau : \delta \to^* \delta'$ in $\mathcal{D}(\Gamma; A)$ and $\tau^- \sqsubset \mathbf{G}(\psi)$, *i.e.*, $M \to^* M'$ via ψ and there is a polyadic reduction approximating ψ which may be given a "valid" type as a reduction $\delta \to^* \delta'$.

In particular,

M is $(\mathcal{D},\mathbf{G})$-typable iff there exists δ in $\mathcal{D}(\Gamma; A)$ such that $\delta^- \sqsubset \mathbf{G}(M)$,

exactly as in the correspondence described at the end of Sect. 2.1.

2.3.2 Capturing dynamic properties

Most well known intersection type systems for the λ-calculus (or minor reformulations of them) arise from the above construction, *i.e.*, they are isomorphic

to $\mathbf{p}[\mathcal{D}, \mathbf{G}]$ for some choice of $\mathcal{D}$ and $\mathbf{G}$. For instance, $\mathbf{p}[\mathbf{Poly}_{\mathsf{l}}, \mathbf{G}_0]$ (where $\mathbf{Poly}_{\mathsf{l}}$ are linear derivations and $\mathbf{G}_0 : \Lambda_0 \to \Lambda_!$ is Girard's call-by-name encoding defined in Sect. 1.2.2) gives rise to Gardner's system [Gar94] as discussed in Sect. 2.1. Many more examples will be given in Sect. 2.4.2. What is more important, however, is that this abstract setting also allows us to prove a quite general theorem through which the usual properties of intersection type systems may be recovered.

Intersection types are known for their ability to capture dynamic (or runtime) properties of programs, most notably various kinds of termination. We will start by giving a somewhat general definition of what a "dynamic property" may be in our context.

Definition 19 (dynamic property) *Let $\mathcal{L}$ be a 2-operad. A* strong dynamic property *for $\mathcal{L}$ is a set of 2-arrows of $\mathcal{L}$. Given such a set $\mathcal{R}$, we let $\mathcal{S}(\mathcal{R})$ be the set of all multimorphisms M of $\mathcal{L}$ such that there is no sequence $(\psi_i : M_i \to^* M_{i+1})_{i\in\mathbb{N}}$ of non-identity 2-arrows of $\mathcal{R}$ with $M_0 = M$.*

A weak dynamic property *of $\mathcal{L}$ consists of a triple $(\mathcal{R}, \mathcal{N}, \mathrm{Ctxt})$ such that*

- *$\mathcal{R}$ is a set of 2-arrows of $\mathcal{L}$;*
- *$\mathcal{N}$ is a set of multimorphisms of $\mathcal{L}$;*
- *Ctxt is a set of functions on multimorphisms of $\mathcal{L}$.*

We write $\mathsf{C}\{M\}$ for the multimorphism of $\mathcal{L}$ resulting from the application of $\mathsf{C} \in \mathrm{Ctxt}$ to a multimorphism M. Given such a triple, we let $\mathcal{W}(\mathcal{R}, \mathcal{N}, \mathrm{Ctxt})$ be the set of all multimorphisms M of $\mathcal{L}$ such that there exist $\mathsf{C} \in \mathrm{Ctxt}$, $N \in \mathcal{N}$ and $\psi : \mathsf{C}\{M\} \to^* N$ in $\mathcal{R}$.

The intuition between strong and weak dynamic properties is that they express some kind of strong or weak normalization. For what concerns the first, this is clear from the definition: $\mathcal{R}$ represents a notion of reduction and $\mathcal{S}(\mathcal{R})$ is the set of strongly $\mathcal{R}$-normalizing terms (seen as multimorphisms). For what concerns the latter, $\mathcal{R}$ is still a notion of reduction, $\mathcal{N}$ represents a notion of normal form and Ctxt a notion of "legal" contexts, so $\mathcal{W}(\mathcal{R}, \mathcal{N}, \mathrm{Ctxt})$ is the set of terms having a $\mathcal{N}$-form reachable via a reduction in $\mathcal{R}$, modulo an initial manipulation in Ctxt.

Definition 20 (faithful reduction, full expansion) *Let $\mathcal{D} \hookrightarrow \mathbf{Poly}_{\mathsf{c}}$ and $\mathbf{G} : \mathcal{L} \to \Lambda_!$. Let $\mathcal{R}$ be a strong dynamic property for $\mathcal{L}$. We say that the pair $(\mathcal{D}, \mathbf{G})$ is* faithfully reductive *with respect to $\mathcal{R}$ if, for all $\rho \in \mathcal{R}$ and for all types Γ, A*

subject reduction: *$(\mathrm{Apx}[\mathcal{D}] \circ \mathbf{G})(\rho)_{\Gamma;A}$ is entire; equivalently, $\mathbf{p}[\mathcal{D}, \mathbf{G}]$ oplifts every 2-arrow of $\mathcal{R}$;*

faithfulness: *if $(\mathrm{Apx}[\mathcal{D}] \circ \mathbf{G})(\rho)_{\Gamma;A} \neq \emptyset$ and is the identity (i.e., the diagonal relation), then ρ is an identity.*

Let $(\mathcal{R}, \mathcal{N}, \mathrm{Ctxt})$ be a weak dynamic property for $\mathcal{L}$. We say that the pair $(\mathcal{D}, \mathbf{G})$ is fully expansive *if, for all $\rho \in \mathcal{R}$ and for all types Γ, A*

subject expansion: $(\mathrm{Apx}[\mathcal{D}] \circ \mathbf{G})\,(\rho)_{\Gamma;A}$ *is op-entire (i.e., surjective); equivalently,* $\mathbf{p}[\mathcal{D}, \mathbf{G}]$ *lifts every arrow of* $\mathcal{R}$*;*

fullness: *for all* $u \in \mathcal{N}$*,* u *is* $(\mathcal{D}, \mathbf{G})$*-typable and, for all* $\mathsf{C} \in \mathrm{Ctxt}$*, if* t *is* $(\mathcal{D}, \mathbf{G})$*-typable, then so is* $\mathsf{C}\{t\}$*.*

Lemma 18 *Let* $\mathcal{D} \hookrightarrow \mathbf{Poly}_{\mathsf{c}}$ *and* $\mathbf{G} : \mathcal{L} \to \Lambda_!$ *and let* $\mathcal{R}$ *be a strong dynamic property for* $\mathcal{L}$ *with respect to which* $(\mathcal{D}, \mathbf{G})$ *is faithfully reductive. Then, if* $\psi \in \mathcal{R}$ *is not an identity, there is at least one oplifting* (ψ, τ) *of* ψ *with respect to* $\mathbf{p}[\mathcal{D}, \mathbf{G}]$ *such that* τ *is not an identity.*

Proof. Let $\psi : M \Rightarrow M' \in \mathcal{R}$ be a non-identity arrow with M in the image of $\mathbf{p}[\mathcal{D}, \mathbf{G}]$, *i.e.*, there exist Γ, A and $\delta \in \mathcal{D}(\Gamma; A)$ such that $(M, \delta) \in \mathcal{E}[\mathcal{D}, \mathbf{G}](\Gamma; A)$, which means that $\delta \in (\mathrm{Apx}[\mathcal{D}] \circ \mathbf{G})(M)(\Gamma; A)$. Now, by hypothesis, not only $(\mathrm{Apx}[\mathcal{D}] \circ \mathbf{G})(\psi)_{\Gamma;A}$ is entire, which implies the existence of $\delta' \in (\mathrm{Apx}[\mathcal{D}] \circ \mathbf{G})(M')(\Gamma; A)$ such that there is $\tau : \delta \Rightarrow \delta'$, giving an oplifting (ψ, τ) of ψ, but it is not the diagonal, which means that we may choose $\delta' \neq \delta$ and, in particular, $\tau \neq \mathrm{id}_\delta$. □

Lemma 19 *Let* $\mathcal{D} \hookrightarrow \mathbf{Poly}_{\mathsf{c}}$ *and* $\mathbf{G} : \mathcal{L} \to \Lambda_!$*.*

1. *Let* $\mathcal{R}$ *be a strong dynamic property for* $\mathcal{L}$ *and suppose that* $(\mathcal{D}, \mathbf{G})$ *is faithfully reductive with respect to it. Then, for every multimorphism* M *of* $\mathcal{L}$*,* M $(\mathcal{D}, \mathbf{G})$*-typable implies* $M \in \mathcal{S}(\mathcal{R})$*.*
2. *Let* $(\mathcal{R}, \mathcal{N}, \mathrm{Ctxt})$ *be a weak dynamic property for* $\mathcal{L}$ *and suppose that* $(\mathcal{D}, \mathbf{G})$ *is fully expansive with respect to it. Then, for every multimorphism* M *of* $\mathcal{L}$*,* $M \in \mathcal{W}(\mathcal{R}, \mathcal{N}, \mathrm{Ctxt})$ *implies* M $(\mathcal{D}, \mathbf{G})$*-typable.*

Proof. Let us start with point 1. Let M be $(\mathcal{D}, \mathbf{G})$-typable and suppose, for the sake of absurdity, that there is a sequence $(\psi_i : M_i \Rightarrow M_{i+1})_{i \in \mathbb{N}}$ of non-identity 2-arrows of $\mathcal{R}$ with $M_0 = M$. Since M is $(\mathcal{D}, \mathbf{G})$-typable, there exist Γ, A and a derivation δ such that $(M, \delta) \in \mathcal{E}[\mathcal{D}, \mathbf{G}](\Gamma; A)$. In particular, M is the image of (M, δ) via $\mathbf{p}[\mathcal{D}, \mathbf{G}]_{\Gamma;A}$. Now, since $\psi_0 \in \mathcal{R}$, this functor oplifts it by the subject reduction hypothesis, so there is $(\psi_0, \tau_0) : (M, \delta) \Rightarrow (M_1, \delta_1)$ in $\mathcal{E}[\mathcal{D}, \mathbf{G}](\Gamma; A)$. In particular, M_1 too is in the image of $\mathbf{p}[\mathcal{D}, \mathbf{G}]_{\Gamma;A}$. Furthermore, by Lemma 18, we may suppose τ_1 not to be an identity. We may now re-apply the reasoning to M_1 and ψ_1, then to M_2 and ψ_2, and so on, obtaining a sequence $(\tau_i : \delta_i \Rightarrow \delta_{i+1})_{i \in \mathbb{N}}$ of non-identity arrows in $\mathcal{D}(\Gamma; A)$, with $\delta_0 = \delta$. We have therefore shown that δ is not strongly normalizing, contradicting Proposition 5 (remember that $\mathcal{D} \hookrightarrow \mathbf{Poly}_{\mathsf{c}}$).

For what concerns point 2, suppose $M \in \mathcal{W}(\mathcal{R}, \mathcal{N}, \mathrm{Ctxt})$, *i.e.*, there exist $\mathsf{C} \in \mathrm{Ctxt}$, $N \in \mathcal{N}$ and $\psi : \mathsf{C}\{M\} \Rightarrow N$ in $\mathcal{R}$. By fullness, there are types Γ, A and a derivation ε such that $(N, \varepsilon) \in \mathcal{E}[\mathcal{D}, \mathbf{G}](\Gamma; A)$. In particular, N is the image of (N, ε) via $\mathbf{p}[\mathcal{D}, \mathbf{G}]_{\Gamma;A}$. Now, since $\psi \in \mathcal{R}$, this functor lifts it by the

subject expansion hypothesis, so there is $(\psi, \tau) : (\mathsf{C}\{M\}, \delta) \Rightarrow (N, \varepsilon)$, showing in particular that $\mathsf{C}\{M\}$ is $(\mathcal{D}, \mathbf{G})$-typable, so we conclude by fullness. □

Combined, the two parts of the above result immediately give what we may call "the fundamental theorem of intersection types":

Theorem 20 *Let $\mathcal{L}$ be a 2-operad, let $\mathcal{R}$ and $(\mathcal{R}_0, \mathcal{N}, \mathrm{Ctxt})$ be a strong and a weak dynamic property for it such that $\mathcal{S}(\mathcal{R}) \subseteq \mathcal{W}(\mathcal{R}_0, \mathcal{N}, \mathrm{Ctxt})$, and suppose that $(\mathcal{D} \hookrightarrow \mathbf{Poly}_{\mathsf{c}}, \mathbf{G} : \mathcal{L} \to \Lambda_!)$ is both faithfully reductive with respect to the former and fully expansive with respect to the latter. Then, for every multimorphism M of $\mathcal{L}$, the following are equivalent:*

1. *M is $(\mathcal{D}, \mathbf{G})$-typable;*
2. *$M \in \mathcal{S}(\mathcal{R})$;*
3. *$M \in \mathcal{W}(\mathcal{R}_0, \mathcal{N}, \mathrm{Ctxt})$.*

Observe that the proof of the implication (3)⇒(1) of Theorem 20, which is given by point 2 of Lemma 19, is completely standard: we apply subject expansion to typability of normal forms. On the other hand, the proof of (1)⇒(2), which is point 1 of Lemma 19, is an abstraction of an argument of Bucciarelli, Piperno and Salvo [BPS03], who reduced the soundness of Coppo, Dezani and Venneri's intersection types system [CDCV81] to the strong normalization of the simply-typed λ-calculus. In Theorem 20, the soundness of every well-known system of intersection types is, essentially, reduced to the strong normalization of simply-typed polyadic calculi (Proposition 5) which, in turn, reduces to the strong normalization of propositional linear logic. This is interesting because it gives a simple, uniform proof, instead of plenty of *ad hoc* reducibility/logical relation arguments (such as those found in [Kri93]) or, in the non-idempotent case, *ad hoc* combinatorial arguments (*e.g.* [Kfo00, BL13, KV16]).

Theorem 20 also tells us something interesting about intersection types disciplines: they always relate, albeit often in a hidden way, a dynamic property of "universal" flavor (strong normalization) with one of "existential" flavor (weak normalization).

2.4 Applications

2.4.1 A worked out example

We will now present a series of interesting instances of Theorem 20, starting from a detailed example. First, however, let us give a sufficient condition for faithfulness with respect to a strong dynamic property, which will turn out be quite useful:

Lemma 21 *Let $\mathcal{D} \hookrightarrow \mathbf{Poly}_{\mathsf{c}}$, $\mathbf{G} : \mathcal{L} \to \Lambda_!$ and let $\mathcal{R}$ be a strong dynamic property for $\mathcal{L}$ such that $\mathbf{p}[\mathcal{D}, \mathbf{G}]$ oplifts every 2-arrow of $\mathcal{R}$. Suppose furthermore that, for all $\psi \in \mathcal{R}$:*

1. *if* $\mathbf{G}(\psi)$ *is an identity, then so is* ψ*;*
2. *if* δ *is an identity 2-arrow of* $\mathcal{D}$ *such that* $\delta^- \sqsubset \mathbf{G}(\psi)$*, then* $\mathbf{G}(\psi)$ *is an identity.*

Then, $(\mathcal{D}, \mathbf{G})$ *is faithfully reductive with respect to* $\mathcal{R}$*.*

PROOF. Subject reduction holds by hypothesis, so we need to prove faithfulness. Let $\psi \in \mathcal{R}$, let Γ, A be types and suppose $(\mathrm{Apx}[\mathcal{D}] \circ \mathbf{G})(\psi)_{\Gamma;A} \neq \emptyset$ and to be the diagonal. This means that there exists δ such that $\tau : \delta \to^* \delta$ and $\tau^- \sqsubset \mathbf{G}(\psi)$. Now, since polyadic simply-typed terms normalize (Proposition 5), we must have $\tau = \delta$ (the identity), so $\mathbf{G}(\psi)$ is an identity by hypothesis 2, hence ψ is an identity by hypothesis 1, as desired. □

Let us discuss Lemma 21. Its hypothesis (1) says that the embedding $\mathbf{G}$ does not loose too much information, *i.e.*, it does not map non-identity reductions of $\mathcal{R}$ to identity reductions. This is a very mild requirement: in fact, it is true in the most general way (*i.e.*, with $\mathcal{R}$ = all 2-arrows) for all the embeddings we consider. This is because $\Lambda_!$ is usually *finer* than $\mathcal{L}$.

The interesting property is hypothesis (2): it says that we never type an "important" non-identity reduction (*i.e.*, one of $\mathcal{R}$) with the identity reduction. How can this happen? Suppose that ψ is not an identity (so, by (1), $\varphi := \mathbf{G}(\psi)$ is not an identity) and suppose that $\delta^- \sqsubset \varphi$, against hypothesis (2). By functoriality, $t := \delta^-$ is also an identity. By inspecting Fig. 1.11, we see that the only way that $t \sqsubset \varphi$ may be derivable is by using the rules box and theta; more specifically, $\varphi : T \to^* T'$ and the reduction happens "inside" a box of T which is approximated by $\langle\rangle$ in t. So, the intuition behind hypothesis (2) is that we must make sure that the typing derivations of $\mathcal{D}$ do not make "important" redexes disappear by abusing empty approximations.

The above discussion is interesting because it gives the intuition behind faithfulness: not only do we want subject reduction, we also want it to reflect the computation being performed. Indeed, subject reduction may hold because, when $M \to M'$, the derivation δ typing M' is the same as that typing M. Typically, as explained above, this happens because the redex fired in M to obtain M' is not typed by δ, so the modifications induced by the reduction are invisible to the typing. While this is not forbidden in general, it must *not* happen for the reductions of which we are trying to capture termination (*i.e.*, those in $\mathcal{R}$).

Let us give an example, which we will work out in detail. Our goal is to show that the multimorphisms of $\mathcal{E}[\mathbf{Poly}_\mathrm{I}, \mathbf{G}_0]$ are isomorphic to the derivations of Gardner-de Carvalho's non-idempotent intersection type system [Gar94] (Fig. 2.1). Before we dive in, let us point out that, for all types Γ, A and for every reduction ψ of Λ_k, $(\mathrm{Apx}[\mathbf{Poly}_\mathrm{I}] \circ \mathbf{G}_0)(\psi)_{\Gamma;A}$ is bientire, which means that this system will satisfy both subject reduction and subject expansion with respect to *every* reduction. However, it is *faithfully* reductive only with respect to *head* reduction. For instance, let $I := \lambda y.y$, $M := x(II)$ and $M' := xI$, and consider the (non-head) reduction $\psi : M \to M'$. Let $t := x\langle\rangle$ and let δ be the obvious linear derivation of $x : \langle\rangle \multimap \alpha; \vdash t : \alpha$ (referring to

Fig. 1.9, δ is obtained by applying an app rule to the conclusions of a pvar rule and a nullary box rule). Note that $t \sqsubset \mathbf{G}_0(M)$, because $\langle\rangle \sqsubset !T$ for every T. Now, $\mathbf{G}_0(\psi) : x!(\mathbf{G}_0(I)!\mathbf{G}_0(I)) \to^* x!\mathbf{G}_0(I)$, *i.e.*, the reduction happens inside a $!(-)$. Using the box rule of Fig. 1.11, we get $t \sqsubset \mathbf{G}_0(\psi)$. Since $\delta^- = t$, we have $(\delta, \delta) \in \mathrm{Apx}[\mathbf{Poly}_\mathsf{I}, \mathbf{G}_0](\psi)_{\langle\rangle \multimap \alpha;\alpha}$. In fact, δ is the *only* derivation with that type, so the above relation is a non-empty diagonal, giving the desired counterexample to faithfulness.

By contrast, head redexes never appear under a $!(-)$ via $\mathbf{G}_0$, so they may never be "forgotten" by approximations. Hence, by Lemma 21, $(\mathbf{Poly}_\mathsf{I}, \mathbf{G}_0)$ is faithfully reductive for $\mathcal{R} =$ head reductions (we are of course working with $\mathcal{L} = \Lambda_k$). Note that $\mathcal{S}(\mathcal{R})$ is the set of terms whose head reduction terminates (in fact, it is the of term whose head reduction *strongly* terminates, but that makes no difference because head reduction is deterministic).

Let now

$$\begin{aligned} \mathcal{R}_0 &= \text{all reductions}, \\ \mathcal{N} &= \text{head normal forms}, \\ \mathrm{Ctxt} &= \{\mathrm{id}\}. \end{aligned}$$

We have that $\mathcal{W}(\mathcal{R}_0, \mathcal{N}, \mathrm{Ctxt})$ is the set of terms having a head normal form. It is straightforward to see that head normal forms are typable (just assign a type of shape $\langle\rangle \multimap \cdots \multimap \langle\rangle \multimap \alpha$ to the head variable), so $(\mathbf{Poly}_\mathsf{I}, \mathbf{G}_0)$ is fully expansive. It is obviously the case that $\mathcal{S}(\mathcal{R}) \subseteq \mathcal{W}(\mathcal{R}_0, \mathcal{N}, \mathrm{Ctxt})$ (if head reduction terminates for M, then M certainly has a head normal form). Therefore, by Theorem 20, $(\mathbf{Poly}_\mathsf{I}, \mathbf{G}_0)$-typability characterizes having a head normal form and, moreover, we get for free that having a head normal form is the same as saying that head reduction terminates. This latter fact is not immediate: syntactic proofs require a form of standardization.

So, we know we have a type system characterizing head normalization, but what does it look like?[2] Since no restrictions on $\mathbf{Poly}_\mathsf{I}$ are imposed, its set of types is equal to all polyadic simple types. Its derivations are pairs (M, δ) consisting of a λ-term M and a linear polyadic simply-typed derivation δ of $\Theta; \vdash t : A$ such that $t \sqsubset \mathbf{G}_0(M)$. We know that the context Θ is entirely polyadic because λ-terms only have variables of type v, and $\mathbf{G}_0(\mathsf{v}) = \mathsf{c}$. Moreover, we know that $t : \mathsf{I}$ because we are considering the restriction of $\mathbf{G}_0$ to Λ_0, the subcategory of Λ_k in which only terms of type t are considered, and $\mathbf{G}_0(\mathsf{t}) = \mathsf{I}$.

Let $\mathrm{fv}(M) \subseteq \{x^1, \ldots, x^n\}$. If we make explicit the approximation context, *i.e.*, which free polyadic variables of t approximate which free variables of $\mathbf{G}_0(M)$ (which coincide with those of M), we get

$$\overline{x}^1 \sqsubset x^1, \ldots, \overline{x}^n \sqsubset x^n \vdash t \sqsubset \mathbf{G}_0(M).$$

This also means that $\Theta = \overline{x}^1 : \overline{C}^1, \ldots, \overline{x}^n : \overline{C}^n$. Superposing approximation judgments and typing judgments (as we did in Sect. 2.1), we get judgments of

[2] This is a curious consequence of abstraction: we have shown a non-trivial property concerning head reduction without even having *seen* the type system characterizing it!

the form

$$\overline{x}^1 \sqsubset x^1 : \overline{C}^1, \ldots, \overline{x}^n \sqsubset x^n : \overline{C}^n \vdash t \sqsubset \mathbf{G}_0(M) : A,$$

where $\overline{x}^i \sqsubset x^i : \overline{C}^i$ abbreviates $x^i_1 \sqsubset x^i : C^i_1, \ldots, x^i_{k_i} \sqsubset x^i : C^i_{k_i}$. Now, by inspecting Fig. 1.11 and Fig. 1.9, we see that linear approximations and linear derivations are both syntax-directed. Therefore, the structure of M guides the structure of δ, and we have only three possibilities:

$$\frac{}{x_0 \sqsubset x : A; \vdash x_0 \sqsubset \mathbf{G}_0(x) : A}\ \mathsf{pvar}$$

$$\dfrac{\dfrac{\dfrac{}{; a : \vec{A} \vdash a : \vec{A}}\ \mathsf{var} \qquad \Gamma, \overline{x} \sqsubset x : \overline{A}; \vdash t \sqsubset \mathbf{G}_0(M) : B}{\Gamma; a : \vec{A} \vdash t[\langle \overline{x} \rangle := a] \sqsubset \mathbf{G}_0(M)[!x := a] : B}\ \mathsf{let}}{\Gamma; \vdash \lambda a.t[\langle \overline{x} \rangle := a] \sqsubset \mathbf{G}_0(\lambda x.M) : \vec{A} \multimap B}\ \mathsf{lam}$$

$$\dfrac{\Gamma; \vdash t \sqsubset \mathbf{G}_0(M) : \vec{A} \multimap B \qquad \dfrac{\Gamma_1; \vdash u_1 \sqsubset \mathbf{G}_0(N) : A_1 \quad \ldots \quad \Gamma_n; \vdash u_n \sqsubset \mathbf{G}_0(N) : A_n}{\Gamma'_1, \ldots, \Gamma'_n; \vdash \langle u_1, \ldots, u_n \rangle \sqsubset !\mathbf{G}_0(N) : \vec{A}}\ \mathsf{box}}{\Gamma, \Gamma'_1, \ldots, \Gamma'_n; \vdash t\langle u_1, \ldots, u_n \rangle \sqsubset \mathbf{G}_0(MN) : B}\ \mathsf{app}$$

where $\vec{A} = \langle A_1, \ldots, A_n \rangle$. If we only retain the purple decorations, forget the intermediate steps not typing terms of the form $\mathbf{G}_0(-)$ and if, in the context, we write $x : \langle A_1, \ldots, A_n \rangle$ instead of $x_1 \sqsubset x : A_1, \ldots, x_n \sqsubset x : A_n$, we obtain precisely Gardner's system [Gar94]. Actually, we also have a fourth rule allowing to permute types in sequences in the context, resulting from the exchange rule on $\mathbf{Poly}_!$ contexts (this is the perm rule of Fig. 2.1).

2.4.2 Other examples

Head normalization. It is easy to see that, more generally, $(\mathbf{Poly}_\mathsf{p}, \mathbf{G}_0)$ for any p induces a type system characterizing head normalization. The most general, when $\mathsf{p} = \mathsf{c}$, is depicted in Fig. 2.2. This is just a (strict, in the sense of [vB95]) reformulation of the standard system called $D\Omega$ in [Kri93], the variant with Ω of [CDCV81]. The other systems are obtained by discarding one or both of the rules weak and cntr: the former enables basic subtyping (*i.e.*, $A \wedge B \leq A$) and makes the system non-relevant; the latter makes the system idempotent. The rule weak_0 *cannot* be discarded, it is necessary to make the system complete (*e.g.* it is needed to type $\lambda x.y$).

Solvability. The above systems actually characterize solvability. It is a classic result of Wadsworth [Wad71] that solvability and head normalization coincide, so this would be a trivial remark if it were not for the fact that we may prove it *independently* of Wadsworth's result, thus yielding an alternative, type-theoretic proof of his theorem. Indeed, let $\mathcal{R}$ be head reductions, as above, so that $\mathcal{S}(\mathcal{R})$ is the set of all λ-terms having a head normal form, and

Types: $A, B ::= \alpha \mid \langle A_1, \ldots, A_n \rangle \multimap B$
Rules:

$$\frac{}{x : \langle A \rangle \vdash x : A}\ \mathsf{var} \qquad \frac{\Gamma, x : \vec{A} \vdash M : B}{\Gamma \vdash \lambda x.M : \vec{A} \multimap B}\ \mathsf{lam}$$

$$\frac{\Gamma \vdash M : \langle A_1, \ldots, A_n \rangle \multimap B \quad \Delta_1 \vdash N : A_1 \quad \ldots \quad \Delta_n \vdash N : A_n}{\Gamma \cdot \Delta_1 \cdots \Delta_n \vdash MN : B}\ \mathsf{app}$$

$$\frac{\Gamma, x : \langle A_1, \ldots, A_n \rangle \vdash M : C}{\Gamma, x : \langle A_{\sigma(1)}, \ldots, A_{\sigma(n)} \rangle \vdash M : C}\ \mathsf{exch} \qquad \frac{\Gamma \vdash M : C}{\Gamma, x : \langle \rangle \vdash M : C}\ \mathsf{weak}_0 \quad x \notin \Gamma$$

$$\frac{\Gamma, x : \langle B_1, \ldots, B_n \rangle \vdash M : C}{\Gamma, x : \langle B_1, \ldots, B_n, A \rangle \vdash M : C}\ \mathsf{weak} \qquad \frac{\Gamma, x : \langle B_1, \ldots, B_n, A, A \rangle \vdash M : C}{\Gamma, x : \langle B_1, \ldots, B_n, A \rangle \vdash M : C}\ \mathsf{cntr}$$

Figure 2.2: Cartesian intersection type system characterizing head normalization. In the app rule, $\Gamma \cdot \Delta$ is concatenation as in Fig. 2.1. Non-idempotent or relevant variants are obtained by removing the rule cntr or weak, respectively (but not weak_0). Gardner's system (Fig. 2.1) is obtained by removing both.

let

$$\begin{aligned} \mathcal{R}_0 &= \text{all reductions}, \\ \mathcal{N} &= \{I\}, \\ \mathrm{Ctxt} &= \text{applicative contexts}, \end{aligned}$$

where I is the identity λ-term and applicative contexts are of the form $\{\cdot\}N_1 \cdots N_n$. By definition, $\mathcal{W}(\mathcal{R}_0, \mathcal{N}, \mathrm{Ctxt})$ is the set of solvable λ-terms.

Now, it is immediate that $\mathcal{S}(\mathcal{R}) \subseteq \mathcal{W}(\mathcal{R}_0, \mathcal{N}, \mathrm{Ctxt})$ (if a λ-term has a head normal form, it is solvable). We know that any of the systems of Fig. 2.2 is faithfully reductive with respect to $\mathcal{R}$. It is easy to see that it is also fully expansive with respect to $(\mathcal{R}_0, \mathcal{N}, \mathrm{Ctxt})$: we know we have subject expansion, I is obviously typable and it is immediate to see that, if $MN_1 \cdots N_n$ is typable, then so must be M. So Theorem 20 applies, and we have that a term is solvable iff it has a head normal form iff it is typable in one of the systems of Fig. 2.2. The possibility of using intersection type systems to give an alternative proof of Wadsworth's result was already pointed out by Bucciarelli, Kesner and Ronchi Della Rocca [BKR14] (specifically, they used Gardner-de Carvalho's system).

Note that this particular application uses a non-trivial set of functions Ctxt. In the sequel, we will always use $\mathrm{Ctxt} = \{\mathrm{id}\}$ (the identity function), so we will never specify it again, and we will write $\mathcal{W}(\mathcal{R}, \mathcal{N})$ for $\mathcal{W}(\mathcal{R}, \mathcal{N}, \{\mathrm{id}\})$.

Strong normalization. Let now $\mathcal{D} \in \{\mathbf{Poly}_{\mathsf{c}}, \mathbf{Poly}_{\mathsf{a}}\}$ and let $\mathcal{D}_{\mathrm{sn}}$ be its full suboperad on the $\langle\rangle$-free types, *i.e.*, the empty sequence $\langle\rangle$ is not allowed, and consider the pair $(\mathcal{D}_{\mathrm{sn}}, \mathbf{G}_0)$. The resulting type system is obtained from Fig. 2.2 by disallowing the use of $\langle\rangle$ in types, which means that the weak_0 rule must be modified to derive $\Gamma, x : \langle A\rangle \vdash M : C$ from $\Gamma \vdash M : C$ (also, in the affine case, the rule cntr must be dropped). The idempotent system (*i.e.*, with contraction) is just a reformulation of the system originally introduced by Coppo, Dezani and Venneri [CDCV81].

Since the type $\langle\rangle$ is the only way to type the term $\langle\rangle$, empty polyadic approximations are not allowed in $\mathcal{D}_{\mathrm{sn}}$; in particular, whenever $t \sqsubset \mathbf{G}_0(M)$ with t typable in $\mathcal{D}_{\mathrm{sn}}$, no redex of M may be "forgotten" by t. This means that the identity reduction cannot approximate a non-identity reduction, ensuring faithfulness with respect to all reductions (subject reduction is easy to show).

It is also easy to show that all normal forms are typable. However, subject expansion fails in general (and for good reasons, see below); it only holds for *non-erasing* reductions. In the λ-calculus, a reduction step firing a redex $(\lambda x.M)N$ is *non-erasing* if $x \notin \mathrm{fv}(M)$ implies N normal.

Indeed, if $(\lambda x.M)N \to M$ because $x \notin \mathrm{fv}(M)$, and if $t \sqsubset M$ is a typable approximation, we cannot use the approximation $(\lambda a.t[\langle\rangle := a])\langle\rangle \sqsubset \mathbf{G}_0((\lambda x.M)N)$ in order to expand, because this is not typable in $\mathcal{D}_{\mathrm{sn}}$. However, we know that N is normal, hence typable, hence there exists $u \sqsubset N$ typable in $\mathcal{D}_{\mathrm{sn}}$, so we may use the approximation $(\lambda a.t[\langle z\rangle := a])\langle u\rangle$, where z does not appear in t and may be given the type of u. This latter point shows the necessity of weakening.

So, if $\mathcal{R}$ denotes the set of all reductions, $\mathcal{R}_0$ the set of non-erasing reductions and $\mathcal{N}$ the set of normal forms, we have that $\mathcal{S}(\mathcal{R})$ is the set of strongly normalizable λ-terms, whereas $\mathcal{W}(\mathcal{R}_0, \mathcal{N})$ is the set of λ-terms having a normal form via a non-erasing reduction. Since being strongly normalizable implies having a normal form under any kind of reduction, Theorem 20 applies and we have that a λ-term is strongly normalizable iff its normal form may be found by non-erasing reduction iff it is $(\mathcal{D}_{\mathrm{sn}}, \mathbf{G}_0)$-typable.

Note that $\mathcal{R}_0$ may be restricted to *any* set of non-erasing reductions whose normal forms are the normal forms *tout court*. In particular, we may take $\mathcal{R}_0$ to be what Barendregt calls the *perpetual strategy* [Bar84], and obtain the result (shown therein) that the perpetual strategy terminates on M iff M is strongly normalizable.

Weak normalization. Let $\mathcal{D}_{\mathrm{wn}}$ be the suboperad of $\mathbf{Poly}_{\mathsf{p}}$ (for arbitrary p) in which typing judgments $\Theta; \Gamma \vdash t : A$ are restricted so that A (resp. a type in Θ, Γ) may only contain occurrences of $\langle\rangle$ in negative (resp. positive) position, and consider the pair $(\mathcal{D}_{\mathrm{wn}}, \mathbf{G}_0)$. The corresponding type systems are obtained from Fig. 2.2 in the obvious way (simply restrict typing judgments).

In this case, subject reduction and expansion are unproblematic for all reductions. It is also easy to show that all normal forms are typable, so the pair is fully expansive with respect to $(\mathcal{R}_0, \mathcal{N})$ where $\mathcal{R}_0$ is all reductions and $\mathcal{N}$ are the normal forms. Note that $\mathcal{W}(\mathcal{R}_0, \mathcal{N})$ is just the set of (weakly) normalizable λ-terms.

This time, what fails in general is faithfulness. Indeed, it holds for reductions which only fire redexes whose applicative depth is minimal among all redexes, the *applicative depth* of a subterm N of M being the number of times one must cross the argument position of an application to reach N from the root of the syntactic tree of M. Such reductions are called *Böhm reductions*, because they iterate head reduction and gradually reveal the Böhm tree of a term: the head redex is at minimum depth; once the head variable is found, one starts entering in its arguments, and so on.

The intuition behind faithfulness for Böhm reductions is the following. Let $\psi : M \to M'$ be a non-identity reduction such that $H\psi$ is a Böhm reduction and suppose that h is a polyadic term typable in $\mathcal{D}_{\mathrm{wn}}$ such that $h\langle\rangle \sqsubset \mathbf{G}_0(H\psi)$, contradicting faithfulness. Since $H\psi$ is a Böhm reduction, H must be of the form $xN_1 \cdots N_k$ (if instead of x we had an abstraction, there would be a redex at strictly lower applicative depth). But then the type of x must be of the form $\vec{C}_1 \multimap \cdots \multimap \vec{C}_k \multimap \langle\rangle \multimap A$ with $\langle\rangle$ appearing negatively, which is not allowed in a context of $\mathcal{D}_{\mathrm{wn}}$.

So, if we take $\mathcal{R}$ to be Böhm reductions, we have that $\mathcal{S}(\mathcal{R})$ is obviously contained in the set of normalizable terms, and Theorem 20 gives us that a term is (weakly) normalizable iff every Böhm reduction starting from it terminates iff it is $(\mathcal{D}_{\mathrm{wn}}, \mathbf{G}_0)$-typable.

With a little bit of work, one may repeat the above with $\mathcal{R}$ equal to the so-called *leftmost strategy*, which always reduces the leftmost redex. We thus obtain as an application of Theorem 20 that a term has a normal form iff the leftmost strategy terminates on it, a classical result whose syntactic proof requires a non-trivial standardization theorem [Bar84].

Non-strict intersection types. So far we have always built systems in which intersections appear only to the left of arrows. As mentioned above, these are sometimes called "strict" intersection types and are a strict subset of the intersection types originally introduced by Coppo and Dezani [CDC80], which allow intersections everywhere (*e.g.*, $A \to A \wedge A$ is a valid intersection type).

As a matter of fact, this is a consequence of our choice of embedding of the λ-calculus in linear logic: we are using the morphism $\mathbf{G}_0 : \Lambda_0 \to \Lambda_!$, where Λ_0 is the sub-operad of Λ_k restricted to terms of type t (*cf.* Sect. 1.2.2). Since variables only have type v, there is an input/output asymmetry which is reflected at the level of intersection types into the "strict" discipline. Indeed, none of the operads of derivations introduced above, which are all of the form $\mathcal{E}[\mathcal{D}, \mathbf{G}_0]$ for some $\mathcal{D}$, is unital: this is visible in the var rule of Fig. 2.2, which is asymmetric.

Systems using "non-strict" intersection types may be obtained by considering the unrestricted encoding

$$\mathbf{G}_k : \Lambda_k \longrightarrow \Lambda_!,$$

which takes as source the full bichromatic presentation of the λ-calculus, with terms of type v as well. In this embedding, the image of a λ-term M seen as value is $!\mathbf{G}_0(M)$, so its approximations will all be of the form $\langle t_1, \ldots, t_1 \rangle$

$$\frac{}{x : A \vdash x : A}\ \mathsf{var} \qquad \frac{\Gamma_1, x : A_1 \vdash N : B_1 \quad \dots \quad \Gamma_n, x : A_n \vdash N : B_n}{\Gamma_1 \cdots \Gamma_n \vdash \lambda x.N : \langle A_1 \multimap B_1, \dots, A_n \multimap B_n \rangle}\ \mathsf{lam}$$

$$\frac{\Gamma \vdash M : \langle A \multimap B \rangle \quad \Delta \vdash N : A}{\Gamma \cdot \Delta \vdash MN : B}\ \mathsf{app}$$

Figure 2.3: Intersection types for the call-by-value λ-calculus. The are also rules exch, weak_0, weak and cntr, not shown because identical to Fig. 2.2.

with $t_i \sqsubset \mathbf{G}_0(M)$. In other words, we have systems with the rules of Fig. 2.2 augmented with

$$\frac{\Gamma_1 \vdash M : A_1 \quad \dots \quad \Gamma_n \vdash M : A_n}{\Gamma_1, \dots, \Gamma_n \vdash M : \langle A_1, \dots, A_n \rangle}\ \mathsf{inter}$$

In particular, the derivations of $\mathcal{E}[\mathcal{D}_{\mathrm{sn}}, \mathbf{G}_k]$, with $\mathcal{D}_{\mathrm{sn}}$ as defined above (the cartesian version), are just a different presentation of Coppo and Dezani's original system [CDC80].

The call-by-value λ-calculus. Our construction has two parameters: a suboperad $\mathcal{D} \hookrightarrow \mathbf{Poly}_{\mathsf{c}}$ and an embedding $\mathbf{G} : \mathcal{L} \to \Lambda_!$ of a calculus $\mathcal{L}$ in linear logic. The first wave of examples fixed $\mathbf{G}$ and showed several possibilities for $\mathcal{D}$; the last example provided a different $\mathbf{G}$, but the calculus was essentially the same.

Let us give an instance of our framework with a somewhat different calculus. In Sect. 1.2.2, we recalled Girard's call-by-value embedding

$$\mathbf{G}_v : \Lambda_v \longrightarrow \Lambda_!.$$

If $\mathcal{R}$ denotes head reduction in Λ_v (also known as the *weak call-by-value* strategy, which is the standard evaluation strategy of many practical programming languages), and if $\mathcal{R}_0$ denotes all reductions and $\mathcal{N}$ weak head normal forms (*i.e.*, arbitrary abstractions or terms of the form $xN_1 \cdots N_k$ with N_i arbitrary), then the pair $(\mathbf{Poly}_{\mathsf{p}}, \mathbf{G}_v)$ for any p is faithfully reductive with respect to $\mathcal{R}$ and fully expansive with respect to $(\mathcal{R}_0, \mathcal{N})$, so Theorem 20 proves that weak head normal forms are reachable with the weak call-by-value strategy iff they are reachable at all, and gives us intersection type systems characterizing weak call-by-value normalization.

The types for these systems are given by

$$A, B ::= \langle A_1 \multimap B_1, \dots, A_n \multimap B_n \rangle$$

($n = 0$ is allowed, which gives the base case of the inductive definition). The shape of types is justified by observing that Girard's call-by-value translation is based on the recursive type $D = !(D \multimap D)$, and remembering that $\langle - \rangle$ approximates $!(-)$. The rules are given in Fig. 2.3. As usual, one has four versions of the system, idempotent or not, relevant or not, by keeping or discarding weak and cntr.

The $\lambda\mu$-calculus. When we introduced the approximation presheaf (Sect. 2.3.1), we mentioned that $\Lambda_!$ and the polyadic calculi that approximate it are not the most general for which one may formulate this construction. In fact, polyadic approximations exist in linear logic and this latter is broader than the intuitionistic fragment represented by $\Lambda_!$.

It is possible to reformulate the whole theory of polyadic approximations (Chapter 1) using proof nets, which are the most general syntax for linear logic proofs, without intuitionistic restrictions. The details of this have been developed by Luc Pellissier in his forthcoming Ph.D. thesis. The development is based on *cyclic 2-operads*, of which the proof net syntax is a paradigmatic example. A cyclic operad is an operad in which inputs and outputs may be permuted at will, which corresponds to the behavior of classical negation. There is a cyclic 2-operad **LL** of untyped proof nets and cyclic 2-operads of (simply-typed) polyadic proof nets, on which a notion of approximation may be defined by straightforwardly extending Fig. 1.11.

The Grothendieck construction adapts without problems, too. For this, one considers the full sub-operad $\mathfrak{DiscRel}$ (resp. $\mathfrak{DiscRel}_*$) of $\mathfrak{Rel}$ (resp. $\mathfrak{Rel}_*$) whose objects are sets. This is a cyclic bioperad, and the Grothendieck construction is computed by pulling back along the forgetful functor $\mathbf{U} : \mathfrak{DiscRel} \to \mathfrak{DiscRel}_*$. The approximation presheaf, for a choice of sub-operad $\mathcal{D}$ of simply-typed polyadic proof nets, becomes then a lax morphism

$$\mathrm{Apx}[\mathcal{D}] : \mathbf{LL} \longrightarrow \mathfrak{DiscRel}.$$

In fact, we have already observed that, even in the intuitionistic case, our definition of approximation presheaf actually always lands in $\mathfrak{DiscRel}$. It is interesting to remark that, in the classical case, we have no choice, because $\mathfrak{Rel}$ does not have a cyclic structure. A suitable reformulation of Theorem 20 holds.

In this augmented set-up, we may consider Parigot's $\lambda\mu$-calculus [Par92], a well-known extension of the λ-calculus capturing classical reasoning (and control operators). This admits a natural presentation as a cyclic 2-operad ΛM, which may be embedded in **LL** by means of Laurent's translation [Lau03]

$$\mathbf{L} : \Lambda M \longrightarrow \mathbf{LL}.$$

Then, for any choice $\mathcal{D}$ of simply-typed polyadic proof nets, the Grothendieck construction applied to $\mathrm{Apx}[\mathcal{D}] \circ \mathbf{L}$ gives us a type system for the $\lambda\mu$-calculus. Its types, ranged over by A, B, C, are as follows:

$$\begin{aligned} A, B, C &::= \langle\langle P_1, \dots, P_n \rangle\rangle, \\ P &::= \Theta \multimap B, \\ \Theta &::= \langle A_1, \dots, A_n \rangle, \end{aligned}$$

where $\langle\langle - \rangle\rangle$ is the dual of $\langle - \rangle$, just like the modality $?(-)$ is dual to $!(-)$ in classical linear logic. The shape of the types is immediately justified by noting that Laurent's translation uses the recursive type $D = ?(!D \multimap D)$, keeping in mind that $\langle - \rangle$ approximates $!(-)$ and $\langle\langle - \rangle\rangle$ approximates $?(-)$. In the literature on intersection types for the $\lambda\mu$-calculus, these latter are known as

$$\frac{}{x : \langle A \rangle \vdash x : A}\ \mathsf{var}$$

$$\frac{\Gamma, x : \Theta \vdash M : B}{\Gamma \vdash \lambda x.M : \langle\langle \Theta \multimap B \rangle\rangle}\ \mathsf{lam} \qquad \frac{\Gamma, \alpha : A, \beta : B \vdash M : C}{\Gamma, \alpha : C \cdot A \vdash \mu\beta.\lceil\alpha\rceil M : B}\ \mathsf{name}$$

$$\frac{\Gamma \vdash M : \langle\langle \Theta_1 \multimap B_1, \ldots, \Theta_n \multimap B_n \rangle\rangle \quad \Delta_i^j \vdash N : \Theta_i(j)}{\Gamma \cdot \Delta_1^1 \cdots \Delta_1^{k_1} \cdots \Delta_n^1 \cdots \Delta_n^{k_n} \vdash MN : \langle\langle B_1, \ldots, B_n \rangle\rangle}\ \mathsf{app} \qquad \begin{array}{l} 1 \leq i \leq n \\ 1 \leq j \leq k_i \end{array}$$

$$\frac{\Gamma, \xi : [X_1, \ldots, X_n] \vdash M : C}{\Gamma, \xi : [X_{\sigma(1)}, \ldots, X_{\sigma(n)}] \vdash M : C}\ \mathsf{exch} \qquad \frac{\Gamma \vdash M : C}{\Gamma, \xi : [] \vdash M : C}\ \mathsf{weak}_0 \quad \xi \notin \Gamma$$

$$\frac{\Gamma, x : \langle B_1, \ldots, B_n \rangle \vdash M : C}{\Gamma, x : \langle B_1, \ldots, B_n, A \rangle \vdash M : C}\ \mathsf{weak} \qquad \frac{\Gamma, x : \langle B_1, \ldots, B_n, A, A \rangle \vdash M : C}{\Gamma, x : \langle B_1, \ldots, B_n, A \rangle \vdash M : C}\ \mathsf{cntr}$$

Figure 2.4: Intersection types for the $\lambda\mu$-calculus. The notation $\Gamma \cdot \Delta$ is concatenation as in Fig. 2.1, extended to sequences of the form $\langle\langle - \rangle\rangle$ as well. In the exch and weak_0 rules, $[-]$ stands for either $\langle - \rangle$ or $\langle\langle - \rangle\rangle$.

union types [Lau04]. Given a sequence $\Theta = \langle A_1, \ldots, A_n \rangle$, we write $\Theta(i)$ for A_i. Given two types A, B, which are always sequences, we write $A \cdot B$ for their concatenation. The typing judgments are of the form

$$x_1 : \Theta_1, \ldots, x_m : \Theta_m, \alpha_1 : B_1, \ldots, \alpha_n : B_n \vdash M : A,$$

where x_i are λ-variables and α_j are μ-variables. The typing rules are given in Fig. 2.4.

One may check that the right conditions for applying (the augmented version of) Theorem 20 are met by this system with respect to head reduction of $\lambda\mu$-terms, which is therefore characterized by typability. Interestingly, the linear version of this system (without weak and cntr) turns out to coincide with a system for the $\lambda\mu$-calculus recently introduced by Kesner and Vial [KV16].

2.4.3 Discussion and perspectives

It is has been known for a long time that intersection types and linear logic are related. Regnier (crediting Duquesne) started this line of investigation in his Ph.D. thesis [Reg92]; later on, Kfoury [Kfo00] discovered an intriguing relationship between intersection types and a linearization of the λ-calculus. Retrospectively, his is a sort of an embryo of our linear polyadic calculus. In the meantime, as mentioned earlier, non-idempotent intersection types had been introduced by Gardner [Gar94] and investigated both syntactically [CPWK04, NM04] and semantically [dC09], revealing their ties

with linear logic. More recently, other connections were drawn in the work of Pimentel et al. [PRR12], who developed a proof-theoretic analysis of intersection types using ideas coming from linear logic. However, to the best of our knowledge none of this previous work described the correspondence between approximations and intersection types in the sharp, synthetic and broad framework we propose.

Concerning (linear) approximations, the idea that the exponential modality of linear logic is a limit of its multiplicative approximations is of course as old as linear logic itself, by analogy with classical constructions from linear algebra. In the logical context, the intuition has been formalized in several ways: as a Taylor expansion [ER08], as a categorical limit [MTT09] and as a topological limit [Maz12], of which we gave an order-theoretic reformulation in Chapter 1. It is also quite explicitly used in games semantic constructions [AJM00, HO00], especially in the AJM version.

Ehrhard and Regnier's approach [ER08] is very similar to ours: they too express Girard's approximations directly in a calculus containing terms of the form $\langle t_1, \dots, t_n \rangle$, which morally correspond to the n-ary tensors of Girard's approximations. In their case, it is Boudol's *resource λ-calculus* [Bou93]; we favor polyadic calculi because they are syntactically simpler (there are no formal sums of terms) and more faithful to Girard's approximations (the tensor is not literally commutative, unlike in the resource λ-calculus). Nevertheless, the affinities between our approximations and the Taylor expansion of [ER08] are obvious and we believe that our work may be entirely reformulated in that context. As a starting point, it is immediate to define an embedding $(-)^\circ$ of $\Lambda_!^{\mathrm{p}}$ into the resource λ-calculus, and it is obvious that, for T in $\Lambda_!$ and t linear, $t \sqsubset T$ iff $t^\circ \in \mathrm{Taylor}(T)$, where $\mathrm{Taylor}(T)$ is the support of the Taylor expansion of T.

Moving on to other work concerning intersection types, we already mentioned Bucciarelli, Piperno and Salvo's paper [BPS03], in which they exhibit an encoding from intersection types derivations to simply-typed λ-terms and show the soundness (*i.e.*, typable $\Rightarrow$ SN) of Coppo, Dezani and Venneri's system [CDCV81] using this encoding. As already noted, our work (vastly) generalizes this approach to proving soundness.

Finally, we can never put too much emphasis on the influence of Melliès and Zeilberger's paper [MZ15], without which this work could not exist in its present form. It is worth mentioning that the fibrational perspective on type systems is already explored to some depth in [MZ15]. In this respect, all we do is "taking it to dimension two", in the sense that we consider reduction sequences and see (op)fibrations as witnesses of the subject expansion (reduction) properties. Melliès and Zeilberger's work focuses on the **Set**-enriched case and therefore fibrations have a different meaning there.

The main message of this chapter is perhaps the following: *as soon as a programming language may be meaningfully encoded in linear logic, there is an intersection type discipline for it*. Of course, what this intersection type discipline may do depends on the encoding and its "meaningfulness". However, we hope that we have shown enough applications to justify the claim that our approach is rather broad.

Our framework also gives a systematic explanation to certain aspects of intersection types:

- non-idempotency and relevance are just reflections of the absence of structural rules (weakening and contraction, respectively) in polyadic systems;
- strictness of intersections arises from the asymmetric version of Girard's encoding;
- as mentioned above, soundness is ultimately a consequence of strong normalization of propositional linear logic.

We would also like to stress that our construction does not have only an "explanatory" power but also has a "predictive" value: the last two instances given in Sect. 2.4.2, albeit simple, exemplify how our framework may be used to almost automatically synthesize intersection type systems without necessarily knowing them in advance (the call-by-value system is possibly implicit in the semantic study of [PRR99], but we did not know of it; the system for $\lambda\mu$ had been developed independently of our work). Another example, which we did not develop here, concerns calculi with explicit substitutions: these are well-known to be encodable in linear logic [KL07, AK12]; we therefore automatically get intersection type systems for such calculi, in the style of [KV14]. More interestingly, we are currently studying the possibility of applying our construction to Ehrhard and Laurent's encoding of the π-calculus in proof nets [EL10], which would yield what is perhaps the first application of intersection types to concurrent calculi.

An intriguing aspect of Theorem 20 is that it brings to light how, contrarily to common understanding, intersection types systems do not characterize dynamic properties *per se* but, rather, *relate* them to one another. That is, intersection types are actually a bridge between a "universal" property (strong normalization) and an "existential" property (weak normalization), and their apparent ability to characterize dynamic properties results from judiciously choosing such properties so that the "universal" implies the "existential". Admittedly, our current formulation of dynamic property (Definition 19) is quite naive and *ad hoc*, and was chosen only in view of making Theorem 20 applicable in the cases we know of. It would be interesting to pursue a more general approach in this direction.

In this respect, at some point above we observed how, in the case of the pair $(\mathcal{D}_{\mathrm{sn}}, \mathbf{G}_0)$, one cannot expect subject expansion to hold for *any* reduction. Indeed, we have the following special case of Theorem 20:

Corollary 22 *Let $\mathcal{R}$ be the set of all 2-arrows of a 2-operad $\mathcal{L}$, and let $\mathcal{N}$ be the set of its normal forms. Suppose that $(\mathcal{D}, \mathbf{G} : \mathcal{L} \to \Lambda_!)$ is faithfully reductive and fully expansive with respect to $\mathcal{R}$ and $(\mathcal{R}, \mathcal{N})$. Then, strong normalization and weak normalization coincide in $\mathcal{L}$.*

So, since $(\mathcal{D}_{\mathrm{sn}}, \mathbf{G}_0)$ is faithfully reductive everywhere, and since normal forms are typable, if it also enjoyed subject expansion everywhere, we would have

that a λ-term is strongly normalizable iff it is weakly normalizable! While obviously false for pure λ-terms, this is true of many classes of typable λ-terms (those of the λ-cube, for instance). In fact, it is a conjecture of Barendregt, Geuvers and Klop that strong and weak normalization coincide in all so-called *pure type systems* [Bar92]. It is unclear at present whether Corollary 22 is relevant to this conjecture.

As a last point, let us mention that intersection types are of course well-known for being the basis of a host of denotational models of the λ-calculus (see [BDS13] for a general account, and [PPR17] for an in-depth study of the relational case). We currently do not have a synthetic description of how to build denotational models of the λ-calculus from our framework. Hyland's operadic perspective [Hyl17] will perhaps be of help here: in fact, $\mathrm{Apx}[\mathcal{D}] \circ \mathbf{G}_!$ is nothing but an algebra in $\mathfrak{Rel}$ for the monochromatic 2-operad Λ, so the question becomes finding a way of turning this into a **Set**-algebra for the operad $\Lambda/\simeq_\beta$.

Chapter 3

Parsimony

3.1 A Quantitative Look at Approximations

3.1.1 The "modulus of continuity"

An immediate consequence of the computational version of Girard's Approximation Theorem (Theorem 16) is a form of (Scott-)continuity of computation in the λ-calculus:

Proposition 23 (continuity) *Suppose that*

$$M \to^* N$$

in the λ-calculus. Then, for all u in $\Lambda^{\mathrm{p}}_{\mathsf{a}}$ such that $u \sqsubset N$, there exists $t \sqsubset M$ such that

$$t \to^* u.$$

Proof. Any λ-calculus computation $\psi : M \to^* N$ is an ideal $[\![\psi]\!]$ of computations of $\Lambda^{\mathrm{p}}_{\mathsf{a}}$ such that, in particular, the set of its targets is the ideal $[\![N]\!]$. Therefore, given $u \sqsubset N$, there is a computation $\rho : t \to^* u$ in $[\![\psi]\!]$ and, by definition, $t \sqsubset M$. (For simplicity, we are ignoring renaming equivalence and are transparently using Girard's embedding, *i.e.*, writing $t \sqsubset M$ instead of $t \sqsubset \mathbf{G}_0(M)$). □

In other words, if we want an approximation of the result of a λ-calculus computation, all we need is to perform a computation in $\Lambda^{\mathrm{p}}_{\mathsf{a}}$ starting from an approximation of the initial term.

Suppose now that we are interested in computations ending with a Boolean value $\underline{b}$, *i.e.*, $\lambda x.\lambda y.x$ or $\lambda x.\lambda y.y$ (this is so that the size of the output is constant and we do not have to worry about it in the coming discussion). Suppose that the computation

$$M \to^* \underline{b}$$

takes l steps. Since $\mathbf{G}_0(\underline{b})$ does not contain boxes (it is of the form $\lambda a.\lambda b.z[!y := b][!x := a]$ with $z \in \{x, y\}$), it admits "itself" as approximation, *i.e.*, the term

$\lambda a.\lambda b.z[\langle y\rangle := b][\langle x\rangle := a]$, which we still abusively denote by $\underline{b}$. Applying Proposition 23, we obtain $t \sqsubset M$ such that

$$t \to^* \underline{b}.$$

What can we say about the size of t as a function of l? Observe that this is roughly the same as asking an upper bound on the length of the reduction $t \to^* \underline{b}$, because t is affine and its size bounds the length of any reduction starting from it.

The answer is that a relationship exists, but it is somewhat unreasonable: the size of t may be exponential in l. This is due to the uncontrolled behavior of duplication in the λ-calculus, which causes β-reduction to apparently perform an exponential amount of work in a linear number of steps, even under extremely minimalist reductions strategies, such as head reduction. A computationally uninteresting but straightforward example is

$$XXI \to\to XX(II) \to\to XX(II(II)) \to\to XX(II(II)(II(II))) \to^* \cdots$$

with $X := \lambda x.\lambda a.xx(aa)$.

Informally, we may call the relationship between the length of reductions and the size of affine approximations "modulus of continuity". Although technically wrong, this terminology is intuitively appropriate and useful.

So, why do we find it unreasonable that the λ-calculus has an exponential modulus of continuity? First of all because this seems to happen for wrong reasons: the fact that a β-reduction step $M(\!|x \leftarrow N|\!)$ immediately dispatches a copy of N to *all* occurrences of x in M does not mean that those copies are actually needed right away, or needed at all. In fact, abstract machines and practical implementations of functional programming languages tell us that this is far from being the case.

Jokingly, we could say that a λ-term is a very inefficient representation of itself. More seriously, the intermediate λ-terms in a β-reduction sequence $\psi : M \to^* \underline{b}$ may be very inefficient representations of the *states* that a machine actually goes through when performing the computation expressed by ψ. If one thinks directly in terms of how λ-terms and β-reduction may be implemented on a machine, then it becomes possible to define reasonable cost semantics directly on λ-terms, both for runtime and space usage [BG95, SBHG08].

Nevertheless, there are other reasons to be unhappy with an exponential modulus of continuity. These are related to computational complexity and motivate the search for a calculus with a polynomial modulus of continuity regardless of its machine implementations, as we are about to discuss.

3.1.2 Higher order circuits

It is well known that the runtime of a Turing machine is strongly related to the size of a Boolean circuit simulating it, where the *size* of a circuit is just the number of its gates. Given a family $(C_n)_{n\in\mathbb{N}}$ of Boolean circuits (on the standard fan-in 2 basis $\{\neg, \wedge, \vee\}$), such that C_n has n inputs and 1 output,

we say that it *decides* a language $L \subseteq \{0,1\}^*$ if, for all $x \in \{0,1\}^n$, $x \in L$ iff $C_{|x|}(x) = 1$.

Theorem 24 *Let M be a Turing machine deciding a language L in time $f(n)$. Then, there is a family of circuits of size $O(f(n)^2)$ deciding L.*

This may be seen as a quantitative version, for Turing machines and Boolean circuits, of Proposition 23: it says that Turing machines have a polynomial modulus of continuity.[1] By contrast, Proposition 23 only implies that a λ-term M deciding a language L in $f(n)$ steps (of, say, head reduction) induces a family of affine polyadic terms deciding L of size $O(2^{f(n)})$, and we observed that the upper bound is tight.

The above theorem is a fundamental result of structural complexity theory and has important implications, most notably the Cook-Levin theorem (the statement that satisfiability of propositional formulas is NP-complete). It seems morally wrong that the λ-calculus should be so desperately inadequate to deal with such basic computational phenomena, even if one is willing to admit (as we are) that quantitative reasoning is inherently less evident with λ-terms then it is with Turing machines.

In fact, it is tempting to consider the following "equation"

$$\frac{\text{affine } \lambda\text{-terms}}{\lambda\text{-terms}} \quad = \quad \frac{\text{Boolean circuits}}{\text{Turing machines}} \tag{3.1}$$

and ask to what extent it may be taken as a guideline for importing quantitative concepts from the world of Turing machines to that of the λ-calculus. Equation 3.1 is supported by several analogies: like Boolean circuits, affine λ-terms may only compute finite functions (on Church binary strings); like Boolean circuits, the (sequential) runtime of an affine λ-term coincides with its size; finally, Boolean circuits may be seen as morphisms of a free symmetric monoidal category (in fact, a PROP), while (normal) affine λ-terms are morphisms in a free *closed* symmetric monoidal category (with terminal unit), and closure is exactly the difference between first-order computation (Turing machines) and higher-order computation.

One way of looking at Theorem 24 is as a sort of "transform" taking from the domain of Turing machines, for which runtime analysis is highly nontrivial, to the domain of circuits, where runtime is completely evident (it is the size), in a way somewhat reminiscent of how the Fourier transform reveals the fundamental frequencies of a signal, which are otherwise hidden in its time development. So an exponential modulus of continuity sort of obfuscates this perspective and, on the contrary, a polynomial modulus of continuity for functional programs would be potentially interesting for quantitative static analysis.

[1] The quadratic bound may actually be improved to $f(n) \log f(n)$, but this is irrelevant for the discussion.

3.1.3 Non-uniform computation

From the perspective of Equation 3.1, there is a third reason why an exponential modulus of continuity is inadequate, which is related to the notion of *non-uniform computation*, of central importance in complexity theory. Analyzing this issue will actually point us in the direction of what we call *parsimony*, a possible way of obtaining a calculus with a polynomial modulus of continuity.

The family of Boolean circuits $(C_n)_{n\in\mathbb{N}}$ given by Theorem 24 has the property of being *uniform*, *i.e.*, there is an algorithm which, given n, outputs a description of C_n. This is because all these circuits come from the same Turing machine. However, when considering circuit families as a model of computation of its own, it is more natural to disregard any uniformity requirement: $(C_n)_{n\in\mathbb{N}}$ is just a sequence of circuits, each C_n having n inputs and 1 output, without any *a priori* relationship between them; the sequence may very well be uncomputable.

In such non-uniform models of computation, size becomes a fundamental parameter: if no restriction on the size of C_n is imposed, then *any* language becomes decidable. In fact, any language $L \subseteq \{0,1\}^*$ is decidable by a (non-uniform) family of size $O(2^n)$: simply hard-wire in C_n the characteristic function of the finite set $L \cap \{0,1\}^n$. By contrast, circuit families of polynomial size are very important in complexity theory: Theorem 24 implies that they correspond to a complexity class called $\mathsf{P/poly}$, which, albeit containing undecidable languages, is widely believed not to contain any NP-hard problem (this, of course, would imply $\mathsf{P} \neq \mathsf{NP}$, because $\mathsf{P} \subset \mathsf{P/poly}$).

The class $\mathsf{P/poly}$ is an instance of a so-called *advice class*: the usual, uniform model (in this case, deterministic polytime Turing machines, hence the letter P) is augmented with access to an *advice string*, which depends only on the length of the input (not on the input itself) and which is restricted to be of polynomial length (hence the poly). In the particular case of $\mathsf{P/poly}$, one may equivalently think in terms of a deterministic polytime Turing machine augmented with an infinite, read-only tape containing arbitrary (but fixed) information. Since the machine is polytime, only an initial segment of polynomial length of this string will be accessible to the machine. The statement that $\mathsf{P/poly}$ does not contain any NP-hard problem is equivalent to $\mathsf{P/poly} \neq \mathsf{NP/poly}$. Although technically stronger that $\mathsf{P} \neq \mathsf{NP}$, the non-uniform version is currently believed not to be any harder to prove in practice, which explains the relevance of non-uniform computation to structural complexity. In fact, as curious as it may seem, no-one at present knows how to use uniformity in proving lower bounds; in other words, no-one sees how one could separate P from NP without actually separating $\mathsf{P/poly}$ from $\mathsf{NP/poly}$.

Now, an interesting aspect of the viewpoint brought forth in Chapter 1 is that it nicely unifies the "family approach" and the "advice approach" to non-uniform computation: in view of Equation 3.1, higher-order circuits are (polyadic) affine terms, and a circuit family should be seen as an ideal of such terms; but a (not necessarily uniform) ideal of polyadic affine terms is an infinite λ-term, *i.e.*, a program which is allowed to contain an infinite string of advice as discussed above. Let us see how, in the context of the λ-calculus, untamed non-uniformity trivializes computation (every language

becomes decidable), and how this is related to the modulus of continuity being exponential. We will use an infinitary syntax of polyadic affine terms as an informal means of representing ideals (in fact, as already noted above, such a syntax may be made formal [Maz12, Maz14]).

We may represent binary strings with a suitable adaptation of the standard Church encoding; for instance,

$$\underline{0010} = \lambda s_0.\lambda s_1.\lambda a.x_0^0(x_1^0(x_2^1(x_3^0 a)))[\langle \overline{x}^1\rangle := s_1][\langle \overline{x}^0\rangle := s_0],$$

where, in general, $\overline{y}$ stands for the infinite sequence $y_0, y_1, y_2, \ldots$ of affine variables. Let $w \in \{0,1\}^*$, let $\#w$ be the non-negative integer whose dyadic representation is w, and let $L \subseteq \{0,1\}^*$. We may represent L as an infinite stream of Booleans

$$\mathsf{T}_L := \langle \underline{b_0}, \underline{b_1}, \underline{b_2}, \ldots\rangle$$

where b_i is 1 just if $i = \#w$ and $w \in L$. Consider now

$$\begin{aligned}
\mathsf{even} &:= \lambda s.\langle x_0, x_2, x_4, \ldots\rangle[\langle \overline{x}\rangle := s],\\
\mathsf{odd} &:= \lambda s.\langle x_1, x_3, x_5, \ldots\rangle[\langle \overline{x}\rangle := s],\\
M_L &:= \lambda w.y_1[\langle \overline{y}\rangle := w\langle \mathsf{even}, \mathsf{even}, \mathsf{even}, \ldots\rangle\langle \mathsf{odd}, \mathsf{odd}, \mathsf{odd}, \ldots\rangle \mathsf{T}_L].
\end{aligned}$$

On input a binary string $\underline{w}$ (encoded as above), M_L applies a number of times odd or even to T_L according to the bits of w, resulting in the extraction of $\underline{b_i}$ where i is the integer whose dyadic representation is w:

$$M_L\, \underline{w} \to^* \underline{b_{\#w}}.$$

So M_L decides L. Moreover, the length of the above reduction is $O(n)$ where n is the length of w; and yet, since the bit $\underline{b_{\#w}}$ is exponentially "far down" in T_L (*i.e.*, at position $\Omega(2^n)$), any approximation $t \sqsubset M_L$ such that $t\,\underline{w} \to^* \underline{b_{\#w}}$ will have to be of size $\Omega(2^n)$. So this is a computationally non-trivial example of exponential modulus of continuity.

It is interesting to observe that the terms even and odd result from contraction, which is the primitive behind indiscriminate duplication in the λ-calculus. Indeed, contraction may be represented by the infinitary term

$$\lambda a.\langle x_0, x_1, x_2, \ldots\rangle \otimes \langle x_1, x_3, x_5, \ldots\rangle[\langle \overline{x}\rangle := a] : !A \multimap !A \otimes !A$$

(although we never formally introduced it, the tensor may be added to the syntax of all our calculi without problems). It is an instance of "Hilbert's hotel": an infinity (of type $!A$) is split into two infinities. The terms even and odd are obtained by splitting an infinity in two and then discarding one copy. This is the fundamental insight leading to parsimony: in the non-uniform setting, computation is not trivialized by the presence of infinite advice (like the term T_L) but by the fact that one may access an exponential amount of such an advice. Since splitting an infinity in two seems to be the culprit, we are led to think that forbidding such a behavior will result in a calculus with a polynomial modulus of continuity. In the next section we will see that this intuition is, indeed, correct.

$$\frac{}{;a \rhd a}\ \mathsf{avar} \qquad \frac{\Theta;\Gamma \rhd M}{\Theta;\Gamma \setminus \{a\} \rhd \lambda a.M}\ \mathsf{lam} \qquad \frac{\Theta;\Gamma \rhd M \quad \Theta';\Gamma' \rhd N}{\Theta,\Theta';\Gamma,\Gamma' \rhd MN}\ \mathsf{app}$$

$$\frac{\Theta;\Gamma \rhd M \quad \Theta';\Gamma' \rhd N}{\Theta,\Theta';\Gamma,\Gamma' \rhd M \otimes N}\ \mathsf{tens} \qquad \frac{\Theta;\Gamma \rhd M \quad \Theta';\Gamma' \rhd N}{\Theta,\Theta';\Gamma \setminus \{a,b\},\Gamma' \rhd M[a \otimes b := N]}\ \mathsf{let}_{\otimes}$$

$$\frac{}{x_i;\rhd x_i}\ \mathsf{evar} \qquad \frac{\Theta;\Gamma \rhd M \quad \Theta';\Gamma' \rhd N}{\Theta \setminus \{x\},\Theta';\Gamma,\Gamma' \rhd M[!x := N]}\ \mathsf{let}_!$$

$$\frac{\Theta;\rhd M}{\uparrow\Theta;\rhd !M}\ \mathsf{box}\,(*) \qquad \frac{\Theta;\Gamma \rhd M \quad \Theta';\Gamma' \rhd N}{\Theta,\Theta';\Gamma,\Gamma' \rhd M :: N}\ \mathsf{cons}$$

Figure 3.1: Parsimonious terms. In rule bang, condition $(*)$ is that every exponential variable has at most one occurrence in Θ.

3.2 The Parsimonious Lambda-Calculus

3.2.1 Terms and reduction

We let a, b (resp. x, y) range over a countably infinite set of *affine* (resp. *exponential*) variables. The use of affine variables rather than linear is not essential but simplifies programming. An *occurrence* of exponential variable is of the form x_i, where $i \in \mathbb{N}$. Occurrences of exponential variables are naturally ordered by $x_i \leq y_j$ whenever $x = y$ and $i \leq j$. Let Θ be a set of occurrences of exponential variables. We write $\uparrow\Theta$ for the upward closure of Θ. We denote by $\Theta \setminus \{x\}$ the set obtained from Θ by removing all occurrences of the form x_i (if any).

Parsimonious terms are defined in Fig. 3.1. In $\Theta;\Gamma \rhd M$, Γ is the set of free affine variables of M and Θ is the set of free *virtual occurrences* of exponential variables. This latter set may be infinite: the rule box causes this to happen. The notation Θ,Θ' (or Γ,Γ') denotes, as usual, the union of the two sets *supposing that they are disjoint*.

It is easy to see that a parsimonious term respects the following discipline (modulo Barendregt's convention):

- affine variables appear at most once;
- if x_i, x_j are different occurrences of the same exponential variable, then $i \neq j$;
- boxes (*i.e.*, terms of the form $!M$) contain no free affine variable;
- an exponential variable occurs in at most one box, at most once therein and with the highest index with respect to the rest of the term.

$$
\begin{array}{rcll}
(\lambda a.M)[-]N & \to_\beta & M\{N/a\}[-] & \\
M[a \otimes b := (N \otimes P)[-]] & \to_\otimes & M\{N, P/a, b\}[-] & \\
\mathsf{S}_*\{x_0\}[!x := (N :: P)[-]] & \to_{::} & \mathsf{S}_*^{x--}\{N\}[!x := P][-] & \\
\mathsf{S}_*\{x_0\}[!x := (!N)[-]] & \to_{\mathsf{d}} & \mathsf{S}_*^{x--}\{N\}[!x := !N^{++}][-] & x \in \mathrm{fv}(\mathsf{S}_*\{x_0\}) \\
\mathsf{S}\{!\mathsf{S}'\{x_0\}\}[!x := (!N)[-]] & \to_{!} & \mathsf{S}\{!\mathsf{S}'\{N\}\}[-] & \\
M[!x := (!N)[-]] & \to_{\mathsf{w}} & M[-] & x \notin \mathrm{fv}(M)
\end{array}
$$

Figure 3.2: Reduction rules for the parsimonious λ-calculus.

We denote by M^{+k} the term obtained from M by replacing each free occurrence of any exponential variable x_i with x_{i+k}. We simply write M^{++} for M^{+1}. Similarly, if x_0 does not occur free in M, we denote by M^{x--} the term obtained from M by replacing each free occurrence x_i of x (and of x only!) with x_{i-1}.

The intuition behind boxes is that they satisfy the equation

$$!M = M :: !(M^{++}).$$

For instance, $!x_1$ morally stands for the infinite term $\langle x_1, x_2, x_3, \ldots \rangle$. This is the reason behind the parsimonious discipline: without it, the above equation would violate affinity. Note that this corresponds to the intuition given in the previous section: it is impossible to write a parsimonious term corresponding to $\langle x_0, x_2, x_4, \ldots \rangle$.

We now define reduction. As usual, we define *substitution contexts* by

$$[-] ::= \{\cdot\} \mid [-][\mathsf{p} := N],$$

where p stands for $a \otimes b$ or $!x$. *Shallow contexts*, ranged over by S, are contexts in which the hole does not appear under a $!(-)$. We write S_* to denote a shallow context in which the hole may not appear at all (*i.e.*, a shallow context *or* a term). The reduction rules are as in Fig. 3.2, and they are closed under *shallow* contexts (*i.e.*, we do not reduce under a $!(-)$). We denote by $\mathrm{p}\Lambda$ the calculus thus obtained.

Note that the three rules $\to_{\mathsf{d}}$, $\to_{!}$ and $\to_{\mathsf{w}}$ are not in superposition: the latter applies only if x does not occur free at all; if x does occur, then one checks if the occurrence x_0 appears under a $!(-)$; if it does, by parsimony it is the only occurrence of x and rule $\to_{!}$ is applied; otherwise, rule $\to_{\mathsf{d}}$ is applied, whether or not x_0 is present, and all the other occurrences are shifted down by 1.

Note how the syntax allows one to recover unambiguously whether a variable is affine or exponential. For this reason, we will occasionally use any letter to denote any kind of variable. It will also be convenient to use the abbreviations

$$\lambda a \otimes b.M := \lambda c.M[a \otimes b := c] \qquad \lambda !x.M := \lambda a.M[!x := a],$$

as well as combinations such as $\lambda a \otimes !x.M := \lambda c.M[!x := d][a \otimes d := c]$.

The parsimonious λ-calculus may be seen as a programming language with built-in manipulation of ultimately constant, affine streams. Exponential variables are stream variables; x_i means that we are asking for the i-th element of the stream, which we may only extract once. If we want to use a whole stream as a parameter, we just write $!x_0$, whereas $!x_1$ shifts a stream one position to the left and $t :: !x_0$ shifts it one position to the right, adding t on top.

It is easy to see that pΛ is Turing-complete. Indeed, one may define a *linear* fixpoint combinator

$$\begin{aligned} \mathsf{X} &:= \lambda !x.\lambda !f.f_0(x_0 !x_1 !f_1) \\ \mathsf{Y}_\ell &:= \mathsf{X}!\mathsf{X} \end{aligned}$$

and the reader may check that

$$\mathsf{Y}_\ell !F \to^* F(\mathsf{Y}_\ell !F^{++}).$$

This fixpoint combinator is linear because F must be of the form $\lambda a.F'$, not $\lambda !y.F'$, otherwise reduction gets stuck. This means that F uses its argument once (it may use it zero times but it would be pointless). Anyhow, linear fixpoints are enough to define while loops (indeed, a while loop is a linear *tail* recursion), and while loops are enough to achieve Turing-completeness.

The reader may have noticed that, besides the situations in which a would-be redex is stuck because of an obvious type mismatch (*e.g.*, $M[!x := \lambda a.N]$), we have a well-typed redex (with respect to the types that we will introduce in the next section)

$$\mathsf{S}\{!\mathsf{S}'\{x_0\}\}[!x := (N :: P)[-]]$$

for which there is no reduction rule defined in Fig. 3.2. Indeed, to reduce a term of the form $M[!x := N :: P]$, we look for x_0 in M: if this does not appear or if it appears in shallow position (*i.e.*, not in a box), then we apply the $\to_{::}$ rule; otherwise, the term is stuck, and the above redex is precisely this case. The missing reduction rule is

$$\mathsf{S}\{!\mathsf{S}'\{x_0\}\}[!x := (N :: P)[-]] \quad \to_{\mathsf{a}} \quad \mathsf{S}\{\mathsf{S}'\{N\} :: !(\mathsf{S}')^{++}\{x_0\}\}[!x := P][-].$$

Since this rule is bit complex to write and is not needed for expressiveness, we decided not to include it in the "official" set of rules, but it is otherwise unproblematic to add it.

3.2.2 Affine approximations and quantitative continuity

The reader will have noticed that reduction in pΛ is more "atomic" than in $\Lambda_!$. In fact, it applies the decomposition of exponential reduction steps we mentioned in Sect. 1.2.1. As stated therein, such a decomposition could be applied to $\Lambda_!$ as well, but in that context we preferred the more synthetic, global formulation. Here, the quantitative viewpoint pushes us to adopt the finer grained definition. The consequence is that the polyadic calculus $\Lambda^{\mathrm{p}}_{\mathsf{a}}$ must also be slightly modified in order make its reduction semantics match that of pΛ.

$$\begin{array}{rcl}(\lambda a.t)[-]u & \to_\beta & t\{u/a\}[-] \\ t[a\otimes b := (u\otimes v)[-]] & \to_\otimes & t\{u,v/a,b\}[-] \\ \mathsf{C}_*\{x_0\}[\langle \overline{x}, x_m\rangle := (u :: v)[-]] & \to_{::} & \mathsf{C}_*^{x--}\{u\}[\langle \overline{x}\rangle := v][-] \\ t[\langle\rangle := \langle \overline{u}\rangle[-]] & \to_{\mathsf{w}} & t[-]\end{array}$$

Figure 3.3: Reduction rules for the modified polyadic affine calculus. The hole may not appear in C_* (in which case u is discarded).

We give a quick overview of the syntax. The terms are given by

$$\begin{array}{c} t,u ::= a \mid \lambda a.t \mid tu \mid t\otimes u \mid t[a\otimes b := u] \\ \mid x \mid !\bot \mid t :: u \mid t[\langle \overline{x}\rangle := u]\end{array}$$

For convenience, we suppose that, in $t[\langle \overline{x}\rangle := u]$, the binder is always of the form $\overline{x} = x_0, x_1, \ldots, x_{m-1}$, so we may have an exact correspondence with the occurrences of exponential variables in pΛ. The constructor $t :: u$ too is introduced to match that of pΛ. The usual notation for polyadic boxes becomes syntactic sugar:

$$\langle t_1, \ldots, t_n\rangle := t_1 :: \cdots :: t_n :: !\bot = t_1 :: (\cdots :: (t_n :: !\bot)).$$

Another difference, of course, is that the linear variables (those ranged over by $a, b \ldots$) are replaced by affine ones.

The modified reduction rules are given in Fig. 3.3. The one-step reduction $t[\langle x_0, \ldots, x_{k-1}\rangle := \langle u_1, \ldots, u_n\rangle] \to t\{u_{i+1}/x_i\}$ of $\Lambda_{\mathsf{a}}^{\mathsf{p}}$ is decomposed here into $k+1$ steps. Observe that, since we are not considering the term $\bot$, we must have $n \geq k$, otherwise the term is "stuck". Also note that the "reindexing" performed in the $\to_{::}$ reduction is only needed to keep up with our naming conventions, so pΛ reduction may be matched with minimal fuss in the notations. Contrarily to pΛ, polyadic affine variables do *not* appear with any index in terms, *i.e.*, the rule may actually be written

$$t[\langle x, y_1, \ldots, y_m\rangle := (u :: v)[-]] \quad \to_{::} \quad t\{u/x\}[\langle y_1, \ldots, y_m\rangle := v][-].$$

This is relevant for complexity purposes: no reindexing is needed to implement reduction of affine polyadic terms; it is just plain, affine β-reduction.

It is straightforward to adapt Fig. 1.11 and define affine approximations for parsimonious λ-terms. This is done in Fig. 3.4. We dispose of approximation contexts by adopting the convention that $x_i \sqsubset x_i$ always, *i.e.*, the evar rule actually derives $y \sqsubset x_i \vdash y \sqsubset x_i$ and we simply stipulate to denote y by x_i. This is consistent with our convention on let binders. As usual, the approximation relation is extended to contexts by treating $\{\cdot\}$ as an affine variable.

We will now prove the result motivating parsimony: the parsimonious λ-calculus has a polynomial "modulus of continuity". First, a couple of technical definitions.

$$\frac{}{a \sqsubset a}\ \mathsf{avar} \qquad \frac{t \sqsubset M}{\lambda a.t \sqsubset \lambda a.M}\ \mathsf{lam} \qquad \frac{t \sqsubset M \quad u \sqsubset N}{tu \sqsubset MN}\ \mathsf{app}$$

$$\frac{t \sqsubset M \quad u \sqsubset N}{t \otimes u \sqsubset M \otimes N}\ \mathsf{tens} \qquad \frac{t \sqsubset M \qquad u \sqsubset N}{t[a \otimes b := u] \sqsubset M[a \otimes b := N]}\ \mathsf{let}_{\otimes}$$

$$\frac{}{x_i \sqsubset x_i}\ \mathsf{evar} \qquad \frac{t \sqsubset M \qquad u \sqsubset N}{t[\langle \overline{x} \rangle := u] \sqsubset M[!x := N]}\ \mathsf{let}_{!}$$

$$\frac{}{!\bot \sqsubset !M}\ \mathsf{empty} \qquad \frac{t \sqsubset M \quad u \sqsubset !M^{++}}{t :: u \sqsubset !M}\ \mathsf{cons}_{!} \qquad \frac{t \sqsubset M \quad u \sqsubset N}{t :: u \sqsubset M :: N}\ \mathsf{cons}$$

Figure 3.4: The approximation relation for $\mathrm{p}\Lambda$.

Definition 21 (rank, exponential depth) *The* rank *of a polyadic affine term t, denoted by $\mathrm{rk}(t)$, is the maximum k such that $\langle u_1, \dots, u_k \rangle$ is a subterm of t.*

The exponential depth *of a parsimonious term M, denoted by $\mathrm{d}(M)$, is the maximum nesting level of its boxes.*

In what follows, we denote by $|\cdot|$ the size of terms.

Lemma 25 *Let S denote a shallow context.*

1. *$\mathsf{S}_0 \sqsubset \mathsf{S}$ iff $\mathsf{S}_0^{x++} \sqsubset \mathsf{S}^{x++}$;*
2. *$t \sqsubset \mathsf{S}\{M\}$ iff $t = \mathsf{S}_0\{u\}$ for some $u \sqsubset M$ and $\mathsf{S}_0 \sqsubset \mathsf{S}$;*

Proof. Both points are straightforward inductions. □

Proposition 26 (quantitative continuity) *Let $M \in \mathrm{p}\Lambda$ and let $M \to N$. Then, for all $u \sqsubset N$ there exists $t \sqsubset M$ such that $t \to^* u$. Moreover, $\mathrm{rk}(t) \leq \mathrm{rk}(u) + 1$.*

Proof. By definition, we have $M = \mathsf{S}\{M_0\}$ and $N = \mathsf{S}\{N_0\}$, with S a shallow context and M_0, N_0 matching the left and right hand side, respectively, of one of the rewriting rules of Fig. 3.2. The proof is by induction on S. The only interesting case is $\mathsf{C} = \{\cdot\}$, the rest is straightforward, using point 2 of Lemma 25.

We check the case of a rule $\to_{\mathsf{d}}$; the other cases are similar and, in fact, yield $\mathrm{rk}(t) = \mathrm{rk}(u)$. We have $M_0 = \mathsf{S}\{x_0\}[!x := (!P)[-]]$ and $u \sqsubset \mathsf{S}^{x--}\{P\}[!x := !P^{++}][-]$. By definition of approximation and point 2 of Lemma 25, we have $u = \mathsf{S}_0\{v\}[\langle \overline{x} \rangle := w][-]'$ with $\mathsf{S}_0 \sqsubset \mathsf{S}^{x--}$, $v \sqsubset P$, $w \sqsubset !P^{++}$ and $[-]' \sqsubset [-]$. Now, by point 1 of Lemma 25, $\mathsf{S}_0^{x++} \sqsubset \mathsf{S}$, so $\mathsf{S}_0^{x++}\{x_0\} \sqsubset \mathsf{S}\{x_0\}$ by point 2 of Lemma 25. Then, if we let $t := \mathsf{S}_0^{x++}\{x_0\}[!x := (v_0 :: w)[-]']$, we have $t \sqsubset M_0$, $t \to_{::} u$ and $\mathrm{rk}(t) = \mathrm{rk}(u) + 1$, as desired. □

Lemma 27 (size bound) *$t \sqsubset M$ implies $|t| \leq |M|(\mathrm{rk}(t)+1)^{\mathrm{d}(M)}$.*

Proof. By induction on M. We only check the case $M = !N$, the rest is straightforward. We claim that $t \sqsubset !N$ implies $t = \langle u_0, \ldots, u_{n-1}\rangle$ with $u_i \sqsubset N^{+i}$ for all $0 \leq i < n$, which is itself easily shown by induction (on the derivation of $t \sqsubset !N$). Using the induction hypothesis, the fact that $\mathrm{rk}(t)$ bounds n as well as all $\mathrm{rk}(u_i)$, and that $|M| = |N|+1$ and $\mathrm{d}(M) = \mathrm{d}(N)+1$, we have

$$\begin{aligned} |t| &= 1 + \sum_{i=0}^{n-1} |u_i| \leq 1 + \sum_{i=0}^{n-1} |N|(\mathrm{rk}(u_i)+1)^{\mathrm{d}(N)} \\ &\leq 1 + |N|(\mathrm{rk}(t)+1)^{\mathrm{d}(N)+1} \leq |M|(\mathrm{rk}(t)+1)^{\mathrm{d}(M)}, \end{aligned}$$

as desired. □

The above results are already enough to prove that $\mathrm{p}\Lambda$ has a polynomial modulus of continuity: if $(M_i)_{i\in I}$ is a family of terms of exponential depth bounded by d and $M_i \to^* \underline{b_i}$ of length $l(i)$ and with b_i a Boolean, Proposition 26 gives us a family $(t_i)_{i\in I}$ such that $t_i \sqsubset M_i$, $t_i \to^* \underline{b_i}$ and $\mathrm{rk}(t_i) \leq l(i)$, so Lemma 27 gives us $|t_i| = O(|M_i| l(i)^d)$.

Although meaningful in the context of non-uniform computation, such a result is not very useful when we care about uniformity, because it says nothing about the family $(t_i)_{i\in I}$. For this, we need a slightly stronger version.

Definition 22 (homogeneous approximations) *Given $k \in \mathbb{N}$ and $M \in \mathrm{p}\Lambda$, the* homogeneous approximation of M of rank k *of M, denoted by $\lfloor M \rfloor_k$, is defined by induction on M, as follows:*

$$\begin{aligned} \lfloor a \rfloor_k &:= a; \\ \lfloor \lambda a.N \rfloor_k &:= \lambda a.\lfloor N \rfloor_k; \\ \lfloor NP \rfloor_k &:= \lfloor N \rfloor_k \lfloor P \rfloor_k; \\ \lfloor N \otimes P \rfloor_k &:= \lfloor N \rfloor_k \otimes \lfloor P \rfloor_k \\ \lfloor N[a \otimes b := P] \rfloor_k &:= \lfloor N \rfloor_k [a \otimes b := \lfloor P \rfloor_k] \\ \lfloor x_i \rfloor_k &:= x_i; \\ \lfloor !N \rfloor_k &:= \langle \lfloor N \rfloor_k, \ldots, \lfloor N^{+(k-1)} \rfloor_k \rangle; \\ \lfloor N[!x := P] \rfloor_k &:= \lfloor N \rfloor_k [\langle \overline{x} \rangle := \lfloor P \rfloor_k]; \\ \lfloor N :: P \rfloor_k &:= \lfloor N \rfloor_k :: \lfloor P \rfloor_k. \end{aligned}$$

Lemma 28 *For all $M \in \mathrm{p}\Lambda$ and $t \sqsubset M$, $t \sqsubseteq \lfloor M \rfloor_k$ for all $k \geq \mathrm{rk}(t)$.*

Proof. A straightforward induction on M. □

$$\frac{}{\vdash A^\perp, A} \qquad \frac{\vdash \Gamma, A \quad \vdash \Delta, A^\perp}{\vdash \Gamma, \Delta} \qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B} \qquad \frac{\vdash \Gamma, A \quad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}$$

$$\frac{\vdash \Gamma_\bullet, A}{\vdash \Gamma_\bullet, A^\bullet} \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, A_\bullet} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, A_\bullet}$$

$$\frac{\vdash \Gamma, A}{\vdash ?\Gamma, !A} \qquad \frac{\vdash \Gamma, A^\bullet \quad \vdash \Delta, !A}{\vdash \Gamma, \Delta, !A} \qquad \frac{\vdash \Gamma, A_\bullet, ?A}{\vdash \Gamma, ?A} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, ?A}$$

Figure 3.5: Parsimonious logic.

Theorem 29 (polynomial "modulus of continuity") *Let $(M_i)_{i\in I}$ be a family of parsimonious terms of exponential depth bounded by d, and let $M_i \to^* \underline{b_i}$ in $l(i)$ steps and with b_i a Boolean. Then, there exists a family of polyadic affine terms $(t_i)_{i\in I}$ such that, for all $i \in I$, $t_i \sqsubset M_i$, $t_i \to^* b_i$ and $|t_i| = O(|M_i| l(i)^d)$. Moreover, we may take $t_i := \lfloor M_i \rfloor_{l(i)}$.*

Proof. The first part is immediate from Proposition 26, Lemma 27. The "moreover" part follows from Lemma 28 and monotonicity of reduction: if $t_i \to^* \underline{b_i}$, then $\lfloor M_i \rfloor_{l(i)} \to^* \underline{b_i}$ as well because we know that $\mathrm{rk}(t_i) \leq l(i)$ and so $t_i \sqsubseteq \lfloor M \rfloor_{l(i)}$. □

3.2.3 Parsimonious logic

The parsimonious λ-calculus is not just an artificial language which happens to have a nice quantitative property. It is the term calculus corresponding, via Curry-Howard, to a well behaved logical system, a variant of linear logic which we call *parsimonious logic*. Let us introduce its propositional fragment (quantifiers of any order may be added using the standard rules).

The formulas are those of multiplicative exponential linear logic, plus two extra modalities:

$$A, B ::= \alpha \mid \alpha^\perp \mid A \otimes B \mid A \parr B \mid !A \mid ?A \mid A^\bullet \mid A_\bullet.$$

The dual of a formula A, denoted by $A^\perp$, is defined by De Morgan duality, as usual, with the addition that $(-)^\bullet$ is dual to $(-)_\bullet$.

The rules of parsimonious sequent calculus are given in Fig. 3.5. The connectives $\parr$ and $\otimes$ behave just as in linear logic. The modality $(-)^\bullet$ is a "weakening modality", *i.e.*, it behaves similarly to the modality $!(-)$ of linear logic, but allows only weakening and not contraction. The parsimonious exponentials are quite different from those of linear logic. There is a "functorial promotion" rule, a weakening rule for $?(-)$, a sort of asymmetric contraction, which is also known as *absorption* in linear logic, and a dual co-absorption rule for $!(-)$.

All of the rules of Fig. 3.5 are derivable in linear logic with the definition $A^\bullet := A \,\&\, 1$ (and $A_\bullet = A \oplus \bot$). However, to see the true nature of parsimonious logic one has to look at the cut-elimination rules. The most important one is

$$\dfrac{\dfrac{\vdash \Gamma, A^\bullet \quad \vdash \Delta, !A}{\vdash \Gamma, \Delta, !A} \quad \dfrac{\vdash \Sigma, A_\bullet^\bot, ?A^\bot}{\vdash \Sigma, ?A^\bot}}{\vdash \Gamma, \Delta, \Sigma} \quad \to \quad \dfrac{\vdash \Gamma, A^\bullet \quad \dfrac{\vdash \Delta, !A \quad \vdash \Sigma, A_\bullet^\bot, ?A^\bot}{\Delta, \Sigma, A_\bullet^\bot}}{\vdash \Gamma, \Delta, \Sigma}$$

Thanks to this rule, one may prove the isomorphism

$$!A \cong A^\bullet \otimes !A,$$

meaning that there are proofs of

$$\vdash ?A^\bot, A^\bullet \otimes !A \qquad \text{and} \qquad \vdash A_\bullet^\bot \parr ?A, !A$$

which, when cut against each other, normalize to an axiom (modulo η-expansion). We call the above isomorphism *Milner's law* because of the similarity with the π-calculus law $!P \equiv P \mid !P$. Milner's law is *not* valid in linear logic, which is why parsimonious logic cannot be seen as a subsystem of linear logic, merely a variant of it. What makes parsimonious logic diverge from linear logic is the co-absorption rule: without it, the system we obtain *is* a subsystem of linear logic.

Speaking of co-absorption, the acquainted reader will have noticed a similarity with *differential* linear logic [ER06], in which the rule

$$\dfrac{\vdash \Gamma, A \quad \vdash \Delta, !A}{\vdash \Gamma, \Delta, !A}$$

is cut-free derivable and represents the derivative (the operation which, from a smooth map $f : A \Rightarrow B$, produces a map $Df : A \Rightarrow (A \multimap B)$ yielding, for every vector $a \in A$, the best linear approximation $Df(a)$ of f around a). Albeit formally similar, this is not the same as co-absorption and Milner's law is not valid in differential linear logic either.

Although we will not do it here, one may of course define proof nets for parsimonious logic and prove cut-elimination using them, which is much nicer than sequent calculus. At the level of correctness criteria, the co-absorption rule behaves as a tensor and the absorption rule as a par. The promotion rule for $(-)^\bullet$ and the functorial promotion rule for $!(-)$ use boxes. For the rest, nothing new happens. In particular, all of the annoying troubles concerning weakenings (and the lack of connectedness it generates) remain.

One may of course consider intuitionistic parsimonious logic, which is where pΛ comes from. In fact, pΛ corresponds to intuitionistic *affine* parsimonious logic, in which weakening is allowed on all formulas and we may therefore discard the $(-)^\bullet$ modality. Milner's law then becomes

$$!A \cong A \otimes !A,$$

which renders the similarity with the π-calculus even more apparent.

$$\frac{}{;a : A \vdash a : A}\ \mathsf{var}$$

$$\frac{\Theta;\Gamma, a : A \vdash t : B}{\Theta;\Gamma \vdash \lambda a.t : A \multimap B}\ \multimap\mathsf{I} \qquad \frac{\Theta;\Gamma \vdash M : A \multimap B \quad \Theta';\Gamma' \vdash N : A}{\Theta,\Theta';\Gamma,\Gamma' \vdash MN : B}\ \multimap\mathsf{E}$$

$$\frac{\Theta;\Gamma \vdash M : A \quad \Theta';\Gamma' \vdash N : B}{\Theta,\Theta';\Gamma,\Gamma' \vdash M \otimes N : A \otimes B}\ \otimes\mathsf{I}$$

$$\frac{\Theta';\Gamma' \vdash N : A \otimes B \quad \Theta;\Gamma, a : A, b : B \vdash M : C}{\Theta,\Theta';\Gamma,\Gamma' \vdash M[a \otimes b := N] : C}\ \otimes\mathsf{E}$$

$$\frac{\Theta;\Gamma \vdash M : C}{\Theta;\Gamma, a : A \vdash M : C}\ \mathsf{weak} \qquad \frac{\Theta;\Gamma \vdash M : C}{\Theta, x : A;\Gamma \vdash M : C}\ \mathsf{!weak}$$

$$\frac{;\overline{a} : \Gamma \vdash M : A}{\overline{x} : \Gamma;\vdash\ !M\{\overline{x}_0/\overline{a}\} : !A}\ \mathsf{!I} \qquad \frac{\Theta';\Gamma' \vdash N : !A \quad \Theta, x : A;\Gamma \vdash M : C}{\Theta,\Theta';\Gamma,\Gamma' \vdash M[!x := N] : C}\ \mathsf{!E}$$

$$\frac{\Theta, x : A;\Gamma, a : A \vdash M : C}{\Theta, x : A;\Gamma \vdash M^{x++}\{x_0/a\} : C}\ \mathsf{abs} \qquad \frac{\Theta;\Gamma \vdash M : A \quad \Theta';\Gamma' \vdash N : !A}{\Theta,\Theta';\Gamma,\Gamma' \vdash M :: N : !A}\ \mathsf{coabs}$$

Figure 3.6: The simply typed parsimonious calculus.

Via Curry-Howard, propositional intuitionistic affine parsimonious logic induces a discipline of simple types for pΛ. The *simple types* are the formulas of intuitionistic propositional linear logic, generated by

$$A, B ::= \alpha \mid A \multimap B \mid A \otimes B \mid !A.$$

The typing rules are given in Fig. 3.6. As usual, typing judgments are of the form $\Theta;\Gamma \vdash M : A$ where Θ (resp. Γ) gathers all exponential (resp. affine) variables. For this reason, it would be superfluous to write the $!(-)$ modalities in front of every type of Θ, so we omit them.

If we forget the term annotations, we get a natural deduction system for intuitionistic affine parsimonious logic. Milner's law is realized by the terms

$$\lambda!x.x_0 \otimes !x_1 : !A \multimap A \otimes !A \qquad \lambda a \otimes !x.a :: !x_0 : A \otimes !A \multimap !A,$$

which use absorption and co-absorption, respectively. The first splits a stream into head and tail; the second takes an element a, a stream x and pushes a on top of x. The fact that these two terms induce an isomorphism is why we speak of streams and not merely lists: the empty list would break the isomorphism.

From the viewpoint of categorical logic, parsimonious logic corresponds to an extremely simple structure: a category $\mathcal{C}$ with

- a $*$-autonomous structure $(\otimes, 1, (-)^{\perp})$;

- free co-pointed objects, denoted by $A^\bullet$;
- a lax monoidal endofunctor $!(-)$ with a natural isomorphism

$$!A \cong A^\bullet \otimes !A.$$

The intuitionistic affine case is even simpler: a symmetric monoidal closed category with terminal unit and a lax monoidal endofunctor $!(-)$ satisfying Milner's law $!A \cong A \otimes !A$. In comparison, the categorical axiomatization of (intuitionistic affine) multiplicative exponential linear logic is fairly imposing [BBdPH93, Mel09].

The relationship between the categorical semantics and the parsimonious λ-calculus is more easily seen if we use a different syntax, in which only affine variables appear and there are explicit constructs for functorial promotion and absorption (we also add the unit for completeness; we omitted it in the above formulation because we never use it):

$$\begin{aligned} M, N ::= a \mid \lambda a.M \mid MN \mid M \otimes N \mid M[a \otimes b := N] \mid * \mid M[* := N] \\ \mid !M[a_1 := N_1, \dots, a_n := N_n] \mid M :: N \mid M[a :: b := N] \end{aligned}$$

with the constraint that, modulo Barendregt's convention, variables appear at most once. Furthermore, in $!M[a_1 := N_1, \dots, a_n := N_n]$, which we also abbreviate as $!M[\overline{a} := \overline{N}]$, we must have $\mathrm{fv}(M) \subseteq \{a_1, \dots, a_n\}$, and all these variables become bound.

The first line of the above grammar corresponds to the symmetric monoidal closed structure (the fact that the unit is terminal is reflected in the affinity of variables, which would otherwise be strictly linear). The constructs in the second line correspond, from left to right, to the monoidal endofunctor and the two directions of the natural isomorphism given by Milner's law (which we called co-absorption and absorption above).

The computational rules induced by the symmetric monoidal closed structure are obvious. The other rules are:

$$\begin{aligned} M[a :: b := (N :: P)[-]] &\to M\{N, P/a, b\}[-] \\ !M[a := (!N[\overline{b} := \overline{P}])[-], \overline{c} := \overline{Q}] &\to (!M\{N/a\}[\overline{b} := \overline{P}, \overline{c} := \overline{Q}])[-] \\ M[a :: b := (!N[\overline{c} := \overline{P}])[-]] &\to (M\{N, !N[\overline{c} := \overline{c}']/a, b\})[\overline{c} :: \overline{c}' := \overline{P}][-] \end{aligned}$$

$$\begin{aligned} !M[a := (N :: P)[-], \overline{b} := \overline{Q}] \quad &\to \\ (M\{\overline{b}'/\overline{b}\}\{N/a\} :: !M[a := P, \overline{b} := \overline{b}'])[\overline{b}' :: \overline{b}'' := \overline{Q}][-]&. \end{aligned}$$

The first comes from Milner's law; the second from the functoriality of $!(-)$; the last two come from the naturality of Milner's law.

It is not too hard to translate from the syntax of pΛ to the "categorical" syntax, and back. The categorical syntax shows that the use of indexed exponential variables in pΛ is just a notational convention and is by no means intrinsic to parsimony. The rewriting rules in categorical syntax, which are not particularly readable (especially the last, which corresponds to the rule we omitted from pΛ, *cf.* the very end of Sect. 3.2.1), show on the other hand the usefulness of indexed variables and motivate our presentation of pΛ.

$$
\begin{aligned}
\mathsf{Nat} &:= !(o \multimap o) \multimap o \multimap o \\
\underline{n} &:= \lambda!s.\lambda z.s_0(\ldots s_{n-1} z \ldots) : \mathsf{Nat} \\
\mathsf{succ} &:= \lambda n.\lambda!s.\lambda z.s_0(n\,!(s_1)\,z) : \mathsf{Nat} \multimap \mathsf{Nat} \\
\mathsf{pred} &:= \lambda n.\lambda!s.\lambda z.n\,((\lambda a.a) :: !s_0)\,z : \mathsf{Nat} \multimap \mathsf{Nat} \\
\mathsf{dup} &:= \lambda n.\mathsf{It}(n, \lambda m_1 \otimes m_2.(\mathsf{succ}\ m_1) \otimes (\mathsf{succ}\ m_2), \underline{0} \otimes \underline{0}) : \mathsf{Nat}[] \multimap \mathsf{Nat} \otimes \mathsf{Nat} \\
\mathsf{store} &:= \lambda n.\mathsf{It}(n, \lambda!x.!(\mathsf{succ}\ x_0), !\underline{0}) : \mathsf{Nat}[] \multimap !\mathsf{Nat}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Bool} &:= o \otimes o \multimap o \otimes o \\
\underline{1} &:= \lambda c \otimes d.c \otimes d : \mathsf{Bool} \\
\underline{0} &:= \lambda c \otimes d.d \otimes c : \mathsf{Bool} \\
\mathsf{not} &:= \lambda b.\lambda c \otimes d.b(d \otimes c) : \mathsf{Bool} \multimap \mathsf{Bool} \\
\mathsf{xor} &:= \lambda b.\lambda b'.\lambda c.b(b'c) : \mathsf{Bool} \multimap \mathsf{Bool} \multimap \mathsf{Bool} \\
\mathsf{and} &:= \lambda b.\lambda b'.c[c \otimes d := b(b' \otimes \underline{0})] : \mathsf{Bool}[] \multimap \mathsf{Bool} \multimap \mathsf{Bool}
\end{aligned}
$$

$$
\mathsf{Str} := !(o \multimap o) \multimap !(o \multimap o) \multimap o \multimap o
$$

Figure 3.7: Unary integers, Booleans and binary strings in $\mathrm{p}\Lambda_{\mathrm{ST}}$.

3.3 Parsimony and Computational Complexity

3.3.1 Simply-typed parsimonious programming

We will now give some examples of how basic datatypes and operations may be represented in the simply-typed parsimonious λ-calculus, which we call $\mathrm{p}\Lambda_{\mathrm{ST}}$ for short. We will use only one atom, o, and, if A, B are simple types, we will denote by $A[B]$ the type $A\{B/o\}$. We will also use the notation $A[]$ to mean that *some* B which we do not want to specify is substituted to o in A. If D and E are datatypes, we will represent a function $\mathsf{D} \to \mathsf{E}$ by means of a term of type $\mathsf{D}[] \multimap \mathsf{E}$, as is customary with simple types. Representations of type $\mathsf{D} \multimap \mathsf{E}$ (without type expansion), as well as the type itself, will be called *flat*. Note that arbitrary type expansions do not affect composition: it is immediate to check that, if $;a : A \vdash M : B$, then $;a : A[C] \vdash M : B[C]$ for all C, so that types $A[] \multimap B$ and $B[] \multimap C$ compose to yield $A[] \multimap C$.

The types Nat, Bool and Str of unary integers, Booleans and binary strings, respectively, are defined in Fig. 3.7, together with the encoding of integers and Booleans. For binary strings, the encoding is similar: if $w = w_1 \cdots w_n \in \mathbb{W}$, we have

$$\underline{w} := \lambda!s^0.\lambda!s^1.\lambda z.s_0^{w_1}(\ldots s_i^{w_n} z \ldots),$$

where the j-th occurrence of s^0 from the left has index $j-1$, and similarly for s^1. For instance, $\underline{001} = \lambda!s^0.\lambda!s^1.\lambda z.s_0^0(s_1^0(s_0^1 z))$.

The type Nat supports iteration $\mathsf{It}(n, \mathsf{step}, \mathsf{base}) := n\,!(\mathsf{step}')\,\mathsf{base}$, typed as:

$$\frac{;\Delta \vdash \mathsf{step} : A \multimap A \quad \Gamma;\Sigma \vdash \mathsf{base} : A}{\Gamma, \Delta'; \Sigma, n : \mathsf{Nat}[A] \vdash \mathsf{It}(n, \mathsf{step}', \mathsf{base}) : A}$$

where Δ' and step' are the results of systematically replacing linear variables by exponential ones. Note that the type of step must be flat.

Unary successor and predecessor are implemented as in Fig. 3.7. Since their types are flat, they may be iterated to obtain addition and subtraction, of type $\mathsf{Nat}[] \multimap \mathsf{Nat} \multimap \mathsf{Nat}$. This is again flat with respect to the second argument, so a further iteration on addition leads to multiplication, of type $\mathsf{Nat}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Nat}$. Unary integers are duplicable and storable as shown in Fig. 3.7. Using addition, multiplication, subtraction and duplication we may represent any polynomial with integer coefficients as a closed term of type $\mathsf{Nat}[] \multimap \mathsf{Nat}$.

These constructions can all be extended to the type Str, which also supports iteration, flat successors and predecessor, concatenation, and is duplicable and storable.

For the Booleans, we adopt the multiplicative type Bool used in [Ter04]. This type too is duplicable and storable. An advantage of multiplicative Booleans is that they support flat exclusive-or in addition to flat negation (see Fig. 3.7). On the other hand, conjunction (and disjunction) has one non-flat argument (see again Fig. 3.7). This would be the case of exclusive-or too if we had chosen the traditional Boolean type $o \multimap o \multimap o$. We write $\mathsf{if}\ b\ \mathsf{then}\ t\ \mathsf{else}\ u$ for $c[c \otimes \_ := b\,(t \otimes u)]$, which has type A if $t, u : A$ and $b : \mathsf{Bool}[A]$.

In the sequel we will abusively use affine variables of duplicable types non-linearly, *e.g.*, if $n : \mathsf{Nat}[]$ we write $n \otimes n$ meaning $n' \otimes n''[n' \otimes n'' := \mathsf{dup}[n]]$. Similarly, if a term step contains a free affine variable a of storable type A, we will abusively consider the result of its iteration to still have a free variable $a : A[]$ (instead of an exponential variable of type $!A$), by implicitly composing with store.

It will be useful to consider for loops, derived from iteration. Given

$$\mathsf{step}[i] : A \multimap A$$

containing a free affine variable $i : \mathsf{Nat}[]$, we define

$$\mathsf{step}_+ := \lambda!j \otimes a.!(\mathsf{succ}\, j_1) \otimes (\mathsf{step}[j_0/i]\, a) : !\mathsf{Nat}[] \otimes A \multimap !\mathsf{Nat}[] \otimes A$$

and, given $\mathsf{base} : A$, we set

$$\mathsf{for}\ i < n\ \mathsf{from\ base\ do\ step} \quad := \quad \mathsf{It}(n, \mathsf{step}_+, !\underline{0} \otimes \mathsf{base}).$$

3.3.2 Expressing logspace computation

We will now see how $\mathrm{p}\Lambda_{\mathrm{ST}}$ is able to express logarithmic space computation. More precisely, we will show that, for every $L \in \mathsf{L}$, where L is the class of problems solvable by deterministic Turing machines with a logarithmically bounded work tape, there exists

$$M_L : \mathsf{Str}[] \multimap \mathsf{Bool}$$

$$
\begin{aligned}
\mathsf{len} &:= \lambda w.\mathsf{It}(w, \mathsf{succ}, \mathsf{succ}, \underline{0}) : \mathsf{Str}[] \multimap \mathsf{Nat} \\
\mathsf{shift} &:= \lambda !x.!x_1 : !A \multimap !A \\
\mathsf{toStrm} &:= \lambda w.\mathsf{It}(w, \lambda s.(\underline{0} :: s), \lambda s.(\underline{1} :: s), !\underline{0}) : \mathsf{Str}[] \multimap !\mathsf{Bool} \\
\mathsf{leq} &:= \lambda m.\lambda n.x_0[!x := \mathsf{It}(n, \mathsf{shift}, \mathsf{It}(m, \lambda s.(\underline{0} :: s), !\underline{1}))] \\
&\quad : \mathsf{Nat}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Bool} \\
\mathsf{isOne} &:= \lambda w.\lambda n.x_0[!x := \mathsf{It}(n, \mathsf{shift}, \mathsf{toStrm}\, w)] : \mathsf{Str}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Bool}
\end{aligned}
$$

Figure 3.8: Other basic logspace functions.

in $\mathrm{p}\Lambda_{\mathrm{ST}}$ which decides L. In order to do this, we will use a nice presentation of L in terms of descriptive complexity, due to Immerman [Imm99]: it corresponds to first-order logic over totally ordered finite structures with the addition of a deterministic transitive closure operator. This may be equivalently presented in recursion-theoretic terms, as we do below.

Let $\mathbb{B} := \{0,1\}$ and $\mathbb{W} := \mathbb{B}^*$. We consider the following set of basic functions:

- the constant $0 \in \mathbb{N}$;
- negation $\mathsf{not} : \mathbb{B} \to \mathbb{B}$ and conjunction $\mathsf{and} : \mathbb{B}^2 \to \mathbb{B}$;
- the predicates $\mathsf{leq} : \mathbb{N}^2 \to \mathbb{B}$, $\mathsf{sum}, \mathsf{mul} : \mathbb{N}^3 \to \mathbb{B}$ corresponding to the integer relations $m \leq n$, $m + n = k$ and $m \cdot n = k$;
- the function $\mathsf{len} : \mathbb{W} \to \mathbb{N}$ returning the length of a string;
- the predicate $\mathsf{isOne} : \mathbb{W} \times \mathbb{N} \to \mathbb{B}$ such that $\mathsf{isOne}(w, i) = 1$ iff the i-th bit of w is 1.

Now call $\mathcal{L}$ the smallest set of functions containing the above basic functions and closed by composition and the following schemata:

- **universal quantification:** if $R : \Gamma \times \mathbb{N}^2 \to \mathbb{B} \in \mathcal{L}$, then $\forall R : \Gamma \times \mathbb{N} \to \mathbb{B}$ mapping $(\gamma, m) \mapsto 1$ iff $R(\gamma, m, i) = 1$ for all $i < m$ is also in $\mathcal{L}$;
- **deterministic transitive closure:** let $R : \Gamma \times \mathbb{N}^{2k+1} \to \mathbb{B} \in \mathcal{L}$. This induces a partial map $R_* : \Gamma \times \mathbb{N}^{k+1} \to \mathbb{N}^k$ mapping $(\gamma, m, \overline{n}) \mapsto \overline{n}'$ if $\overline{n}'$ is unique s.t. $R(\gamma, m, \overline{n}, \overline{n}') = 1$, or undefined otherwise. Then, $\mathcal{L}$ also contains $\mathsf{DTC}(R) : \Gamma \times \mathbb{N}^{2k+1} \to \mathbb{B}$ mapping $(\gamma, m, \overline{n}, \overline{n}') \mapsto 1$ iff there exist $\overline{n}_0, \ldots, \overline{n}_l \in \mathbb{N}^k$, with $\overline{n}_0 = \overline{n}$, $\overline{n}_l = \overline{n}'$ and $\overline{n}_i \in \{0, \ldots, m-1\}^k$ for all $0 < i \leq l$, such that $R_*(\gamma, m, \overline{n}_i) = \overline{n}_{i+1}$ for all $0 \leq i < l$.

The class L corresponds exactly to the predicates $\mathbb{W} \to \mathbb{B}$ in $\mathcal{L}$.

The basic functions are easily representable in $\mathrm{p}\Lambda_{\mathrm{ST}}$: some are already in Fig. 3.7 and what was not covered by the previous section may be found in Fig. 3.8. The universal quantification scheme is represented by the higher order term Univ defined in Fig. 3.9. The idea is the following: given $R : \mathbb{N}^2 \to \mathbb{B}$ and $m \in \mathbb{N}$, we use iteration to build a stream of Booleans whose first m bits

$$
\begin{aligned}
\mathsf{strmToW} &:= \lambda!x.\lambda m.m\,!(\mathsf{if}\ x_0\ \mathsf{then}\ \mathsf{succ}^1\ \mathsf{else}\ \mathsf{succ}^0)\,\underline{\varepsilon} : !\mathsf{Bool}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Str} \\
\mathsf{forall} &:= \lambda w.\mathsf{It}(w, \lambda b.\underline{0}, \lambda b.b, \underline{1}) : \mathsf{Str}[] \multimap \mathsf{Bool} \\
\mathsf{Univ} &:= \lambda!R.\lambda m.\mathsf{forall}(\mathsf{strmToW}(\mathsf{for}\,k < m\,\mathsf{from}\,!\underline{0}\,\mathsf{do}\,\lambda s.(R\,m\,k) :: s)) \\
&\quad : !(\mathsf{Nat}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Bool}[]) \multimap \mathsf{Nat}[] \multimap \mathsf{Bool}
\end{aligned}
$$

Figure 3.9: Universal quantification.

contain $R(m,0),\ldots,R(m,m-1)$; then, we use $\mathsf{strmToW}$ to convert this into a string and we check with forall that it consists entirely of ones.

Note that the variable $!R$ representing the relation on which universal quantification is applied is exponential because it appears free in the subterm $\lambda s.(R\,m\,k) :: s$, which is iterated. This means that, when we want to apply universal quantification to $t : \Gamma \multimap \mathsf{Nat}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Bool}$ representing a function in $\mathcal{L}$, we will first have to convert it to a term of type $!\Gamma \multimap !(\mathsf{Nat}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Bool})$ and then apply Univ to obtain a term of type $!\Gamma \multimap \mathsf{Nat}[] \multimap \mathsf{Bool}$. The extra modalities in $!\Gamma$ may then be removed because all types in Γ are storable (they are either Nat, Bool or Str). The same remark tacitly applies below.

Let us turn to representing $\mathsf{DTC}(R)$ with $R : \Gamma \to \mathbb{N}^{2k+1} \to \mathbb{B}$. First of all, we will restrict to the case $k = 1$. The general case may be treated by encoding a pairing function, which we omit here for briefness.[2] Second, we observe that the particular determinization R_* of R used in the definition of DTC is inessential: we may as well define $R_*(\gamma, m, i)$ to be the smallest j such that $R(\gamma, m, i, j) = 1$, or undefined otherwise. Indeed, the important case is when R is already deterministic (*i.e.*, a partial function), in which the determinization is irrelevant. We will adopt the second definition here; it is possible to deal with the first at the expense of a more complex encoding.

The representation of deterministic transitive closure is given in Fig. 3.10, which we now explain (in that figure, for aesthetic reasons we use the notation $\mathsf{let\,p} := N\,\mathsf{in}\,M$ instead of $M[\mathsf{p} := N]$). If $R : \mathbb{N}^3 \to \mathbb{B}$, computing $\mathsf{DTC}(R)(m,n,n')$ amounts to determining whether there is a path from n to n' in a graph G whose nodes are $[m] := \{0,\ldots,m-1\}$ and s.t. there is an edge (n,n') iff $R_*(m,n) = n'$, so the out-degree of G is at most 1 (*i.e.*, it is a forest). To do this, we imagine a token traveling in G, its position being represented by a stream of type $!\mathsf{Bool}$ which is $\underline{0}$ everywhere except where the token is. The edges of G may now be seen as a stream transformation $\varphi : !\mathsf{Bool} \multimap !\mathsf{Bool}$. Initially, the stream is $\underline{1}$ at position n; applying φ will make the token move, and we may determine the existence of a path by checking the value at position n' after at most m applications of φ.

The idea behind the definition of φ is best explained with an example. Suppose that $m = 4$ and that the edges of G are $\{(0,1),(1,1),(3,2)\}$. Then, $\varphi = \lambda!x.\underline{0} :: (\mathsf{xor}\,x_0\,x_1) :: x_3 :: \underline{0} :: !x_4$. This works because the input stream $!x$

[2]The curious reader may find an encoding of Cantor's bijective pairing in the appendix of the version of [Maz15] available on the author's web page. It involves programming division by two and square root.

$$
\begin{aligned}
\mathsf{mkDep}_R &:= \mathsf{if}\ (R\,m\,i\,j)\ \mathsf{then}\ \lambda b \otimes\, !x \otimes\, !y.(\mathsf{xor}\, b\, x_0) \otimes\, !x_1 \otimes \underline{0} :: !y_0 \\
&\qquad\qquad \mathsf{else}\ \lambda b \otimes\, !x \otimes\, !y.b \otimes\, !x_1 \otimes x_0 :: !y_0 \\
&\quad : (\mathsf{Bool} \otimes\, !\mathsf{Bool} \otimes\, !\mathsf{Bool}) \multimap (\mathsf{Bool} \otimes\, !\mathsf{Bool} \otimes\, !\mathsf{Bool}) \\
\\
\mathsf{rev} &:= \lambda s.s'[s' \otimes \_ := \mathsf{It}(m, \lambda !x \otimes\, !y.(y_0 :: !x_0) \otimes\, !y_1, !\underline{0} \otimes s)] \\
&\quad : !\mathsf{Bool} \multimap\, !\mathsf{Bool} \\
\\
\mathsf{mkFun}_R &:= \lambda s.\mathsf{let}\, s' \otimes \_ := \mathsf{for}\, j < m\, \mathsf{from}\, !\underline{0} \otimes s\, \mathsf{do} \\
&\qquad\qquad \lambda p \otimes q.\mathsf{let}\, b \otimes \_ \otimes q' := \mathsf{for}\, i < m\, \mathsf{from}\, (\underline{0} \otimes q \otimes\, !\underline{0})\, \mathsf{do} \\
&\qquad\qquad\qquad\qquad \mathsf{mkDep}_R[m,i,j]) \\
&\qquad\qquad\quad \mathsf{in}\ (b :: p) \otimes (\mathsf{rev}\, q') \\
&\qquad\ \mathsf{in}\ \mathsf{rev}\, s' \\
&\quad : !\mathsf{Bool} \multimap\, !\mathsf{Bool} \\
\\
\mathsf{DTC}_R &:= \lambda m.\lambda n.\lambda n'.\mathsf{for}\, k < m\, \mathsf{from}\, \underline{0}\, \mathsf{do} \\
&\qquad \mathsf{let}\, !x := \mathsf{It}(n', \mathsf{shift}, \mathsf{It}(k, \mathsf{mkFun}_R[m], \mathsf{It}(n, \lambda p.\underline{0} :: p, \underline{1} :: !\underline{0}))) \\
&\qquad \mathsf{in}\ \mathsf{if}\ x_0\ \mathsf{then}\ \lambda b.\underline{1}\ \mathsf{else}\ \lambda b.b \\
&\quad : \mathsf{Nat}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Bool}
\end{aligned}
$$

Figure 3.10: Deterministic transitive closure.

contains exactly one bit set to $\underline{1}$, so at most one of x_0, x_1 will be $\underline{1}$ and exclusive-or is equivalent to disjunction. We cannot use disjunction because it is not flat. Observe by the way that the simultaneous presence of flat disjunction and flat duplication (*i.e.*, if dup had type $\mathsf{Bool} \multimap \mathsf{Bool} \otimes \mathsf{Bool}$ instead of its present type $\mathsf{Bool}[\mathsf{Bool} \otimes \mathsf{Bool}] \multimap \mathsf{Bool} \otimes \mathsf{Bool}$) would allow this solution to work for arbitrary relations (*i.e.*, graphs of arbitrary out-degree) and we would be able to compute arbitrary transitive closures, which, by the forthcoming results, is impossible unless $\mathsf{L} = \mathsf{NL}$ (non-deterministic logspace).

The tricky task now is to compute φ from R. This is realized by mkFun_R, which operates by manipulating two streams p and q, the latter being initialized as the input stream s. For each $j \in [m]$, we determine its dependencies, *i.e.*, those $i \in [m]$ s.t. $R(m,i,j) = 1$. This is done by iterating over all $i \in [m]$ the term mkDep_R: if $R(m,i,j) = 0$, the i-th element is saved in an auxiliary stream (it may contain the token, so we must preserve it); otherwise, we xor the current result with the i-th element and set this element to $\underline{0}$, so that it won't be considered later (if the token was there, it has now moved). This yields the determinization of R we defined above. At the end of this, the result is pushed to p and we start over with the (possibly) modified q (q also needs to be reversed because traversing it and pushing its elements into an auxiliary stream reversed their order). When we exit the outer loop, p contains the desired

$$\frac{\Theta;\Gamma \vdash M : A}{\Theta;\Gamma \vdash M : \forall\alpha.A}\ \forall\mathsf{I},\ \alpha\notin\Gamma,\Theta \qquad \frac{\Theta;\Gamma \vdash M : \forall\alpha.A}{\Theta;\Gamma \vdash M : A\{B/\alpha\}}\ \forall\mathsf{E},\ B\ \text{!-free}$$

Figure 3.11: Typing rules for linear polymorphism.

stream (but, again, in reverse order).

Finally, the term DTC_R does nothing but looping through all $0 \leq k < m$ to determine whether, after k iterations of mkFun_R, the token has moved from n to n'.

3.3.3 Linear polymorphism and polytime computation

Polymorphism may be added to $\mathrm{p}\Lambda_{\mathrm{ST}}$ in the standard way, thus obtain what we may call "parsimonious system F". We will be particularly interested in a limited form of polymorphism: in logical terms, we restrict the comprehension scheme to linear types, *i.e.*, types not containing $!(-)$.

The obvious typing rules for linear polymorphism, to be added to those of Fig. 3.6, are given in Fig. 3.11. We call this augment system $\mathrm{p}\Lambda_{\ell\forall}$. In this more powerful type system, the term and of Fig. 3.7 may be given a flat type:

$$\mathsf{and} : \mathsf{Bool} \multimap \mathsf{Bool} \multimap \mathsf{Bool}.$$

Since negation is already flat, disjunction may also be given a flat type. The same is true of Boolean duplication:

$$\mathsf{dup} := \lambda b.\mathsf{if}\ b\ \mathsf{then}\ \underline{1} \otimes \underline{1}\ \mathsf{else}\ \underline{0} \otimes \underline{0} : \mathsf{Bool} \multimap \mathsf{Bool} \otimes \mathsf{Bool}.$$

This means that we may encode a Boolean circuit with n inputs and m outputs as a term of type $\mathsf{Bool}^{\otimes n} \multimap \mathsf{Bool}^{\otimes m}$. From this, it is not too hard to encode the transition function of an arbitrary Turing machine M by means of a term $\mathsf{trans}_M : !\mathsf{Bool} \multimap !\mathsf{Bool}$ (this is actually the main observation behind the standard proof of Theorem 24). Since we may compute polynomials, and since trans_M is flat, we may iterate trans_M a polynomial number of times (in the size of the input) starting from the initial configuration (of type !Bool, which may easily be built from an input of type Str using toStrm), thus simulating any deterministic Turing machine whose running time is bounded by a polynomial.

We have thus proved that, for any $L \in \mathsf{P}$ (the class of problems solvable in polynomial time by a deterministic Turing machine), there exists a term

$$M_L : \mathsf{Str}[] \multimap \mathsf{Bool}$$

deciding L typable in $\mathrm{p}\Lambda_{\ell\forall}$. In Sect. 3.3.6 we will show that the converse is true. The following result will be instrumental in the proof:

Proposition 30 *Let M : Bool be typable in $\mathrm{p}\Lambda_{\ell\forall}$. Then, M normalizes in a number of steps bounded by $|M|^k$, where k depends on the exponential depth of M (Defini-*

tion 21) and the exponential height *of M, which is the maximum nesting level of $!(-)$ modalities in the types appearing in the derivation of $M : \mathsf{Bool}$.*

Proof. The proof is essentially a careful cut-elimination theorem. The key technical point is that neither the exponential depth nor the exponential height may increase under reduction: the first because of parsimony (for the reader familiar with linear logic slang, parsimonious logic does not have "digging"), the second because polymorphism is restricted to linear types, whose exponential height is zero: the exponential height of $A\{B/\alpha\}$ is the same as that of A if B is !-free. The requirement that M be of type Bool, which may actually be replaced by any !-free type, is so that the set of rules of Fig. 3.2 is enough to reach the normal form (*i.e.*, we do not need the reduction rule mentioned at the end of Sect. 3.2.1). The details are a bit too technical and not interesting enough to be included here; the curious reader may find them in the version of [MT15] available on the author's web page. □

3.3.4 Polyadic geometry of interaction

The name "geometry of interaction", or GoI, originally designated a research program initiated by Girard [Gir89b] which aimed at interpreting cut-elimination as the inversion of a certain operator in a suitable algebra: from C^*-algebras [Gir89a] to von Neumann algebras [Gir11]. Over time, Girard's results were reformulated in different settings, from more concrete (token machines) to more abstract (traced monoidal categories). Today, the term GoI encompasses a diverse body of techniques whose only common ground is, perhaps, that they describe the execution of a program under a radically different form than the one adopted by more traditional operational semantics, be it "big step", "small step" or environment-based abstract machines.

In particular, and this is the main reason behind our interest in it, the GoI allows one to execute a program *without rewriting it* and *without explicitly computing closures*, *i.e.*, associating values to variables. This turns out to be fundamental if one wants to develop a meaningful complexity theory of space, in particular because neither rewriting nor environments are suitable for dealing with sublinear bounds, a fundamental case in space complexity:

- rewriting is obviously out of the question (how do we not count the size of the initial term?);
- explicitly computing the value of variables is also problematic: even if done with pointers (to subterms of the initial term), the number of variables is still linear in the size of the program under execution.

The GoI offers a scalable, general solution to the above issues. This was perhaps first understood by Schöpp [Sch06, Sch07] and later fully developed by Dal Lago and Schöpp [DLS10a], as well as Terui, who lectured several times on the subject (but never published any related work as far as we know).

The approach we follow here is to see the GoI semantics as a token machine, in the style of the *interaction abstract machine* originally introduced by

▲	a	$\mathsf{C}\{\lambda a.\mathsf{C}'\}$	S	$\rightsquigarrow$	▼	$\lambda a.\mathsf{C}'\{a\}$	C	$\mathsf{p}\cdot S$
▼	t	$\mathsf{C}\{\lambda a.\{\cdot\}\}$	S	$\rightsquigarrow$	▼	$\lambda a.t$	C	$\mathsf{q}\cdot S$
▼	u	$\mathsf{C}\{t\{\cdot\}\}$	S	$\rightsquigarrow$	▲	t	$\mathsf{C}\{\{\cdot\}u\}$	$\mathsf{p}\cdot S$
▲	tu	C	S	$\rightsquigarrow$	▲	t	$\mathsf{C}\{\{\cdot\}u\}$	$\mathsf{q}\cdot S$
▼	t	$\mathsf{C}\{\{\cdot\}\otimes u\}$	S	$\rightsquigarrow$	▼	$t\otimes u$	C	$\mathsf{p}\cdot S$
▼	u	$\mathsf{C}\{t\otimes\{\cdot\}\}$	S	$\rightsquigarrow$	▼	$t\otimes u$	C	$\mathsf{q}\cdot S$
▲	a	$\mathsf{C}[a\otimes b:=u]$	S	$\rightsquigarrow$	▲	u	$\mathsf{C}\{a\}[a\otimes b:=\{\cdot\}]$	$\mathsf{p}\cdot S$
▲	b	$\mathsf{C}[a\otimes b:=u]$	S	$\rightsquigarrow$	▲	u	$\mathsf{C}\{b\}[a\otimes b:=\{\cdot\}]$	$\mathsf{q}\cdot S$
▼	t	$\mathsf{C}\{\{\cdot\}::u\}$	S	$\rightsquigarrow$	▼	$t::u$	C	$0\cdot S$
▼	u	$\mathsf{C}\{t::\{\cdot\}\}$	$i\cdot S$	$\rightsquigarrow$	▼	$t::u$	C	$(i+1)\cdot S$
▲	x_i	$\mathsf{C}[\langle\overline{x}\rangle:=u]$	S	$\rightsquigarrow$	▲	u	$\mathsf{C}\{x_i\}[\langle\overline{x}\rangle:=\{\cdot\}]$	$i\cdot S$
▼	t	$\mathsf{C}\{\{\cdot\}[\mathsf{p}:=u]\}$	S	$\rightsquigarrow$	▼	$t[\mathsf{p}:=u]$	C	S

Figure 3.12: The transitions of the interaction abstract machine (IAM). In the second to last last row, $\overline{x}=x_0,\dots,x_m$ with $m\geq i$. In the last row, p stands for either $a\otimes b$ or $\overline{x}$.

Danos and Regnier [DR99]. Danos and Regnier's definition, as well as the ones that followed it (including very recent developments extending it in all sorts of directions [DLFVY17]), use proof nets. Here, we formulate the machine directly on terms, drawing inspiration from an unpublished note of Accattoli and Dal Lago. The machine will execute affine polyadic terms, in the modified syntax of Sect. 3.2.2.

Definition 23 (polyadic interaction abstract machine) *The* polyadic interaction abstract machine*, or pIAM, is a machine whose states are tuples of the form*

$$d \mid t \mid \mathsf{C} \mid S$$

where

- $d\in\{\blacktriangle,\blacktriangledown\}$ *is a* direction*;*
- t *is a closed affine polyadic term;*
- C *is an affine polyadic context;*
- S *is a* stack*, which is a finite string over* $\{\mathsf{p},\mathsf{q}\}\cup\mathbb{N}$*. We write* ε *for the empty stack and* $s\cdot S$ *for a stack whose first symbol is* s*.*

The transitions of the pIAM are given in Fig. 3.12. In addition, for each transition

$$d \mid t \mid \mathsf{C} \mid S \rightsquigarrow d' \mid t' \mid \mathsf{C}' \mid S'$$

therein, there is a transition

$$\overline{d'} \mid t' \mid \mathsf{C}' \mid S' \rightsquigarrow \overline{d} \mid t \mid \mathsf{C} \mid S,$$

where $\overline{\blacktriangle} := \blacktriangledown$ *and* $\overline{\blacktriangledown} := \blacktriangle$.

For a closed term t *and stack* S*, we let*

$$\mathsf{init}(t, S) := \blacktriangle \mid t \mid \{\cdot\} \mid S,$$
$$\mathsf{fin}(t, S) := \blacktriangledown \mid t \mid \{\cdot\} \mid S.$$

The former are called initial states*, the latter* final states*. Given a term* t*, we define a binary relation on stacks by*

$$S \overset{t}{\leftrightsquigarrow} S' \quad \textit{just if} \quad \mathsf{init}(t, S) \rightsquigarrow^* \mathsf{fin}(t, S').$$

Why is the pIAM a "token machine"? The intuition is that the pIAM has a read-only memory in which the term-graph of a fixed closed affine polyadic term is stored, and at any given time a token is placed on a node of this term-graph. The token carries information consisting of one bit and a stack (as defined above). A state $d \mid t \mid \mathsf{C} \mid S$ represents the machine with its memory containing $\mathsf{C}\{t\}$ and the token placed at the root of the subterm t, carrying the information (d, S). Depending on the current position of the token and its current information, the transitions move the token on an adjacent node of the term-graph and update the information. As a consistency check, we invite the reader to verify that $d \mid t \mid \mathsf{C} \mid S \rightsquigarrow d' \mid t' \mid \mathsf{C}' \mid S'$ implies $\mathsf{C}\{t\} = \mathsf{C}'\{t'\}$.

Observe that the pIAM is *bideterministic*: given a state, there is at most one future state *and* at most one past state. This makes the relation $\overset{t}{\leftrightsquigarrow}$ symmetric. In Girard's original viewpoint [Gir89a], it corresponds to the fact that proofs/programs are hermitian operators.

Proposition 31 (soundness of the pIAM) *Let* t *be a closed affine polyadic term. Then,*

$$t \to t' \quad \textit{implies} \quad \overset{t}{\leftrightsquigarrow} = \overset{t'}{\leftrightsquigarrow}.$$

Proof. This is "the" standard result for GoI token machines. In this case, it holds without restrictions because we are considering closed affine terms (no exponential boxes). An equivalent result, stated for parsimonious proof nets, may be found in [Maz15]. The proof in this context is morally identical and consists of a case analysis of Fig. 3.12 (which is modeled on proof nets). □

3.3.5 The return of intersection types

The ideas of Chapter 2 turn out to be useful also in the context of space bounds. The lesson we learned in there is that an intersection types discipline arises as soon as we have a well-behaved notion of approximation. The parsimonious λ-calculus fits the bill, and we may build an intersection type system for it by applying the Grothendieck construction to a suitable approximation presheaf

$$\mathrm{pApx} : \mathrm{p}\Lambda \longrightarrow \mathfrak{Rel}$$

based on Fig. 3.4. The 2-operad $\mathrm{p}\Lambda$ should be obvious: its colors are a and e (for affine and exponential variables), its multimorphisms terms with free

$$\frac{}{\vdash_{\mathsf{p}} !\bot : \langle\rangle}\ \textsf{empty} \qquad \frac{\Theta;\vdash t : A_0 \quad \Theta' \vdash_{\mathsf{p}} u : \langle A_1, \ldots, A_n\rangle}{\Theta, \Theta' \vdash_{\mathsf{p}} t :: u : \langle A_0, A_1, \ldots, A_n\rangle}\ \textsf{cons}_!$$

$$\frac{\Theta; \Gamma \vdash t : A_0 \quad \Theta'; \Gamma' \vdash u : \langle A_1, \ldots, A_n\rangle}{\Theta, \Theta'; \Gamma, \Gamma' \vdash t :: u : \langle A_0, A_1, \ldots, A_n\rangle}\ \textsf{cons}$$

Figure 3.13: Modified typing rules for polyadic simple types.

$$\frac{}{\alpha \sqsubset \alpha} \qquad \frac{A \sqsubset \sigma \quad B \sqsubset \tau}{A \multimap B \sqsubset \sigma \multimap \tau} \qquad \frac{A_1 \sqsubset \sigma \quad \ldots \quad A_n \sqsubset \sigma}{\langle A_1, \ldots, A_n\rangle \sqsubset\ !\sigma} \qquad \frac{\langle \overline{A}\rangle \sqsubset\ !\sigma}{\langle \overline{A}\rangle_{\mathsf{p}} \sqsubset \#\sigma}$$

Figure 3.14: Approximation relation for parsimonious types.

variables matching the types and its 2-arrows reductions modulo permutation equivalence.

In reality, since we are interested in the simply-typed parsimonious λ-calculus, we need something slightly different, namely a presheaf

$$\mathrm{pApx}_{\mathrm{ST}} : \mathrm{p}\Lambda_{\mathrm{ST}} \longrightarrow \mathfrak{Rel},$$

where $\mathrm{p}\Lambda_{\mathrm{ST}}$ is the 2-operad of (Church-style) simply-typed parsimonious terms. We will denote simple types by σ, τ to avoid confusion with *polyadic* simple types (which the construction "turns" into intersection types), for which we use A, B. Let us recall them:

$$\sigma, \tau ::= \alpha \mid \sigma \multimap \tau \mid \sigma \otimes \tau \mid\ !\sigma,$$
$$A, B ::= \alpha \mid A \multimap B \mid A \otimes B \mid \langle A_1, \ldots, A_n\rangle.$$

In the sequel, for the sake of brevity we will ignore tensors, their treatment being unproblematic and in all respects similar to that of the linear arrow.

The 2-operad $\mathrm{p}\Lambda_{\mathrm{ST}}$ has as colors the simple types σ (refining the color a of $\mathrm{p}\Lambda$), plus colors of the form $\#\sigma$ with σ a simple type (refining the color e of $\mathrm{p}\Lambda$). Its multimorphisms and 2-arrows are (Church-style) terms and reductions, as usual.

Since we are using a slightly different syntax for polyadic terms than the one introduced in Sect. 1.2.3 (in order to match parsimonious terms more closely, we have the term $!\bot$ and the construct :: instead of just one construct $\langle \ldots \rangle$, *cf.* Sect. 3.2.2), we need to adapt the typing rules of Fig. 1.9 to this syntax. The typing rules for polyadic simple types are as in Fig. 1.9, except that:

- the box rule is removed and replaced with the rules given in Fig. 3.13;
- of course, we have the weak rule but *not* the cntr rule.

Now, the *approximation relation* between polyadic simple types and simple types is presented in Fig. 3.14, and the approximation presheaf is defined as follows:

$$\frac{}{\Theta;\Gamma, a^{\sigma} : A \vdash a : A}\ \mathsf{avar}$$

$$\frac{\Theta;\Gamma, a^{\sigma} : A \vdash M : B}{\Theta;\Gamma \vdash \lambda a^{\sigma}.M : A \multimap B}\ \mathsf{lam} \qquad \frac{\Theta;\Gamma \vdash M : A \multimap B \quad \Theta;\Gamma' \vdash N : A}{\Theta;\Gamma,\Gamma' \vdash MN : B}\ \mathsf{app}$$

$$\frac{m \geq i}{\Theta, x^{\sigma} : \langle A_0, \ldots, A_m \rangle;\Gamma \vdash x_i : A_i}\ \mathsf{evar}$$

$$\frac{!M\{\overline{x^{\varphi}}\} : !\sigma}{\overline{x^{\varphi}} : \Theta;\Gamma \vdash !M : \langle\rangle}\ \mathsf{empty} \qquad \frac{\Theta;\Gamma \vdash M : A \quad \Theta;\Gamma' \vdash !M^{++} : \langle \overline{B} \rangle}{\Theta;\Gamma,\Gamma' \vdash !M : \langle A, \overline{B} \rangle}\ \mathsf{cons}_!$$

$$\frac{\Theta;\Gamma \vdash M : A \quad \Theta;\Gamma' \vdash N : \langle \overline{B} \rangle}{\Theta;\Gamma,\Gamma' \vdash M :: N : \langle A, \overline{B} \rangle}\ \mathsf{cons}$$

$$\frac{\Theta;\Gamma' \vdash N : \langle \overline{A} \rangle \quad \Theta, x^{\sigma} : \langle \overline{A} \rangle;\Gamma \vdash M : C}{\Theta;\Gamma,\Gamma' \vdash M[!x^{\sigma} := N] : C}\ \mathsf{let}_!$$

Figure 3.15: Intersection types for simply-typed parsimonious terms.

- on colors,
$$\mathrm{pApx}_{\mathrm{ST}}(\sigma) := \{A \mid A \sqsubset \sigma\};$$
- on multimorphisms,
$$\mathrm{pApx}_{\mathrm{ST}}(M)(\Theta, \Gamma; A) := \{\delta \text{ derivation of } \Theta;\Gamma \vdash t : A \mid t \sqsubset M\},$$
where the approximation relation is that of Fig. 3.4 and derivations are taken to be those of Fig. 1.9 amended as described above (with the rules of Fig. 3.13);
- on 2-arrows, the definition is similar, using the obvious approximation relation for reduction terms which may be inferred from Fig. 3.4.

The Grothendieck construction applied to $\mathrm{pApx}_{\mathrm{ST}}$ yields the type system shown in Fig. 3.15, let us call it $\mathbf{IT}\mathrm{p}\Lambda_{\mathrm{ST}}$. Typing judgments are of the form
$$\Theta;\Gamma \vdash M : A,$$
where

- Θ is composed of statements of the form $x^{\varphi} : \langle B_1, \ldots, B_k \rangle$ such that, for all $1 \leq i \leq k$, $B_i \sqsubset \varphi$;
- Γ is composed of statements of the form $a^{\tau} : C$ such that $C \sqsubset \tau$;
- M is a (Church-style) simply-typed parsimonious term.

One may check that, if $M : \sigma$ and the above judgment is derivable, then $A \sqsubset \sigma$. This is the reason behind the premise of the empty rule, which means that $!M$ is a term of type $!\sigma$ with free variables $\overline{x} = x^1, \ldots, x^n$ of respective types $\overline{\varphi} = \varphi_1, \ldots, \varphi_n$.

By removing the simple-type decorations (*i.e.*, σ, τ, etc.) and by discarding the hypothesis of rule empty (which becomes nullary), we obtain from Fig. 3.15 the intersection type system for pΛ (*i.e.*, pure parsimonious terms) mentioned above, resulting from the Grothendieck construction applied to pApx. Let us call this system **IT**pΛ. It is easy to see, using the techniques of Chapter 2, that typability in **IT**pΛ characterizes head-normalizable parsimonious terms. Since simply-typed parsimonious terms (strongly) normalize, they are all typable; moreover, they are actually typable in **IT**pΛ_{ST}. This remark will be useful later.

As introduced in the previous subsection, the pIAM executes untyped terms. However, if fed a simply-typable term, it behaves consistently with the type information. Albeit not so surprising (Girard originally conceived the GoI in a typed context), this property will give intersection types a key role in dealing with space complexity in the parsimonious λ-calculus.

Definition 24 (stack matching a type; well-typed state) *A stack S of the pIAM* matches *a simple polyadic type A if:*

- $S = \varepsilon$ *and* A *is atomic;*
- $S = \mathsf{p} \cdot S'$, $A = A' \multimap B$ *or* $A = A' \otimes B$ *and* S' *matches* A'*;*
- $S = \mathsf{q} \cdot S'$, $A = B \multimap A'$ *or* $A = B \otimes A'$ *and* S' *matches* A'*;*
- $S = i \cdot S'$, $A = \langle A_0, \ldots, A_n \rangle$, $0 \leq i \leq n$ *and* S' *matches* A_i.

Let δ be a closed simply-typed affine polyadic term and let $d \mid t \mid \mathsf{C} \mid S$ be a state of the pIAM such that $\mathsf{C}\{t\} = \delta^-$. We say that such a state is well-typed *if S matches the type of t within δ.*

Lemma 32 *The transitions of the pIAM preserve well-typedness.*

Proof. A case-by-case inspection of Fig. 3.12. □

3.3.6 Implicit and explicit complexity via parsimony

Deterministic polynomial time (P) and deterministic logarithmic space (L) are quite important classes in computational complexity theory because they are taken to embody the concept of "feasible computation" with respect to running time and memory usage, respectively.

These classes are also very robust, in that they admit a host of alternative characterizations by means of a variety of models of computation. In particular, both classes may be seen as the "uniformization" of a non-uniform model of computation restricted to families of polynomial size:

Theorem 33 *We have:*

1. $\mathsf{P} = \mathsf{POLYSIZE}_{\mathrm{unif}}$;
2. $\mathsf{L} = \mathsf{POLYBP}_{\mathrm{unif}}$.

In point 1, $\mathsf{POLYSIZE}_{\mathrm{unif}}$ is the class of problems decided by *uniform* polysize families of Boolean circuits. We will not enter into the the details of what "uniform" means; it suffices to think of it in the terms described in Sect. 3.1, *i.e.*, as the fact that there is a (very efficient) algorithm computing the description of the n-th circuit of the family from $n \in \mathbb{N}$. The key technical part is proving Theorem 24 mentioned in Sect. 3.1. The reader may find the details in any computational complexity textbook, *e.g.* [Pap94].

In point 2, one considers the non-uniform model of *branching programs*. These may be intuitively described as "decision trees with sharing", and $\mathsf{POLYBP}_{\mathrm{unif}}$ is the class of languages decided by *uniform* families of polysize branching programs. We will not need a formal definition, which the reader may find *e.g.* in [Vol99], together with the proof that such a class equals L.

As mentioned in Sect. 3.1, results like Theorem 33 may be seen as "unraveling" the complexity analysis of a program, a highly non-trivial task, into a family of "micro-programs" for which the complexity analysis is trivial (it is given by their size). This is important because it opens the way to study computational complexity from the non-uniform viewpoint, which is the predominant one in structural complexity theory. That is, once Theorem 33 is established, one may forget about uniformity and try to prove lower bounds directly on $\mathsf{POLYSIZE}$ and POLYBP, which are of purely combinatorial nature. Although still formidably difficult, this approach is at present the only one for which any viable technique has been developed.

In this section we will see how parsimony allows us to establish a similar framework in a higher-order setting. In particular, we will prove results similar to Theorem 33, stating the equality of fours kinds of classes:

$$\text{implicit} = \text{explicit} = \text{non-uniform} = \text{traditional},$$

where:

- by "traditional" we mean a complexity class in the usual definition, *i.e.*, one of P or L.
- By "explicit" we mean the higher-order equivalent of the "traditional" complexity class, *i.e.*, the one we would define if we did not know Turing machines and had to resort to notions which are purely internal to the λ-calculus. This is especially tricky for space complexity and we will see that our solution uses intersection types in a crucial way.
- By "non-uniform" we mean the higher-order equivalent of classes defined in therms of non-uniform models (circuits, branching programs). In our case, we will use families of affine polyadic terms, which are "higher-order circuits" according to our Equation 3.1 (Sect. 3.1).

- By "implicit" we mean a class defined without any explicit reference to complexity bounds, of any sort. In our case, these classes will be defined by means of standard type disciplines.

There are two novel aspects that the higher-order setting offers with respect to the "traditional", first-order setting. The first, of course, is the implicit characterization. Much has already been said about that in the literature, we will not insist any more here. The second aspect is more interesting. The characterization in terms of the non-uniform model must necessarily employ some uniformity condition; in the traditional case, we skipped the details but we mentioned that uniformity is an *algorithmic* notion, *i.e.*, it is about the existence of an algorithm generating the members of the family of "microprograms" of the non-uniform model (circuits, branching programs...). In our setting, it turns out that uniformity has a more *algebraic* flavor: a family $(t_n)_{n\in\mathbb{N}}$ of affine polyadic terms (*i.e.*, of "higher order circuits") is uniform just if there is a parsimonious term M such that $t_n \sqsubset M$ for all $n \in \mathbb{N}$ (Definition 27). From Chapter 1, we know that, intuitively, this means that all t_n follow the same "shape". In other words, uniformity may be reformulated as a coherence relation between pairs of "microprograms", rather than an algorithm generating such "microprograms". Such a coherence relation is explicitly defined, in the context of resource λ-terms, by Ehrhard and Regnier [ER08]. It is nice to see it resurface here; its role in computational complexity is still to be clarified, but it seems worth further investigation in the future.

Let us implement the framework sketched above. We start with the definition of "explicit" complexity classes. In the following, we refer to the encoding of Booleans and binary strings described in Fig. 3.7, which is of course available also in the untyped setting. We say that a closed term $M \in \mathrm{p}\Lambda$ decides a language L if, for all $w \in \{0,1\}^*$, we have

$$\psi_w : M\,\underline{w} \to^* \underline{b_w}$$

with b_w a Boolean and $b_w = 1$ iff $w \in L$. We need to define how much time and how much space M takes in deciding L. For what concerns time, the length of reductions to normal form is the obvious choice. We will adopt an equivalent but more sophisticated definition, which will give us the possibility of dealing with space too.

Note that, for all $w \in \{0,1\}^*$, we obviously have that

$$;\vdash \underline{b_w} : \mathsf{Bool}$$

is derivable in the system of intersection types $\mathbf{IT}\mathrm{p}\Lambda$. By subject expansion along ψ_w, we have a derivation of

$$;\vdash M\,\underline{w} : \mathsf{Bool},$$

from which we deduce the existence of a family of derivations $(\delta_w)_{w\in\{0,1\}^*}$ of

$$;\vdash M : A_w \multimap \mathsf{Bool},$$

for some types $(A_w)_{w\in\{0,1\}^*}$ such that $;\vdash \underline{w} : A_w$ is derivable. Observe that the family δ_w depends on ψ_w, so it is not uniquely defined in general. For our

purposes, we will take ψ_w to be a head reduction, which is unique and always guaranteed to lead to $\underline{b_w}$. Other choices are possible if one wishes to. This detail having been fixed, there is a unique family $(\delta_w)_{w\in\{0,1\}^*}$ associated with a term M deciding a language, constructed as above. We call such a family the *execution envelope* of M, and denote it by $\mathrm{env}(M)$, so that, if $w \in \{0,1\}^*$, $\mathrm{env}(M)_w = \delta_w$.

Definition 25 (type depth) *Let δ be a Church-style simply-typed affine polyadic term. The* type depth *of δ, denoted by $\mathrm{td}(\delta)$, is the maximum depth of the types (as syntactic trees) appearing in it. Note that this definition also applies to intersection types derivations of $\mathbf{IT}\mathrm{p}\Lambda$ and $\mathbf{IT}\mathrm{p}\Lambda_{\mathrm{ST}}$, because by definition these are isomorphic to Church-style simply-typed affine polyadic terms.*

Recall that, given a Church-style simply-typed affine polyadic term, δ^- is the underlying pure polyadic term, without type annotations.

Definition 26 (complexity classes in $\mathrm{p}\Lambda$) *Let $M \in \mathrm{p}\Lambda$ decide a language. We define:*

- *the* running time *of M to be the function*

$$n \mapsto \max_{|w|=n} |\mathrm{env}(M)^-_w|;$$

- *the* execution space *of M to be the function*

$$n \mapsto \max_{|w|=n} \left((\mathrm{td}(\mathrm{env}(M)_w) + 1) \cdot \log |\mathrm{env}(M)^-_w|\right).$$

We then define:

- $\mathrm{p}\lambda\mathsf{P}$ *to be the class of languages decided by closed parsimonious terms in polynomial running time;*
- $\mathrm{p}\lambda\mathsf{L}$ *to be the class of languages decided by closed parsimonious terms in logarithmic execution space.*

Some comments are in order. For what concerns running time, the above definition is actually unsurprising: in view of the polynomial modulus of continuity, and of the fact that a derivation of $\mathbf{IT}\mathrm{p}\Lambda$ is just an approximation, modulo a polynomial (which is negligible in time complexity) bounding the size is exactly the same as bounding the length of reductions to normal form.

The definition of execution space is justified by Lemma 36 below. Underlying it, there is the assumption that the GoI is a space-efficient execution mechanism for λ-terms.

Let us move on to our non-uniform model. Observe that, for all $w \in \{0,1\}^*$, $\underline{w}$ has no boxes, so $\lfloor \underline{w} \rfloor_k = \lfloor \underline{w} \rfloor_0$ for all $k \in \mathbb{N}$. Because of this, in the sequel we are going to abusively write $\underline{w}$ both for the parsimonious term representing w and the affine polyadic term $\lfloor w \rfloor_0$.

Definition 27 (higher order affine circuit classes) *Let $(t_n)_{n\in\mathbb{N}}$ be a family of affine polyadic terms. We say that such a family decides a language L if, for all $n \in \mathbb{N}$ and $w \in \{0,1\}^n$,*

$$t_n\, \underline{w} \to^* \underline{b_w}$$

with b_w a Boolean and $b_w = 1$ iff $w \in L$.

We say that a family $(t_n)_{n\in\mathbb{N}}$ is uniform *if there exists $M \in \mathrm{p}\Lambda$ such that $t_n \sqsubset M$ for all $n \in \mathbb{N}$.*

We denote by

- $\mathsf{HOPOLYSIZE}$ *($\mathsf{HOPOLYSIZE}_{\mathrm{unif}}$) the class of languages decided by (uniform) families of affine polyadic terms of polynomial size;*
- HOAC^0 *($\mathsf{HOAC}^0_{\mathrm{unif}}$) the class of languages decided by (uniform) families of simply-typable affine polyadic terms of polynomial size and bounded type depth.*

We may now prove the results announced above.

Lemma 34 *The normal form of an affine polyadic term t may be computed in deterministic time $O(|t|^k)$, with k constant.*

Proof. Let $m := |t|$. We know that m bounds the length of any reduction sequence starting from t, so there is a naive quadratic deterministic algorithm for finding the normal form: scan for a redex (*e.g.* from the left); halt if none is found, otherwise reduce it; start over. This terminates in at most m rounds, and the size of the terms shrinks at each round, so each round takes time $O(m)$, hence the quadratic bound. □

Theorem 35 *Let $\mathsf{C}(\mathrm{p}\Lambda_{\ell\forall})$ be the class of languages decided by terms of type*

$$\mathsf{Str}[] \multimap \mathsf{Bool}$$

in $\mathrm{p}\Lambda_{\ell\forall}$ (i.e., parsimonious with linear polymorphism). Then,

$$\mathsf{C}(\mathrm{p}\Lambda_{\ell\forall}) = \mathrm{p}\lambda\mathsf{P} = \mathsf{HOPOLYSIZE}_{\mathrm{unif}} = \mathsf{P}.$$

Proof. We show the cycle of four inclusions:

- $\mathsf{C}(\mathrm{p}\Lambda_{\ell\forall}) \subseteq \mathrm{p}\lambda\mathsf{P}$: let $M : \mathsf{Str}[] \multimap \mathsf{Bool}$ in $\mathrm{p}\Lambda_{\ell\forall}$ and let L be the language it decides. We know that, for all $w \in \{0,1\}^*$,

  $$M\, \underline{w} \to^{l(w)} \underline{b_w},$$

  where b_w is a Boolean and $l : \{0,1\}^* \to \mathbb{N}$ is the number of reduction steps. By Proposition 30, we know that $l(w) \leq |M\, \underline{w}|^k$. Now, $|M\, \underline{w}| = O(|w|)$. For what concerns k, we have that:

  - the exponential depth of $M\, \underline{w}$ is equal to the exponential depth of M, because the exponential depth of $\underline{w}$ is zero;

– the exponential height of $M\,\underline{w}$ is equal to the exponential height of M, because the simple typing $\underline{w} : \mathsf{Str}[]$ may be derived using only subtypes of $\mathsf{Str}[]$.

Therefore, relative to w, $k = O(1)$. So $l(w)$ is polynomial in $|w|$ and we conclude $L \in \mathrm{p}\lambda\mathsf{P}$ by the polynomial modulus of continuity (more precisely, Proposition 26 and Lemma 27).

- $\mathrm{p}\lambda\mathsf{P} \subseteq \mathsf{HOPOLYSIZE}_{\mathsf{unif}}$: let $M \in \mathrm{p}\Lambda$ decide $L \in \mathrm{p}\lambda\mathsf{P}$ and let, for $n \in \mathbb{N}$,

$$r(n) = \max_{|w|=n} \mathrm{rk}(\mathrm{env}(M)^{-}_{w}).$$

By Lemma 28, we have $\mathrm{env}(M)_w \sqsubseteq \lfloor M \rfloor_{r(|w|)}$ for all $w \in \{0,1\}^*$. So, if we let $t_n := \lfloor M \rfloor_{r(n)}$, by monotonicity we have

$$t_{|w|}\underline{w} \to^* \underline{b_w}$$

for all $w \in \{0,1\}^*$, which means that $(t_n)_{n\in\mathbb{N}}$ is a family of affine polyadic terms deciding L, which is is uniform ($t_n \sqsubset M$ for all $n \in \mathbb{N}$) and of polynomial size (by definition, $r(n) = O(n^k)$, so we conclude by Lemma 27).

- $\mathsf{HOPOLYSIZE}_{\mathsf{unif}} \subseteq \mathsf{P}$: Let $(t_n)_{n\in\mathbb{N}}$ be uniform, of polynomial size and decide L. We know there is a polynomial r such that $\mathrm{rk}(t_n) = O(p(n))$. By Lemma 28, $(\lfloor M \rfloor_{r(n)})_{n\in\mathbb{N}}$ still decides L. It is now immediate to construct a deterministic algorithm deciding L in polynomial time: on input $w \in \{0,1\}^n$, compute $\lfloor M \rfloor_{r(n)}$, which is obviously doable in polynomial time in n, because $r(n)$ is polynomial; then, construct $\lfloor M \rfloor_{r(n)}\,\underline{w}$ (which takes $O(n)$ time) and normalize it, which is polytime in n by Lemma 34.

- $\mathsf{P} \subseteq \mathsf{Cp}\Lambda_{\ell\forall}$: this was discussed in Sect. 3.3.3.

The proof is thus complete. □

Lemma 36 *The normal form of a closed simply-typable affine polyadic term $t = \delta^-$ with $\delta : \mathsf{Bool}$ may be computed in deterministic space*

$$O\left((\mathrm{td}(\delta) + 1)\log|t|\right).$$

Proof. We invite the reader to check that, if $\underline{b}$ is a Boolean, we have

$$b = 1 \quad \text{implies} \quad \mathsf{pp} \overset{\underline{b}}{\leftrightsquigarrow} \mathsf{qp};$$

$$b = 0 \quad \text{implies} \quad \mathsf{pp} \overset{\underline{b}}{\leftrightsquigarrow} \mathsf{qq}.$$

Therefore, by Proposition 31 and determinism of the pIAM, computing the normal form of t amounts to computing the final state of $\mathsf{init}(t, \mathsf{pp})$ on the pIAM. Such a computation may obviously be simulated by a Turing machine with a read-only input tape on which the input t is stored and with two work

tapes, one containing a pointer to the current position of the token in t, the other containing the information carried by the token.

The first work tape is obviously bounded by $\log |t|$. For what concerns the second work tape, it is enough to bound the space occupied by the stack. Observe that $\mathsf{init}(t, \mathsf{pp})$ is well-typed, so by Lemma 32 every state encountered during the execution of the pIAM will also be well-typed, which means that the stack will always match the types of δ. Therefore, the length of the stack will not exceed d. As for its elements, these are either bits or integers bounded by $|t|$, hence the stated bound. □

Theorem 37 *Let* $\mathsf{C}(\mathrm{p}\Lambda_{\mathrm{ST}})$ *be the class of languages decided by terms of type*

$$\mathsf{Str}[] \multimap \mathsf{Bool}$$

in $\mathrm{p}\Lambda_{\mathrm{ST}}$ *(i.e., parsimonious simple types). Then,*

$$\mathsf{C}(\mathrm{p}\Lambda_{\mathrm{ST}}) = \mathrm{p}\lambda\mathsf{L} = \mathsf{HOAC}^0_{\mathrm{unif}} = \mathsf{L}.$$

Proof. We show the cycle of four inclusions:

- $\mathsf{C}(\mathrm{pApx}_{\mathrm{ST}}) \subseteq \mathrm{p}\lambda\mathsf{L}$: we are given a closed simply-typed parsimonious term $M : \mathsf{Str}[] \multimap \mathsf{Bool}$. By the same reasoning of the proof of the first inclusion of Theorem 35 (which invokes Proposition 30), we have, for all $w \in \{0,1\}^*$,
$$\psi_w : M\,\underline{w} \to^{l(w)} \underline{b_w}$$
with $l(w)$ polynomial in $|w|$. Therefore, by Proposition 26 and Lemma 27, $|\mathrm{env}(M)^-_w| = O(|w|^k)$ for some constant k. Now observe that, for all $w \in \{0,1\}^*$,
$$;\vdash \underline{b_w} : \mathsf{Bool}$$
is derivable in $\mathbf{IT}\mathrm{p}\Lambda_{\mathrm{ST}}$, so the derivations of $\mathrm{env}(M)$ (which are computed by expanding along ψ_w) are actually in $\mathbf{IT}\mathrm{p}\Lambda_{\mathrm{ST}}$. By definition of $\mathbf{IT}\mathrm{p}\Lambda_{\mathrm{ST}}$, $\mathrm{td}(\delta_w)$ is bounded by the depth of the simple types in the derivation of $M : \mathsf{Str}[] \multimap \mathsf{Bool}$, which is fixed, call it d. Therefore, the execution space is $O((d+1)\log(|w|^k)) = O(\log |w|)$, as desired.

- $\mathrm{p}\lambda\mathsf{L} \subseteq \mathsf{HOAC}^0_{\mathrm{unif}}$: let $M \in \mathrm{p}\Lambda$ decide $L \in \mathrm{p}\lambda\mathsf{L}$. Since, by definition, $(\mathrm{td}(\mathrm{env}(M)_w) + 1)\log|\mathrm{env}(M)^-_w| = O(\log|w|)$, we must have $\mathrm{td}(\mathrm{env}(M)_w) = O(1)$ and $|\mathrm{env}(M)^-_w| = O(|w|^k)$. Moreover, by definition of $\mathbf{IT}\mathrm{p}\Lambda$, $\mathrm{env}(M)^-_w \sqsubset M$. So we actually already have a polysize, type-depth-bounded and uniform family of affine polyadic terms; the only problem is that such a family is indexed by $\{0,1\}^*$ and not $\mathbb{N}$. This is easily amended: we let, for $n \in \mathbb{N}$,
$$r(n) := \max_{|w|=n} \mathrm{rk}(\mathrm{env}(M)^-_w)$$
and we define $t_n := \lfloor M \rfloor_{r(n)}$. One may check that, for all $w \in \{0,1\}^n$,

since $\mathrm{env}(M)^{-}_{w}$ is typable, then so is t_n, so we conclude by Lemma 28 and monotonicity.

- $\mathsf{HOAC}^0_{\mathrm{unif}} \subseteq \mathsf{L}$: we are given a typable family $(t_n)_{n\in\mathbb{N}}$ of bounded type depth and polynomial size such that there exists a parsimonious term M such that $t_n \sqsubset M$ for all $n \in \mathbb{N}$. As in the proof of Theorem 35, we may replace it with $(\lfloor M \rfloor_{r(n)})_{n\in\mathbb{N}}$ for a suitable polynomial r. Observe now that, for fixed M (as in our case), $\lfloor M \rfloor_k$ may be computed in logspace in k. Indeed, in order to determine a given node in the syntactic tree of $\lfloor M \rfloor_{k(n)}$ (or the absence of such), all we need is a pointer to the syntactic tree of M and an integer counter bounded by $k+m$, where m is the maximum index of exponential occurrences in M (a constant). Storing this counter in binary only occupies logarithmic space in k. So we compose this algorithm with the one given by Lemma 36, and conclude.
- $\mathsf{L} \subseteq \mathsf{C}(\mathrm{pApx}_{\mathrm{ST}})$: this was shown in Sect. 3.3.2.

The proof is thus complete. □

3.4 Further results and perspectives

3.4.1 Non-uniform complexity

So far we only dealt with uniform complexity classes. And yet, in Sect. 3.1.3, we motivated parsimony by an analysis of non-uniform computation in infinitary affine terms. In fact, by nature, the parsimonious λ-calculus admits a non-uniform extension which is, in a sense, the λ-calculus equivalent of Turing machines with advice. We will now make a brief survey of it.

The non-uniform parsimonious λ-calculus, or nupΛ, is defined similarly to pΛ, except that boxes (*i.e.*, terms of the form $!M$) are replaced by

$$!\langle M_1, \dots, M_k \rangle_f$$

where $f : \mathbb{N} \to \{1, \dots, k\}$ is *arbitrary*. This means of course that the syntax of nupΛ is infinitary, which is to be expected if one has to somehow introduce non-uniformity. The above is called a *non-uniform box* (or just box) and the terms M_i are its *components*. The parsimonious requirements are always the same: only free exponential variables are allowed in a box and, modulo Barendregt's convention, each exponential variable in a term appears in at most one component of at most one box, with the highest index with respect to other occurrences of the same variable in the term.

The intuitive meaning of non-uniform boxes is

$$!\langle M_1, \dots, M_k \rangle_f = M_{f(0)} :: M^{+}_{f(1)} :: M^{+2}_{f(2)} :: M^{+3}_{f(3)} :: \cdots,$$

that is, a non-uniform box is an arbitrary stream on the finite "alphabet" $M_1, \dots, M_k$. The reduction rules of nupΛ may be obtained from those of pΛ in an obvious way by keeping in mind the above semantics. Of course, uniform boxes $!M$ are the special case in which $k = 1$, but now the infinite sequences

of Booleans mentioned in Sect. 3.1.3 become available as well. Because of this, *every* language is decidable in nupΛ.

The above fact does not prevent us from, on the one hand, restating Definition 26 and Definition 27 in terms of nupΛ instead of pΛ, and, on the other hand, to endow nupΛ with a discipline of simple types, with or without linear polymorphism, like we did for pΛ. For the former, we obtain two classes that we denote by nupλP and nupλL. For the latter, the typing rule for non-uniform boxes is a straightforward generalization of the uniform case:

$$\frac{;\overline{a}^1\Gamma_1 \vdash M_1 : A \qquad \dots \qquad ;\overline{a}^k : \Gamma_k \vdash M_k : A}{\overline{x}^1 : \Gamma_1, \dots, \overline{x}^k : \Gamma_k; \vdash !\langle M_1\{\overline{x}_0^1/\overline{a}^1\}, \dots, M_k\{\overline{x}_0^k/\overline{a}^k\}\rangle_f : !A}$$

We denote by nupΛ_{ST} (nup$\Lambda_{\forall\ell}$) the simply-typed (with linear polymorphism) non-uniform parsimonious λ-calculus.

Every result of Sect. 3.3 smoothly extends to the non-uniform case. In fact, the proofs are sometimes easier because we do not have to deal with uniformity. We therefore have:

Theorem 38 *Let* $\mathsf{C}(\mathrm{nup}\Lambda_{\mathrm{ST}})$ *(resp.* $\mathsf{C}(\mathrm{nup}\Lambda_{\forall\ell})$*) be the class of languages decided by terms of type*

$$\mathsf{Str}[] \multimap \mathsf{Bool}$$

in nupΛ_{ST} *(resp.* nup$\Lambda_{\forall\ell}$*). Then:*

1. $\mathsf{C}(\mathrm{nup}\Lambda_{\forall\ell}) = \mathrm{nup}\lambda\mathsf{P} = \mathsf{HOPOLYSIZE} = \mathsf{P/poly}$;
2. $\mathsf{C}(\mathrm{nup}\Lambda_{\mathrm{ST}}) = \mathrm{nup}\lambda\mathsf{L} = \mathsf{HOAC}^0 = \mathsf{L/poly}$.

Proof. The inclusions $\mathsf{C}(\mathrm{nup}\Lambda_{\forall\ell}) \subseteq \mathrm{nup}\lambda\mathsf{P} \subseteq \mathsf{HOPOLYSIZE} \subseteq \mathsf{P/poly}$ and $\mathsf{C}(\mathrm{nup}\Lambda_{\mathrm{ST}}) \subseteq \mathrm{nup}\lambda\mathsf{L} \subseteq \mathsf{HOAC}^0 \subseteq \mathsf{L/poly}$ are proved much in the same way as those of Theorem 35 and Theorem 37, respectively. The key technical points are the polynomial modulus of continuity (*i.e.*, Theorem 29), which of course is enjoyed also by nupΛ, and the non-uniform version of Proposition 30, which is the one originally proved in [MT15]. The only notable difference is in the inclusions $\mathsf{P/poly} \subseteq \mathsf{C}(\mathrm{nup}\Lambda_{\forall\ell})$ and $\mathsf{L/poly} \subseteq \mathsf{C}(\mathrm{nup}\Lambda_{\mathrm{ST}})$: the former has been hinted at in Sect. 3.3.3 (Boolean circuits may be encoded in nup$\Lambda_{\forall\ell}$); the latter is proved by encoding polysize families of branching programs, as detailed in [MT15]. □

To our knowledge, the above theorem is the first of its kind in implicit computational complexity: non-uniform complexity classes were considered previously (*e.g.* [Ter04]) but no implicit characterizations in logical terms were given. This shows the fruitfulness of the parsimonious approach to implicit complexity. Indeed, we want to stress that the best-known linear-logical approach to implicit complexity, namely Girard's *light linear logic* [Gir98], is inadequate to deal with non-uniform computation, because it includes contraction. In other words, if one took the infinitary affine λ-calculus $\Lambda_{\mathsf{a}}^{\infty}$ and imposed on it the constraints of light linear logic (which is possible!), one would still obtain a calculus in which every language is decidable. It is not clear whether

Lafont's alternative approach [Laf04] suffers from this problem and we do not know whether one can develop all of the above results in it.

The equality $\mathsf{HOAC}^0 = \mathsf{L/poly}$ is interesting in its own right, for two reasons. The first is that it clarifies Terui's above mentioned result [Ter04], stating that the class of languages decidable by bounded-depth, polysize families of so-called *Boolean proof nets*, let us denote it by BPN^0, is equal to $\mathsf{AC}^0[\textsc{ustcon}_2]$, *i.e.*, the class of languages decidable by bounded-depth, polysize families of Boolean circuits with unbounded fan-in gates and the addition of a certain kind of reachability gates ($\textsc{ustcon}_2$).

These extra gates are quite *ad hoc* and their computational meaning rather technical: a $\textsc{ustcon}_2$ gate of order n has $n(n+1)/2$ inputs, which are interpreted as the adjacency matrix of an undirected graph on the nodes $\{1,\ldots,n\}$; assuming that such a graph has degree at most 2 (*i.e.*, at most two incident edges on each node), the gate outputs 1 iff the nodes 1 and n are in the same connected component (if the input graph has degree greater than 2, the output of the gate is arbitrary).

Now, it is very easy to see that $\mathsf{BPN}^0 = \mathsf{HOAC}^0$, so Theorem 38 immediately implies $\mathsf{BPN}^0 = \mathsf{L/poly}$, showing, on the one hand, that Terui's original characterization was not optimal, and, on the other hand, that bounded-depth, polysize Boolean proof nets correspond to a very well known and robust complexity class, instead of a non-standard and rather *ad hoc* class.[3]

The second reason is more abstract: the fact that $\mathsf{HOAC}^0 = \mathsf{L/poly}$ tells us that the presence of higher-order makes a huge difference in the world of small complexity classes. Indeed, one of the very few unconditional lower bounds known in complexity theory states that $\textsc{parity} \notin \mathsf{AC}^0$ [Ajt83, FSS84], where AC^0 is the class of languages decidable by bounded-depth, polysize families of Boolean circuits with unbounded fan-in gates and $\textsc{parity}$ is the following problem: on input a binary string, tell whether it contains an even number of 1's. Now, not only is $\textsc{parity}$ extremely easy to implement with higher order primitives,[4] but bounded-depth higher-order circuits go well beyond that: they decide, for instance, $\textsc{majority}$ (given a binary string, tell if it contains strictly more 1's than 0's) and all the way up to the quintessential logspace problem, *i.e.*, reachability for directed forests, which we met under the disguise of deterministic transitive closure in Sect. 3.3.2.

So there is a fairly big gap between AC^0 and HOAC^0, and it is unclear how one may deal with complexities below logarithmic space in a λ-calculus

[3] Incidentally, we must note that, a few years after Terui's characterization of [Ter04], Allender et al. [ABC$^+$09] proved that the problem $\textsc{ustcon}_2$ described above is L-complete under AC^0 reductions, which immediately implies $\mathsf{AC}^0[\textsc{ustcon}_2] = \mathsf{L/poly}$. We learned of this result only after proving Theorem 38 in our joint work with Terui [MT15]. In fact, our theorem gives an independent proof of the L-completeness of $\textsc{ustcon}_2$.

[4] With the Church encoding of binary strings, $\textsc{parity}$ is decidable by the simply-typed program

$$\lambda w.w\ I\ \mathsf{not}\ \underline{1} : \mathsf{Str}[\mathsf{Bool}] \multimap \mathsf{Bool}$$

as soon as negation is available with "flat" type

$$\mathsf{not} : \mathsf{Bool} \multimap \mathsf{Bool}.$$

One must obviously do something tricky, with the calculus and/or with the definition of Booleans, for this *not* to be the case.

setting. By contrast, first-order "catches up" with higher-order as soon as we allow the depth to be unbounded: if POLYSIZE denotes the class of languages decidable by polysize families of Boolean circuits, then Theorem 38 tells us that POLYSIZE = HOPOLYSIZE because it is well-known that POLYSIZE = P/poly (this is Theorem 24 mentioned in Sect. 3.1).

3.4.2 The "logic of while loops"?

Implicit characterizations of L abound: of recursive-theoretic nature [Nee04, Kri05], using imperative languages [Jon99, Bon06] and higher-order languages [Sch06, Sch07, DLS10a]. Of these, only the latter are immediately comparable to our work. Another paper explicitly relating streams and logarithmic space is [RL11], which however does not have much of a connection with our work: the authors consider there corecursive definitions, *i.e.*, algorithms on infinite streams (as opposed to finite strings) and the space complexity they refer to is not the usual decision problem complexity.

The GoI plays a key role in both [Sch07, DLS10a]. The difference here is not so much in the use of the GoI, which is quite similar, but in the underlying programming language: in that work, the author(s) take the standpoint that the fundamental primitive of sublinear space computation is interaction (a point of view already taken in [Sch06]) and forge their programming language around this. This leads, for instance, to the use of non-standard types for encoding strings, namely $\mathsf{Nat} \multimap \mathsf{Three}$ (a binary string x is seen as a function mapping i to $x_i \in \{0,1\}$ or to $\bot$ if $i \geq |x|$), whereas the language of [Sch06] has an explicit list type.

With respect to the above, we believe that the highlight of our characterization is that it is closer to the original spirit of applying linear logic to implicit complexity [Gir98]: it is purely logical (there is no primitive datatype) and employs standard types. Our characterization also improves on previous ones in terms of simplicity: the types of [Sch07] include full polymorphism and indexed exponential modalities, whereas the categorical construction of [DLS10a], while elegant, also yields a sort of indexed exponential modality in types, making type inference not straightforward (see [DLS10b]). By contrast, our calculus is simply-typed, has only 11 typing rules (Fig. 3.6) which are essentially syntax-directed, so type inference is easier. Programming is of course restricted but, as hopefully showcased by Sect. 3.3.2, quite reasonably so if we consider that all programs must run in logarithmic space.

We also want to stress (again) that parsimony offers a truly novel approach to applying linear logic to implicit complexity, which is not just a variant of existing "light logics" (such as bounded, light or soft linear logic) or of systems such as those of [Hof97, Hof03]. The most prominent difference with respect to "light logics" is the absence of stratification or other structural principles enforcing bounded-time cut-elimination: as mentioned above, the untyped parsimonious λ-calculus is Turing-complete, whereas light λ-calculi normalize with the same runtime independently of types. This is because parsimony is not about the global complexity of normalization but the local complexity of single reduction steps, via the notion of continuous linear approximations.

This allows dealing with non-uniform computation, a perspective not offered by previous work on implicit complexity.

An interesting research direction is to consider second-order quantification, *i.e.*, parsimonious system F. It is easy to encode primitive recursive functions as terms of type $\mathsf{Nat}^{\otimes n} \multimap \mathsf{Nat}$ in such a system; however, all the usual ways of encoding the Ackermann function fail. In fact, we conjecture that parsimonious system F captures exactly primitive recursion, although at present we do not even have a proof strategy for showing this. Apart from our failed attempts at coding non-primitive recursive functions, our reason for believing in such a conjecture is that we think parsimonious logic to be, in a certain sense, the "logic of while loops". We believe that system F forces every while loop to terminate in a hereditary way which, in the end, is equivalent to having only for loops.

The reason why we think of parsimonious logic as the "logic of while loops" becomes clear if we extend simple parsimonious types along a different direction, that of "parsimonious PCF". The most peculiar feature of this calculus is that recursive definitions are restricted to be *linear*, as described in Sect. 3.2.1. In other words, we may only use recursive definitions that coincide with a slightly liberalized form of while loops, where the unique recursive call is not required to be in tail position.

For instance, a "parsimonious OCaml" programmer would be allowed to write

```
let rec fun x = <BODY> ;;
```

only if `<BODY>` contains exactly one occurrence of `fun` in every execution branch (*e.g.*, `fun` may actually appear twice, once in each branch of an `if then else` statement). Typically,

```
let rec fib n = if n < 2 then 1 else fib(n-1) + fib(n-2);;
```

would be rejected. Instead, one would have to resort (for example) to the following solution:

```
let rec fib_aux(n,m,d,s) =
    if d = -1 & s = [] then m
    else let (n',m',d',s') =
        if d = 1 then
            if n = 0 or n = 1 then (0,1,-1,s)
            else (n-1,0,1,(n,-1)::s)
        else let (a,b)::r = s in
            if (b = -1) then (a-2,0,1,(a,m)::r)
            else (0,m+b,-1,r)
        in fib_aux(n',m',d',s');;

let fib_lin n = fib_aux(n,0,1,[]);;
```

The above code has been obtained, essentially, by linearizing the tree of the recursive calls that `fib` generates. The function `fib_aux`, which is linear tail recursive, manipulates the following arguments:

- an integer `n`, which represents a question of the form "how much is `fib(n)`?";
- an integer `m`, which is an answer to a question as above;
- a bit `d` telling us which between the question and the answer is currently meaningful: `1` indicates that we are asking for the value of `fib(n)`, `-1` that we are returning the answer `m`;
- a list `s` which represents the call stack, made of pairs of integers `(a,m)` where `a` is a question and `m` an answer, or `-1` if we do not yet have an answer (because we are in between two recursive calls, *i.e.*, we computed `fib(a-1)` but are waiting for `fib(a-2)`).

We may call parsimonious programming "higher-order imperative programming": as in usual functional programming, higher-order functions are basic primitives but, unlike functional programming, recursive calls are restricted to being while loops. Retrospectively, this also justifies the nice behavior of the parsimonious λ-calculus in terms of complexity: the size-explosion problem is solved by forcing recursion to be linear.

Such a hybrid programming style may perhaps be of interest as a kind of intermediate language in a compilation process. In this respect, while we do think that the transformation leading from `fib` to `fib_lin` shown above may be automatized, we do not know whether it corresponds to a canonical semantic construction. Oddly, we have not been able to find anything in the existing literature mentioning a systematic study of linearization of recursive calls, at least not in the context of full-fledged functional programming.

Chapter 4

Church Meets Cook and Levin

4.1 The Cook-Levin Theorem

4.1.1 Motivation

The Cook-Levin theorem [Coo71, Lev73] is a central result of structural complexity theory. It states that SAT, *i.e.*, the problem of deciding whether a given propositional formula is satisfiable, is NP-complete. It is actually easy to show the existence of artificially constructed NP-complete problems, *i.e.*, problems built with the sole purpose of being complete. However, the fact that there are *natural* NP-complete problems, *i.e.*, problems that have a practical interest like SAT, is far from immediate and is what makes the Cook-Levin theorem valuable.

Let us give a quick review of the meaning of NP-completeness. First of all, we recall that the class NP, although traditionally defined in terms of non-deterministic Turing machines, is well-known to admit the following alternative definition:

Definition 28 (the class NP) *A problem L is in* NP *just if there exists another problem $L^{\text{wit}} \in \mathsf{P}$ and a polynomial q such that, for all $w \in \{0,1\}^*$, $w \in L$ iff there is $m \leq q(|w|)$ and $w' \in \{0,1\}^m$ such that $(w, w') \in L^{\text{wit}}$.*

The string w' is called a *witness*, or *membership certificate* of w to L; so NP is the class of problems admitting "short" and "quickly verifiable" membership certificates, where "short" and "quick" both mean polynomially long in the size of the instance. This brings to light the importance of NP in everyday life: there are literally hundreds of naturally occurring search problems such that, for a fixed instance w, the number of possible solutions is exponential in $|w|$ (*i.e.*, possible solutions are "short" compared to w) and, given a purported solution, it is easy (*i.e.*, "quick") to check whether this is indeed a solution or not.

The idea of completeness is based on the notion of *many-one reduction* from

a decision problem L to a decision problem L' (which, we remind, are both subsets of $\{0,1\}^*$). This is a total recursive function

$$r : \{0,1\}^* \to \{0,1\}^*$$

such that, for all $w \in \{0,1\}^*$, $w \in L$ iff $r(w) \in L'$. Therefore, modulo r, solving L reduces to solving L'. In the context of complexity theory, where one is interested not only in solving a problem but also in doing it with some degree of efficiency, the nature of r must be restricted further. For NP-completeness, a natural choice is requiring r to be polynomial-time computable, yielding so-called *Karp reductions* [Kar72].

Definition 29 (NP-completeness) *We write $L \leq_{\mathrm{K}} L'$ if there exists a Karp reduction from L to L' and we denote by $L{\downarrow_{\mathrm{K}}}$ the principal ideal of L with respect to $\leq_{\mathrm{K}}$. A problem L_0 is* NP-complete *(under Karp reductions) if*

$$L_0{\downarrow_{\mathrm{K}}} = \mathsf{NP}.$$

In other words, $L_0 \in \mathsf{NP}$ and, for all $L \in \mathsf{NP}$, $L \leq_{\mathrm{K}} L_0$.
In case $\mathsf{NP} \subseteq L_0{\downarrow_{\mathrm{K}}}$, i.e., if $L \leq_{\mathrm{K}} L_0$ for all $L \in \mathsf{NP}$, we say that L_0 is NP-hard.

The path that modern textbooks (*e.g.* [Pap94, Gol08, AB09]) take to prove the Cook-Levin theorem passes through Boolean circuits. Indeed, it is fairly easy to show that CIRCUIT SAT $\leq_{\mathrm{K}}$ SAT, where CIRCUIT SAT is the following problem (which is obviously in NP, just as it is obvious that SAT $\in$ NP): given a Boolean circuit C with n inputs and one output, decide whether there exist $b_1, \ldots, b_n \in \{0,1\}$ such that $C(b_1, \ldots, b_n) = 1$; the size of the instance is the number of gates of C. Essentially, a Boolean circuit with one output is a propositional formula which allows "sharing" of subformulas, so it is a generalization of SAT. The above-mentioned reduction shows that, with respect to complexity, the generalization is only apparent.

At this point, the heart of the Cook-Levin theorem is showing that all NP problems reduce to CIRCUIT SAT. The intuition is the following. Let $L \in \mathsf{NP}$ and let L^{wit} be its associated witness language, with q the polynomial bounding the length of witnesses. We know that there is a deterministic Turing machine M running in polynomial time p such that, for all $w \in \{0,1\}^*$ and $w' \in \{0,1\}^{q(|w|)}$, $M(w, w') = 1$ iff w' is a membership certificate of w to L. Now, for a fixed pair of inputs (w, w'), if the running time of M is $t(w, w')$ and the space used is $s(w, w')$, the whole execution of M on input (w, w') may be represented by a $t(w, w') \times s(w, w')$ matrix $\delta_{w,w'}$ such that $\delta_{w,w'}(i, j) = (s, q)$ where s is the symbol contained at position j at the i-th step and q is the state of the machine at step i if the head was at position j at that time, or $\perp$ otherwise. In other words, $\delta_{w,w'}$ is the "space-time" of the execution of M on (w, w').[1]

The fundamental observation now is that, for all i, j, $\delta_{w,w'}(i+1, j)$ only depends on $\delta_{w,w'}(i, j-1)$, $\delta_{w,w'}(i, j)$ and $\delta_{w,w'}(i, j+1)$, *i.e.*, if Σ is the alphabet

[1] Observe that our *execution envelope* of Sect. 3.3.6 is meant to be the corresponding notion, for a higher-order program, of this idea of "space-time" of a Turing machine.

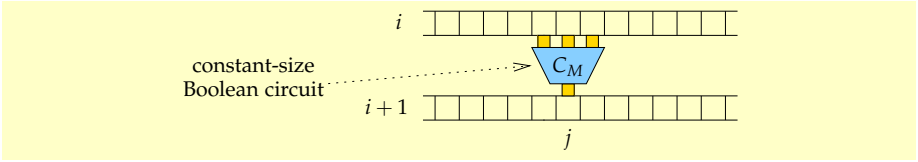


Figure 4.1: The essence of the Cook-Levin theorem: "computation is local".

of M and Q its set of states (assumed not to include $\perp$), there is a function

$$f_M : (\Sigma \times (Q \cup \{\perp\}))^3 \longrightarrow \Sigma \times (Q \cup \{\perp\})$$

such that

$$\delta_{w,w'}(i+1,j) = f_M(\delta_{w,w'}(i,j-1), \delta_{w,w'}(i,j), \delta_{w,w'}(i,j+1)).$$

In other words, as complexity theorists often say, "computation is local". Furthermore, the function f_M itself does not depend on any of w, w', i, j, but only on the transition table of M, *i.e.*, we might add that "computation is isotropic". Therefore, if c is the least integer such that $2^c \geq |\Sigma|(|Q|+1)$, the function f may be encoded by a constant (*i.e.*, depending only on M) Boolean circuit C_M with $3c$ inputs and c outputs (see Fig. 4.1).

By the above discussion, one step of M on a tape of length s may be implemented by a constant-depth circuit C_M^s consisting of s copies of C_M, side by side (with adjacent copies sharing some inputs), and a t-step computation of M may be simulated by a circuit $C_M^{t,s}$ composed of t copies of C_M^s stacked one on top of the other. The size of $C_M^{t,s}$ is thus $k \cdot t \cdot s$, where k is the size of C_M.

We may therefore define a reduction from L to CIRCUIT SAT as follows: on input $w \in \{0,1\}$, let $m := p(|w| + q(|w|))$ and:

1. compute $C_M^{m,m}$;
2. fix its inputs so that they represent the initial configuration with w on the tape, leaving only $q(|w|)$ inputs where w' would be plugged in;
3. compose the obtained circuit with a circuit with m inputs and one output which extracts the answer of M from the output of $C_M^{m,m}$.

By construction, the resulting circuit is satisfiable iff $w \in L$. Moreover, such a circuit may obviously be computed in polynomial time in $|w|$, because $C_M^{m,m}$ is a polynomially big (more precisely, of size $O(p(|w| + q(|w|))^2)$) repetition of a constant circuit, and the procedures (2) and (3) are obviously linear in m.

Although intuitively clear, the above proof sketch hides a host of technical details: how is C_M defined? How are the circuits at steps (2) and (3) of the reduction defined? In the definition of C_M^s, what happens at the "edges" of the array? Then, of course, several standardization assumptions on M must be made in order for the idea to work at all, the most important ones being that:

- the running time of M does not depend on the input but only on its size;

- the length of witnesses for instances of size n is always exactly $q(n)$.

To simplify the construction, some authors (*e.g.* [AB09]) make the further assumption that M is *oblivious*, *i.e.*, that also the position of the head at step i does not depend on the input but only on its length. Of course, the proof that all these assumptions may be made "for free", albeit unproblematic, must be taken into account in the global complexity of the proof of the Cook-Levin theorem.

The point, of course, is not that some important insight is lurking in the missing details or, worse, that some mistake may be hiding in them. No-one doubts the correctness of the above proof and no-one claims that some deep technique must be developed in order to fill in the details. However, this may leave us wondering:

1. if the low-level details are so unimportant, maybe it is because there is a proof in which those details are simply not needed? Or maybe the details are still there, but under a form which is more easily amenable to formalization?

2. Ignoring the details, the essence of the Cook-Levin theorem emerges as being the idea that "computation is local". This is an informal statement; can it be given a more mathematically precise meaning?

It turns out that the techniques developed in Chapters 2 and 3 are also useful to address the above two questions. In [Maz16] we gave a proof of the Cook-Levin theorem taking the parsimonious λ-calculus, instead of Turing machines, as the underlying model of computation. The key points are:

- the "locality" of computation becomes the quantitative continuity property expressed by Proposition 26: if

$$M \to^{l} N \quad \text{and} \quad u \sqsubset N,$$

  there exists $t \sqsubset M$ such that

$$t \to^{*} u \quad \text{and} \quad \mathrm{rk}(t) \leq \mathrm{rk}(u) + l.$$

- The "higher-order" version of the Cook-Levin theorem is then implied by Theorem 35: a language in P admits a polystep parsimonious term M deciding it, which in turn yields a family of affine polyadic terms, or "higher-order circuits". This family is uniform, which means that one may build a Karp reduction from it.

- In this way, one obtains the NP-completeness of a higher-order version of CIRCUIT SAT, let us call it HO CIRCUIT SAT: given an affine polyadic term $t : \mathsf{Bool}^{\otimes n} \multimap \mathsf{Bool}$, determine whether there exist inputs $b_1, \ldots, b_n \in \{0,1\}$ such that $t(\underline{b}_1 \otimes \cdots \otimes \underline{b}_n) \to^{*} \underline{1}$. To complete the proof, one must then show that HO CIRCUIT SAT $\leq_{\mathrm{K}}$ CIRCUIT SAT. In [Maz16], this is done via proof nets: using ideas similar to those of [Ter04], one "compiles" a proof net into an equivalent Boolean circuit.

The proof presented in [Maz16] is already of significance with respect to question (2) above: compared to the original proof, it certainly gives a more satisfactory meaning to the "locality of computation". However, it falls short of addressing question (1): the compilation of higher-order circuits in first-order circuits is still full of technical details of the same nature as those of the original proof.

The latter reason pushes us to develop here a different proof, which does not take the "higher-order route": we work directly with a first-order programming language, with basically nothing more than while loops, which plays the role of Turing machines; then, we apply our techniques to such a language, using directly Boolean circuits to approximate programs and thus avoiding the higher-order-to-first-order compilation. This also offers us a further example of how far the abstract construction of Chapter 2 can go: thanks to it, we will be able to see a Boolean circuit (in the literal sense!) as an intersection types derivation for a language which, after all, is not too far from Turing machines.

The resulting proof is not bureaucracy-free: there are still plenty of technical, uninteresting details which are swept under the rag of lemmas proved by "a straightforward induction".[2] However, we point out that these technical details now concern term syntaxes and type systems which are much more amenable to formalization (for instance, in a proof assistant) than those of the Turing-machine-based proof. In this sense, we are dealing with question (1) in a much more satisfactory way than before. At the same time, we stress that ours is not really a "new" proof of the Cook-Levin theorem: it is merely a different presentation of the original one, giving a (hopefully) mathematically clearer meaning to its essence and making the details less low-level.

A final note: although the influence of the techniques developed in the previous chapter is obvious (and will be explicitly pointed out at times), in the following presentation we refrain from explicitly using any "advanced" notion, and try to present the proof so that an undergraduate student with basic knowledge in programming languages and type systems should be able to follow it.

4.1.2 A minimalist programming language

In Fig. 4.2 we introduce the programming language that we will use to play the role of Turing machines. We call it Mowl, for "monoidal while language". Indeed, it should be seen as the presentation of a symmetric monoidal category with a Boolean object, a binary string object, and the ability to write recursive programs. Since we do not need multiple outputs, we present it in operadic style, although we do not introduce reduction terms explicitly (for the sake of maintaining an introductory level, as mentioned above).

The other significant change with respect to a monoidal framework is that the operad is actually cartesian. This is because, even in a strictly monoidal

[2] We may have overused the word "straightforward" in this document, but we never abused its meaning: it is usually understood that a proof by induction is straightforward when it unravels without any unexpected complication and the "obvious" way of doing things is the way that works.

Types: $\sigma, \tau ::= \mathsf{Bool} \mid \mathsf{Str}$

Terms:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}\ \mathsf{var} \qquad \frac{\Gamma \vdash N : \sigma \quad \Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash M[x \leftarrow N] : \tau}\ \mathsf{share}$$

$$\frac{}{\Gamma \vdash \mathsf{b} : \mathsf{Bool}}\ \mathsf{bool}\ \mathsf{b}\in\{0,1\} \qquad \frac{\Gamma \vdash P : \mathsf{Bool} \quad \Gamma \vdash M : \sigma \quad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathsf{if}\ P\ \mathsf{then}\ M\ \mathsf{else}\ N : \sigma}\ \mathsf{if\ then\ else}$$

$$\frac{}{\Gamma \vdash \varepsilon : \mathsf{Str}}\ \mathsf{empty} \qquad \frac{\Gamma \vdash M : \mathsf{Str}}{\Gamma \vdash \mathsf{b}M : \mathsf{Str}}\ \mathsf{succ}\ \mathsf{b}\in\{0,1\}$$

$$\frac{\Gamma \vdash Q : \mathsf{Str} \quad \Gamma \vdash M : \sigma \quad \Gamma, x : \mathsf{Str} \vdash N : \sigma \quad \Gamma, x : \mathsf{Str} \vdash P : \sigma}{\Gamma \vdash \mathsf{case}\ Q\ \mathsf{of}\ \varepsilon.M \mid 0x.N \mid 1x.P : \sigma}\ \mathsf{case}$$

$$\frac{\Gamma, x : \sigma, y : \tau \vdash P : \tau \quad \Gamma, x : \sigma \vdash M : \sigma \quad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathsf{while}\ y.P\ \mathsf{do}\ M\ \mathsf{to}\ x := N : \tau}\ \mathsf{while}$$

Figure 4.2: The language Mowl.

(*i.e.*, linear) framework, Booleans and strings are duplicable and erasable, as are all inductive datatypes. For this reason, we adopt the more liberal cartesian syntax, but include a sharing construct which implements explicit duplication of Booleans and strings.

The evaluation rules are given in Fig. 4.3. They are completely standard, except the evaluation of while, which is perhaps a bit different from what one would expect. A more standard formulation would be to include a construct

$$\mathsf{while}\ P\ \mathsf{do}\ M\ \mathsf{to}\ x := N$$

with P : Bool and a reduction rule

$$\mathsf{while}\ P\ \mathsf{do}\ M\ \mathsf{to}\ x := v\ \rightarrow\ \mathsf{if}\ P\{v/x\}\ \mathsf{then}\ (\mathsf{while}\ P\ \mathsf{do}\ M\ \mathsf{to}\ x := M\{v/x\})\ \mathsf{else}\ v,$$

which unfolds the while loop in the usual way. Our formulation has the advantage of not requiring any Boolean primitive, showing that the type Bool is only a commodity and that Mowl makes sense even when restricted only to the Str type.

It is more or less obvious that Mowl is Turing-complete and, more importantly, that its evaluation rules may simulate and may be simulated by Turing machines with a polynomial slowdown:

Proposition 39 NP *is the class of decision problems* $L \subseteq \{0,1\}^*$ *such that there exists a* Mowl *program*

$$x : \mathsf{Str}, y : \mathsf{Str} \vdash M : \mathsf{Bool}$$

Values: $v ::= 0 \mid 1 \mid \varepsilon \mid 0v \mid 1v$.
For $w \in \{0,1\}^*$, we write $\underline{w}$ for the obvious value of type Str representing w.

Substitution contexts: $[-] ::= \{\cdot\} \mid [-][x \leftarrow u]$.

Evaluation rules:

$$
\begin{array}{rcl}
M[x \leftarrow v[-]] & \to & M\{v/x\}[-] \qquad v \text{ value} \\
\mathsf{if}\ \mathsf{b}[-]\ \mathsf{then}\ M\ \mathsf{else}\ N & \to & \begin{cases} M[-] & \text{if } \mathsf{b} = 1 \\ N[-] & \text{if } \mathsf{b} = 0 \end{cases} \\
\mathsf{case}\ \underline{w}[-]\ \mathsf{of}\ \varepsilon.M \mid 0x.N \mid 1x.P & \to & \begin{cases} M[-] & \text{if } w = \varepsilon \\ N\{\underline{w}'/x\}[-] & \text{if } w = 0w' \\ P\{\underline{w}'/x\}[-] & \text{if } w = 1w' \end{cases} \\
\mathsf{while}\ y.P\ \mathsf{do}\ M\ \mathsf{to}\ x := (v[-]) & \to & \\
\multicolumn{3}{c}{P\{v/x\}[y \leftarrow \mathsf{while}\ y.P\ \mathsf{do}\ M\ \mathsf{to}\ x := M\{v/x\}][-]}
\end{array}
$$

Figure 4.3: The operational semantics of Mowl.

and two polynomials p, q such that, for all $w, w' \in \{0,1\}^*$

$$M[x \leftarrow \underline{w}][y \leftarrow \underline{w}'] \to^{l(w,w')} \mathsf{b}_{w,w'}$$

with $l(w,w') \leq p(|w| + |w'|)$ and, moreover, there exists $m \leq q(|w|)$ and $w' \in \{0,1\}^m$ such that $\mathsf{b}_{w,w'} = 1$ iff $w \in L$.

In the sequel, we will take Proposition 39 as our definition of NP.

4.1.3 Boolean circuits

The most common definition of Boolean circuit in the setting of complexity theory uses the binary and and or gates and the (unary) not gate. There would be no problem in adopting that definition in our setting, but we prefer to use if then else as a primitive because it is closer to the spirit of programming languages and it is more minimalist. Of course, this changes nothing from the point of view of complexity: one may translate between our circuits and circuits on the basis $\{\wedge^2, \vee^2, \neg\}$ with a constant overhead in size.

Our formal definition of Boolean circuits is given in Fig. 4.4. For the same reasons mentioned in the case of Mowl, we adopt a cartesian syntax, although circuits really form what some would call a PROP (a symmetric monoidal category in which every object is a tensor power of a single generating object, Bool in our case). Of course, we include a sharing construct, which is fundamental in circuits. Contrarily to Mowl, we do need circuits with more than one output, which is why we include n-ary tensors $\langle t_1, \ldots, t_n \rangle$ and their associated destructor $t[\langle x_1, \ldots, x_n \rangle := u]$, in perfect analogy with polyadic calculi.

The syntax also includes a special symbol $\bullet$, which may only appear in

Types: $A, B ::= \mathsf{Bool}^n \quad n \in \mathbb{N}$

Terms:

$$\frac{}{\Gamma, x : \mathsf{Bool} \vdash x : \mathsf{Bool}}\ \mathsf{var} \qquad \frac{}{\Gamma \vdash_\bullet \bullet : A}\ \mathsf{undef} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash_\bullet t : A}\ \bullet$$

$$\frac{\Gamma \vdash_\bullet u : \mathsf{Bool} \quad \Gamma, x : \mathsf{Bool} \vdash t : A}{\Gamma \vdash t[x \leftarrow u] : A}\ \mathsf{share} \qquad \frac{\Gamma \vdash t_1 : \mathsf{Bool} \quad \dots \quad \Gamma \vdash t_n : \mathsf{Bool}}{\Gamma \vdash \langle t_1, \dots, t_n \rangle : \mathsf{Bool}^n}\ \mathsf{box}$$

$$\frac{\Gamma \vdash u : \mathsf{Bool}^n \quad \Gamma, x_1 : \mathsf{Bool}, \dots, x_n : \mathsf{Bool} \vdash t : A}{\Gamma \vdash t[\langle x_1, \dots, x_n \rangle := u] : A}\ \mathsf{let}$$

$$\frac{}{\Gamma \vdash \mathsf{b} : \mathsf{Bool}}\ \mathsf{bool}\ \mathsf{b} \in \{0,1\} \qquad \frac{\Gamma \vdash p : \mathsf{Bool} \quad \Gamma \vdash_\bullet t : A \quad \Gamma \vdash_\bullet u : A}{\Gamma \vdash \mathsf{if}\ p\ \mathsf{then}\ t\ \mathsf{else}\ u : A}\ \mathsf{if\ then\ else}$$

Figure 4.4: Boolean circuits.

Substitution contexts: $[-] ::= \{\cdot\} \mid [-][x \leftarrow u] \mid [-][\langle \overline{x} \rangle := u]$.

Evaluation rules:

$$\begin{aligned}
t[\langle x_1, \dots, x_n \rangle := \langle u_1, \dots, u_n \rangle [-]] &\to t\{u_1, \dots, u_n / x_1, \dots, x_n\}[-] \\
t[x \leftarrow \mathsf{b}[-]] &\to t\{\mathsf{b}/x\}[-] \qquad \mathsf{b} \in \{0,1\} \\
\mathsf{if}\ \mathsf{b}[-]\ \mathsf{then}\ t\ \mathsf{else}\ u &\to \begin{cases} t[-] & \text{if } \mathsf{b} = 1,\ t \neq \bullet \\ u[-] & \text{if } \mathsf{b} = 0,\ u \neq \bullet \end{cases}
\end{aligned}$$

Figure 4.5: Evaluation rules for Boolean circuits.

certain places, namely in sharing constructs $t[x \leftarrow \bullet]$ and as a branch of an if then else statement. This symbol may be treated as a free variable impervious to substitution; it is necessary in the definition of approximation order (to be given below), but may otherwise be ignored.

The evaluation rules for Boolean circuits are given in Fig. 4.5 and are completely standard. The only non-standard point is the symbol $\bullet$, whose presence blocks evaluation. Indeed, as the next definition will show, $\bullet$ stands for an unknown circuit.

Now we start preparing in view of the Cook-Levin theorem.

Definition 30 (approximation order) *The* approximation order *on Boolean circuits, denoted by $\sqsubseteq$, is defined by the rules of Fig. 4.6.*

Although we chose to keep advanced notions out of the presentation, the reader who remembers Chapter 1 will read in Figures 4.4 and 4.5 the presentation of a 2-operad. In this viewpoint, the above definition should be given at the level of 2-arrows, so that one obtains a **DblPos**-operad and, in fact,

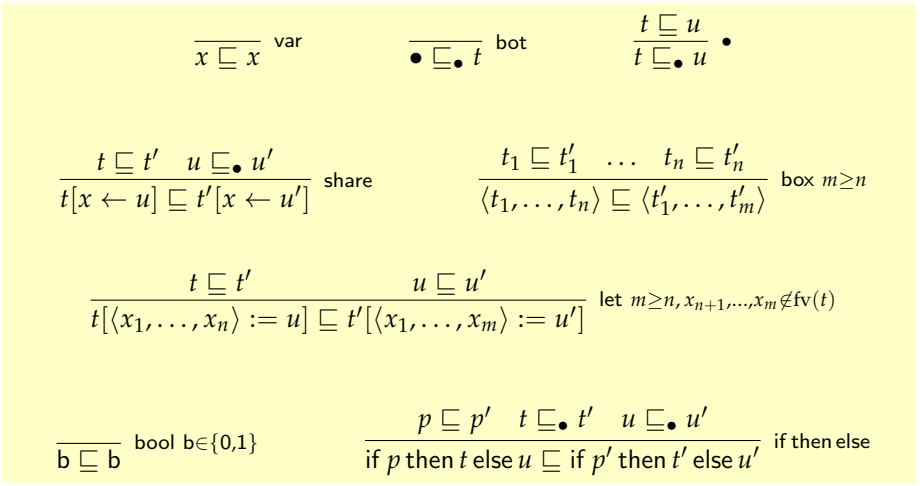


Figure 4.6: Approximation order on Boolean circuits.

a **MntDblPos**-operad, to which the ideal completion of Sect. 1.3.3 may be applied. Here, we will content ourselves whit the following key result:

Lemma 40 (monotonicity) *If $t \to u$ and $t \sqsubseteq t'$, then $t' \to u'$ such that $u \sqsubseteq u'$.*

Proof. A straightforward induction on t. □

The next result is not needed for the Cook-Levin theorem but we mention it anyway because it may be used to give a simple, conceptual proof of the classic Theorem 24 (Sect. 3.1.2).

Lemma 41 (compatible suprema) *Let $t, t' \sqsubseteq u$. Then their supremum $t \sqcup t'$ exists.*

Proof. One defines $t \sqcup t'$ (as well as $t \sqcup_\bullet t'$) by induction on u, in the obvious way, and checks that it is indeed the least upper bound. □

4.1.4 Intersection types, again

We now introduce the main technical tool for the proof of the Cook-Levin theorem. We already mentioned that Lemma 40 is actually the shadow of a stronger result stating that the **DblPos**-operad of Boolean circuits is monotonic. At this point, the reader will have guessed that the 2-operad presenting Mowl embeds in the ideal completion of Boolean circuits much in the same way as linear logic embeds in the ideal completion of the affine polyadic calculus.

From the above-mentioned embedding, one may infer an approximation relation $t \sqsubset M$ between Boolean circuits and Mowl programs (actually, between reductions of the former and reductions of the latter). On types, we have the approximation relations

$$\mathsf{Bool} \sqsubset \mathsf{Bool} \qquad \mathsf{Str}_n \sqsubset \mathsf{Str}$$

$$\frac{m \geq n}{\Gamma, x : \mathsf{Str}_m \vdash_{\langle \overline{x} \rangle} x : \mathsf{Str}_n}\ \text{var} \qquad \frac{\Gamma \vdash_u N : \mathsf{Str}_n \quad \Gamma, x : \mathsf{Str}_n \vdash_t M : A}{\Gamma \vdash_{t[\overline{x} \Leftarrow u]} M[x \leftarrow N] : A}\ \text{share}$$

$$\frac{}{\Gamma \vdash_{\langle 0,0,\ldots,0,0 \rangle} \varepsilon : \mathsf{Str}_n}\ \text{empty} \qquad \frac{\Gamma \vdash_t M : \mathsf{Str}_n}{\Gamma \vdash_{\langle 1,\mathsf{b},\overline{x} \rangle [\langle \overline{x} \rangle := t]} \mathsf{b}M : \mathsf{Str}_{n+1}}\ \text{succ } \mathsf{b} \in \{0,1\}$$

$$\frac{\Gamma \vdash_q Q : \mathsf{Str}_{n+1} \qquad \begin{array}{r} [\Gamma \vdash_t M : A] \\ [\Gamma, x : \mathsf{Str}_n \vdash_u N : A] \\ [\Gamma, x : \mathsf{Str}_n \vdash_p P : A] \end{array}}{\Gamma \vdash_{\text{if } x'_0 \text{ then (if } x_0 \text{ then } p \text{ else } u) \text{ else } t} \mathsf{case}\, Q\, \mathsf{of}\, \varepsilon.M \mid 0x.N \mid 1x.P : A}\ \text{case}$$

$$\frac{\begin{array}{r} \ldots \Gamma, x : A_i, y : B_{i+1} \vdash_{p_i} P : B_i \ldots \\ \ldots \Gamma, x : A_{j-1} \vdash_{t_i} M : A_j \ldots \end{array} \quad \Gamma \vdash_u N : A_0}{\Gamma \vdash_w \mathsf{while}\, y.P\, \mathsf{do}\, M\, \mathsf{to}\, x := N : B_0}\ \text{while} \quad 0 \leq i \leq k,\ 0 < j \leq k$$

where $w = p_0[\overline{y} \Leftarrow p_1[\overline{y} \Leftarrow \ldots p_k[\overline{y} \Leftarrow \bullet][\overline{x} \Leftarrow t_k] \ldots][\overline{x} \Leftarrow t_1]][\overline{x} \Leftarrow u]$

Figure 4.7: Intersection types for Mowl. In the case rule, the premises in brackets are not required to be present.

where $\mathsf{Str}_n := \mathsf{Bool}^{\otimes(2n)}$, $n \geq 1$. The reason why Str_n is defined in this way will be explained momentarily.

At this point, we switch on the machinery of Chapter 2: we define an approximation presheaf, we apply the Grothendieck construction, and we get an intersection types system for Mowl programs, whose proofs are isomorphic to Boolean circuits.

The type system in question is show in Fig. 4.7, ignoring the gray annotations, whose meaning will be explained shortly. For brevity, we only show it for the fragment of Mowl restricted to Str. As observed above, the language keeps being meaningful and Turing-complete with this restriction, so this is the non-trivial part of the type system. Indeed, Bool is approximated by itself and the typing rules for the Boolean constructs of Mowl for intersection types are identical to those for the "simple types" of Fig. 4.2.

It is important to observe that, strictly speaking, our proof of the Cook-Levin theorem does not need the machinery of Chapter 2, in the sense that it does not directly invoke any of the theorems shown therein. For the sake of the proof, the type system of Fig. 4.7 "fell from the sky" and its properties are proved independently of the results of Chapter 2.

Let us explain the meaning of the approximation $\mathsf{Bool}^{\otimes(2n)} \sqsubset \mathsf{Str}$. This is where we subtly insert the low-level coding required for circuits to simulate programs (in some sense, the coding necessary for the definition of the circuit C_M of Fig. 4.1). In the limit, a binary string w (of type Str in Mowl) of arbitrary

length is encoded by an infinite stream of Booleans. However, since strings are finite, we need to know where the string actually ends, which is why we use two bits to encode one bit:

$$w'_1, w_1, w'_2, w_2, w'_3, w_3, \dots$$

If $w'_i = 1$, then w_i is meaningful and contains the value of the i-th bit of w; otherwise, it marks the end of the string, regardless of what comes next. The finite approximations of type $\mathsf{Str}_n = \mathsf{Bool}^{\otimes(2n)}$ are therefore able to encode strings of length at most $n-1$, which is why we require $n \geq 1$ (we always need at least one pair containing the end-of-string marker).

In the above perspective, for facilitating the correspondence between derivations in intersection types and Boolean circuits, we associate with each variable $x : \mathsf{Str}$ of Mowl a fixed sequence of Boolean variables $x'_0, x_0, x'_1, x_1, x'_2, x_2, \dots$. Such a sequence, or a suitable initial prefix of it (the length of which will be explicitly given if not clear from the context) will be denoted by $\overline{x}$ (this is similar to the notion of "supervariable" of Sect. 1.2.3).

In the following, we use the notation

$$t[\overline{x} \Leftarrow u] := \begin{cases} t[\overline{x} \leftarrow \overline{y}][\langle \overline{y} \rangle := u] & \text{if } u \neq \bullet \\ t[\overline{x} \leftarrow \overline{\bullet}] & \text{otherwise} \end{cases}$$

where by $t[\overline{x} \leftarrow \overline{v}]$ we mean $t[x'_0 \leftarrow v'_0][x_0 \leftarrow v_0] \dots [x'_n \leftarrow v'_n][x_n \leftarrow v_n]$.

Definition 31 (underlying approximation) *We associate with each derivation δ of the system of Fig. 4.7 a Boolean circuit, denoted by δ^-, which is defined inductively. The definition is given in Fig. 4.7 itself, as the gray annotation t in the judgment $\Gamma \vdash_t M : A$, with the following indications:*

- *in the* var, share *and* succ *rule, $\overline{x}$ is of length $2n$;*
- *in the* empty *rule, there are $2n$ occurrences of 0;*
- *in the* while *rule, the lengths of the various occurrences of $\overline{x}$ and $\overline{y}$ are determined in each case by the various types A_i and B_i.*

Of course, the above definition would be automatic (and pointless) if we had introduced the approximation relation $t \sqsubset M$ and presented the type system directly in terms of the Grothendieck construction. With our current approach, it becomes necessary to give it explicitly.

We now show the main properties of the type system. The first is a quantitative form of subject expansion, which is the type-theoretic version of quantitative continuity.

Definition 32 (rank and width) *Let δ be an intersection types derivation. Its* rank, *denoted by $\mathrm{rk}(\delta)$, is the maximum k such that a* while *rule with arity $2k+2$ appears in δ. Its* type width, *denoted by $\mathrm{tw}(\delta)$, is the maximum m such that the type Str_m appears in δ.*

Lemma 42 (quantitative subject expansion) *Let δ be an intersection types derivation of $\Gamma \vdash M : A$ and let*

$$M' \to M.$$

Then, there exists a derivation δ' of $\Gamma \vdash M' : A$ such that

$$(\delta')^- \to^* \delta^-.$$

Moreover, $\mathrm{rk}(\delta') \leq \mathrm{rk}(\delta) + 1$ *and* $\mathrm{tw}(\delta') \leq \mathrm{tw}(\delta) + 1$.

Proof. The proof follows a rather classic pattern. One first proves the following auxiliary claim: for all Γ, A, N and $w \in \{0,1\}^*$, if $\Gamma \vdash N\{\underline{w}/x\} : A$ is derivable, then there exists B such that $\Gamma \vdash \underline{w} : B$ and $\Gamma, x : B \vdash N : A$ are derivable, with rank and width bounded by those of the original derivation. This is shown by induction on N.

The proof then proceeds by induction on the context C such that $M' = \mathsf{C}\{M'_0\}$ and $M = \mathsf{C}\{M_0\}$ with $M'_0 \to M_0$ and M'_0 a redex. The only really interesting case is $\mathsf{C} = \{\cdot\}$, in which there are three subcases, corresponding to the possible redexes. The verification, although somewhat lengthy, is completely unproblematic. The above claim is used in every case; subtyping (*i.e.*, the fact that a declaration $x : \mathsf{Str}_n$ in a typing context may always be replaced with $x : \mathsf{Str}_m$ for any $m \geq n$, without affecting derivability) is used in the while case. For what concerns the quantitative part, the rank increases in the case of while reductions, whereas the width increases for case reductions. □

The second main property concerns the existence of *uniform derivations*, which may be efficiently computed. These are to our proof what the circuits $C_M^{t,s}$ are to the original proof of the Cook-Levin theorem.

Definition 33 (uniform typings) *Let M be a* Mowl *program, let*

$$\Gamma = x_1 : \mathsf{Str}_{n_1}, \ldots, x_p : \mathsf{Str}_{n_p}$$

be a context such that $\mathrm{fv}(M) \subseteq \{x_1, \ldots, x_p\}$ *and let $k, m \in \mathbb{N}$ with m at least equal to the number of binary successors in M (i.e., the number of* succ *rules in its typing derivation, Fig. 4.2). The* uniform typing *of M in context Γ of rank k and width m, denoted by $\lfloor M \rfloor_{k,m}^{\Gamma}$, is an intersection types derivation of the judgment*

$$\Gamma \vdash M : \mathsf{Str}_m$$

defined inductively as follows:

- $\lfloor x \rfloor_{k,m}^{\Gamma, x:\mathsf{Str}_n}$:

$$\frac{}{\Gamma, x : \mathsf{Str}_n \vdash x : \mathsf{Str}_m}\ \mathsf{var}$$

- $\lfloor M[x \leftarrow N] \rfloor^{\Gamma}_{k,m}$:

$$\dfrac{\begin{array}{cc}\vdots\ \lfloor N \rfloor^{\Gamma}_{m,k} & \vdots\ \lfloor M \rfloor^{\Gamma, x:\mathsf{Str}_m}_{m,k} \\ \Gamma \vdash N : \mathsf{Str}_m & \Gamma, x : \mathsf{Str}_m \vdash M : \mathsf{Str}_m\end{array}}{\Gamma \vdash M[x \leftarrow N] : \mathsf{Str}_m}\ \mathsf{share}$$

- $\lfloor \varepsilon \rfloor^{\Gamma}_{k,m}$:

$$\dfrac{}{\Gamma \vdash \varepsilon : \mathsf{Str}_m}\ \mathsf{empty}$$

- $\lfloor \mathsf{b}M \rfloor^{\Gamma}_{k,m}$:

$$\dfrac{\begin{array}{c}\vdots\ \lfloor M \rfloor^{\Gamma}_{k,m-1} \\ \Gamma \vdash M : \mathsf{Str}_{m-1}\end{array}}{\Gamma \vdash \mathsf{b}M : \mathsf{Str}_m}\ \mathsf{succ}$$

- $\lfloor \mathsf{case}\, Q\, \mathsf{of}\, \varepsilon.M \mid 0x.N \mid 1x.P \rfloor^{\Gamma}_{k,m}$:

$$\dfrac{\begin{array}{cccc}\vdots\ \lfloor Q \rfloor^{\Gamma}_{k,m+1} & \vdots\ \lfloor M \rfloor^{\Gamma}_{k,m} & \vdots\ \lfloor N \rfloor^{\Gamma, x:\mathsf{Str}_m}_{k,m} & \vdots\ \lfloor P \rfloor^{\Gamma, x:\mathsf{Str}_m}_{k,m} \\ \Gamma \vdash Q : \mathsf{Str}_{m+1} & \Gamma \vdash M : \mathsf{Str}_m & \Gamma, x : \mathsf{Str}_m \vdash N : \mathsf{Str}_m & \Gamma, x : \mathsf{Str}_m \vdash P : \mathsf{Str}_m\end{array}}{\Gamma \vdash \mathsf{case}\, Q\, \mathsf{of}\, \varepsilon.M \mid 0x.N \mid 1x.P : \mathsf{Str}_m}\ \mathsf{case}$$

- $\lfloor \mathsf{while}\, P\, \mathsf{do}\, M\, \mathsf{to}\, x := N \rfloor^{\Gamma}_{k,m}$:

$$\dfrac{\begin{array}{ccccc}\vdots\ \lfloor P \rfloor^{\Gamma, x:\mathsf{Str}_m, y:\mathsf{Str}_m}_{k,m} & & \vdots\ \lfloor M \rfloor^{\Gamma, x:\mathsf{Str}_m}_{k,m} & & \vdots\ \lfloor N \rfloor^{\Gamma}_{k,m} \\ \Gamma, x : \mathsf{Str}_m, y : \mathsf{Str}_m \vdash P : \mathsf{Str}_m & \ldots & \Gamma, x : \mathsf{Str}_m \vdash M : \mathsf{Str}_m & & \Gamma \vdash N : \mathsf{Str}_m\end{array}}{\Gamma \vdash \mathsf{while}\, P\, \mathsf{do}\, M\, \mathsf{to}\, x := N : \mathsf{Str}_m}\ \mathsf{while}$$

  with $k+1$ premises typing P and k typing M. The uniformity is in the fact these premises all come from the same two derivations, $\lfloor P \rfloor^{\Gamma, x:\mathsf{Str}_m, y:\mathsf{Str}_m}_{k,m}$ and $\lfloor M \rfloor^{\Gamma, x:\mathsf{Str}_m}_{k,m}$, respectively.

It is very important to observe the following: the above definition, which works for all k and sufficiently large m, tells us that *every* Mowl program is typable in intersection types. This should sound as extremely troublesome to the reader acquainted with intersection types, because usually typability implies normalization, and obviously not every Mowl program terminates. Luckily, something goes wrong here in the usual soundness proof: the intersection types system of Fig. 4.7 does *not* verify subject *reduction*; therefore, the fact that every Mowl program is typable has no consequence concerning termination.

Lemma 43 *For a fixed* Mowl *program M, the Boolean circuit $(\lfloor M \rfloor^{\Gamma}_{k,m})^-$ may be computed in polynomial time in k and m.*

Proof. First of all, one proves by induction on M that, if d is the number of nested while constructs found in M, then the size of $(\lfloor M \rfloor^{\Gamma}_{k,m})^-$ is bounded by $|M|(2m(k+1))^d$. Then, we observe that the circuit in question is built from a linear exploration of M, so the size of the circuit bounds the running time.

Since $|M|$ and d are both $O(1)$ with respect to k and m, the result follows. □

Lemma 44 *Let δ be an intersection types derivation of the judgment $\Gamma \vdash M : A$, with M containing c binary successors. Then, for all $k \geq \mathrm{rk}(\delta)$ and $m \geq \mathrm{tw}(\delta) + c$, we have*

$$\delta^- \sqsubseteq (\lfloor M \rfloor_{k,m}^{\Gamma'})^-,$$

where Γ' is Γ in which every type is replaced by Str_m.

Proof. A straightforward induction on δ. □

4.1.5 The proof

We already mentioned that there is a well-known Karp reduction from CIRCUIT SAT to SAT (see for instance [Pap94]), so the Cook-Levin theorem reduces to proving the NP-completeness of CIRCUIT SAT.

Such a proof should start with showing that CIRCUIT SAT $\in$ NP. As obvious as it may be, a formal proof of this fact would require us to write a program for CIRCUIT SAT, prove it correct and prove that it has the right complexity properties. In this respect, using Mowl as programming language is a minuscule improvement over Turing machines, so we leave this to the reader's intuition and concentrate on hardness. Besides, formal program verification and automatic inference of complexity bounds have been and are mainstream research topics in the theory of programming languages, so we are confident that this part of the proof too may be fully "absorbed" into our paradigm.

Theorem 45 (Cook-Levin) CIRCUIT SAT *is* NP*-hard.*

Proof. Let $L \in \mathsf{NP}$; we must build a Karp reduction from L to CIRCUIT SAT. By definition, there are two polynomials p, q and a Mowl program

$$x : \mathsf{Str}, y : \mathsf{Str} \vdash M : \mathsf{Bool}$$

such that, for all $w, w' \in \{0,1\}^*$,

$$M[x \leftarrow \underline{w}][y \leftarrow \underline{w}'] \to^{l(w,w')} \mathsf{b}_{w,w'}$$

with $l(w, w') \leq p(|w| + |w'|)$ and, moreover, $w \in L$ iff there exists $w' \in \{0,1\}^m$ with $m \leq q(|w|)$ such that $\mathsf{b}_{w,w'} = 1$. Obviously,

$$\vdash \mathsf{b}_{w,w'} : \mathsf{Bool}$$

is derivable in intersection types. By quantitative subject expansion (Lemma 42), we have, for all $w, w' \in \{0,1\}^*$, a derivation

$$\begin{array}{c} \vdots\ \varepsilon_{w,w'} \\ \vdash M[x \leftarrow \underline{w}][y \leftarrow \underline{w}'] : \mathsf{Bool} \end{array}$$

and $\mathrm{rk}(\varepsilon_{w,w'}), \mathrm{tw}(\varepsilon_{w,w'})$ are both bounded by $l(w, w')$. The intersection types

system is syntax directed, so from $\varepsilon_{w,w'}$ we infer the existence of derivations

$$\begin{array}{ccc} \vdots\ \delta_{w,w'} & \vdots\ \omega_w & \vdots\ \omega'_{w'} \\ x : A_w, y : B_{w'} \vdash M : \mathsf{Bool} & \vdash \underline{w} : A_w & \vdash \underline{w'} : B_{w'} \end{array}$$

with rank and width still bounded by $l(w, w')$. Let c be the number of binary successors in M, which is $O(1)$ with respect to w and w'. For $n, m \in \mathbb{N}$, let

$$\begin{aligned} r(n,m) &:= \max_{|w|=n,|w'|=m} \mathrm{rk}(\delta_{w,w'}), \\ s(n,m) &:= \max_{|w|=n,|w'|=m} \mathrm{tw}(\delta_{w,w'}) + c. \end{aligned}$$

With $\Gamma_{n,m} := x : \mathsf{Str}_{r(n,m)}, y : \mathsf{Str}_{s(n,m)}$, let

$$\varphi_{n,m} := \lfloor M \rfloor^{\Gamma_{n,m}}_{r(n,m),s(n,m)}.$$

Note that this induces a Boolean circuit

$$\overbrace{\ldots x'_i : \mathsf{Bool}, x_i : \mathsf{Bool}, \ldots}^{2r(n,m)}, \overbrace{\ldots, y'_j : \mathsf{Bool}, y_j : \mathsf{Bool}, \ldots}^{2s(n,m)} \vdash \varphi^-_{n,m} : \mathsf{Bool}.$$

By Lemma 44, $\delta^-_{w,w'} \sqsubseteq \varphi^-_{|w|,|w'|}$ for all $w, w' \in \{0,1\}^*$. Now, remember that Lemma 42 also tells us that

$$\delta^-_{w,w'}[\overline{x} \Leftarrow \omega^-_w][\overline{y} \Leftarrow (\omega'_{w'})^-] \to^* \mathsf{b}_{w,w'},$$

(applying $(\cdot)^-$ to a derivation typing a Boolean constant gives us that same constant). Hence, by Lemma 40, we also have

$$\varphi^-_{|w|,|w'|}[\overline{x} \leftarrow \overline{u}_w][\overline{y} \leftarrow \overline{u}_{w'}] \to^* \mathsf{b}_{w,w'},$$

with $\langle \overline{u}_w \rangle : \mathsf{Bool}^{2r(n,m)}$ and $\langle \overline{u}_{w'} \rangle : \mathsf{Bool}^{2s(n,m)}$ bit representations of w and w'.

We are now ready do define our reduction. For $n \in \mathbb{N}$, let

$$\begin{aligned} r_1(n) &:= r(n, q(n)), \\ s_1(n) &:= s(n, q(n)). \end{aligned}$$

Note that both $r_1(n)$ and $s_1(n)$ are $O(p(n + q(n)))$, *i.e.*, they may be replaced by two polynomials $\widetilde{r}_1$ and $\widetilde{s}_1$ which bound them from above for all $n \geq k$ for some fixed $k \in \mathbb{N}$. Moreover, $\widetilde{r}$, $\widetilde{s}$ and k depend only on M, so we may consider them hard-wired, together with M itself, in our reduction, which is defined follows. On input $w \in \{0,1\}^*$, it outputs the description of

$$\varphi^-_{\widetilde{r}(|w|),\widetilde{s}(|w|)}[\overline{x} \leftarrow \overline{u}_w]$$

which may be done in polynomial time in $|w|$ by Lemma 43. This is a Boolean circuit which, by definition, is satisfiable if and only if $w \in L$. □

4.2 A Glimpse Beyond

4.2.1 What now?

Having reproved the Cook-Levin theorem with logic-related tools might be seen, in itself, as a quite amusing but otherwise futile exercise, at least from the standpoint of complexity theory. As a matter of fact, seeing this new proof will hardly have any impact on one's understanding of NP-completeness, and for a good reason: as we already pointed out, this "new" proof is really the usual one presented in a different language, so there is nothing in it that a complexity theorist did not already know. Of course, one may object that it is always good in principle to look at well-known matters from a new vantage point, as significant insights may come from the least expected direction. But what are we looking for?

Structural complexity theorists are looking for *lower bounds*. They want to be able to prove results of the form: in the model of computation X, problem L needs at least $R(n)$ computational resources to be solved, where n is the size of the problem instance. Some strong results are known when X is a restricted model of computation (*e.g.* decision trees), but as soon as X has a minimum of expressiveness, the question becomes so hard that it is generally considered hopeless. For instance, when $X =$ deterministic Turing machines and $L = \textsc{sat}$, proving a superlinear time lower bound would already be considered a huge breakthrough (yes, super*linear*, not superpolynomial!).

In this respect, logical approaches have proved to be of no use so far. Descriptive complexity [Imm99], based on finite model theory, has been quite successful at characterizing nearly every well-known complexity class. However, it has provided no workable ideas concerning lower bounds. Implicit computational complexity, which is one of the topics we touched upon in this thesis, is situated at the other end of the logical spectrum, at the interface between recursion theory, type theory and the theory of programming languages. This approach too seems to be worthless when it comes to lower bound techniques.

Logical approaches to computational complexity usually focus on capturing complexity classes: the classic result in these fields is that a language belongs to a complexity class C if, and only if, it may be decided by a program of a given form or expressed by a formula of a certain sort. The non-triviality of such results resides in the fact that, of course, neither the program nor the formula make any reference to the complexity bounds defining the class C. Moreover, since the algorithms for solving the problems of C are "normalized" to a given form which is usually of logical nature, one may hope that a new arsenal of tools becomes available for studying C.

However, by inspecting more closely the characterization results described above, one realizes that they are composed of two parts bearing little relevance with the question of lower bounds: a completeness part, which is generally a programming exercise of limited theoretical interest; and a soundness part, which is a *global upper bound* result, *i.e.*, a statement saying that all programs of the given form may be evaluated within the resources allotted by the definition of C. Now, if the question is showing that $H \notin \mathsf{C}$ for some purportedly

hard problem H, we see that such a characterization leaves us completely clueless. If anything, it is the far-reaching conjectures of structural complexity theory that tell us something about these characterizations, not the other way around. For instance, we may safely conjecture that there is no simply-typed parsimonious λ-term of type $\mathsf{Str}[] \multimap \mathsf{Bool}$ deciding SAT on no further ground than the widespread belief that there is no logspace algorithm for SAT, and not because we have any deep intuition about the limits of what simply-typed parsimonious λ-terms may do.

To make things worse, not only do we not gain any intuition on the limits of computation with bounded resources but, more often than not, we actually *lose* the ability to prove that such limits exist! For instance, it is a consequence of the well-known space hierarchy theorem that $\mathsf{L} \subsetneq \mathsf{PSPACE}$; and yet, if we had to show that the quantified Boolean formula problem (the prototypical PSPACE-complete problem) is not decidable by any simply-typed parsimonious term of type $\mathsf{Str}[] \multimap \mathsf{Bool}$, we would have absolutely no clue on how to proceed by logical means. Note that we are not trying to follow some "purity of methods" craze here; we are saying that, if the definition of the class L were "the problems solvable by simply-typed parsimonious terms of type $\mathsf{Str}[] \multimap \mathsf{Bool}$", we would have to resort to a characterization using deterministic logspace-bounded Turing machines in order to infer any non-trivial separation concerning L, while our hope with implicit computational complexity was that such transfers of results would go in the opposite direction.

We must be realistic and admit that, for the time being, there is no hope of making progress in structural complexity theory by use of tools from logic and programming languages theory. This is particularly frustrating because such tools, like types, categories and rewriting, seem to be at least as good as the combinatorial ones in describing computation as a dynamic phenomenon. If anything, having re-proved the Cook-Levin theorem in type-theoretic language serves as an additional witness to this fact.

This motivates us to keep investigating the world of computational complexity with a logical eye, with the goal of finding out whether the logical approach may indeed bear fruits, or gain a clear understanding of why it cannot. In the meantime, we are confident that the technical challenges we will encounter will spur the development of finer tools for the quantitative study of programming languages, which is a field of interest in its own right. The key point is to go beyond the idea of using logic and programming languages solely as a means of characterizing complexity classes. Below we describe a possible research path in this direction.

4.2.2 Projections and the non-uniform perspective

One might say that the Cook-Levin theorem is the mother of all completeness results: after showing the relevance of NP-completeness, and opening the door to the wave of NP-complete problems that were found soon after [Kar72], it also set the stage for investigating the notion of C-completeness for an arbitrary complexity class C, and today we know the existence of non-artificial complete problems for just about any standard complexity class that is known

to have complete problems at all. Investigating this general notion of completeness seems like a natural direction to follow.

A somewhat annoying aspect of completeness is that it is not a fixed notion, independent of complexity classes: for a given complexity class C, we have a possibly quite large number of legitimate notions of many-one reduction to be used in the definition of C-completeness. Basically, any monoid of functions on $\{0,1\}^*$ which does not use more resources than those defining C and which does not trivially give the full power of C itself will do.

For instance, NP-completeness was originally defined in terms of Karp reductions (*i.e.*, polytime computable functions), but it may also be considered relative to logspace computable functions. For P-completeness, the latter are a common choice, while the former are of little interest because any non-trivial problem in P is P-complete under Karp reductions. The same issue appears with logspace reductions if we wish to define L-completeness, for which an even stricter notion of reduction must be used. There would be no problem if these different notions of completeness coincided (*e.g.*, NP-completeness under Karp or logspace reductions), but it is of course wide open whether this is the case or not, except in some limited cases.[3]

This lack of canonicity does not diminish the relevance of completeness: as long as a notion of reduction R is legitimate for a class C which admits a complete problem L_0 relative to R, then, in presence of an inclusion $\mathsf{B} \subseteq \mathsf{C}$ in which B is closed under R-reductions:

- if one believes that $\mathsf{B} = \mathsf{C}$, one may concentrate on showing $L_0 \in \mathsf{B}$, which suffices to prove equality;
- if one believes that $\mathsf{B} \neq \mathsf{C}$, then one may concentrate on proving lower bounds for L_0, which is the "hardest" problem in C, in the sense that it is the least likely to be in B (for the above reason).

So the non-canonicity may even be advantageous, in that it is a source of flexibility.

Nevertheless, in all this arbitrariness there is a remarkable empirical phenomenon: every known "natural" complete problem for every "natural" complexity class is actually complete under an extremely weak form of reduction, known as *quantifier-free projection* [Imm99]. This is a uniform version of a notion of reduction originally introduced by Valiant [Val82], which we will call *projective reduction* here:

Definition 34 (projective reduction) *In the following, if $n \in \mathbb{N}$, we let $[n] := \{1,\dots,n\}$ and $[\overline{n}] := \{\overline{1},\dots,\overline{n}\}$. A* projection *of type $n \to m$ is a tuple*

$$(p^1,\dots,p^m) \in ([n] \cup [\overline{n}] \cup \{k_0,k_1\})^m.$$

This induces a function $p : \{0,1\}^n \to \{0,1\}^m$ defined as follows: given $w =$

[3] In [AAI+01], an artificially constructed NP-complete problem under Karp reductions (actually, under uniform ACC^0 reductions) is shown *not* to be NP-complete under non-uniform AC^0 reductions.

$w_1 \cdots w_n \in \{0,1\}^n$ *and* $1 \leq j \leq m$,

$$p(w)_m := \begin{cases} w_{p_m} & \text{if } p_m \in [n] \\ \neg w_{p_m} & \text{if } p_m \in [\overline{n}] \\ b & \text{if } p_m = k_b \end{cases}$$

where $\neg 0 = 1$ *and* $\neg 1 = 0$.

A projective reduction *of degree* $k \in \mathbb{N}$ *is a sequence* $(p_n)_{n\in\mathbb{N}}$ *such that, for all* $n \in \mathbb{N}$, p_n *is a projection of type* $n \to n^k$.

A projection of type $n \to m$ is in fact nothing but an extremely spartan Boolean circuit of depth 1, with n inputs and m outputs: each output is either a copy of one of the inputs, or the negation of such, or a constant. A projective reduction is just a polysize family of such circuits, one for each input length. Note that projective reductions are non-uniform: the family is allowed to be arbitrary, including uncomputable, as long as the size is polynomially-bounded.

Quantifier-free projections are "very uniform" projective reductions, *i.e.*, the structure of the various p_n is described by a single propositional formula, parametric in n, in a certain language containing basic arithmetic primitives. Such a language is so simple that quantifier-free projections are computable within any resource bound defining any complexity class of interest, including the smallest ones like AC^0 (which is much smaller than L).

What is remarkable about this empirical phenomenon is not so much that there is a sort of "smallest" notion of reduction which seems to fit all classes, but that such a notion is computationally trivial. In fact, these reductions are almost void of computational content: they merely copy the bits of their input, perhaps flip some of them, and return them, perhaps mixed with some constant bits. In comparison, a Karp reduction is capable of mind-bogglingly sophisticated manipulations!

We are tempted to say that projections reveal the true nature of completeness: it is not an algorithmic notion but an algebraic notion. This is especially evident with the projective reductions of Definition 34, in which even the faint algorithmic content hidden in the uniformity condition disappears. Adopting the non-uniform approach to complexity, which we first mentioned in Sect. 3.1.3, we may state the following:

Proposition 46 $\mathsf{P/poly} = \mathsf{NP/poly}$ *iff there is a projective reduction from* CIRCUIT SAT *to* CIRCUIT VALUE.

In the above, CIRCUIT VALUE is the problem of deciding whether a circuit with no free input (*i.e.*, in which all inputs have been assigned a value) evaluates to 1. This is the prototypical P-complete problem. In fact, $\mathsf{P/poly} = \mathsf{NP/poly}$ iff there exists a projective reduction from *some* NP-complete problem to *some* problem in P, but picking two specific problems makes the statement more concrete: it tells us that the answer to one of the deepest open problems of computer science depends on whether or not the instances of CIRCUIT SAT may be transformed to equivalent instances of CIRCUIT VALUE via a manipulation that has basically zero algorithmic content. This is probably the best way

of showing why, since the early days of complexity (essentially the mid-70s), people found the non-uniform perspective attractive: it shifts the focus from computation to algebra and combinatorics.

4.2.3 Reductions as monoidal functors

Complexity theorists often speak of "gadget reductions". This is an informal notion which roughly corresponds to the following situation. The instances of a problem are usually composed of basic elements: the nodes and arcs of a graph, the gates of a circuit, the clauses of a CNF formula, etc. We write informally $w = [e_1, \ldots, e_n]$ to express that the instance w is composed of the basic elements $e_1, \ldots, e_n$. When reducing a problem L to a problem L', it often happens that each basic element e forming an instance of L is mapped to a fixed configuration $r(e)$ of basic elements of instances of L'; such fixed configurations are called "gadgets". So an instance $w = [e_1, \ldots, e_n]$ of L is reduced to an instance $r(w) = [r(e_1), \ldots, r(e_n)]$ of L' by "implementing" its basic elements with gadgets, and then assembling them together.

Let us give an example. We saw that a key step in our proof of the Cook-Levin theorem is exhibiting a Karp reduction from CIRCUIT SAT to SAT. In fact, one may easily reduce CIRCUIT SAT to 3SAT, which is the satisfiability problem restricted to formulas in conjunctive normal form (CNF), *i.e.*, of the form

$$(\alpha_1^1 \vee \cdots \vee \alpha_{k_1}^1) \wedge \cdots \wedge (\alpha_1^n \vee \cdots \vee \alpha_{k_n}^n),$$

(where α_j^i is either an atom or the negation of such) in which, additionally, we require $k_i \leq 3$ for all $1 \leq i \leq n$. A disjunction of the form $\alpha_1^1 \vee \cdots \vee \alpha_{k_i}^i$ above is called a *clause* and its atomic components α_j^i are called *literals*, so a 3CNF is just a CNF in which all clauses have at most three literals.

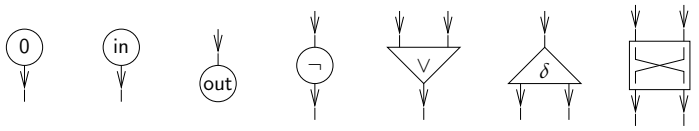
The reduction r from CIRCUIT SAT to 3SAT is defined as follows. Let C be a Boolean circuit. If C is a circuit with inputs $x_1, \ldots, x_n$, whose gates are $g_1, \ldots, g_m$, our 3CNF formula $r(C)$ will be over the atoms $x_1, \ldots, x_n, g_1, \ldots, g_m$. We assign to each gate g a "gadget" $r(g)$, which will be a set of clauses made of at most three literals, depending on the type of g:

- g is an input gate associated with the variable x: $r(g) := \{(\neg g \vee x), (g \vee \neg x)\}$ (corresponding to $g \Leftrightarrow x$);
- g is a constant 0 gate: $r(g) := \{(\neg g)\}$ (corresponding to $g \Leftrightarrow 0$);
- g is a not gate whose input is the output of the gate h: $r(g) := \{(\neg g \vee \neg h), (g \vee h)\}$ (corresponding to $g \Leftrightarrow \neg h$);
- g is an or gate whose inputs are the outputs of the gates h and h': $r(g) := \{(\neg h \vee g), (\neg h' \vee g), (h \vee h' \vee \neg g)\}$ (corresponding to $g \Leftrightarrow (h \vee h')$).

In case g is also the output gate, we add the clause (g) to $r(g)$. (For brevity, we omit and and constant 1 gates, which may anyway be defined from or, not and 0). It is immediate to see that C is satisfiable iff the 3CNF formed of all the clauses $r(g_1), \ldots, r(g_m)$ is satisfiable. It is also fairly obvious that r is polynomial-time computable in m (the size of C).

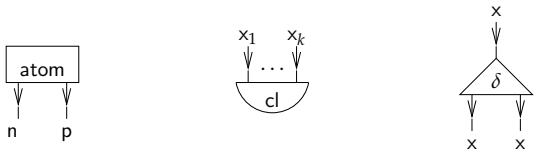
To try and formalize the intuition of "gadget reduction", it is tempting to consider problem instances as being the elements of some free algebra on the basic elements $e_1, e_2, e_3, \ldots$, so that a gadget reduction becomes just a homomorphism. Interestingly, this seems to work seamlessly on all examples we have have tried. Let us apply it to the above reduction.

The instances of CIRCUIT SAT may be naturally presented as endomorphisms of the unit object of the free PRO (strict monoidal category whose objects are all tensor powers of a single generating object) generated by

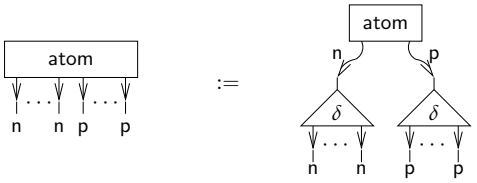


(We use string diagram notations to make the presentation more intuitive). We denote such a category by **Circ**. The generator 0 represents the Boolean constant "false"; the generators in and out represent an input and an output gate, respectively; the other generators are hopefully self-explanatory. An instance of CIRCUIT SAT is a morphism in $\mathbf{Circ}(1,1)$ (*i.e.*, a string diagram with no pending wires) containing one occurrence of out.

For 3SAT, we may see its instances as endomorphisms of the unit object of the following free strict symmetric monoidal category **3CNF**. Its objects are generated by p and n. Its morphisms are generated by



The generator atom stands for an atom, with p (resp. n) representing its positive (resp. negative) form. The generators cl represent clauses with $k \leq 3$ literals, with $\mathsf{x}_1, \ldots, \mathsf{x}_k \in \{\mathsf{p}, \mathsf{n}\}$ depending on whether a literal occurs positively or negatively. Of course, we may define atoms with an arbitrary number of positive and negative occurrences by using trees of δ generators:



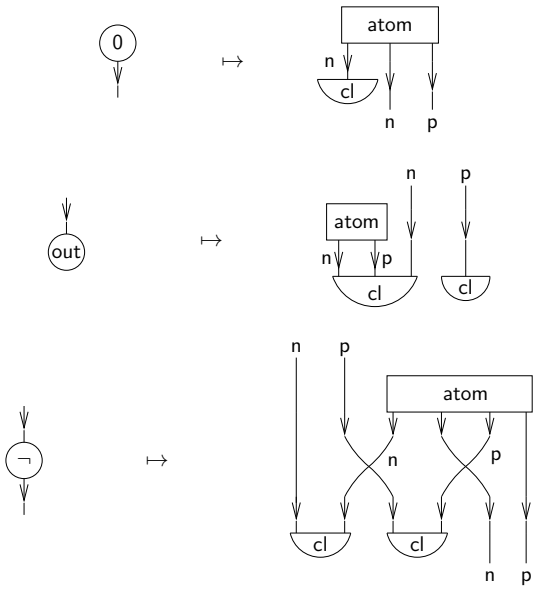
It is now fairly straightforward to implement the reduction r described above as a strict monoidal functor

$$r : \mathbf{Circ} \longrightarrow \mathbf{3CNF}.$$

On objects, if we call $*$ the generating object of **Circ**, we define

$$r(*) := \mathsf{n} \otimes \mathsf{p}.$$

On morphisms, it is of course enough to define r on the generators; without being too verbose, let us give a few examples:



This hopefully gives the idea of how r is defined following the definition of the "gadget reduction" introduced above.

The fact that the instances of a problem live in a monoidal category is justified by the well-known observation, originally made in the context of descriptive complexity, that "everything is a graph" (see Exercise 3.7 of [Imm99]). In other words, all computational structures may be seen, more or less naturally, as graphs, which are two-dimensional objects corresponding to morphisms of monoidal categories.

In this viewpoint, a decision problem on a type of instances $\mathcal{C}$ (a strict monoidal category) is just a strict monoidal functor

$$L : \mathcal{C} \longrightarrow \mathbf{Rel}.$$

This is because instances are endomorphisms in $\mathcal{C}(1,1)$ and the unit object of **Rel** has only two endomorphisms (it acts as a classifier). For instance,

$$\textsc{circuit sat} : \mathbf{Circ} \longrightarrow \mathbf{Rel}$$

maps $*$ to the set $\{0,1\}$ and maps the generators to the following relations:

$$\begin{aligned}
0 &\mapsto \{(*,0)\},\\
\mathsf{in} &\mapsto \{(*,0),(*,1)\},\\
\mathsf{out} &\mapsto \{(1,*)\},\\
\neg &\mapsto \{(0,1),(1,0)\},\\
\vee &\mapsto \{((b_1,b_2),b_1 \vee b_2) \mid b_1,b_2 \in \{0,1\}\},\\
\delta &\mapsto \{(0,(0,0)),(1,(1,1))\},\\
\mathsf{exch} &\mapsto \{((b_1,b_2),(b_2,b_1)) \mid b_1,b_2 \in \{0,1\}\},
\end{aligned}$$

where $*$ denotes the only element of the monoidal unit $1 = \{*\}$ of **Rel**. The reader can check that, if C is a circuit seen as a morphism of $\mathbf{Circ}(1,1)$, C is satisfiable iff CIRCUIT SAT$(C) = \{(*,*)\} = \mathrm{id}_1$ (otherwise, CIRCUIT SAT$(C) = \emptyset$, the only other endomorphism of 1 in **Rel**).

For what concerns

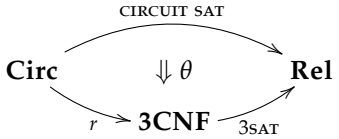
$$3\textsc{sat} : \mathbf{3CNF} \longrightarrow \mathbf{Rel},$$

it maps both p and n to $\{0,1\}$ and

$$\begin{aligned} \mathsf{atom} &\mapsto \{(*,(0,1)),(*,(1,0))\}, \\ \mathsf{cl} &\mapsto \{((b_1,\ldots,b_k),*) \mid b_1,\ldots,b_k \in \{0,1\},\ \textstyle\bigvee b_i = 1\}. \end{aligned}$$

and δ is as for CIRCUIT SAT. Again, we invite the reader to check that, if $\varphi \in \mathbf{3CNF}(1,1)$, then $3\textsc{sat}(\varphi) = \mathrm{id}_1$ iff φ represents a satisfiable 3CNF.

The above functor $r : \mathbf{Circ} \to \mathbf{3CNF}$ is a reduction because there is a monoidal natural transformation



such that $\theta_* = \{(0,(1,0)),(1,(0,1))\}$, that is, the truth value $b \in \{0,1\}$ is mapped to $(\neg b, b) \in \{0,1\}^2$. Thanks to this, we have, for every circuit C, CIRCUIT SAT$(C) = 3\textsc{sat}(r(C))$, as desired.

It is not hard to see that, as a function $r : \{0,1\}^* \to \{0,1\}^*$, the gadget reduction from CIRCUIT SAT to 3SAT introduced above is actually projective. In fact, projective reductions may be considered as very restrictive forms of gadget reductions. So one is naturally led to wonder whether, from the above perspective, strict monoidal functors correspond to projective reductions. The goal would be to reformulate Proposition 46 roughly as follows: the instances of CIRCUIT VALUE are obviously a subset of those of CIRCUIT SAT (they have no input), corresponding to the subcategory $\mathbf{Circ}_0$ of **Circ** without the in generator; then, one could hope that

$$\mathsf{P/poly} = \mathsf{NP/poly} \quad \text{iff} \quad \begin{array}{l} \text{there is a strict monoidal functor } r : \mathbf{Circ} \to \mathbf{Circ}_0 \\ \text{together with a monoidal natural transformation} \\ \theta : \textsc{circuit sat} \Rightarrow \textsc{circuit value} \circ r \end{array}$$

holds. This hope is misguided: while the backward implication certainly holds, it seems quite plausible that the right hand side is false, without this having any bearing on the fact that $\mathsf{P/poly} \neq \mathsf{NP/poly}$, as widely believed.

So the general picture, whatever it is, must be less naive. In particular:

- on the one hand, we cannot restrict to monoidal categories with a finite number of generating objects, like **Circ** or **CNF**. Indeed, for the forward implication of the above reformulation of Proposition 46 to hold, every projective reduction must induce a strict monoidal functor, and this cannot be the case if we have only finitely many generating objects.

- On the other hand, if our monoidal categories must have infinitely many generating objects, then strict monoidal functors will be too broad to match projective reductions; we must impose a size restriction corresponding to that of projective reductions.

We are currently working on a framework that seems to be able to yield the expected property, *i.e.*, that $\mathsf{P}/\mathsf{poly} = \mathsf{NP}/\mathsf{poly}$ iff there is no structure-preserving functor from a certain category to another. This framework requires moving to autonomous categories and makes, again, use of fibrations along similar lines as Chapter 2. It is however too soon to present it here.

At any rate, we want to stress that such a framework would *not* provide any new lower bound technique in itself. Just as for the Cook-Levin theorem, it would merely be a translation of, in this case, basic complexity-theoretic definitions in a new language, closer to the one we use in logic, type theory and the theory of programming languages. It is impossible to say whether this new language will inspire new ideas, but such ideas will certainly not magically sprout from the translation.

The important point is that, compared with the logical routes followed so far (descriptive complexity, implicit computational complexity) this approach does not try to translate the definition of a complexity class, but the definition of decision problem itself, *i.e.*, it is at a fundamentally deeper level, which is precisely the spirit the we intend to follow.

Bibliography

[AAI+01] Manindra Agrawal, Eric Allender, Russell Impagliazzo, Toniann Pitassi, and Steven Rudich. Reducing the complexity of reductions. *Computational Complexity*, 10(2):117–138, 2001.

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[ABC+09] Eric Allender, David A. Mix Barrington, Tanmoy Chakraborty, Samir Datta, and Sambuddha Roy. Planar and grid graph reachability problems. *Theory Comput. Syst.*, 45(4):675–723, 2009.

[ABKL14] Beniamino Accattoli, Eduardo Bonelli, Delia Kesner, and Carlos Lombardi. A nonstandard standardization theorem. In *Proceedings of POPL*, pages 659–670, 2014.

[ABM14] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. Distilling abstract machines. In *Proceedings of ICFP*, pages 363–376, 2014.

[ABM15] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. A strong distillery. In *Proceedings of APLAS*, pages 231–250, 2015.

[AC98] Roberto Amadio and Pierre-Louis Curien. *Domains and Lambda-calculi*. Cambridge University Press, 1998.

[Acc12] Beniamino Accattoli. An abstract factorization theorem for explicit substitutions. In *Proceedings of RTA*, pages 6–21, 2012.

[AJM00] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Inform. Comput*, 163(2):409–470, 2000.

[Ajt83] Miklós Ajtai. Σ^1_1-formulae on finite structures. *Annals of Pure and Applied Logic*, 24(1):1–48, 1983.

[AK12] Beniamino Accattoli and Delia Kesner. Preservation of strong normalisation modulo permutations for the structural lambda-calculus. *Logical Methods in Computer Science*, 8(1), 2012.

[AL16] Beniamino Accattoli and Ugo Dal Lago. (leftmost-outermost) beta reduction is invariant, indeed. *Logical Methods in Computer Science*, 12(1), 2016.

[Bar84] Henk P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Elsevier, 1984.

[Bar92] Henk P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, vol. 2*. Oxford University Press, 1992.

[BBdPH93] P. N. Benton, Gavin M. Bierman, Valeria de Paiva, and Martin Hyland. A term calculus for intuitionistic linear logic. In *Proceedings of TLCA*, pages 75–90, 1993.

[BDS13] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013.

[Bén00] Jean Bénabou. Distributors at work. Unpublished lecture notes, 2000.

[BG95] Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, pages 226–237, 1995.

[BKR14] Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. The inhabitation problem for non-idempotent intersection types. In *Proceedings of IFIP TCS*, pages 341–354, 2014.

[BL13] Alexis Bernadet and Stéphane Lengrand. Non-idempotent intersection types and strong normalisation. *Logical Methods in Computer Science*, 9(4), 2013.

[Bon06] Guillaume Bonfante. Some programming languages for logspace and ptime. In *Proceedings of AMAST*, pages 66–80, 2006.

[Bou93] Gérard Boudol. The lambda-calculus with multiplicities (abstract). In *Proceedings of CONCUR*, pages 1–6, 1993.

[BPS03] Antonio Bucciarelli, Adolfo Piperno, and Ivano Salvo. Intersection types and lambda-definability. *Mathematical Structures in Computer Science*, 13(1):15–53, 2003.

[CDC80] Mario Coppo and Mariangiola Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.

[CDCV81] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Math. Log. Q.*, 27(2-6):45–58, 1981.

[CG01] Andrea Corradini and Fabio Gadducci. Categorical rewriting of term-like structures. *Electr. Notes Theor. Comput. Sci.*, 51:108–121, 2001.

[CO17] Pierre-Louis Curien and Jovana Obradovic. Categorified cyclic operads. Technical Report 1706.06788 [math.CT], ArXiv, 2017.

[Coo71] Stephen Cook. The complexity of theorem proving procedures. In *Proceedings of STOC*, pages 151–158, 1971.

[CPWK04] Sébastien Carlier, Jeff Polakow, J. B. Wells, and A. J. Kfoury. System E: expansion variables for flexible typing with linear and non-linear types and intersection types. In *Proceedings of ESOP 2004*, pages 294–309, 2004.

[dC09] Daniel de Carvalho. Execution time of lambda-terms via denotational semantics and intersection types. *CoRR*, abs/0905.4251, 2009.

[dCPTdF11] Daniel de Carvalho, Michele Pagani, and Lorenzo Tortora de Falco. A semantic measure of the execution time in linear logic. *Theor. Comput. Sci.*, 412(20):1884–1902, 2011.

[DLFVY17] Ugo Dal Lago, Claudia Faggian, Benoît Valiron, and Akira Yoshimizu. The geometry of parallelism: classical, probabilistic, and quantum effects. In *Proceedings of POPL*, pages 833–845, 2017.

[DLS10a] Ugo Dal Lago and Ulrich Schöpp. Functional programming in sublinear space. In *Proceedings of ESOP*, pages 205–225, 2010.

[DLS10b] Ugo Dal Lago and Ulrich Schöpp. Type inference for sublinear space functional programming. In *Proceedings of APLAS*, pages 376–391, 2010.

[DR99] Vincent Danos and Laurent Regnier. Reversible, irreversible and optimal lambda-machines. *Theor. Comput. Sci.*, 227(1-2):79–97, 1999.

[EL10] Thomas Ehrhard and Olivier Laurent. Interpreting a Finitary Pi-Calculus in Differential Interaction Nets. *Information and Computation*, 208(6):606–633, 2010.

[ER06] Thomas Ehrhard and Laurent Regnier. Differential interaction nets. *Theor. Comput. Sci.*, 364(2):166–195, 2006.

[ER08] Thomas Ehrhard and Laurent Regnier. Uniformity and the Taylor expansion of ordinary lambda-terms. *Theor. Comput. Sci.*, 403(2–3):347–372, 2008.

[FPT99] Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proceedings of LICS*, pages 193–202, 1999.

[FSS84] Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984.

[Gar94] Philippa Gardner. Discovering needed reductions using type theory. In *Proceedings of TACS*, pages 555–574, 1994.

[Gir87] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987.

[Gir89a] Jean-Yves Girard. Geometry of interaction I: Interpretation of system F. In *Proccedings of Logic Colloquium 1988*, pages 221–260, 1989.

[Gir89b] Jean-Yves Girard. Towards a geometry of interaction. In *Categories in Computer Science*, volume 92 of *Contemporary Mathematics*, pages 69–108. AMS, 1989.

[Gir98] Jean-Yves Girard. Light linear logic. *Inf. Comput.*, 143(2):175–204, 1998.

[Gir01] Jean-Yves Girard. Locus solum. *Math. Struct. Comput. Sci.*, 11(3):301–506, 2001.

[Gir11] Jean-Yves Girard. Geometry of interaction V: logic in the hyperfinite factor. *Theor. Comput. Sci.*, 412(20):1860–1883, 2011.

[Gol08] Oded Goldreich. *Computational complexity - a conceptual perspective*. Cambridge University Press, 2008.

[Her00] Claudio Hermida. Representable multicategories. *Adv. Math.*, 151(2):164–225, 2000.

[Hil96] Barney P. Hilken. Towards a proof theory of rewriting: The simply typed 2lambda-calculus. *Theor. Comput. Sci.*, 170(1-2):407–444, 1996.

[Hir13] Tom Hirschowitz. Cartesian closed 2-categories and permutation equivalence in higher-order rewriting. *Logical Methods in Computer Science*, 9(3), 2013.

[HO00] Martin Hyland and Luke Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285–408, 2000.

[Hof97] Martin Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In *Proceedings of CSL*, pages 275–294, 1997.

[Hof03] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Inf. Comput.*, 183(1):57–85, 2003.

[Hyl17] Martin Hyland. Classical lambda-calculus in modern dress. *Math. Struct. Comput. Sci.*, 27(5):762–781, 2017.

[Imm99] Neil Immerman. *Descriptive complexity*. Graduate texts in computer science. Springer, 1999.

[Joh99] Peter Johnstone. A note on discrete conduché fibrations. *Theory Appl. Categ.*, 5(1), 1999.

[Jon99] Neil D. Jones. Logspace and ptime characterized by programming languages. *Theor. Comput. Sci.*, 228(1-2):151–174, 1999.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations*, pages 85–103, 1972.

[Kfo00] Assaf J. Kfoury. A linearization of the lambda-calculus and consequences. *J. Log. Comput.*, 10(3):411–436, 2000.

[KKSdV95] Richard Kennaway, Jan Willem Klop, Ronan Sleep, and Fer-Jan de Vries. Transfinite reductions in orthogonal term rewriting systems. *Inform. Comput.*, 119(1):18–38, 1995.

[KKSdV97] Richard Kennaway, Jan Willem Klop, Ronan Sleep, and Fer-Jan de Vries. Infinitary lambda calculus. *Theor. Comput. Sci.*, 175(1):93–125, 1997.

[KL07] Delia Kesner and Stéphane Lengrand. Resource operators for lambda-calculus. *Inf. Comput.*, 205(4):419–473, 2007.

[Kri93] Jean-Louis Krivine. *Lambda Calculus, Types and Models*. Ellis Horwood, 1993.

[Kri05] Lars Kristiansen. Neat function algebraic characterizations of logspace and linspace. *Computational Complexity*, 14(1):72–88, 2005.

[KV14] Delia Kesner and Daniel Ventura. Quantitative types for the linear substitution calculus. In *Proceedings of IFIP-TCS*, pages 296–310, 2014.

[KV16] Delia Kesner and Pierre Vial. Types as resources for classical natural deduction. *Submitted*, 2016.

[Laf04] Yves Lafont. Soft linear logic and polynomial time. *Theor. Comput. Sci.*, 318(1-2):163–180, 2004.

[Lau03] Olivier Laurent. Polarized proof-nets and lambda-mu-calculus. *Theor. Comput. Sci.*, 290(1):161–188, 2003.

[Lau04] Olivier Laurent. On the denotational semantics of the untyped lambda-mu calculus. Unpublished note, 2004.

[Lei04] Tom Leinster. *Higher Operads, Higher Categories*. Cambridge University Press, 2004.

[Lev73] Leonid Levin. Universal search problems. *Problems of Information Transmission*, 9(3):115–116, 1973.

[Maz12] Damiano Mazza. An infinitary affine lambda-calculus isomorphic to the full lambda-calculus. In *Proceedings of LICS*, pages 471–480, 2012.

[Maz13] Damiano Mazza. Non-linearity as the metric completion of linearity. In *Procedings of TLCA*, pages 3–14, 2013.

[Maz14] Damiano Mazza. Non-uniform polytime computation in the infinitary affine lambda-calculus. In *Proceedings of ICALP, Part II*, pages 305–317, 2014.

[Maz15] Damiano Mazza. Simple parsimonious types and logarithmic space. In *Proceedings of CSL*, pages 24–40, 2015.

[Maz16] Damiano Mazza. Church meets Cook and Levin. In *Proceedings of LICS*, pages 827–836, 2016.

[Maz17] Damiano Mazza. Infinitary affine proofs. *Math. Struct. Comput. Sci.*, 27(5):581–602, 2017.

[Mel04] Paul-André Melliès. Asynchronous games 1: A group-theoretic formulation of uniformity. Technical Report PPS//04//06//n°31, Preuves, Programmes et Systèmes, 2004.

[Mel09] Paul-André Melliès. Categorical semantics of linear logic. In *Interactive models of computation and program behaviour,* Panorama et synthéses 27. Société Mathématique de France, 2009.

[Mel17] Paul-André Melliès. *Une étude micrologique de la négation*. Thése d'habilitation à diriger les recherches, Université Paris 7, 2017.

[Mil07] Robin Milner. Local bigraphs and confluence: Two conjectures: (extended abstract). *Electr. Notes Theor. Comput. Sci.*, 175(3):65–73, 2007.

[MT15] Damiano Mazza and Kazushige Terui. Parsimonious types and non-uniform computation. In *Proceedings of ICALP, Part II*, pages 350–361, 2015.

[MTT09] Paul-André Melliès, Nicolas Tabareau, and Christine Tasson. An explicit formula for the free exponential modality of linear logic. In *Proc. ICALP 2009*, pages 247–260, 2009.

[MZ15] Paul-André Melliès and Noam Zeilberger. Functors are type refinement systems. In *Proceedings of POPL*, pages 3–16, 2015.

[Nee04] Peter Møller Neergaard. A functional language for logarithmic space. In *Proceedings of APLAS*, pages 311–326, 2004.

[Nie04] Susan Niefield. Change of base for relational variable sets. *Theory Appl. Categ.*, 12(7):248–261, 2004.

[NM04] Peter Møller Neergaard and Harry G. Mairson. Types, potency, and idempotency: why nonlinearity and amnesia make a type system work. In *Proceedings of ICFP*, pages 138–149, 2004.

[Pap94] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[Par92] Michel Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning: International Conference LPAR '92 St. Petersburg, Russia, July 15–20, 1992 Proceedings*, pages 190–201, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[PPR17] Luca Paolini, Mauro Piccolo, and Simona Ronchi Della Rocca. Essential and relational models. *Mathematical Structures in Computer Science*, 27(5):626–650, 2017.

[PRR99] Alberto Pravato, Simona Ronchi Della Rocca, and Luca Roversi. The call-by-value lambda-calculus: a semantic investigation. *Mathematical Structures in Computer Science*, 9(5):617–650, 1999.

[PRR12] Elaine Pimentel, Simona Ronchi Della Rocca, and Luca Roversi. Intersection types from a proof-theoretic perspective. *Fundam. Inform.*, 121(1-4):253–274, 2012.

[Reg92] Laurent Regnier. *Lambda-calcul et réseaux*. Phd thesis, Université Paris 7, 1992.

[RL11] Ramyaa Ramyaa and Daniel Leivant. Ramified corecurrence and logspace. *Electr. Notes Theor. Comput. Sci.*, 276:247–261, 2011.

[SBHG08] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. *J. Funct. Program.*, 20(5-6):417–461, 2008.

[Sch06] Ulrich Schöpp. Space-efficient computation by interaction. In *Proceedings of CSL*, pages 606–621, 2006.

[Sch07] Ulrich Schöpp. Stratified bounded affine logic for logarithmic space. In *Proceedings of LICS*, pages 411–420, 2007.

[See87] R. A. G. Seely. Modelling computations: A 2-categorical framework. In *Proceedings of LICS*, pages 65–71, 1987.

[Shu08] Michael Shulman. Framed bicategories and monoidal fibrations. *Theory Appl. Categ.*, 20(18):650–738, 2008.

[Shu10] Michael Shulman. Constructing symmetric monoidal bicategories. Technical Report arXiv:1004.0993 [math.CT], ArXiv, 2010.

[Sta16] Michael Stay. Compact closed bicategories. *Theory Appl. Categ.*, 31:755–798, 2016.

[Ter03] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[Ter04] Kazushige Terui. Proof nets and boolean circuits. In *Proceedings of LICS*, pages 182–191, 2004.

[Val82] Leslie Valiant. Reducibility by algebraic projections. *L'enseignement mathématique*, 28(3–4):253–268, 1982.

[vB95] Steffen van Bakel. Intersection type assignment systems. *Theor. Comput. Sci.*, 151(2):385–435, 1995.

[Vol99] Heribert Vollmer. *Introduction to circuit complexity - a uniform approach*. Texts in theoretical computer science. Springer, 1999.

[Wad71] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. Phd thesis, University of Oxford, 1971.