The monitoring power of forcing program transformations

Aloïs Brunel

Université Paris Nord — Paris 13
Sorbonne Paris-Cité
Laboratoire d'Informatique de Paris Nord
Thèse de doctorat Spécialité informatique

THE MONITORING POWER OF FORCING TRANSFORMATIONS

Aloïs Brunel

Soutenue publiquement le 27 juin 2014

Directeurs:
Stefano Guerrini
Damiano Mazza

Rapporteurs:
Lars Birkedal
Alexandre Miquel

Jury:
Pierre-Louis Curien
Stefano Guerrini
Martin Hofmann
Martin Hyland
Damiano Mazza
Paul-André Melliès
Michele Pagani

Remerciements

Si j'ai pu mener au bout ce travail de thèse, c'est grâce à un certain nombre de personnes que j'aimerais maintenant remercier.

Je remercie chaleureusement Stefano Guerrini pour avoir dirigé ma thèse et m'avoir accueilli avec bonne humeur au sein du LIPN ! Nos échanges, même si ponctuels, ont toujours été instructifs.

Je voudrais maintenant adresser ma grande reconnaissance à Damiano Mazza, pour avoir accepté de co-encadrer cette thèse et m'avoir fait bénéficié de son temps et son expérience. Il a su me faire confiance en me laissant carte blanche quant au sujet de mes travaux, et accueillir mes idées (même les plus loufoques) avec le grand enthousiasme qui le caractérise ! Je ne cacherais pas non plus une ceraine admiration pour son travail scientifique, dont la grande variété et l'originalité des sujets a toujours été pour moi une grande source d'inspiration.

Je souhaite également remercier les rapporteurs et tous les membres du jury, pour m'avoir fait l'honneur d'accepter d'en faire partie.

Si cette thèse existe, c'est également grâce à une série de personnes qui ont su me guider jusqu'à elle, et à qui je souhaite exprimer toute ma gratitude, dans l'ordre chronologique. C'est Christophe Pointier qui m'a initié aux joies de la programmation fonctionnelle, quand je n'étais encore qu'en première année de prépa à Marcelin Berthelot, et qui a su m'en faire voir la beauté.

La personne qui m'a réellement propulsé dans l'univers complètement fou de la réalisabilité classique et du forcing est Alexandre Miquel, dont le travail constitue de maniére évidente le point de départ de cette thèse. Il a su m'expliquer, avec la légendaire pédagogie qu'on lui connaît, des concepts pourtant ardus, et guider mes premiers pas de chercheur lors de mon stage de L3. Je le remercie de m'avoir toujours encouragé durant les sept dernières années. Enfin je suis honoré qu'il ait accepté de rapporter ce travail de thèse,

Merci également à mes autres encadrants de stage. D'abord Kazushige Terui, qui m'a accueilli dans une des villes les plus belles du monde: Kyoto. Il m'a appris la rigueur scientifique et m'a montré qu'il était possible et nécessaire de s'intéresser à une multitude de sujets scientifiques. Il est également celui qui m'a soufflé l'idée de rapprocher réalisabilité quantitative et réalisabilité classique de Krivine, idée qui constitue le point de départ de cette thèse. Finalement, merci à Olivier Hermant et Gilles Dowek, qui ont encadré mon stage de M2 avec énormément d'enthousiasme.

Merci également à Alexis Saurin, qui a toujours pris le temps de me conseiller, c'est à lui que je dois d'avoir atterri à Paris 13.

Bien entendu, cette thèse a bénéficié de plusieurs collaborations. Merci d'abord à Antoine Madet. Notre travail commun a constitué un tournant dans ma recherche scientifique. Sans parler des nombreuses bières et sakés partagés au cours de ces dernières années :-) Marco

Gaboardi, collaborateur prolifique et qui a toujours été derrière moi, tant scientifiquement qu'humainement. Merci de m'avoir accueilli à Philadelphie. J'y ai découvert les meilleurs food trucks, les pizzas à la Burrata et la notion de Monitoring Algebra.

Merci aux relecteurs très attentifs, et en particulier Pierre-Évariste Dagand et Adrien Guatto qui ont produit une liste incroyable de remarques et corrections !

La bande du GdT: Adrien, Anne-Sophie, Charles, Guillaume, Marc, ainsi que le vieil homme du Pantalon. Pour les nombreuses conversations mêlant logique et Triple Karmeliet !

La bande de l'Oregon: Andrei, Antoine, Clément, Matthias, Marc, Stéphane, et tous les occupants de la maison hippie.

Les joyeux occupants du bureau B309: Hanene, Manisha, Zayd. Je vous souhaite à tous le meilleur !

Les chevaliers de la pesanteur, grands garants du Koo, occupants temporaires de la B35 et autres joyeux lutins du BDE Mogwai et assimilés. Alvaro, Anne-Laure, Baptiste, Camille, Dimy, Fofi, Geekou, JR, Quentin le poète. Les visiteurs de l'appartement de la rue d'Ulm: Clément, Franck, Xavier, et tous les autres que j'ai peut-Ãłtre oublié (merci à ces derniers, pour m'avoir déjà pardonné :)). Augustin, qui s'évertue à faire décoller l'Ariane 5 en la nourrissant du fameux calvados de la distillerie Boulard. Puissent les dieux sacrés de la déontique veiller sur vous tous.

Parce qu'ils sont mes amis depuis toujours, et que je sais pouvoir toujours compter sur eux: Ben, Christophe, Clément, Frosio, Johan, Let, Marlène, Mélanie, et Michele qui est arrivé plus récemment, mais qui a déjà prouvé sa valeur ! Merci à vous tous, pour votre amour du vin, les baignades dans le Nahon et les soirées à Menetou !

Et comme la musique est une part importante de ma vie, il faut absolument remercier les chevaucheurs de poney fou: Baptiste, Guillaume, Séverine, Victor. Et les plus vieux ont aussi leur place: Giraud et Maxime, je pense à vous !

Ma famille: mon frère Camille, ma soeur Clotilde et mon/ma futur(e) neveu/nièce dont le nom sera ajouté en temps voulu ! Mes parents, Ghislain, Marion, pour l'amour et le soutien qu'ils m'ont apportés déjà bien avant le début de cette thèse :-) Sans eux, il n'y aurait pas non plus eu de pot de thèse ! Pascale, Alain, et Liliane pour qui j'ai une pensée très chère, j'aurais aimé qu'elle puisse me voir docteur.

Et enfin, merci à Samantha, pour ces trois précieuses années, et pour je l'espère tout ce qui est à venir.

Contents

List of Notations

Chapter I

Introduction

The last few decades of research in programming language theory have seen the introduction of a myriad of very advanced type systems, intended to enforce increasingly complex computational properties of programs. For instance, in the field of functional *differential privacy* [GHH+13, RP10] one aims at developing programing languages whose type system prevents information leaks in the context of database requests. Another example is *functional reactive programming* [Kri13, KBH12]: several memory leak problems (that are common in reactive programming) are ruled out by a sophisticated linear discipline that tames space consumption. A last example would be bounded-time programming languages, based on substructural logics like *light logics* [GDR09, GMRDR12, Gir98], that ensure that the time or space complexity of programs remains reasonable. In all those cases, one needs to prove the correctness of the language with respect to the given computational property we want to ensure. The guarantee of correctness can be critical in certain environments, and much energy has been dedicated to formalize and mechanically verify such proofs. Those proofs are done syntactically or semantically. When it is possible, semantics proofs are usually preferred since they are more modular and easier to formalize.

Realizability semantics

In this thesis, we are particularly interested in *realizability semantics*. This technique consists in defining a binary relation $\Vdash$ between programs and types, called the **realizability relation**:

$$t \Vdash A$$

In that context $t \Vdash A$ means "t *follows the computational specification represented by* A". We also say that t is a realizer of A. This relation is usually defined by induction on the grammar of types. The goal is then to prove a *soundness theorem* of the form:

$$\text{If } \vdash t : A \text{ then } t \Vdash A.$$

This theorem makes the link between *syntactic constraints* enforced by the type system and *computational correctness* represented by the realizability relation. One generally deduces from

this soundness theorem the correctness of typable programs, like termination without clash, or safety. Using the realizability technique has several advantages:

- It is modular, in the sense that the theorems do not break easily when the definitions are changed, and the proofs stay more or less the same. This allows to adapt realizability techniques to other similar cases.
- The proofs are usually much simpler than the syntactic ones. Once the definition of the realizability relation is given, the soundness tends to follow by a long and tedious but straightforward induction.

However, we can identify several unsatisfactory points to this approach:

- These realizability semantics are not quite as modular as we often say. Indeed, extending a programming language often implies changing all the definitions of the realizability relation and reproving the entirety of the soundness theorem, even though the inductive cases of the proof are said to be *similar* to the previous case.
- Moreover, it is really difficult to *combine* those semantics, in order to combine different programming features together, since there is no general and structured enough *space of realizability semantics*. When it comes to stacking up different programming features, one often has to try to combine different realizability techniques and see if they work without having formal results on the possibility of doing so.

Overall, there is a lack of *structure*. The success in proving the correctness of complex languages often rests upon the experience and intuitive understanding of the common patterns underlying realizability semantics and on how to combine them. We advocate the need for a more algebraic framework that formally unveils these common patterns, in which it is possible to:

- Explore new realizability semantics *generated* by some simple algebraic structure.
- Combine those semantics by combining the underlying algebraic structures.
- Prove pieces of soundness results once and for all, that can be reused for various realizability semantics *out of the box*.
- Develop algebraic constructions that can be reused in various correctness proofs, for example to add specific programming features and automatically derive the associated soundness property.

To summarize, we want to adopt a more *analytical* point of view on realizability semantics for the correctness of programming languages.

Annotated realizability

For the languages we mentionned before, there is a common particularity concerning the realizability semantics used to prove the correctness: the presence of an *additional annotation of*

the programs [BM12, KBH12, RP10]. We call such an approach **annotated realizability**. In that context the unary realizability relation $\Vdash$ becomes a relation between types and *annotated* (or *weighted*) *programs*:

$$(t, l) \Vdash A$$

The annotation l can be of a completely different nature in each case. It can be an integer bounding the consumption of space of the associated program, an abstract quantity bounding the time complexity, step-indices used to stratify the execution, or even a real number in the case of differential privacy. In general, we can see l as a way to refine the specification represented by the type A and hence the observation made on programs, by providing tests that the program has to pass. By studying how this annotation is used, one soon notices many common patterns in those realizability semantics. Amongst these patterns, one could cite:

- In much examples, the "annotation part" of the realizability relation satisfies similar properties. In the case of the functional arrow $\to$, we often have the following:

  $$(t, p) \Vdash A \to B \Longrightarrow \forall (u, q) \Vdash A, ((t)u, p + q) \Vdash B$$

  Where the operation $+$ on the set of annotation depends of the case, but always satisfies the same basic properties (associativity, commutativity, the presence of a neutral element).

- The presence of a certain preorder $\preceq$ on those annotations such that the interpretation is closed by $\preceq$:

  $$(t, p) \Vdash A \wedge p \preceq q \Longrightarrow (t, q) \Vdash A$$

  This preorder expresses how much information is carried by an annotation, or at least how it compares to another annotation. In this case, it says that if a program is correct with respect to an annotation that is *finer* than another one, then it is also correct with respect to the latter.

This suggests that:

- It may be possible to look for a **unifying framework for these semantics**, by abstracting over the set of annotations.

- In certain cases [BM12, KBH12], the annotation, the operation $+$ and the preorder $\preceq$ are defined as products of more elementary annotations and operations. In those works, this results in the combination of the properties induced by each of the elementary annotations. This suggest that by combining the annotations in that framework, one could combine the realizability semantics (and the languages being interpreted).

This thesis

The goal of the present thesis is to introduce a framework based on the idea of abstracting over the set of annotations in an algebraic way. The ultimate goal is to dramatically reduce the complexity of developing realizability semantics and prove soundness results, by allowing

to stack up abstract results. This framework allows to define a *space of realizability semantics* that is flexible enough to contain many known semantics, but constrained enough so that we can adopt an algebraic view on them. Most of the complexity is contained in a simple algebraic notion, which allows to *combine* semantics, *reuse* parts of soundness results, *understand* known phenomenon and techniques (for instance, value restriction, step-indexing) in an algebraic fashion.

This thesis develops the basis of this theory along with elementary results and tools. We show that it is already possible to use it to re-obtain difficult correctness results. We also show that this framework can be used to *explore* new possibilities of realizability semantics, leading to new results. The main elements used to develop such a theory are *forcing*, *Krivine's classical realizability* and the technique of *iteration*. The sections 1.1 to 1.4 are devoted to the introduction of the main tools underlying our work: the forcing technique and Krivine's realizability. The reader will find in Section 1.5 a summary of each chapter of this thesis, in which we can find the main definitions and results accompanied with various intuitions and remarks. We recommend the reading of this section before the reading of the actual chapters. Finally, Section 1.6 contains a very brief summary of the contributions of this thesis.

1.1 | Forcing

The main ingredient of our methodology is a variant of Cohen's forcing and Girard's phase semantics. These techniques are both based on simple algebraic structures, which will be used in our framework to represent annotations. We give some historical as well as technical details about those techniques.

1.1.1 Cohen's forcing

The **Continuum Hypothesis** (or **CH**) states that every infinite subset $S \subseteq \mathbb{R}$ is either denumberable or in bijection with $\mathbb{R}$. The question of the validity of this conjecture was first raised by Georg Cantor in 1878. It became one of the major open problems in the foundations of mathematics, and was the first of Hilbert's famous 23 problems.

Inner models

In 1938, Kurt Gödel [GB40] proved that the negation of the continuum hypothesis could not be derived from the axioms of **ZFC**, *i.e.*

$$\mathbf{ZFC} \nvdash \neg CH$$

His construction, called the **constructible sets**, is an example of what is now known as an **inner model**. It consists in starting with a model V of **ZF** that does not necessarily entail the axiom of choice or the continuum hypothesis, and find a subclass of V that possesses a new membership relation which satisfies these two principles.

Cohen's discovery

It was not until 1963 that the problem of the consistency of the negation of the continuum hypothesis was solved by Paul Cohen. He introduced in a series of two papers [Coh63, Coh64] the celebrated technique known as *forcing*, which he used to finally prove

$$\mathbf{ZFC} \nvdash CH$$

The technique has since then been extensively used to prove many consistency results. Starting from a standard model of **ZFC** (that is, a model whose objects are well-founded sets built using the empty set $\varnothing$, the union and powerset operations, and whose membership relation is the usual $\in$), it consists in adjoining to it a new set that does not exist in that ground model. In the case of the original forcing, Cohen shows how to add a set X that is strictly between $\mathbb{N}$ and $\mathbb{R}$. The trick consists in describing this set using approximations, *i.e.* elementary statements about X which are called **forcing conditions**.

Sketch of the argument

Definition 1. *In an ambient **ZFC** standard model $\mathcal{M}$, a **forcing notion** (or **forcing poset**) is a partially ordered set $(P, \leq)$, whose elements are called **forcing conditions**.*

Consider the forcing notion based on the set of of all finite functions:

$$p : \omega \to \{0, 1\}$$

This set is orderered by the reverse inclusion $\supseteq$. Each forcing condition p represents a finite number of statements on a subset X of ω saying: "x is in X" or "x is not in X", depending on wether $p(x) = 0$ or $p(x) = 1$. These are *conditions* on the possible elements of X, hence the name *forcing conditions*. We say that two forcing conditions are **compatible** if they agree on their common domain. Eventually, with enough compatible forcing conditions, we obtain the full information on the new set we are trying to add. Here comes the idea of taking a generic P-ultrafilter G, *i.e.* a subset of P that satisfies various conditions, in particular it intersects every dense subset of P[1]. This is where one builds a *new model* $\mathcal{M}[G]$ which is a generic extension of $\mathcal{M}$. In this new model, it is possible to consider:

$$g = \bigcup G$$

Because of the conditions satisfied by G, it is possible to prove that g is in fact a total function $\omega \to \{0, 1\}$. One can then pose $X = g^{-1}(1)$. Because G is generic, it can be shown that X is a *set* that was not previously in $\mathcal{M}$. Indeed if A is a subset of $\mathbb{N}$, then the following set is dense and hence intersects G:

$$D_A = \{\ p \mid \exists n \in \mathbb{N}, p(n) = 1 \Leftrightarrow n \notin A\ \}$$

[1] This generic P-ultrafilter always exists if $\mathcal{M}$ is a denumberable model. Those can always be considered due to Löwenheim-Skolem theorem that allows us to replace any uncountable model with a countable one

By generalizing this argument and replacing ω with $\omega \times \aleph_2$ ($\aleph_2$ is the second uncountable ordinal) and choosing the forcing poset of finite functions:

$$p : \omega \times \aleph_2 \to \{0, 1\}$$

One obtains $\aleph_2$ new distinct subsets of $\mathbb{N}$ in the model $\mathcal{M}[G]$, hence an injection $\aleph_2 \to \mathcal{P}(\omega)$. Modulo some technicalities that show that cardinals cannot collapse (this is a consequence of the *countable chain condition*), one obtains that $\mathcal{M}[G]$ is a model that violates the continuum hypothesis.

The proof-theoretical point of view

From the proof-theoretical point of view, the interesting part happens when one constructs the model $\mathcal{M}[G]$, that somehow integrates the forcing conditions. Internally, it consists in defining in $\mathcal{M}$ a **forcing relation** between forcing conditions and formulas of set theory:

$$p \Vdash A$$

This relation is defined inductively[2]. For example, the case of the implication connective is as follows:

$$p \Vdash A \Rightarrow B \quad \equiv \quad \forall q \leq p \, (q \Vdash A \Rightarrow p \Vdash B)$$

For the "and" connective, we define:

$$p \Vdash A \wedge B \quad \equiv \quad p \Vdash A \wedge p \Vdash B$$

We can then prove the following properties:

1. If $\vdash A$ is provable, then so is $\vdash \forall p \in P, p \Vdash A$.
2. There exists A such that $\vdash \forall p \in P, \neg(p \Vdash A)$.

The first property morally says that $\Vdash$ induces a model of **ZFC** [3] and the second represents the consistency of this model. We can in fact see the relation $\Vdash$ as a *logical translation*:

$$A \mapsto p \Vdash A$$

The property 1. can be seen as the *soudness* of this logical translation. The recent works of Krivine [Kri11] and Miquel [Miq11] unveil the *program transformation* behind this logical transformation, which will be the starting point of this thesis.

[2] Most of the complexity of the definition lies in the interpretation of the membership relation $a \in b$.
[3] In fact, to obtain an actual model of **ZFC**, one has to perform a quotient by G.

1.1.2 Forcing and orthogonality

The forcing technique can benefit from a reformulation using **orthogonality**. Orthogonality underlies many mathematical concepts. In linear algebra, we say that two vectors x and y are orthogonal and denote by $x \perp y$ if their inner product is zero:

$$x \perp y \Leftrightarrow \langle x|y\rangle = 0$$

Many denotational semantics of linear logic are based on a form of orthogonality. For example, in coherence spaces [Gir87], Girard uses the orthogonality relation between two cliques c and c' in a graph, which is defined as:

$$c \perp c' \Leftrightarrow \#(c \cap c') \leq 1 \qquad (\# \text{ being the cardinality})$$

In general, one can understand orthogonality as a way to express the *good interaction* between two objects. When we reformulate forcing using orthogonality, we basically obtain a set-theoretical version of the so-called **phase semantics** of linear logic, introduced as a semantics of truth by Girard in his seminal paper [Gir87]. Since this is the style of forcing we will consider in this thesis, we give a brief account of the phase semantics.

Definition 2. *A **phase space** is given by:*

- *A commutative monoid* $(P, \cdot, \mathbf{1})$.
- *A subset* $\perp\!\!\!\perp$ *of* P*, called the **pole**.*

- Each idempotent phase space (whose operation $\cdot$ is such that $p \cdot p = p$) defines a forcing poset in the sense of Cohen's forcing. Indeed, we can define a partial order $\leq$ using $\perp\!\!\!\perp$:

$$p \leq q \Leftrightarrow \forall r \in P, q \cdot r \in \perp\!\!\!\perp \Rightarrow p \cdot r \in \perp\!\!\!\perp$$

- For example, the original Cohen's forcing notion can be retrieved as a phase space. We take P as the set of relations $\omega \times \{0,1\}$, the $\cdot$ operation is the union of two relations and the pole $\perp\!\!\!\perp$ consists in the set of relations that define a finite function $\omega \to \{0,1\}$. $\mathbf{1}$ is then the empty relation.

Given a phase space, we define an orthogonality relation using its pole, as follows:

$$p \perp q \Longleftrightarrow p \cdot q \in \perp\!\!\!\perp$$

This orthogonality operation lifts to sets $X \subseteq P$[4]:

$$X^{\perp} = \{\ q \in P \mid \forall p \in X, p \perp q\ \}$$

This operation then satisfies several properties. In particular, the biorthogonality operator $(.)^{\perp\perp}$ is a closure operator:

[4]The same happens in the case of an inner product space: the notion of orthogonality between two vectors lifts to a notion of orthogonal of a set of vectors X, by considering all the vectors that are orthogonal to those of X.

- $X \subseteq X^{\perp\perp}$
- $X \subseteq Y$ implies $Y^{\perp} \subseteq X^{\perp}$
- $X^{\perp\perp\perp} = X^{\perp}$

Here are some of the cases of the definition of the interpretation of linear logic in phase spaces. The idea is to associate to each type a set *closed by bi-orthogonality, i.e.* a set of the form $X^{\perp}$. As examples, we give the interpretation of $\otimes$ and $\multimap$, which are linear logic versions of the conjunction and the implication respectively:

$$\begin{array}{rcl} (A \otimes B)^* & = & \{\ p \cdot q \mid p \in A^* \wedge q \in B^*\ \}^{\perp\perp} \\ (A \multimap B)^* & = & \{\ p \cdot q \mid p \in A^* \wedge q \in B^{*\perp}\ \}^{\perp} \end{array}$$

We can then define the relation $\Vdash$ as follows:

$$p \Vdash A \equiv p \in A^*$$

For example, the definition of the $\multimap$ connectives is equivalent to the following definition:

$$p \Vdash A \multimap B \Leftrightarrow \forall q \Vdash A, p \cdot q \Vdash B$$

One can check that if $\cdot$ is idempotent, then by by reformulating this property with the partial order $\leq$, one retrieves the definition of the forcing interpretation of $\Rightarrow$.

Remark 3. *There are several benefits from the orthogonality approach.*

- *It simplifies the definitions and proofs, since one can now prove the membership of a condition p to the interpretation of a type A* interactively, *i.e. by testing the good interaction with all the elements of $A^{*\perp}$.*
- *It is very flexible and can be used to prove many different results just by changing the pole: one can obtain cut-elimination proofs [Oka99, Oka02], decidability and undecidability results [Laf96, DLM04], finite model properties [Laf97], or a proof of Girard's approximation theorem [Gir87].*
- *As we will see it also allows for a much easier analysis of the program transformation behind forcing.*
- *Finally, it better isolates the different parameters of the forcing construction, which allows to generalize or specialize it more easily. For example, a first generalisation is to consider a non-idempotent operation $\cdot$, as it is the case in phase spaces. In this thesis, we will push the generalization further by considering two operations $+$ and $\bullet$ (one for the tensor, the other for the implication) instead of just $\cdot$.*

1.1.3 Iterated forcing

The forcing technique starts from a standard model $\mathcal{M}$, and given a poset P inside $\mathcal{M}$, and a P-generic filter G, it builds a new standard model $\mathcal{M}[G]$. One can *iterate* this construction, and choose a new forcing poset and a generic ultrafilter G' *inside* $\mathcal{M}[G]$, then constructing $\mathcal{M}[G][G']$. It is possible to show that this construction, which is the combination of a forcing model and another forcing model built inside it, can be seen as a single forcing construction. This is called the 2-**steps forcing iteration**. One can even iterate forcing a *transfinite number κ of times*, where κ is any ordinal. This technique has been introduced by Solovay and Tennenbaum [ST71] in 1965 in their proof of the consistency of Souslin's Hypothesis. Iterated forcing has since then played a central role in set-theory and the quest for new consistency results. From the point of view of the forcing relation the 2-steps iteration amounts to study:

$$p \Vdash_{P_1} (q \Vdash_{P_2} A)$$

where P_1 is a forcing notion and P_2 is a forcing notion taken *inside* a model induced by P_1. One can show that there exists a single forcing notion $P_1 \star P_2$ such that

$$p \Vdash_{P_1} (q \Vdash_{P_2} A) \Leftrightarrow (p,q) \Vdash_{P_1 \star P_2} A$$

Remark 4. *From the point of view of the orthogonality, given a phase space P_1, we choose a P_1-phase space, i.e. a monoid P_2 and a relativized pole $\perp\!\!\!\perp_2 : P_1 \to \mathcal{P}(P_2)$. We then form the phase space $P_1 \star P_2$ as follows:*

- *The product of the monoids.*
- *The pole is defined as*

$$\perp\!\!\!\perp = \{ (p,q) \mid \forall r \in P_1, q \notin \perp\!\!\!\perp_2(r) \Rightarrow p + r \in \perp\!\!\!\perp_1 \}$$

1.2 | Forcing as a program transformation

1.2.1 Krivine/Miquel program transformation

The forcing technique as introduced by Cohen has been studied and used for decades by set theory specialists, but it had escaped any proof theory investigation until Krivine showed [Kri10a] how to combine his theory of classical realizability with forcing. He defines and uses a program transformation that turns any proof term t of a formula A into a *realizer* of the formula $\mathbf{1} \Vdash A$ in a suitable realizability model. Subsequently, Miquel [Miq11] showed that this transformation can be turned into a proper fully-typed syntactic program transformation in an extension of higher-order arithmetic called $\mathbf{PA}\omega^+$. In this context, the soundness of the forcing interpretation becomes a type preservation theorem. Moreover, Miquel studies the computational behavior of the transformed programs, which reminds of the ring protection mechanism used in most processors.

We briefly remind the extension $\mathbf{PA}\omega^+$ of higher-order arithmetic that Miquel defines, as some parts of our core type system are strongly inspired by it. It is based on a higher-order logic whose **kinds** are built using three constructors: ι (the kind of individuals, or integers), o (the kind of propositions) and $\sigma \to \tau$ (the functional kind). The higher-order terms contain the usual λ-calculus augmented with the control operator call-cc[5], integers (including a recursor) and logical constructions:

- The classical implication $\Rightarrow$.
- The universal quantifier $\forall x^\tau M$ on all kinds τ.
- The equational implication $M =_\tau M' \mapsto A$, where M and M' are two terms of kind τ. This construction is not standard and its intuitive meaning is as follows:

$$M =_\tau M' \mapsto A \equiv \begin{cases} A & \text{if } M \text{ equals } M' \\ \top & \text{otherwise} \end{cases}$$

  where $\top$ represents the type of *all* λ-terms. This equational implication is provably equivalent to a usual implication $M =_\tau M' \Rightarrow A$ (where $=_\tau$ is the Leibniz equality at kind τ), but it yields more compact proof terms, as the equality is transparent from the point of view of the proof terms. It provides a convenient way to simplify the forcing translation.

As we said earlier, forcing is usually parametrized by a poset of forcing conditions. However, in [Kri11] Krivine considers forcing conditions as the elements of an upward closed subset C of a meet semi-lattice $(P, \cdot, \mathbf{1})$ ($\cdot$ is a meet operation on P and $\mathbf{1}$ is the least element). Morally, he uses the orthogonality-based presentation of forcing (very much in the spirit of phase spaces). C represents the complement of the pole $\bot$ of the phase space. One obtains an orthogonality relation by defining:

$$p \bot q \Leftrightarrow p \cdot q \notin \mathsf{C}$$

Concretely, in Miquel's framework, a **forcing structure** is given by:

- A kind κ
- A closed predicate $\mathsf{C} : \kappa \to o$.
- A closed term $\cdot : \kappa \to \kappa \to \kappa$.
- A closed term $\mathbf{1} : \kappa$
- Several closed terms α_i which constitute the proof that C is upward closed and that its elements constitutes a meet semi-lattice. Here are some of those terms:

$$\begin{array}{lcl} \alpha_0 & : & \mathsf{C}[\mathbf{1}] \\ \alpha_4 & : & \forall p^\kappa \mathsf{C}[p] \Rightarrow \mathsf{C}[pp] \\ \alpha_6 & : & \forall p^\kappa \forall q^\kappa \forall r^\kappa (\mathsf{C}[p(qr)] \Rightarrow \mathsf{C}[(pq)r]) \\ \alpha_7 & : & \forall p^\kappa \forall q^\kappa \forall r^\kappa (\mathsf{C}[(pq)r)] \Rightarrow \mathsf{C}[p(qr)]) \end{array}$$

[5] We do not emphasize on control operators in this introduction, since it is not the focus of this thesis.

For example, α_6 and α_7 constitute the proof of the associativity of $\cdot$ (relatively to C).

As an example of such structure, Miquel explains how original Cohen's original forcing can be reformulated in this framework. Basically, the kind is taken as $\tau \to \iota \to o$, that is the binary relations over $\tau \times \mathbb{N}$. The term $\cdot$ is chosen so it performs the union of two such binary relations. The predicate C is set so as $\mathsf{C}[p]$ expresses the fact that $p \subseteq \tau \times \{0,1\}$, p is a functional relation and p is finite.

Using forcing structures, it is then possible to formalize *inside* $\mathbf{PA}\omega^+$ the forcing interpretation. Forcing usually assigns to every formula A of the theory a set A^* of forcing conditions. Here, when τ is a type (that is a term $\tau : o$), then its forcing translation is $\tau^* : \kappa \to o$, that is a predicate over κ. $(.)^*$ is in fact defined on all kinds and all higher-order terms by induction. Finally the forcing interpretation is defined as

$$p \Vdash A \equiv \forall r^\kappa (\mathsf{C}[pr] \Rightarrow A^*(r))$$

By interpreting C as the complement of an orthogonality pole, this can be understood as the following informal sentence: *For every r in the complement of A^* we have that p is orthogonal to r*. Hence A^* morally represents the complement of the orthogonal of the phase space interpration of a type A.

The main theorem is the soundness of the translation with respect to typing.

Theorem 5 (Miquel 2011)**.** *If the judgment $\mathcal{E}; \Gamma \vdash t : A$ is derivable (in $\mathbf{PA}\omega^+$), then for all forcing conditions p, the sequent $\mathcal{E}^*; (p \Vdash \Gamma) \vdash t^* : (p \Vdash A)$ is derivable too.*

This purely syntactic theorem is the formalization of the soundness of the forcing interpretation. Indeed, it says that if $\vdash A$ is provable, then we have $\mathbf{1} \Vdash A$. But what is interesting here is what happens with the proof-term t.

1.2.2 The **KFAM**

Miquel studies the computational behavior of the transformed programs of the form t^*. He considers the Krivine Abstract Machine [Kri07] with explicit environments: the basic objects are **closures**, **environments** and **stacks**. A closure is a pair (t, e), where t is a term and e an environment, and an environment is a list of assignments of closures to variables:

$$[x_1 := c_1, \ldots, x_n := c_n]$$

A stack is an element of the grammar:

$$\pi \quad ::= \quad \mathsf{nil} \mid c.\pi \qquad \text{where } c \text{ is a closure}$$

The reduction rules are defined in the standard following way (for simplicity, here we do not consider call-cc):

$$\begin{array}{rcccrcc} x[e, y := c] & \star & \pi & \succ & x[e] & \star & \pi \\ x[e', x := c] & \star & \pi & \succ & c & \star & \pi \\ \lambda x.t[e] & \star & c.\pi & \succ & t[e, x := c] & \star & \pi \\ tu[e] & \star & \pi & \succ & t[e] & \star & u[e].\pi \end{array}$$

Miquel unveils the behavior of the transformed programs, which can be synthetized as the following translated reduction steps:

$$\begin{array}{rcccrcc} x[e, y := c]^* & \star & c_0.\pi & \succ^* & x[e]^* & \star & \alpha_9 c_0.\pi \\ x[e', x := c]^* & \star & c_0.\pi & \succ^* & c & \star & \alpha_{10} c_0.\pi \\ \lambda x.t[e]^* & \star & c_0.c.\pi & \succ^* & t[e, x := c]^* & \star & \alpha_6 c_0.\pi \\ tu[e]^* & \star & c_0.\pi & \succ^* & t[e]^* & \star & \alpha_{11} c_0.u[e]^*.\pi \end{array}$$

The first element of the stack plays the role of a *memory cell*, which is protected: even if there is a $\lambda x.t[e]^*$ in head position, this element is not accessed. Here the parameters α_i s are combinators of a certain type[6] and are pushed on the memory cell depending on the reduction step. Each α is a proof of a certain property of the forcing structure. For example, here are the types of some of the combinators:

$$\begin{array}{rcl} \alpha_6 & : & \forall p^\kappa \forall q^\kappa \forall r^\kappa (\mathsf{C}[p(qr)] \Rightarrow \mathsf{C}[(pq)r]) \\ \alpha_{11} & : & \forall p^\kappa \forall q^\kappa (\mathsf{C}[pq] \Rightarrow \mathsf{C}[p(pq)]) \end{array}$$

α_6 is a proof of one part of the associativity property of $\cdot$ and is pushed on the memory cell during a λ-step (when a term of the form $\lambda x.t$ is in head position), when a closure from the stack goes in the environment of the head closure. α_{11} is pushed when the the application of two terms is splitted on the context: the environment is duplicated. In general, if one looks carefully, the combinator α_i describes at the level of the forcing conditions the moves, duplications, erasing of the environments.

After having examined the translated programs, Miquel introduces an abstract machine that implements *natively* the behavior of those programs. This machine, called the **Krivine Forcing Abstract Machine** (or **KFAM**) , extends the usual Krivine Abstract Machine with a native $(.)^*$ constructor that corresponds to the programs seen through the program transformation. Its reduction rules are exactly those of the usual Krivine abstract machine, extended with the previous reduction steps (except that now c^* is a native constructor and not a transformation). To sum up, we can differentiate two *execution modes*:

- The *kernel mode*, where the execution is performed as usual.
- The *user mode*, when we execute a program of the form t^*. In that case, the reduction steps are similar to the kernel mode, except that informations about the execution are stored in a protected memory cell that can't be accessed by the program.

[6] We keep the notations of Miquel's paper [Miq11], so that the reader can refer to it to see what are the different combinators.

Miquel points out a similarity with the ring protection mechanism of the x86 processors, where for instance a program running in ring3 (known as *user space*) does not have access to the reserved memory segment of ring0 (known as *kernel space*).

1.3 | Classical realizability and forcing

1.3.1 Krivine classical realizability

Curry remarked in 1958 the formal analogy between his combinatory logic and the deduction rules of Hilbert system for intuitionistic propositional logic. Several years later, in 1969, Howard [How69] observed the same analogy between Gentzen natural deduction and the lambda-calculus. The Curry-Howard correspondence (or proof-as-program correspondence) was born. Because it showed a correspondence with a rather weak logical system, it was first perceived as an interesting result, but its importance was not yet fully understood. In particular many believed that *classical* proofs could not be given any computational content. Until Griffin's discovery [Gri89] who showed in 1989 that the instruction call-cc of the Scheme language could be given the type of Peirce's Law:

$$((\neg A \Rightarrow A) \Rightarrow A$$

This opened the way for the extension of the Curry-Howard isomorphism to classical calculi. In particular many classical λ-calculi have since been defined: Parigot's $\lambda\mu$-calculus [Par92], Filinski's Symmetric λ-calculus [Fil89] (which, interestingly, slightly precedes Griffin's discovery), Girard's classical logic **LC** [Gir91], Curien and Herbelin $\bar{\lambda}\mu\tilde{\mu}$-calculus [CH00], and more recently Munch's system **L** [MM09].

Shortly after that discovery, Krivine became interested in analyzing the behavior of classical proof terms obtained not only with propositional classical logic but also classical arithmetic or even set theory. This research program raised the issue of associating a computational content to *axioms* of those theories, like the axiom of choice or the continnum hypothesis. Because the classical calculi previously defined could not help with this question, Krivine turned to *semantics*. He adapted Kleene's realizability [Kle45] for intuitionistic arithmetic to a classical setting, which resulted in his theory of classical realizability introduced in a serie of papers [Kri09, Kri03]. Among the early successes of this theory, one could cite the following:

- The computational content [Kri03] of the axiom of dependent choice **DC**, given by the quote instruction of LISP.
- New models of **ZF+DC** [Kri10b] that violate the axiom of choice and the continuum hypothesis. In particular these models **cannot** be obtained using forcing or the technique of inner models.
- Program extraction procedures for a classical extension of the calculus of constructions [Miq07].

Krivine's classical realizability relies on the notion of *orthogonality* already mentioned. His framework is very close to other orthogonality-based works, most notably Girard's ludics [Gir01] and Pitts and Stark's $\top\top$-closure [PS98].

The principle

We give a quick introduction to Krivine's realizability interpretation. We consider the following simple language:

$$\begin{array}{rcll} t, u & ::= & x \mid \lambda x.t \mid (t)u & \\ \pi & ::= & \mathsf{nil} \mid t.\pi & \text{where } t \text{ is closed} \end{array}$$

We consider a set Λ containing at least the closed terms defined by this grammar, and a set Π containing at least the stacks defined by this grammar. A **configuration** is a pair $t \star \pi$ of a closed term $t \in \Lambda$ and a stack $\pi \in \Pi$. We consider a reduction relation $\succ$ that contains the following rules:

$$\begin{array}{rcl} \lambda x.t \star u.\pi & \succ & t[u/x] \star \pi \\ (t)u \star \pi & \succ & t \star u.\pi \end{array}$$

Remark 6. *One important point of methodology is that we* do not fix a syntax, nor a reduction relation. *We just ask that Λ, Π contain a minimal syntax, and that $\succ$ contains a minimal set of reduction rules. This allows to prove general results, and then add later new primitives and new reduction rules without having to reprove everything.*

The construction is parametrized with a pole $\perp\!\!\!\perp$, that is a set of configurations which is $\succ$-saturated:

$$C \succ C' \wedge C' \in \perp\!\!\!\perp \Rightarrow C \in \perp\!\!\!\perp$$

The pole morally corresponds to the set of **correct** configurations. As in phase spaces, it induces an orthogonality relation, between closed programs and stacks:

$$Y^\perp = \{\ t \in \Lambda \mid \forall \pi \in Y, t \star \pi \in \perp\!\!\!\perp\ \}$$

In Krivine classical realizability, since the reduction is done in a call-by-name fashion, the interpretation is *negative*. It means that every type A is interpreted by *two* sets:

- a set $[\![A]\!] \in \mathcal{P}(\Pi)$ of stacks, known as the **falsity value**. This set is defined by induction on the formula A.
- a set $|A| \in \mathcal{P}(\Lambda)$ of closed terms, known as the **truth value**. This set is defined as the orthogonal of the falsity value:
$$|A| = [\![A]\!]^\perp$$

Here are a few inductive cases of the definition of the set $[\![A]\!]$:

$$\begin{array}{rcl} [\![A \Rightarrow B]\!] & = & \{\ t.\pi \mid t \in |A| \wedge \pi \in [\![B]\!]\ \} \\ [\![\forall x^\iota.A]\!] & = & \bigcup_{n \in \mathbb{N}} [\![A[n/x]]\!] \end{array}$$

Definition 7 (Realizability relation)**.** *The **realizability relation** $\Vdash$ is defined as:*

$$t \Vdash A \stackrel{\text{def}}{=} t \in |A|$$

The following property shows very similar patterns to what we observe in forcing or phase spaces. This is a first clue that these frameworks are somehow related.

Property 8.

- *Suppose that $t \Vdash A \Rightarrow B$. Then $u \Vdash A$ implies $(t)u \Vdash B$.*
- *Suppose that $t \Vdash \forall x^{\iota}.A$. Then for all $n \in \mathbb{N}$, we have $t \Vdash A[x := n]$.*

Krivine classical realizability enjoys a soundness theorem, which is parametric in the chosen pole. A simplified version would be as follows:

Theorem 9 (Krivine 2004)**.** *Let $\perp\!\!\!\perp$ be a saturated pole. If $\vdash t : A$ is provable then $t \Vdash A$.*

The soundness theorem makes the link between the syntactic correctness given by the type system and the computational correctness represented by the realizability relation. The strength of Krivine classial realizability lies in its *modularity*. One can indeed extend this soundness theorem to powerful theories, beginning with classical reasoning and going all the way up to the entire **ZF**+**DC**[7] set theory. This methodology allows one to turn these theories into type systems, and give a computational content to all the proofs using these theories. We give two examples.

- Suppose Λ contains the extension of the term grammar with the two following new primitives: call-cc and k_π where $\pi \in \Pi$. Suppose that $\succ$ contains the following new reduction rules:

$$\begin{array}{rcl} \mathsf{call\text{-}cc} \star t.\pi & \succ & t \star k_\pi.\pi \\ k_\pi \star t.\pi' & \succ & t \star \pi \end{array}$$

  Then the following result can be shown : for any saturated pole $\perp\!\!\!\perp$, we have

$$\mathsf{call\text{-}cc} \Vdash (\neg A \Rightarrow A) \Rightarrow A$$

  This justifies a computational content for Peirce's law.

- Suppose Λ contains the extension of the grammar with the following primitive: quote. We denote by $\underline{n}$ the church encoding of the integer n. Let $t \in \Lambda \mapsto n_t \in \mathbb{N}$ be a bijection between Λ and $\mathbb{N}$. Suppose that $\succ$ contains the following new reduction rule:

$$\mathsf{quote} \star t.\pi \quad \succ \quad t \star \underline{n_t}.\pi$$

[7] Dependent Choice

Then the following result can be shown : there is a program $t_{\mathbf{DC}}$ [8] that uses quote such that for any saturated pole $\bot\!\!\bot$, we have

$$t_{\mathbf{DC}} \Vdash \mathbf{DC}$$

The details can be found in Krivine's paper [Kri09].

1.3.2 Realizability algebras: combining forcing and realizability

Recently, Krivine [Kri11] defined the notion of **realizability algebra**, which generalises his classical realizability by abstracting over the sets Λ of terms, Π of stacks, $\Lambda \star \Pi$ of configurations and the reduction relation $>$. The interpretation of classical second-order arithmetic (and in fact, even set theory) can be done in a generic way and proved to be sound in all realizability algebras. Moreover, classical forcing is shown to give an example of such a realizability algebra, when considering the Krivine/Miquel presentation of forcing. In addition, Krivine shows that it is possible to *combine* forcing and classical realizability into a new realizability algebra. This allows to reobtain consistency proofs obtained using forcing, and more importantly, to understand the computational content of those proofs. This construction has been shown by [Str13] to be an instance of topos iteration (which itself is a generalization of the forcing iteration).

Here is a glimpse of the construction, simplified by Miquel in [Miq11] and called **product realizability algebra**. We start with a realizability algebra $\mathcal{A}$ (for example the standard classical realizability algebra), and a forcing structure $(\kappa, \mathsf{C}, \cdot, \mathbf{1})$ in the sense of Subsection 1.2.1. We build a new realizability algebra $\mathcal{A}^*$ as follows:

- $\Lambda^* = \Lambda \times [\![\kappa]\!]$, where $[\![\kappa]\!]$ is the interpretation of the kind κ. For example if $\kappa = \iota$, then $[\![\kappa]\!] = \mathbb{N}$.
- $\Pi^* = \Pi \times [\![\kappa]\!]$
- $\Lambda^* \star \Pi^* = (\Lambda \star \Pi) \times [\![\kappa]\!]$
- $(t, p) \star (\pi, q) = (t \star \pi, p \cdot q)$
- $\bot\!\!\bot^* = \{\ (t \star \pi, p) \mid \forall c \in \Lambda, c \Vdash_{\mathcal{A}} \mathsf{C}(p) \Rightarrow \langle t, c.\pi \rangle \in \bot\!\!\bot\ \}$

Notice that the definition of $\bot\!\!\bot^*$ makes use of the realizability relation induced by the realizability algebra $\mathcal{A}$. Miquel shows that this construction formally corresponds to the iteration of $\mathcal{A}$ and a forcing model chosen *inside* $\mathcal{A}$, in the form of the **connection theorem**. We give a simplified version of the theorem, that can be found in [Miq11].

[8] This program can be written explicitely and makes use of quote, but it has to be noted that the question of its behavior is not studied in Krivine original work.

Theorem 10 (Connection theorem (Miquel 2011))**.** *Let A be a closed formula, $\mathbf{p} \in [\![\kappa]\!]$ and $c \in \Lambda$. Then:*

$$(c, \mathbf{p}) \Vdash_{\mathcal{A}^\star} A \Leftrightarrow c \Vdash_{\mathcal{A}} (\mathbf{p} \ \mathbb{F} \ A)$$

1.4 | Logical relations under the light of forcing

The particular case of quantitative realizability

The technique of **quantitative realizability** introduced by Dal Lago and Hofmann [DLH05, DLH11] is an extension of Kleene's realizability where realizers are annotated by elements of a **resource monoid**. In a realizer (t, p), this element p represents an abstract bound on the *execution time* of the accompanying program t. They have used this technique and some variants of it to prove the bounded-time termination properties of various *light logics* and bounded linear logic [DLH10a]. We showed in [Bru13] that it was possible to reformulate their ideas in the style of Krivine's classical realizability, by transitioning from the notion of resource monoid to the more general notion of **quantitative monoid**. In this previous work, we showed that:

- The annotated realizability relation used in the obtained framework is extremely similar in its definition to Krivine's product realizability algebra mentioned in Subsection 1.3.2.
- Then, following Miquel's methodology, we exhibited a forcing program transformation and a corresponding abstract machine such that the annotated realizability relation could be decomposed as a unary realizability relation built upon this abstract machine, iterated with a (linear) forcing model chosen inside it. informally:

  $$(t, p) \Vdash A \Leftrightarrow t \Vdash (p \ \mathbb{F} \ A)$$

  This abstract machine exhibits a surprising feature: there is a protected memory cell (similarly to the **KFAM**) that contains an integer that is decremented at each reduction step. When it reaches 0, the computation diverges. Thus, if the execution terminates in this machine, it means we have been able to give a *bound on the execution time*. This memory cell accumulates informations on the execution (here the execution time) and possibly change its course: this is what we call **monitoring the execution**.

These observations constitute the origin of the work presented in this thesis.

A forcing-based framework

In this thesis, we advocate the use of forcing as a structuring concept for the development and study of advanced realizability semantics. We propose to generalize what has been done for quantitative realizability, and study the space of *all realizability semantics that are induced by a certain class of forcing program transformations*, based on what we call **forcing monoids**. This yields the notion of **Monitoring Algebras**, that is the main object of study of this thesis.

The use of forcing allows to transport known forcing constructions (like forcing product or iteration) in the world of programming languages semantics, and its algebraic nature makes it possible to design elementary pieces of semantics that we can reuse in many different correctness proofs. The goal being to accumulate results on the monitoring algebras that would make the future design of semantics and correctness proofs easier and shorter, but also to explore the possibilities suggested by the theory and find new results.

Typically, to prove the correctness of a programming language using the theory of monitoring algebras, we proceed with the following steps:

1. Start with a simple monitoring algebra $\mathcal{A}_0$, that is known to entail the correctness of a core programming language, subset of our language.

2. Use the forcing annotations to refine the computational property we are observing, for example bounded-time execution. Here, we use a system similar Miquel's **KFAM** where combinators are being pushed on a protected element of the stack to track informations about the execution of programs (we will often refer to this element as the **protected memory cell**).

3. Using constructions like forcing product or iteration on $\mathcal{A}_0$, successively adding programming features to the corresponding programming language, reusing known techniques.

4. Show using a library of known results about forcing iteration and monitoring algebras that the final monitoring algebra induces a sound realizability model of the programming language we wanted to study.

A central part in the theory of iterated forcing consists in preservation theorems, *i.e.* finding conditions that imply the preservation of certain axioms *after* an iteration. For example, preserving **CH** by iteration is possible only under certain conditions. Preservation theorems will also be a central tool of our methodology. They will ensure that each stage of our construction, we accumulate new programming features but still have a semantics of the language considered at the previous step, and still observe the desired computational property.

At this point, in addition to Krivine and Miquel work on realizability, we should mention other previous works that are somehow related:

- The work of Jaber and Tabareau on the decomposition of logical relations using intuitionistic forcing [JT11]. Although it is similar in spirit to our work, the tools used are very different: they consider intuitionistic forcing whereas we use classical forcing (even though the programming languages considered are not necessarily classical), they use intuitionistic logical relations while we use Krivine's realizability, they consider a higher-order logic that includes propositions on the reduction of terms and primitive abstract observation, while we only consider a simple first-order extension of Levy's call-by-push-value [Lev99].

- The works of Birkedal et al. [BST09, BSS10, BRS+11, BMSS11] on defining step-indexed logical relations by using metric spaces on the one hand, and using topos theory to simplify the definition of step-indexed logical relations on the other. We will reuse many ideas of the metric spaces construction in our treatment of step-indexing. Concerning the second line of work, it relates to our work in the sense that topos theory can be used to rephrase Cohen's forcing. It is however not clear how this work and ours could be formally related.

1.5 | The monitoring power of forcing program transformations

We now give a synthetic but quite wide presentation of the work and results contained in this thesis. The structure of this presentation follows closely the order in which it is exposed in the thesis. Each subsection corresponds to one specific chapter, and contains intuitions about the results and definitions. Hence it should be read before the actual chapter of the thesis.

1.5.1 Chapter II : a linear call-by-push value

The base language we will consider in the thesis is a variant of Levy's call-by-push-value [Lev99, Lev03] that we call λ_{LCBPV}. We made this choice since it decomposes both the call-by-name and the call-by-value λ-calculi. This will allow us to define a realizability framework that can be specialised at will for the call-by-name and the call-by-value paradigms. Moreover, the main ingredient of the call-by-push-value, namely the **polarisation**, simplifies and makes the definition of both the forcing program transformation and the Krivine style realizability semantics more natural[9]. The polarity distinguishes between **positive** connectives (like $\times$, $\exists$, ...) the **negative** connectives (like $\to$, $\forall$, ...). In the call-by-push-value, at the operational level this translates to a syntactical separation between **positive values** and **negative computations**.

Syntax

Our syntax of values and computation is as follows:

$$
\begin{array}{llll}
\text{Values} & v, w & ::= & x \mid (v, w) \mid * \mid \underline{0} \mid \underline{\mathsf{s}}(v) \mid \mathsf{thunk}(t) \mid \varpi \qquad (\text{where } \varpi \in \mathcal{W}) \\
\text{Computations} & t, u & ::= & \mathsf{force}(v) \mid \mathsf{let} * = v \,\mathsf{in}\, u \mid \mathsf{ret}(v) \mid \lambda x.t \mid \\
 & & & (t)v \mid t \,\mathsf{to}\, x.u \mid \mathsf{let}\,(x, y) = v \,\mathsf{in}\, t \mid \\
 & & & \mathsf{case}\, v \,\mathsf{of}\, x.t \parallel x.u \mid \maltese \mid \zeta \qquad (\text{where } \zeta \in \mathcal{K})
\end{array}
$$

[9] We have to remark that the choice of the call-by-push-value is guided essentially by its synthetic syntax and relative ease to use. In [Bru13], we considered another polarised syntax, called system **L** and introduced by Munch [MM09], and this would have worked too. In fact, system **L** generalises call-by-push-value which can be retrieved as a fragment.

This syntax features the unit, pairs, integers (including a successor and a case constructor). Here are some remarks about the more unusual parts of this syntax:

- The thunk() constructor transforms a computation into a value, hence freezing it into a thunk. Dually, the ret() constructor transforms a value into a computation.
- The application $(t)v$ is restricted to values on the right-hand size. This is unusual, but it makes the choice of the order of evaluation not ambiguous. The sequential composition is then implemented by the constructor t to $x.u$: it reduces first t into a value, and then passes it to u. It is this constructor that is responsible for the choice of the reduction strategy.
- The special computation ✠, called the **daimon**, is inherited from Girard's ludics [Gir01]. Once it is executed it stops immediately the computation and consider it a success. It is useful in the realizability semantics when one wants to test only some parts of a program while ignoring some others.
- Finally, the syntax is parametrized by two sets: $\mathcal{W}$ and $\mathcal{K}$. The first is a set of *additional values* one can consider, and the latter is a set of *additional primitives*. This will allow to add new primitives on the fly, and keep the definitions of semantics independent of any language extension.

Reduction

The reduction is given by an abstract machine inspired by the **KAM**. The object being executed is a **configuration**, which is basically a pair composed by a closed computation and an environment.

Environments	E	::=	$\mathsf{nil} \mid \mathsf{a}(v).E \mid \mathsf{f}(x.t).E$	where $FV(t) \subseteq \{x\}$
Configurations	C	::=	$✠ \mid \langle t, E\rangle_0$	where t is closed

An environment is either the empty environment, or a an environment intuitively representing an evaluation context $E((\bullet)v)$ or $E((t)\bullet)$. Notice that in a configuration, no free variable ever appears. The special configuration ✠ means "success", and will be triggered by the daimon computation, which is denoted by the same symbol.

We now consider a *reduction relation* $\xrightarrow{0}$ on configurations. It is a relation between configurations which contains at least the next rules. In this presentation, we only give some representative rules, but they are all defined in detail in Chapter II . The first two rules correspond to the usual β-rule, which are simpler than in the call-by-value for instance, since the argument is always a value:

$$\begin{array}{rcl} \langle (t)v, E\rangle_0 & \xrightarrow{0} & \langle t, \mathsf{a}(v).E\rangle_0 \\ \langle \lambda x.t, \mathsf{a}(v).E\rangle_0 & \xrightarrow{0} & \langle t[v/x], E\rangle_0 \end{array}$$

The next two rules correspond to the usual rules associated to a let constructor *à la Moggi*.

$$\begin{array}{rcl} \langle t \text{ to } x.u, E\rangle_0 & \xrightarrow{0} & \langle t, \mathsf{f}(x.u).E\rangle_0 \\ \langle \mathsf{ret}(v), \mathsf{f}(x.u).E\rangle_0 & \xrightarrow{0} & \langle u[v/x], E\rangle_0 \end{array}$$

The following reduction step corresponds to *unfreezing* a frozen computation:

$$\langle \mathsf{force}(\mathsf{thunk}(t)), E\rangle_0 \xrightarrow{0} \langle t, E\rangle_0$$

Finally we give the daimon reduction rule, which empties the stack and returns the daimon configuration:

$$\langle \maltese, E\rangle_0 \xrightarrow{0} \maltese$$

Notice that we never suppose that the reduction rules are restricted to these rules, but we only suppose that it contains them. This will allow us, like in Miquel's work [Miq11], to consider new reduction rules later on.

Notation 11. *Each reduction rule is determined by the term constructor in head position. If* $@$ *is such a constructor (for example* λ *or* force*), a* $@$*-step is any reduction step associated with a constructor* $@$*. For example, the following step is a* λ*-step:*

$$\langle \lambda x.t, \mathsf{a}(v).E\rangle_0 \xrightarrow{0} \langle t[v/x], E\rangle_0$$

Types

We consider a type system for a fragment of this language. Reflecting the distinction between values and computations, the types are separated into **positive types** denoted by capital letters $P, Q, \ldots$ and **negative types**, denoted by capital letters $N, M, \ldots$. Values will be typed by positive types, while computations correspond to negative types. The grammar is defined by mutual induction:

$$\begin{array}{rcl} P, Q & ::= & X(e_1, \dots, e_n) \mid \Downarrow N \mid \mathsf{Nat} \mid P \otimes Q \mid \exists x \in S.P \mid \{e \preceq_S f\} \wedge P \\ N, M & ::= & \Uparrow P \mid P \multimap N \mid \forall x \in S.N \mid \{e \succeq_S f\} \mapsto N \end{array}$$

This type system will be used as a target for the programming languages we want to give models of, and hence contains basic connectives. But it will also be the target of the forcing translation as in Miquel [Miq11]. That's why it is enriched with first-order quantifiers and inequational implications.

- The implication is *affine*. It means it represents the type of programs that use their argument *at most* once. Notice also that reflecting the application construction which is restricted to value argument, the implication is from positive to negatives.
- Two particular connectives are introduced because of the polarity: the **positive shift** $\Downarrow$ and the **negative shift** $\Uparrow$. They allow to pass from a negative to a positive, and *vice versa*. They correspond to the $\mathsf{thunk}()$ and $\mathsf{ret}()$ constructors.
- The presence of first-order expressions. Those are built using a given set of sets S, an infinite number of variables of sort S (denoted x^S), together with function symbols that denote real functions on sets. Here is an example of first-order expression:
$$x^{\mathbb{N}} + 2$$
- Notice that the first-order quantifiers do not change the polarity, since they are computationally transparent. The domain of these quantifiers can be *any set*, which is the option we choose to internalize our forcing. This is a main difference with Miquel [Miq11], as he considers a higher-order type system where the quantifiers domains are higher-order kind of the language. But externalizing the domain of the quantifiers will allow us to elude some aspects of the forcing transformation we are not interested in, and simplify the definitions.
- Finally, we add an **inequational implication** and an **inequational conjunction**. They are a slight modification of the equational implication used by Miquel. We consider preordered sets $(S, \preceq_S)$ instead of simply sets. In that context the inequational implication intuitively means
$$\{e \succeq_S f\} \mapsto N \equiv \begin{cases} N & \text{if } f \preceq_S e \\ \top & \text{otherwise} \end{cases}$$
where $\top$ is morally type of all computations. While the inequational conjunction intuition is
$$\{e \preceq_S f\} \wedge P \quad \equiv \quad e \preceq_S f \textbf{ and } P$$
These connectives will be useful to internalize our forcing interpretation.

The typing judgments are of the form:

$$\mathcal{E}; \Gamma \vdash_0 v : P \quad \text{or} \quad \mathcal{E}; \Gamma \vdash_0 t : N$$

where:

- $\mathcal{E}$ is an inequational theory, that is a set of inequations between first-order expressions. An example of such an inequation for a set S possessing a binary operation + is:

$$x^S \preceq_S x^S + y^S$$

- Γ is a context of assignments of positive types to variables.

The inequational theory $\mathcal{E}$ will be used to define a subtyping relation $\sqsubseteq_{\mathcal{E}}$. Without giving the details of its definition, here is an example of subtyping with $\mathcal{E} = f \preceq_S g$:

$$\{e \preceq_S f\} \wedge P \sqsubseteq_{\mathcal{E}} \{e \preceq_S g\} \wedge P$$

We give as an example some of the important rules of λ_{LCBPV} in Figure 1. These rules correspond closely to Levy's call-by-push-value, except for the multiplicative treatment of the context (hence the absence of contraction).

$$\frac{}{\mathcal{E}; \Gamma, x : P \vdash_0 x : P}$$

$$\frac{\mathcal{E}; \Gamma \vdash_0 v : P}{\mathcal{E}; \Gamma \vdash_0 \mathsf{ret}(v) : \Uparrow P} \qquad \frac{\mathcal{E}; \Gamma \vdash_0 t : \Uparrow P \qquad \mathcal{E}; \Delta, x : P \vdash_0 u : M}{\mathcal{E}; \Gamma, \Delta \vdash_0 t \text{ to } x.u : M}$$

$$\frac{\mathcal{E}; \Gamma \vdash_0 t : N}{\mathcal{E}; \Gamma \vdash_0 \mathsf{thunk}(t) : \Downarrow N} \qquad \frac{\mathcal{E}; \Gamma \vdash_0 v : \Downarrow N}{\mathcal{E}; \Gamma \vdash_0 \mathsf{force}(v) : N}$$

$$\frac{\mathcal{E}; \Gamma, x : P \vdash_0 t : N}{\mathcal{E}; \Gamma \vdash_0 \lambda x.t : P \multimap N} \qquad \frac{\mathcal{E}; \Gamma \vdash_0 t : P \multimap N \qquad \mathcal{E}; \Delta \vdash_0 v : P}{\mathcal{E}; \Gamma, \Delta \vdash_0 (t)v : N}$$

$$\frac{\mathcal{E}; \Gamma \vdash_0 v : P \qquad \mathcal{E}; \Delta \vdash_0 w : Q}{\mathcal{E}; \Gamma, \Delta \vdash_0 (v, w) : P \otimes Q} \qquad \frac{\mathcal{E}; \Gamma \vdash_0 v : P \otimes Q \qquad \mathcal{E}; \Delta, x : P, y : Q \vdash_0 t : N}{\mathcal{E}; \Gamma, \Delta \vdash_0 \mathsf{let}\,(x, y) \;=\; v \,\mathsf{in}\, t : N}$$

Figure 1: Some typing rules of λ_{LCBPV}

1.5.2 Call-by-value and call-by-name translations

It is well-known that one can retrieve in CBPV both a call-by-name and a call-by-value as fragments. In fact, the purpose of CBPV is to unveil the elementary particles used to build these two calculi. We show that this is still true when we consider our affine variant λ_{LCBPV} of CBPV. We define an affinely typed λ-calculus called λ_{Aff}. Its syntax is as follows:

$$\begin{array}{llll} \mathsf{Terms} & t, u & ::= & x \mid \underline{\mathsf{n}} \mid \underline{\mathsf{s}}(t) \mid \lambda x.t \mid (t)u \mid \mathsf{case}\, t\, \mathsf{of}\, x.u_1 \,\|\, x.u \mid (t,u) \mid \mathsf{let}\,(x,y) = t\,\mathsf{in}\,u \\ \mathsf{Types} & A, B & ::= & \mathsf{Nat} \mid A \multimap B \mid A \otimes B \end{array}$$

- It is a usual λ-calculus with pairs and integers.
- The types are either the type of integers Nat, the tensor $A \otimes B$ or the *affine* map $\multimap$.

We then define a type system $\vdash_{\mathsf{Aff}}$ manipulating sequents of the form

$$x_1 : A, \ldots, x_n : A \vdash_{\mathsf{Aff}} t : B$$

This type system is completely standard. We then endow λ_{Aff} with two different reduction strategies: call-by-name and call-by-value. They are introduced by considering abstract machines:

- In the case of the call-by-name, we execute configurations of the form $\langle t, E\rangle_{\mathsf{N}}$ in a Krivine abstract machine and the reduction relation is denoted by $\to_{\mathsf{N}}$.
- In the case of the call-by-value, we define a notion of values:

  $$v, w \quad ::= \quad \underline{\mathsf{k}} \mid \lambda x.t \mid (v, w)$$

  We execute configurations of the form $\langle t, E\rangle_{\mathsf{V}}$ in an abstract machine whose reduction is denoted by $\to_{\mathsf{V}}$.

We show how it is possible to give two different *typed translations* of λ_{Aff} in λ_{LCBPV} corresponding to the two different strategies. These translations are shown to be both type and reduction preserving. The call-by-name calculus corresponds to the purely negative part of λ_{LCBPV}, while the call-by-value corresponds to the purely positive part of λ_{LCBPV}.

1.5.3 Chapter II The monitoring abstract machine

Miquel's methodology consisted in considering a forcing program transformation first and then giving an abstract machine that implements the behavior of the transformed programs. Here, we choose to present things in a reverse order: we start with an abstract machine called the **Monitoring Abstract Machine** (or **MAM**), and show later that it is justified by a forcing program transformation. As expected and announced, this machine is very much in the spirit of Miquel's **KFAM**. The syntax of λ_{LCBPV} is extended using the following constructors, n ranging over $\mathbb{N}^+$:

$$\begin{array}{lll} t & ::= & \ldots \mid (\!|t|\!)^{\alpha}_{n} \qquad \text{(where } \alpha \text{ is a closed computation)} \\ E & ::= & \ldots \mid \mathsf{m}_n(t).E \qquad \text{(where } t \text{ is a closed computation)} \\ C & ::= & \ldots \mid \langle t, E\rangle_n \qquad \text{(where } t \text{ is a closed computation)} \end{array}$$

- A configuration of the form $\langle t, E\rangle_n$ is said to be *at level* n. We will in fact be interested in configurations of the form:

$$\langle t, \mathsf{a}(v_1).\dots.\mathsf{a}(v_n).E\rangle_n$$

  At level n, the values $v_1, \dots, v_n$ represent the different memory cells used to track informations about the execution.

- The constructor $(\!|t|\!)^{\alpha}_{n}$, called the **observation**, should be seen as an annotation of the program t. When it comes in head position, it triggers the update of the n-th memory cell. This update is performed using the closed program α, called the **monitor**. Hence, depending on the annotations of a program, the memory cells will be updated in different ways.

- We denote by $\mathbb{V}, \mathbb{P}, \mathbb{E}, \mathbb{C}$ respectively the sets of closed values, closed computations, environments and configurations generated by this extended grammar.

In Section 2.3, we define an infinity of reduction relations $\overset{n}{\to}$ for each level $n \in \mathbb{N}$, but here we focus on the levels 0 and 1, to simplify the presentation. While the full language (at any level) is referred to by the notation λ_{Mon}, the restriction to the level 1 is referred to by $\lambda^{\mathbf{1}}_{\mathsf{Mon}}$. In that case, the machine can be seen as a usual Krivine Abstract Machine augmented with a protected memory cell, with two different modes of execution:

- The level 1 mode, where the execution does not access or modify the memory cell.

- The level 0 mode, which we enter at certain times of the execution, where the memory cell is accessed and updated using a special program called the monitor. Depending on the content of the memory cell, the monitor can take different actions (updating the memory cell, stopping the execution, etc.).

The reduction relation, denoted by $\overset{1}{\to}$, contains the following schemes of rules that are defined using $\overset{0}{\to}$.

- The first scheme of reduction rules only says that $\overset{1}{\to}$ contains $\overset{0}{\to}$. It is defined as a rule, saying every $\overset{0}{\to}$ reduction step induces a corresponding $\overset{1}{\to}$ reduction step.

$$\frac{C \overset{0}{\to} C'}{C \overset{1}{\to} C'}$$

- The second scheme of reduction rules extends the relation $\overset{0}{\to}$ by adding exactly one memory cell. Each $\overset{0}{\to}$ reduction step induces a new reduction step that does not alter the value in that memory cell.

$$\frac{\langle t, E\rangle_0 \xrightarrow{0} \langle t', E'\rangle_0}{\langle t, \mathsf{a}(v).E\rangle_1 \xrightarrow{1} \langle t', \mathsf{a}(v).E'\rangle_1}$$

- We also distinguish the case when the daimon configuration is observed:

$$\frac{\langle t, E\rangle_0 \xrightarrow{0} \maltese}{\langle t, \mathsf{a}(v).E\rangle_1 \xrightarrow{1} \maltese}$$

- The next rule allows to enter what we call the **monitoring mode**. It happens when an observation of the form $(\!|t|\!)^\alpha_1$ is in head position, and triggers the introduction of the monitor α, but at level 1:

$$\langle (\!|t|\!)^\alpha_1, \mathsf{a}(v).E\rangle_1 \xrightarrow{1} \langle \alpha, \mathsf{a}(v).\mathsf{m}_1(t).E\rangle_0$$

  Being at level 1, α can then access the value v and modify it.

- We introduce a rule to exit the monitoring mode at level 1. Once the computation triggered by a previous use of the monitor α has ended on a value, we can use the following rule to put that value into the monitoring state:

$$\langle \mathsf{ret}(v), \mathsf{m}_1(t).E\rangle_0 \xrightarrow{1} \langle t, \mathsf{a}(v).E\rangle_1$$

1.5.4 Examples of reduction at level ≤ 1

To give some intuitions on the monitoring abstract machine, we show several examples of reductions of a same configuration

$$\langle (\!|(\!|I|\!)^\alpha_1 \underline{0}|\!)^\alpha_1, \mathsf{a}(v).E\rangle_1$$

but for different choices of α and different choice of the value v initially put in the state. The reduction steps which are about the manipulation of the monitoring state have been highlighted in blue. More examples are given in Subsection 2.3.2.

Clock

We consider the **clock monitor**

$$\alpha = \lambda x.\mathsf{ret}(\underline{\mathsf{s}}(x))$$

We show the reduction sequence of $(\!|(\!|I|\!)^{\alpha}_1 \underline{0}|\!)^{\alpha}_1$ and the term $\underline{3}$ in the memory cell.

$$\begin{array}{rcl}
\langle (\!|(\!|I|\!)^{\alpha}_1 \underline{0}|\!)^{\alpha}_1, \mathsf{a}(\underline{3}).E\rangle_1 & \xrightarrow{1} & \langle \lambda x.\mathsf{ret}(\underline{\mathsf{s}}(x)), \mathsf{a}(\underline{3}).\mathsf{m}_1((\!|I|\!)^{\alpha}_1 \underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle \mathsf{ret}(\underline{4}), \mathsf{m}_1((\!|I|\!)^{\alpha}_1 \underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle (\!|I|\!)^{\alpha}_1 \underline{0}, \mathsf{a}(\underline{4}).E\rangle_1 \\
& \xrightarrow{1} & \langle (\!|I|\!)^{\alpha}_1, \mathsf{a}(\underline{4}).\mathsf{a}(\underline{0}).E\rangle_1 \\
& \xrightarrow{1} & \langle \lambda x.\mathsf{ret}(\underline{\mathsf{s}}(x)), \mathsf{a}(\underline{4}).\mathsf{m}_1(I).\mathsf{a}(\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle \mathsf{ret}(\underline{5}), \mathsf{m}_1(I).\mathsf{a}(\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle I, \mathsf{a}(\underline{5}).\mathsf{a}(\underline{0}).E\rangle_1 \\
& \xrightarrow{1} & \langle \mathsf{ret}(\underline{0}), \mathsf{a}(\underline{5}).E\rangle_1
\end{array}$$

This example witnesses one possible use of the monitoring state: a counter incremented when certain parts of a λ-term are executed. This can be used to implement a *clock*.

Countdown: divergence

Here is another example. We now choose the following monitor:

$$\alpha = \lambda x.\mathsf{case}\, x\, \mathsf{of}\, x.\Omega \parallel x.\mathsf{ret}(x)$$

We show two different executions: when the state is initially set to $\underline{2}$ and $\underline{1}$.

$$\begin{array}{rcl}
\langle (\!|(\!|I|\!)^{\alpha}_1 \underline{0}|\!)^{\alpha}_1, \mathsf{a}(\underline{2}).E\rangle_1 & \xrightarrow{1} & \langle \alpha, \mathsf{a}(\underline{2}).\mathsf{m}_1((\!|I|\!)^{\alpha}_1 \underline{0}).E\rangle_0 \\
& \xrightarrow{1}{}^{*} & \langle \mathsf{ret}(\underline{1}), \mathsf{m}_1((\!|I|\!)^{\alpha}_1 \underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle (\!|I|\!)^{\alpha}_1 \underline{0}, \mathsf{a}(\underline{1}).E\rangle_1 \\
& \xrightarrow{1} & \langle (\!|I|\!)^{\alpha}_1, \mathsf{a}(\underline{1}).\mathsf{a}(\underline{0}).E\rangle_1 \\
& \xrightarrow{1} & \langle \alpha, \mathsf{a}(\underline{1}).\mathsf{m}_1(I).\mathsf{a}(\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle \mathsf{ret}(\underline{0}), \mathsf{m}_1(I).\mathsf{a}(\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle I, \mathsf{a}(\underline{0}).\mathsf{a}(\underline{0}).E\rangle_1 \\
& \xrightarrow{1} & \langle \mathsf{ret}(\underline{0}), \mathsf{a}(\underline{0}).E\rangle_1
\end{array}$$

Here the counter is decremented and the execution is somewhat similar to the previous example. Now here is what happens when we put $\underline{1}$ in the monitoring state.

$$\begin{array}{rcl}
\langle (\!|(\!|I|\!)^{\alpha}_1 \underline{0}|\!)^{\alpha}_1, \mathsf{a}(\underline{1}).E\rangle_1 & \xrightarrow{1} & \langle \alpha, \mathsf{a}(\underline{1}).\mathsf{m}_1((\!|I|\!)^{\alpha}_1 \underline{0}).E\rangle_0 \\
& \xrightarrow{1}{}^{*} & \langle \mathsf{ret}(\underline{0}), \mathsf{m}_1((\!|I|\!)^{\alpha}_1 \underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle (\!|I|\!)^{\alpha}_1 \underline{0}, \mathsf{a}(\underline{0}).E\rangle_1 \\
& \xrightarrow{1} & \langle (\!|I|\!)^{\alpha}_1, \mathsf{a}(\underline{0}).\mathsf{a}(\underline{0}).E\rangle_1 \\
& \xrightarrow{1} & \langle \alpha, \mathsf{a}(\underline{0}).\mathsf{m}_1(I).\mathsf{a}(\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle \Omega, \mathsf{m}_1(I).\mathsf{a}(\underline{0}).E\rangle_0 \\
& \Uparrow & \\
\end{array}$$

Here, the monitor plays the role of a countdown: it is a counter that makes the computation diverge once it reaches $\underline{0}$. This example shows that changing the value in the memory cell can affect the termination of the resulting configuration. In general, a program will be annotated by various monitors. Observing the termination of such an annotated program will in fact account to observe a more refined property of the source program. For example, the previous example shows that if we annotated a term with an observation on each redex, observing termination for a given value $\underline{k}$ of the memory cell in fact amounts to say that the execution time is bounded by k.

1.5.4.1 Program transformation

In Section 2.4 is exposed a program transformation from the monitoring extension at level 1 $\lambda^{\mathbf{1}}_{\mathsf{Mon}}$ into the base calculus λ_{LCBPV}. It is *untyped* and will be extended to a *typed* program transformation in a Chapter III . This program transformation is shown to preserve the dynamic semantics of the monitoring abstract machine. For the most part, it is an adaptation to the call-by-push-value of the usual monadic program transformation corresponding to the state monad [Mog91], with a straightforward variation concerning the observation constructor. This translation consists in:

- A translation function $\langle\!\langle . \rangle\!\rangle$ that maps $\lambda^{\mathbf{1}}_{\mathsf{Mon}}$ values to λ_{LCBPV} values, and $\lambda^{\mathbf{1}}_{\mathsf{Mon}}$ computations to λ_{LCBPV} computations.
- A translation of execution environments. Since there are two modes of execution, we need two different maps, the first $\langle\!\langle . \rangle\!\rangle$ is used to translate environments used during executions in the normal mode. The second map $[\![.]\!]$ corresponds to the monitoring mode.
- Finally, the translation $\langle\!\langle . \rangle\!\rangle$ of configurations is given as follows:

$$\begin{array}{rcl} \langle\!\langle \langle t, \mathsf{a}(v).E \rangle_1 \rangle\!\rangle & = & \langle \langle\!\langle t \rangle\!\rangle, \mathsf{a}(v).\langle\!\langle E \rangle\!\rangle \rangle_0 \\ \langle\!\langle \langle t, E \rangle_0 \rangle\!\rangle & = & \langle t, [\![E]\!] \rangle_0 \end{array}$$

The simulation theorem 69 holds for the program transformation. It basically says that the transformation preserves the reduction of $\lambda^{\mathbf{1}}_{\mathsf{Mon}}$.

Theorem 12 (Simulation)**.** *The following propositions hold:*

$$C \xrightarrow{1} C' \text{ implies } \langle\!\langle C \rangle\!\rangle \xrightarrow{0}{}^{*} \langle\!\langle C' \rangle\!\rangle$$

1.5.5 Chapter III : a novel forcing transformation

The second main contribution of this thesis, which is the object of Chapter III , explains how the the monitoring abstract machine can be obtained through a forcing program transformation of the abstract machine associated with the linear call-by-push-value. Unlike Krivine's forcing transformation which is based on a notion of forcing structure which is *internal* to realizability in [Kri11], or to a higher-order type system in Miquel [Miq11], ours rely on an *external*, set-theoretic notion of **forcing monoid**.

1.5.5.1 Forcing monoids

Forcing monoids constitute one of the main ingredients of our construction, as it is the structure we will use to represents annotations in our realizability framework. The elements of a forcing monoid are what describes the content of a memory cell in the **MAM**. A **forcing monoid** $\mathcal{M} = (\mathcal{M}, +, \mathbf{0}, \bullet, \leq)$ is given by:

- A preordered commutative monoid $(\mathcal{M}, +, \mathbf{0}, \leq)$.
- A left action $\bullet$ of $\mathcal{M}$ over $\mathcal{M}$. In particular, we require that:

$$\forall p, q, r \in \mathcal{M}, (p + q) \bullet r = p \bullet (q \bullet r)$$

Here is an intuitive explanation of the different components of a forcing monoid.

- The elements of $\mathcal{M}$ will be used to annotate programs (or types). They will represent various notions. They will play the role of bounds on the execution time, step-indices, or security labels.
- The element $\mathbf{0}$ is the element that carries the less information, as it is the least element for $\leq$: $\mathbf{0} \leq p$.
- The commutative operation $+$ corresponds to the tensor connective, hence representing how to combine the annotations of two values when forming a pair. It also corresponds to the application of a computation to a value.
- The action $\bullet$ represents how the annotations should be combined during the *interaction* between a program and an environment. Here is a comparison between the application reduction step and the action law:

$$\begin{array}{lcl} \langle (t)v, E\rangle_0 & \xrightarrow{0} & \langle t, \mathsf{a}(v).E\rangle_0 \\ (p + q) \bullet r & = & p \bullet (q \bullet r) \end{array}$$

Here are some examples of forcing monoids (called **additive** because $\bullet = +$):

- The structure based on the set of integers $\mathbb{N}$, and $+$ and $\bullet$ being both the usual addition on $\mathbb{N}$. The preorder is the usual order on $\mathbb{N}$.
- The structure based on the set of integers $\overline{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$, obtained by adjoining ∞ to $\mathbb{N}$. The $+$ and $\bullet$ operations are then $\min$ (with ∞ being its neutral). Finally the preorder is the reverse order $\geq$ on $\overline{\mathbb{N}}$.

The forcing monoid corresponds to the commutative monoid in a phase space. To be complete we also need what corresponds to the pole of a phase space. This is exactly what the **forcing structure** represents. It is given by:

- A forcing monoid $\mathcal{M}$.

- A positive predicate C of arity $\mathcal{M}$ such that $p \preceq q$ implies that $\mathsf{C}(q) \sqsubseteq \mathsf{C}(p)$.

As in Krivine program transformation, this C represents intuitively the negation of the pole of a phase space. Notice that the forcing structures encapsulate an internal and an external notion.

1.5.5.2 An annotated type system

In Section 3.2, we define an annotated type system, which is parametrized by a forcing structure $\mathcal{F} = (\mathcal{M}, \mathsf{C})$. It manipulates judgments of the form:

$$\mathcal{E}; \Gamma \vdash_n t : (N, p) \quad \text{or} \quad \mathcal{E}; \Gamma \vdash_n v : (P, p)$$

where $n \in \mathbb{N}$ represents a level of the **MAM**, $\mathcal{E}$ is an inequational theory, Γ is a positive context and $p \in \mathcal{M}$ is a forcing condition. This type system should be seen more as a way to assign an element of the forcing monoid to each λ_{LCBPV} derivation. It will also be useful to represent semantical judgments, that is rules that are derivable *inside* a given annotated realizability model. Here is a sample of the annotations of the typing rules of λ_{LCBPV} (which corresponds to the same set of rules we have presented in the previous section).

$$\frac{}{\mathcal{E}; x : P \vdash_n x : (P, \mathbf{0})} \qquad \frac{\mathcal{E}; \Xi \vdash_n v : (P, p)}{\mathcal{E}; \Xi \vdash_n \mathsf{ret}(v) : (\Uparrow P, p)}$$

$$\frac{\mathcal{E}; \Xi \vdash_n t : (\Uparrow P, p) \qquad \mathcal{E}; \Upsilon, x : P \vdash_n u : (N, q)}{\mathcal{E}; \Xi, \Upsilon \vdash_n t \text{ to } x.u : (N, p + q)}$$

$$\frac{\mathcal{E}; \Xi \vdash_n t : (N, p)}{\mathcal{E}; \Xi \vdash_n \mathsf{thunk}(t) : (\Downarrow N, p)} \qquad \frac{\mathcal{E}; \Xi \vdash_n v : (\Downarrow N, p)}{\mathcal{E}; \Xi, \Upsilon \vdash_n \mathsf{force}(v) : (N, p)}$$

$$\frac{\mathcal{E}; \Xi, x : P \vdash_n t : (N, p)}{\mathcal{E}; \Xi \vdash_n \lambda x.t : (P \multimap N, p)} \qquad \frac{\mathcal{E}; \Xi \vdash_n t : (P \multimap N, p) \qquad \mathcal{E}; \Upsilon \vdash_n v : (P, q)}{\mathcal{E}; \Xi, \Upsilon \vdash_n (t)v : (N, p + q)}$$

$$\frac{\mathcal{E}; \Xi \vdash_n v : (P, p) \qquad \mathcal{E}; \Upsilon \vdash_n w : (Q, q)}{\mathcal{E}; \Xi, \Upsilon \vdash_n (v, w) : (P \otimes Q, p + q)}$$

$$\frac{\mathcal{E}; \Xi \vdash_n v : (P \otimes Q, q) \qquad \mathcal{E}; \Upsilon, x : P, y : Q \vdash_n t : (N, p)}{\mathcal{E}; \Xi, \Upsilon \vdash_n \mathsf{let}\,(x, y) \;=\; v \,\mathsf{in}\, t : (N, p + q)}$$

This annotated type system also allows us to type the observation constructor $(\!|.|\!)^{\alpha}_{1}$. Reflecting the fact that the computation rule makes the configuration level go from 1 to 0, the corresponding typing rule uses a premise in the λ_{LCBPV} type system (that is, at level 0).

$$\frac{\mathcal{E};\Gamma \vdash_1 t : (N,p) \qquad \mathcal{E};\vdash_0 \alpha : \forall x \in \mathcal{M}.\mathsf{C}(f(x)) \multimap \mathsf{C}(x)}{\mathcal{E};\Gamma \vdash_1 (\!|t|\!)_1^\alpha : (N, f(p))}$$

This rule explains how a monitor α is allowed to modify the memory cell: it has to stay in the set given by a certain $\mathsf{C}(x)$ and has to make it pass from $\mathsf{C}(f(p))$ to $\mathsf{C}(p)$ for every $p \in \mathcal{M}$. Under certain conditions, it is also possible to give a general rule for the observation at any level, but it is a little bit more complicated and will be treated later in the thesis.

1.5.5.3 The type translation

Still following Miquel's methodology, we introduce a formalization of a linear forcing interpretation, inside λ_{LCBPV}. Morally, it corresponds to a certain phase space interpretation for the call-by-push-value underlying logic. It is similar to the phase semantics we were talking about in Subsection 1.1.2, and with Krivine/Miquel forcing type translation mentioned in Subsection 1.2.1. Here are some remarkable differences:

- In addition to the usual components of a phase space, one has to deal with the preorder.
- A positive type is interpreted as a set of forcing conditions, but is not closed by biorthogonality (unlinke negative types). It is however upward-closed for the preorder $\preceq$.
- A negative type is still interpreted as the orthogonal of another set.
- The interpretation of the two shift connectives $\Uparrow$ and $\Downarrow$ corresponds to the orthogonality operator.

The forcing interpretation is parametrized by the choice of a forcing structure $\mathcal{F} = (\mathcal{M}, \mathsf{C})$. It is then given in the form of three maps:

- For any positive type P, we associate a positive predicate $P^*(p)$.
- For any negative type N, we associate two negative predicates:
  - $N^\circ(p)$, which corresponds to the falsity value $[\![A]\!]$ in Krivine's classical realizability.
  - $N^*(p)$, which is morally defined as the orthogonal of the "set" $N^\circ(.)$ and corresponds to the negative truth value of Krivine's classical realizability.

Here is a fragment of the definition of this type translation:

Positive interpretation

$$\begin{array}{rcl} (\mathsf{Nat})^*(p) & = & \mathsf{Nat} \\ (P \otimes Q)^*(p) & = & \exists q_1 \in \mathcal{M}.\exists q_2 \in \mathcal{M}.\{q_1 + q_2 \preceq_{\mathcal{M}} p\} \wedge (P^*(q_1) \otimes Q^*(q_2)) \\ (\Downarrow N)^*(p) & = & \Downarrow(\forall r \in \mathcal{M}.\mathsf{C}(p \bullet r) \multimap N^\circ(r)) \end{array}$$

Negative falsity value

$$\begin{array}{rcl}(P \multimap M)^\circ(p) & = & \exists q \in \mathcal{M}.\exists r \in \mathcal{M}.\{p \geq_{\mathcal{M}} q \bullet r\} \mapsto (P^*(q) \multimap M^\circ(r)) \\ (\Uparrow P)^\circ(p) & = & \Uparrow(\exists r \in \mathcal{M}.P^*(r) \otimes \mathsf{C}(r \bullet p))\end{array}$$

Negative truth value

$$N^*(p) \quad = \quad \forall r \in \mathcal{M}.\mathsf{C}(p \bullet r) \multimap N^\circ(r)$$

We also define the following notation:

$$p \Vdash_{\mathcal{F}} A \stackrel{\text{def}}{=} A^*(p)$$

Remark 13. *As we said, one can think of* $\mathsf{C}(p \bullet q)$ *as the negation* $\neg(p \perp q)$ *of the orthogonality relation in phase spaces. With this interpretation in mind, if we look at the interpretation of* $\Uparrow$*, and forget about the polarities, we can intuitively interpret:*

$$q \in (\Uparrow P)^* \Leftrightarrow \exists r \in \mathcal{M}, \neg(q \perp r) \wedge r \in P^*$$

Now, if we look at the interpretation of $\Downarrow\Uparrow P$ *we obtain:*

$$p \in (\Downarrow\Uparrow P)^* \Leftrightarrow \forall r \in \mathcal{M}, \neg(r \perp p) \Rightarrow (\exists q \in \mathcal{M}, \neg(q \perp r) \wedge q \in P^*)$$

That is,

$$\begin{array}{rcl} p \in (\Downarrow\Uparrow P)^* & \Leftrightarrow & \forall r \in \mathcal{M}, (\forall q \in \mathcal{M}, q \in P^* \Rightarrow q \perp r) \Rightarrow p \perp r \\ & \Leftrightarrow & p \in P^{*\perp\perp} \end{array}$$

The main result of this section is the following preservation theorem, which internalizes the soundness of this forcing interpretation. It is stated using the annotated type system of the previous subsection, used with the same forcing structure as the forcing translation.

Theorem 14 (Preservation theorem)**.** *The two following statements hold:*

1. *Suppose that the following judgment is derivable:*

$$\mathcal{E}; x_1 : P_1, \ldots, x_n : P_n \vdash_1 v : (P, p)$$

Then we also have for any distinct variables $\iota_1, \ldots, \iota_n$*:*

$$\mathcal{E}; x_1 : P_1^*(\iota_1), \ldots, x_n : P_n^*(\iota_n) \vdash_0 \langle\!\langle v \rangle\!\rangle : P^*(p + \sum_{1 \leq i \leq n} \iota_i)$$

2. *Suppose that the following judgment is derivable:*

$$\mathcal{E}; x_1 : P_1, \ldots, x_n : P_n \vdash_1 t : (N, p)$$

Then we also have for any distinct variables $\iota_1, \ldots, \iota_n$*:*

$$\mathcal{E}; x_1 : P_1^*(\iota_1), \ldots, x_n : P_n^*(\iota_n) \vdash_0 \langle\!\langle t \rangle\!\rangle : N^*(p + \sum_{1 \leq i \leq n} \iota_i)$$

Remark 15. *This program transformation and the corresponding results present very strong similarities with the work of Pottier [Pot11] on a store-passing translation for general references.*

1.5.6 Chapter IV : Krivine style realizability

Exactly like Miquel defines a unary realizability framework based on his **KFAM**, we define a unary realizability framework based on the **MAM**. This unary realizability will constitute the corner-stone of our theory. Like in classical realizability [Kri09], the interpretation is parametrized by what is called a **pole**. It is a set $\perp\!\!\!\perp$ of configurations (at any level), which is $\to$-saturated:

$$C \to C' \wedge C' \in \perp\!\!\!\perp \Rightarrow C \in \perp\!\!\!\perp$$

That pole morally represents the set of *good* configurations, or *correct* configurations. This is the notion of computational correctness we want to study. For example, here are some possible poles:

1. The pole of terminating configurations

$$\perp\!\!\!\perp_{✠} = \{\ C \mid C \text{ reduces to either } \left\{ \begin{array}{l} \langle v, \mathsf{a}(v_n). \ldots . \mathsf{a}(v_1).\mathsf{nil} \rangle_n \\ ✠ \end{array} \right. \ \}$$

   This is an adaptation of the pole used to prove termination, and is also similar to the one used in Girard's ludics [Gir01].

2. The pole of diverging configurations

$$\perp\!\!\!\perp_{\Omega} = \{\ C \mid C \text{ diverges } \ \}$$

This pole induces a notion of **orthogonality**, similarly to what is done in Krivine's realizability.

$$Y^{\perp_n} = \{\ t \mid \forall E \in Y, \langle t, E \rangle_n \in \perp\!\!\!\perp \ \}$$

The interpretation of types is built around this orthogonality operation. The intuition is that we assign to each positive type P a set of values $\|P\|$ called its **truth value**, and to each negative connective N a set of environments $[\![N]\!]$ called its **falsity value**. The **truth value** of a negative N is then the orthogonal of $[\![N]\!]$. Moreover, since the orthogonality operation is indexed by integers (corresponding to different levels of execution), so is the interpretation. We only give the most important inductive cases, and forget about first-order:

Example of positive truth value

$$\begin{array}{rcl} \|P \otimes Q\|_n & = & \{\ (v,w) \mid v \in \|P\|_n \wedge w \in \|Q\|_n\ \} \\ \|\Downarrow N\|_n & = & \{\ \mathsf{thunk}(t) \mid t \in [\![N]\!]_n^{\perp_n}\ \} \end{array}$$

Example of negative falsity value

$$\begin{array}{rcl} [\![P \multimap N]\!]_n & = & \{\ v.E \mid v \in \|P\|_n \wedge E \in [\![N]\!]_n\ \} \\ [\![\Uparrow P]\!]_n & = & \|P\|_n^{\perp_n} \end{array}$$

Definition of the negative truth value

$$\|N\|_n \quad = \quad [\![N]\!]_n^{\perp_n}$$

Here are some remarks about the interpretation:

- If one only looks at the negative part of the interpretation, one recognizes a similar pattern to the usual Krivine classical realizability. For example, a negative is intrepreted as the orthogonal of a set of stacks, as in Krivine realizability. Another similarity concerns the interpretation of the implication, which is almost the same. And indeed, if one looks at the fragment of λ_{LCBPV} interpreting the call-by-name translation, one gets back Krivine classical realizability. The same phenomenon can be observed in [MM09]. [10]
- One important remark is that the interpretation of the implication and of the tensor are somewhat similar. This is because the $\multimap$ connective is in linear logic decomposed as $P \multimap N = P^\perp \parr N = (P \otimes N^\perp)^\perp$. Hence, morally the $[\![N]\!]$ interpretation corresponds to a *positive* interpretation of $N^\perp$.

Our realizability framework enjoys a soudness theorem, exactly like Krivine realizability. This soundness theorem shows the correction of the **level** 0 **interpretation**. Here is a simplified version of this result that we prove in Section 4.1:

Theorem 16. *Let $\perp\!\!\!\perp$ be a saturated pole. Suppose that $\vdash_0 t : N$. Then we have $t \in \|N\|_0$.*

As a corollary, we could very well obtain a proof of the termination of λ_{LCBPV}.

[10] Again, we used call-by-push-value because it is convenient, but any other polarised linear logic based system would have been good. Our interpretation is very similar to Munch system **L** version of Krivine classical realizability [MM09], and could in fact be retrieved from it.

1.5.7 Chapter IV : Monitoring algebras

The central notion of this thesis is the n-**Monitoring Algebra**, or simply n-**MA**. The first intuition is that an n-**MA** internalizes the combination of a unary realizability model *à la Krivine* with a forcing program transformation that is chosen *inside this realizability model*. Its definition and the study of its main basic properties are given in Chapter IV .

Definition

A n-**MA** $\mathcal{A}$ is given by the two following components:

- A forcing monoid $|\mathcal{A}|$
- A **test function** $\mathcal{C}_{\mathcal{A}} : |\mathcal{A}| \to \mathcal{P}(\mathbb{V}^n)$, which is decreasing:

$$p \preceq q \Rightarrow \mathcal{C}_{\mathcal{A}}(q) \subseteq \mathcal{C}_{\mathcal{A}}(p)$$

The intuition behind the two components can be understood as follows:

- The forcing monoid $|\mathcal{A}|$ represents abstract pieces of information about the computation performed by a configuration, which is accumulated during the monitoring execution. Those pieces of information help refining what is *observed*. It is the algebraic part of a n-**MA** that gives them a great flexibility: algebraic constructions on forcing monoids will be reflected at the programming level.
- The test function $\mathcal{C}_{\mathcal{A}}$ associates with every abstract piece of information $p \in |\mathcal{A}|$ a concrete information represented as a set of tuples of values $\mathcal{C}_{\mathcal{A}}(p)$. These values are intended to be put in the memory cells of the monitoring abstract machine. As we have already seen, changing the content of the memory cells can impact the execution itself, hence the name *test function*.

By reusing the pole $\perp\!\!\!\perp$ of the unary realizability, each n-**MA** $\mathcal{A}$ naturally induces a family of orthogonality relations between computations and environments denoted by $\{\bot_{\mathcal{A},k}\}_{k\in\mathbb{N}}$:

$$(t,p)\bot_{\mathcal{A},k}(E,q) \Leftrightarrow \forall (v_1,\dots,v_n) \in \mathcal{C}_{\mathcal{A}}(p \bullet q), \langle t, \mathsf{a}(v_1)\dots\mathsf{a}(v_n).E\rangle_k \in \perp\!\!\!\perp$$

We can reformulate the orthogonality using an intermediate set called the $\mathcal{A}$-**pole**:

$$\perp\!\!\!\perp_{\mathcal{A},k} = \{\ (t,E,p) \mid \forall \overrightarrow{v} \in \mathcal{C}_{\mathcal{A}}(p), \langle t, \mathsf{a}(\overrightarrow{v}).E\rangle_k \in \perp\!\!\!\perp\ \}$$

Then we have $(t,p)\bot_{\mathcal{A},k}(E,q) \Leftrightarrow (t,E,p+q) \in \perp\!\!\!\perp_{\mathcal{A},k}$. Notice the similarity with the definition of Remark 4. Here it is quite clear that the forcing condition helps *refining* the orthogonality, thanks to the values given by the test function $\mathcal{C}_{\mathcal{A}}$. Those values are intended as tests that the interaction of the computation and the environment has to pass. For example, if the computation t is annotated with the observation $(\!|.|\!)^{\alpha}_1$ with α being the monitor that decrements a counter and diverges when it reaches 0, then depending on the values we put in the memory cell, we can observe the number of steps needed by the configuration to terminate.

Interpretation

Each n-**MA** defines an indexed interpretation of types. It is similar to the unary realizability interpretation, with the addition of elements of the forcing monoid. In fact, each type will be interpreted by truth and falsity values, as in the unary case. The set $\mathcal{I}(\mathbb{V}_{\mathcal{A}})$ of **positive truth values** is the set of all sets $X \subseteq \mathbb{V} \times |\mathcal{A}|$ such that if $(v,p) \in X$ and $p \preceq q$, then $(v,q) \in X$. Similarly, the set of **negative truth values** $\mathcal{I}(\mathbb{P}_{\mathcal{A}})$ contains all sets $X \subseteq \mathbb{P} \times |\mathcal{A}|$ such that $(t,p) \in X$ and $p \preceq q$ then $(c,q) \in X$. Finally the set $\mathcal{I}(\mathbb{E}_{\mathcal{A}})$ of **negative falsity values** contains the sets $Y \subseteq \mathbb{E} \times |\mathcal{A}|$ such that if $(E,p) \in Y$ and $p \preceq q$, then $(E,q) \in Y$. We give different examples:

$$\|P \otimes Q\|_{\mathcal{A},k} = \{\ ((v,w), p+q) \mid (v,p) \in \|P\|_{\mathcal{A},k} \wedge (w,q) \in \|Q\|_{\mathcal{A},k}\ \}$$

$$[\![P \multimap N]\!]_{\mathcal{A},k} = \{\ (\mathsf{a}(v).E, p \bullet q) \mid (v,p) \in \|P\|_{\mathcal{A},k} \wedge (E,q) \in [\![N]\!]_{\mathcal{A},k}\ \}$$

$$[\![\Uparrow P]\!]_{\mathcal{A},k} = \|P\|_{\mathcal{A},k}^{\perp_{\mathcal{A},k}}$$

As we see, without the forcing part, the interpretation is just the same as in the unary realizability case. Moreover, if we consider only the forcing part, it really is similar to the phase space interpretation with the slight difference that + becomes $\bullet$ in certain cases. The first important result about n-**MA**s is the soudness theorem, that says that every model induced by a n-**MA** is sound with respect to λ_{LCBPV}, in a slightly different sense than the unary realizability model. Indeed, it basically says that if $\vdash_0 t : N$ is provable, then there exists some forcing condition $p \in |\mathcal{A}|$ such that $(t,p) \in \|N\|_{\mathcal{A},n}$. This p can be computed using the forcing annotated type system. A simplified version of Theorem 140 is as follows:

Theorem 17 (Soundness)**.** *Let $\mathcal{A}$ be a n-**MA**. If $\vdash_n t : (N,p)$ is provable in the annotated type system then $(t,p) \Vdash_{\mathcal{A},n} N$.*

$\mathcal{A}$-monitors

The previous soundness result covers only the annotated version of the λ_{LCBPV} rules, but if one looks at the rules, the forcing condition obtained is only built using $\mathbf{0}$ and the + operation (hence is always $\mathbf{0}$). Hence it means that the programs we can type don't really use the memory cell. That's natural since we have not given any sound annotated typing rule concerning the observation constructor $(\!|.|\!)^{\alpha}_{n}$, which is responsible for the update of the memory cell. It is not possible to give a general typing rule for the observation. However, it is possible to give a general condition such that if it is met, we can type it. We introduce what we call $\mathcal{A}$-**monitors**. Basically, it is a pair (α, f) where α is a closed computation and f a strong function, *i.e.* such that

$$f(p \bullet q) \preceq_{|\mathcal{A}|} f(p) \bullet q$$

and that satisfies a certain condition that implies the following:

$$(t,p) \Vdash_{\mathcal{A}} N \Longrightarrow ((\!|t|\!)^{\alpha}_{k}, f(p)) \Vdash_{\mathcal{A}} N$$

Hence, given a $\mathcal{A}$-monitor (α, f), the following rule is sound:

$$\frac{\mathcal{E};\Gamma \vdash_n t : (N, p)}{\mathcal{E};\Gamma \vdash_n (\!|t|\!)^{\alpha}_n : (N, f(p))}$$

Clearly, the function f is responsible for making the forcing condition change, like the monitor α makes the memory cell content change.

Properties of 1-MAs

Even if the soundness theorem is satisfied by any n-**MA**, some of them behave better than others. This is the case of 1-**MA**s, which enjoy remarkable properties that are studied in Section 4.4.

- The first one concerns the monitors. If $\mathcal{A}$ is 1-**MA**, it is possible to characterize a consequent subset of the $\mathcal{A}$-monitors. Suppose that $\alpha \in \mathbb{P}$ and $f : |\mathcal{A}| \to |\mathcal{A}|$ is a strong function. Suppose that the following condition is satisfied:

  $$\alpha \Vdash_0 \forall x \in |\mathcal{A}|.\mathcal{C}_{\mathcal{A}}(f(x)) \multimap \mathcal{C}_{\mathcal{A}}(x)$$

  Then (α, f) is a $\mathcal{A}$-**monitor**. The fact that the condition exclusively relies on the realizability model at level 0 comes from the fact that during the reduction of the observation at level 1 $(\!|.|\!)^{\alpha}_1$, the configuration passes to the level 0 before the monitor can perform any operation. Remark that this corresponds to the annotated typing rule already used for the program transformation:

  $$\frac{\mathcal{E};\Gamma \vdash_1 t : (N, p) \qquad \mathcal{E};\vdash_0 \alpha : \forall x \in |\mathcal{A}|.\mathsf{C}(f(x)) \multimap \mathsf{C}(x)}{\mathcal{E};\Gamma \vdash_1 (\!|t|\!)^{\alpha}_1 : (N, f(p))}$$

- The second result is the **Connection Lemma**. It intuitively says that any model induced by $\mathcal{A}$ is the iteration of a forcing model *inside* a unary realizability model. This result can be informally formulated as follows.

> **Theorem 18.** *Suppose that $\mathcal{A}$ is a 1-**MA**. Then for all $(t, p) \in \mathbb{P}\times\mathcal{A}$, and every negative type N, we have:*
>
> $$(t, p) \Vdash_{\mathcal{A},n} N \Leftrightarrow t \Vdash_n (p \ \Vdash_{|\mathcal{A}|} N)$$
>
> *where $\Vdash_{|\mathcal{A}|}$ is the forcing interpretation induced by the forcing monoid $|\mathcal{A}|$.*

1.5.8 Chapter V : Iteration

At the heart of our methodology lie different ways of building new n-**MA**s out of previously defined ones. To obtain a model of a complex programming language, we generally start with a simple n-**MA**, and incrementally add new features to the corresponding language by applying different constructions, and show that at each step we do not loose what we gained previously. These constructions, based on forcing iteration, are described and studied in Chapter V .

Direct product

The direct product is the simplest way of combining two different **MA**s. In a sense, it corresponds to the intuition of "combining two programming languages". Start with a k-**MA** $\mathcal{A}$ and a 1-**MA** $\mathcal{B}$. We then build a $(k+1)$-**MA** denoted $\mathcal{A} \times \mathcal{B}$ as follows:

- The carrier $|\mathcal{A} \times \mathcal{B}|$ is the direct product of the two forcing monoids $|\mathcal{A}| \times |\mathcal{B}|$. It means that we take the product of the underlying sets, and we define the operations $\bullet$ and $+$ component wise, e.g:
$$(p, n) + (q, m) = (p + q, n + m)$$
The preorder is the preorder on each component.
- The test function $\mathcal{C}_{\mathcal{A}\times\mathcal{B}}$ is also the product of the two test functions:
$$\mathcal{C}_{\mathcal{A}\times\mathcal{B}}(p, m) = \{\ (\overrightarrow{v}, w) \mid (v_1, \ldots, v_n) \in \mathcal{C}_{\mathcal{A}}(p) \wedge w \in \mathcal{C}_{\mathcal{A}}(m)\ \}$$

The direct product construction corresponds in the forcing theory to the product of two forcing notions. It satisfies many good properties, and in particular, it preserves the $\mathcal{A}$-monitors and the $\mathcal{B}$-monitors, as witnessed by Property 181.

Property 19. *Let $\mathcal{D} = \mathcal{A} \times \mathcal{B}$ be a direct product. Suppose that (α, f) is a $\mathcal{A}$-monitor and (β, g) is a $\mathcal{B}$-monitor. Then the two following pairs are $\mathcal{D}$-monitors:*

- $(\lambda x.(\!|\mathsf{ret}(x)|\!)^{\alpha}_{k}, (a, b) \mapsto (f(a), b))$
- $(\beta, (a, b) \mapsto (a, g(b)))$

To lift the $\mathcal{A}$-monitor into a $\mathcal{A} \times \mathcal{B}$-monitor, we need to use $\lambda x.(\!|\mathsf{ret}(x)|\!)^{\alpha}_{k}$. That's because to access another memory cell than the last, we need to use another monitor. Let's look at an example of a reduction that accesses the second memory cell. Suppose $\beta = \lambda x.\mathsf{ret}(\underline{\mathsf{s}}(x))$ and $\alpha = \lambda x.(\!|\mathsf{ret}(x)|\!)^{\beta}_{1}$, then the configuration $\langle (\!|t|\!)^{\alpha}_{2}, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{a}(v).\mathsf{nil}\rangle_2$ reduces as follows:

$$\begin{array}{lll}
\langle (\!|t|\!)^{\alpha}_{2}, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{a}(v).\mathsf{nil}\rangle_2 & \to & \langle \lambda x.(\!|\mathsf{ret}(x)|\!)^{\beta}_{1}, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{a}(v).\mathsf{m}_2(t).\mathsf{nil}\rangle_1 \\
& \to^* & \langle (\!|\mathsf{ret}(v)|\!)^{\beta}_{1}, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{m}_2(t).\mathsf{nil}\rangle_1 \\
& \to & \langle \lambda x.\mathsf{ret}(\underline{\mathsf{s}}(x)), \mathsf{a}(\underline{\mathsf{n}}).\mathsf{m}_1(\mathsf{ret}(v)).\mathsf{m}_2(t).\mathsf{nil}\rangle_0 \\
& \to^* & \langle \mathsf{ret}(\underline{\mathsf{n}+1}), \mathsf{m}_1(\mathsf{ret}(v)).\mathsf{m}_2(t).\mathsf{nil}\rangle_0 \\
& \to & \langle \mathsf{ret}(v), \mathsf{a}(\underline{\mathsf{n}+1}).\mathsf{m}_2(t).\mathsf{nil}\rangle_1 \\
& \to & \langle t, \mathsf{a}(\underline{\mathsf{n}+1}).\mathsf{a}(v).\mathsf{nil}\rangle_2
\end{array}$$

Here, we used the $\lambda x.(\!|\mathsf{ret}(x)|\!)^{\beta}_{1}$ to access the second memory cell.

Simple iteration

Our second construction is directly inspired by the 2-steps forcing iteration. We call it the **MA simple iteration**. Starting with a k-**MA** $\mathcal{A}$, it consists in taking a new 1-**MA** $\mathcal{B}$ *inside* $\mathcal{A}$. We then can define a new $(k+1)$-**MA** that results of the combination of $\mathcal{A}$ and $\mathcal{B}$. A 1-**MA** *inside another k-MA* $\mathcal{A}$ is called $\mathcal{A}$-**MA**, and is given by the following components:

- A forcing monoid $|\mathcal{B}|$.
- A relativized test function $\mathcal{C}_{\mathcal{B}} : |\mathcal{B}| \to \mathcal{I}(\mathbb{V}_{\mathcal{A}})$ such that:

$$n \preceq_{|\mathcal{B}|} m \Rightarrow \mathcal{C}_{\mathcal{B}}(m) \subseteq \mathcal{C}_{\mathcal{B}}(n)$$

Given $\mathcal{A}$ a k-**MA**, and a $\mathcal{A}$-**MA**, one can define a new $(k+1)$-**MA** as described in Definition 173.

Definition 20 (Simple iteration)**.** *Let $\mathcal{A}$ be a k-**MA**, and $\mathcal{B}$ a $\mathcal{A}$-**MA**. Then we denote by $\mathcal{A} \lhd \mathcal{B}$ the following $(k+1)$-**MA**:*

- $|\mathcal{A} \lhd \mathcal{B}| = |\mathcal{A}| \times |\mathcal{B}|$
- $\mathcal{C}_{\mathcal{A}\lhd\mathcal{B}}(p,n) = \{\ (\overrightarrow{v}, w) \in \mathbb{V}^{k+1} \mid \exists r \in |\mathcal{A}|, (w,r) \in \mathcal{C}_{\mathcal{B}}(n) \wedge \overrightarrow{v} \in \mathcal{C}_{\mathcal{A}}(r \bullet p)\ \}$

Among the good properties satisfied by the simple iteration, we retrieve the properties of the 1-**MA**s.

- A preservation result for $\mathcal{A}$-monitors.
- A sufficient condition to find new $\mathcal{A} \lhd \mathcal{B}$-monitors based on $\mathcal{B}$.

The preservation result tells that the $\mathcal{A}$-monitors can be extended into $\mathcal{A} \lhd \mathcal{B}$-monitors. If (α, f) is a $\mathcal{A}$-monitor, then the following pair is a $\mathcal{A} \lhd \mathcal{B}$-monitor.

$$(\lambda x.(\!|\mathsf{ret}(x)|\!)^{\alpha}_{k}, (a,b) \mapsto (f(a), b))$$

The other property is similar to the way we can find new $\mathcal{A}$-monitors in 1-**MA**s. It basically says that any realizer in the realizability model induced by a k-**MA** $\mathcal{A}$ of the following form induces a $\mathcal{A} \lhd \mathcal{B}$-monitor:

$$(\alpha, q) \Vdash_{\mathcal{A},k} \forall x \in |\mathcal{B}|.\mathcal{C}_{\mathcal{B}}(f(x)) \multimap \Uparrow \mathcal{C}_{\mathcal{B}}(x)$$

The simple iteration really is a 2-steps forcing iteration inside a realizability model, as witnessed by the following generalized version of the connection lemma:

Theorem 21. *Suppose that $\mathcal{D} = \mathcal{A} \lhd \mathcal{B}$ is a simple iteration. Then for all $(t, (p,n)) \in \mathbb{P}_{\mathcal{D}}$, and every negative type N, we have:*

$$(t, (p,n)) \Vdash_{\mathcal{D},k} N \Leftrightarrow (t,p) \Vdash_{\mathcal{A},k} (p \Vdash_{|\mathcal{B}|} N)$$

where $\Vdash_{|\mathcal{B}|}$ is the forcing interpretation induced by the forcing monoid $|\mathcal{B}|$.

Remark 22. *The simple iteration is in some sense to the direct product what the dependent sum is to the product in dependent type theory.*

Semi-direct iteration

Finally, we consider in Section 5.3 a last construction called the semi-direct iteration. The only difference with the simple iteration is that the product of forcing monoid is replaced by a semi-direct product of forcing monoid, hence the name. Despite the name "iteration", this construction does not satisfy the same connection lemma as the simple iteration, hence it is not coming from a 2-steps forcing iteration. However it is very similar and satisfies a monitor preservation property. We will use this construction in a very particular context later.

1.5.9 Chapter VI : Basic blocks

Our methodology strongly relies on designing semantical techniques that can be reused in correctness proofs of various programming languages. We expose in Chapter VI a wide range of such "basic semantical blocks". Those consist in particular constructions on **MA**s that can be used to add different programming features to the language, like new modalities, recursive types or higher-order references, or to modify the property we observe, like bounded-time termination.

1.5.9.1 Modalities

Modalities are ubiquitous in programming languages and logic. They often control different structural aspects of a programming language. For example, the exponential modality of linear logic controls duplication [Gir87]. Nakano's next modality [Nak00] controls the stratification and helps taming recursive types. An extensive use of modalities has been strongly advocated in [AMRV07]. One of the nice outcomes of the monitoring algebras is the ability to define what a modality is simply in terms of forcing monoids, which will allow for an algebraic treatment of those. In the context of our core affine λ-calculus λ_{Aff}, by **modality** we informally mean a term and type constructor $\Box$ such that at least the two following rules hold:

$$\frac{\Gamma \vdash_{\mathsf{Aff}} t : A}{\Box\Gamma \vdash_{\mathsf{Aff}} \Box t : \Box A} \qquad \frac{\Delta \vdash_{\mathsf{Aff}} t : \Box A \qquad \Gamma, x : \Box A \vdash_{\mathsf{Aff}} u : B}{\Gamma, \Delta \vdash_{\mathsf{Aff}} \mathsf{let}\, \Box x \;=\; t \,\mathsf{in}\, u : B}$$

Operationally, we distinguish two kinds of modalities: the **blocking** and **call-by-value** modalities. To illustrate each case, let's consider a call-by-value language. If we extend it with a blocking modality $\Box$, its syntax of values and terms is extended as follows:

$$\begin{array}{rcl} v, w & ::= & \ldots \mid \Box t \\ t, u & ::= & \ldots \mid \Box t \mid \mathsf{let}\, \Box x \;=\; v \,\mathsf{in}\, t \\ E & ::= & \ldots \mid (\Box x.u).E \end{array}$$

and the following rules:

$$\begin{array}{rcl} \langle \mathsf{let}\, \Box x \;=\; t \,\mathsf{in}\, u, E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle t, (\Box x.u).E\rangle_{\mathsf{V}} \\ \langle \Box t, (\Box x.u).E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle u[t/x], E\rangle_{\mathsf{V}} \end{array}$$

Notice that $\Box t$ is a value, hence we cannot reduce under $\Box$. That's why we call it **blocking modality**. Then to translate this extension of the call-by-value language into λ_{LCBPV}, we only need to have the following call-by-push-value promotion rule:

$$\frac{\mathcal{E}; x_1 : P_1, \dots, x_n : P_n \vdash_0 v : Q}{\mathcal{E}; x_1 : \Box P_1, \dots, x_n : \Box P_n \vdash_0 v : \Box Q}$$

Notice that this rule is restricted to positive values. If one wants to extend it to negative, we only have the option of using the positive shift and define $\Box \Downarrow N$. [11] In the context of **MA**s, we can identify a significant subset of those blocking modalities in a purely algebraic way: this is the notion of $\mathcal{A}$-modality.

Definition 23 ($\mathcal{A}$-modality). *A $\mathcal{A}$-modality is a sub-additive function $\Box : |\mathcal{A}| \to |\mathcal{A}|$, i.e.*

$$\Box(p + q) \preceq_{|\mathcal{A}|} \Box(p) + \Box(q)$$

Given a $\mathcal{A}$-modality, and if we extend the interpretation of types as follows:

$$\|\Box P\| \stackrel{\mathsf{def}}{=} \{\ (v, \Box(p)) \mid (v, p) \in \|P\|\ \}$$

Then the following annotated rule is sound:

$$\frac{\mathcal{E}; x_1 : P_1, \dots, x_n : P_n \vdash_0 v : (Q, p)}{\mathcal{E}; x_1 : \Box P_1, \dots, x_n : \Box P_n \vdash_0 v : (\Box Q, \Box(p))}$$

Now consider a call-by-value modality, *i.e.* where we can reduce *under* the modality. It means we have consider the following alternative extension of the syntax:

$$\begin{array}{rcl} v, w & ::= & \dots \mid \Box v \\ t, u & ::= & \dots \mid \Box t \mid \mathsf{let}\, \Box x \;=\; v \,\mathsf{in}\, t \\ E & ::= & \dots \mid \Box.E \mid (\Box x.u).E \end{array}$$

Notice that now $\Box t$ is not a value anymore, hence we need to reduce under $\Box$ to obtain a value. We have added a special environment construction $\Box.E$ and the following two additional reduction rules:

$$\begin{array}{rcl} \langle \Box t, E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle t, \Box.E\rangle_{\mathsf{V}} \\ \langle v, \Box.E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle \Box v, E\rangle_{\mathsf{V}} \end{array}$$

[11] This is very similar to what happens with Melliès and Tabareau decomposition of the linear logic exponential using resource modalities in tensor logic [MT10], where only a similar modality is defined on positives. This phenomenon is called the value restriction.

Now, if one wants to translate this fragment into λ_{LCBPV}, one needs more than just the previous promotion rule. The following promotion rule is needed:

$$\frac{\mathcal{E}; x_1 : P_1, \ldots, x_n : P_n \vdash_0 t : (\Uparrow Q, p)}{\mathcal{E}; x_1 : \Box P_1, \ldots, x_n : \Box P_n \vdash_0 t : (\Uparrow(\Box Q), \Box(p))}$$

But in order to obtain such a rule, we need to have a much stronger notion of modality, called **adjoint modality**. If one looks at the operational side, we have added to the language of environments an *adjoint* to $\Box$, in the form of the environment constructor $\Box.E$. We in fact need the function $\Box$ on the forcing monoid to have a right adjoint, that is a function $\overline{\Box}$ such that

$$\mathcal{C}_{\mathcal{A}}(\Box(p) \bullet q) = \mathcal{C}_{\mathcal{A}}(p \bullet \overline{\Box}(q))$$

This reflects the additional reduction rules we have added. Not all modalities have a right adjoint. But in Section 6.1, we will give a general construction, based on the semi-direct iteration, that gives the possibility to give an adjoint to a $\mathcal{A}$-modality. But by doing so, one might loose some properties of the **MA** under consideration.

1.5.9.2 Bounded-time programming

We define the class of **quantitative** 1-**MAs**. Those particular **MA**s give rise to realizability models that allow the observation of bounded-time termination. This kind of realizability models have been initially defined as an extension of Kleene realizability in a serie of works by Hofmann [Hof03], Dal Lago and Hofmann [DLH05, DLH11, DLH10b][12]. We study this class of 1-**MA**s in Section 6.2. Quantitative **MA**s are based on the notion of **quantitative forcing monoid**[13]. A quantitative forcing monoid disposes of a function $\|.\| : \mathcal{M} \to \overline{\mathbb{N}}$ and an element $\mathbf{1} \in \mathcal{M}$ such that in particular we have:

- $\|p\| + \|q\| \leq \|p \bullet q\|$
- $p \leq q \Rightarrow \|p\| \leq \|q\|$
- $1 \leq \|\mathbf{1}\|$

Intuitively, the elements of the forcing monoid should be seen as abstract representations of concrete quantities, like time, space or resources.

- $\|p\|$ represents the concrete quantity (as an actual number) associated to p. We need this to be able to express concrete bounds.
- The anti-triangular inequality $\|p\| + \|q\| \leq \|p \bullet q\|$ represents the fact that the quantity of resources usage resulting of an interaction of two programs (represented by the operation $\bullet$) is a priori more than the sum of the resource usage generated by these two programs alone (represented by the $+$).

[12] The theory of **MA**s presented in this thesis in fact originates from a Krivine-style extension of this quantitative realizability and the proof that this framework can be decomposed through forcing [Bru13]. It is then no surprise that these models find their place in the more general theory of monitoring algebras.

[13] It constitutes a slight variant of the quantitative monoids of [Bru13], which themselves are a generalization of the resource monoids of [DLH05]

- Finally, $\mathbf{1}$ represents an elementary quantity.

Example 24. *A simple non trivial example of a quantitative forcing monoid is the* $\mathbb{N}$ *endowed with the usual addition and order, with*

- $\|n\| \stackrel{\text{def}}{=} n$ *is the identity*
- $\mathbf{1} \stackrel{\text{def}}{=} 1$

If $\mathcal{M}$ is a quantitative forcing monoid, we can then define the corresponding **quantitative** 1-**MA**:

- $|\mathcal{A}|$ is the quantitative forcing $\mathcal{M}$.
- The test function is defined as $\mathcal{C}_{\mathcal{A}}(p) \stackrel{\text{def}}{=} \{\ \underline{k} \mid k \geq \|p\|\ \}$.

Each such quantitative 1-**MA** disposes of a particular $\mathcal{A}$-monitor $(\alpha_{\mathsf{time}}, f)$ with:

- $\alpha_{\mathsf{time}} \stackrel{\text{def}}{=} \lambda x.\mathsf{case}\, x\, \mathsf{of}\, x.\Omega \parallel x.\mathsf{ret}(x)$
- $f \stackrel{\text{def}}{=} x \mapsto \mathbf{1} \bullet x$

When using this monitor, the memory cell is seen as a counter that is decreased by the monitor if it is greater than $\underline{0}$. Otherwise, the monitor triggers the execution of the diverging term Ω. One of the main interests of this particular monitoring is that it allows to observe bounded-time execution. Indeed, consider a computation t. We define an annotation $\{t\}_k^{\alpha_{\mathsf{time}}}$ of t. On λ constructors it is defined as:

$$\{\lambda x.t\}_k^{\alpha} \quad \stackrel{\text{def}}{=} \quad (\!|\lambda x.\{t\}_k^{\alpha}|\!)_k^{\alpha}$$

and $\{.\}_k^{\alpha}$ commutes with all other constructors. Consider the execution of a configuration of the form

$$\langle \{t\}_1^{\alpha_{\mathsf{time}}}, \mathsf{a}(\underline{k}).\mathsf{nil} \rangle_1$$

If we observe the termination of that configuration, it means that the monitor α_{time} has been triggered less than k times, otherwise it would have made the execution diverge. Hence, by observing the termination for different values of the memory cell, we can deduce a bounded-time termination property. As an example, if we consider the core type system λ_{LCBPV}, we can prove the following theorem as a corollary of the soundness theorem on the quantitative **MA** based on $\mathbb{N}$:

Corollary 25. *Suppose that* $\vdash_0 t : N$. *Then*

$$\langle t, \mathsf{nil} \rangle_0 \text{ terminates on a value in less than } |t|_\lambda \ \lambda\text{-steps}$$

where $|t|_\lambda$ *is the number of* λ *constructors appearing in* t.

To prove this result we use the most simple quantitative 1-**MA** of Example 24, but by choosing more complex ones, which for instance admit additional modalities that can be used to allow some duplication, we can prove bounded-time properties for more complex languages. We will use this class of 1-**MA**s in all the examples of Chapter VII .

1.5.9.3 Stratification

Many programmig features rely on some kind of *circularity*: recursive types, higher-order references, streams or recursive definitions of programs. In all those examples, the treatment of the circularity can be reduced to the treatment of *fixed-points*. Indeed, higher-order references can be justified through a proper program transformation typed using recursive types [Pot11], and fixed-point combinators can be typed using recursive types. In terms of model, it means the ability to prove the existence of fixed-points for certain maps. In our framework, it is possible to give a rather general account of recursivity by considering a certain class of monitoring algebras called **stratified**. A stratified monitoring algebra $\mathcal{A}$ is required to dispose of a map $\phi : |\mathcal{A}| \to \overline{\mathbb{N}}$ such that:

$$p \preceq_{|\mathcal{A}|} q \Rightarrow \phi(q) \leq \phi(p)$$

Using this stratification map, the goal is then to endow the sets $\mathcal{I}(\mathbb{V}_{\mathcal{A}})$ and $\mathcal{I}(\mathbb{E}_{\mathcal{A}})$ with a structure of complete metric space. This will allow us to use Banach fixed-point theorem in order to interpret various kinds of recursive types. We remind the statement of Banach fixed-point theorem:

Theorem 26 (Banach fixed-point theorem). *Let (X, d) be a complete metric space. Any contractive map $T : X \to X$ admits a unique fixed point x^* (i.e. , $T(x^*) = x^*$). Where a contractive map T is such that:*

$$\exists q < 1, \forall x, y \in X, d(T(x), T(y)) \leq q.d(x, y)$$

In order to do define the structure of metric space, we use a technique inspired by previous works on semantics of recursive types and higher-order references [BST09, BSS10, BMSS11]. Following those works, we use the notion of k-approximation of a set X, which amounts to only keep the elements of rank below k:

$$\pi_k(X) \stackrel{\text{def}}{=} \{\ (x, p) \in X \mid \phi(p) \leq k\ \}$$

We can deduce from it a notion of distance $d(X, Y)$ between two sets of $\mathcal{I}(\mathbb{V}_{\mathcal{A}})$ or $\mathcal{I}(\mathbb{E}_{\mathcal{A}})$:

$$\begin{array}{rcl} D(X,Y) & \stackrel{\text{def}}{=} & \sup(\{\ k \in \overline{\mathbb{N}} \mid \pi_k(X) = \pi_k(Y)\ \}) \\ d(X,Y) & \stackrel{\text{def}}{=} & 2^{-D(X,Y)} \end{array}$$

Hence two types are at distance 0 if all their k-approximation (for $k \in \mathbb{N}$) are equal. This in fact not a distance but a pseudo-distance, which does not satisfy the following separation axiom:

$$d(x, y) = 0 \Rightarrow x = y$$

However, it is always possible to quotient by the following equivalence relation:

$$X \approx Y \Leftrightarrow d(X, Y) = 0$$

Hence obtaining a distance. We show in Section 6.3 that $(\mathcal{I}(\mathbb{X}_{\mathcal{A}})/\approx, d)$ is a complete ultrametric space. A map from to the set of semantic types to the set of semantic types $F : \mathcal{I}(\mathbb{V}_{\mathcal{A}}) \to \mathcal{I}(\mathbb{V}_{\mathcal{A}})$ is:

- **Non-expansive** if $\pi_k(X) = \pi_k(Y)$ implies $\pi_k(F(X)) = \pi_k(F(Y))$.
- **Contractive** if $\pi_k(X) = \pi_k(Y)$ implies $\pi_{k+1}(F(X)) = \pi_{k+1}(F(Y))$.

The main result of this section is a corollary of Banach fixed-point theorem, in the form of a general fixed-point theorem for stratified **MA**s.

Theorem 27. *Let $\mathcal{A}$ be a stratified **MA**. Suppose that $F : \mathcal{I}(\mathbb{V}_{\mathcal{A}}) \to \mathcal{I}(\mathbb{V}_{\mathcal{A}})$ is contractive. Then F has a unique fixed-point modulo $\approx$.*

1.5.9.4 Step-indexing

One of the most often used technique to interpret recursive types or higher-order references is the so-called technique of **step-indexing**. In presence of such programming features, one cannot directly prove correctness because of circularity issues already mentionned. The step-indexing technique consists in stratifying the definition of the semantics interpretation by the number of reduction steps available for testing the desired property. For example, if one wants to prove a safety property of programs, one can instead prove a k-safety condition that say the program reduces for at least k steps without clash, or returns a value in less than k steps. This allows for a proof by induction of the correctness property. Since their introduction by Appel and McAllester [AM01], step-indices have been extensively used and improved [DAB09, BH09, BRS+11, Hos12]. It is possible to recast that technique using a specific Monitoring Algebra called the **step-indexing MA**. It is an example of stratified **MA**.

Definition 28. *The **step-indexing MA** $\mathcal{A}_{\mathsf{step}}$ is given by:*

- *The additive forcing monoid $(\overline{\mathbb{N}}, \min, \infty, \geq)$ of integers augmented with the ∞ element, the minimum operation and the reversed order $\geq$.*
- *The test function $\mathcal{C}_{\mathcal{A}_{\mathsf{step}}}(n) = \{\ \underline{k} \mid k \geq n\ \}$.*

This **MA** is obviously stratified (the identity is the stratifying map). Moreover, it is easy to see that all basic connectives induce non-expansive maps, hence by considering a predicate variable X, each type P induces a non-expansive map *via* its interpretation at *any level* k:

$$C \mapsto \|P\|_{\mathcal{A}_{\mathsf{step}},k,\rho[X \leftarrow C]}$$

For instance the orthogonality map is non-expansive:

$$\begin{cases} \mathcal{I}(\mathbb{V}_{\mathcal{A}}) & \to & \mathcal{I}(\mathbb{E}_{\mathcal{A}}) \\ X & \mapsto & X^{\perp_{\mathcal{A}_{\mathsf{step}},k}} \end{cases}$$

To use the fixed-point theorem, non-expansive maps are not enough: we need to find a contractive map. This role is played by the next modality, defined on his action on the monoid:

$$\triangleright n \stackrel{\mathsf{def}}{=} n+1$$

We prove in Section 6.4 that $\triangleright$ defines a contractive map. This modality, or slight variations of that modality, has been introduced by Nakano in [Nak00], and studied in many subsequent works [AMRV07, BSS10, BMSS11, DAB09, JT11] as a way to simplify the technique of step-indexing. Since $\triangleright$ is contractive, all maps induced by types of the form $\triangleright P$ are shown to be contractive. Hence, they have fixed-points (unique modulo $\approx$).

Theorem 29 (Fixed-points)**.** *Let P be a positive predicate of arity $S = S_1 \times \dots S_n$ and X be a positive type variable. Then for any level $k \in \mathbb{N}$ and any $\mathcal{A}_{\mathsf{step}}$-valuation ρ the map $\| \triangleright P \|_{\mathcal{A}_{\mathsf{step}},k,\rho}$ has a fixed-point $\mu X. \| \triangleright P \|_{\mathcal{A}_{\mathsf{step}},k,\rho}$, which is unique modulo $\approx$.*

We use this Theorem to extend the interpretation of positive types by

$$\|\mu X.P\|_{\mathcal{A}_{\mathsf{step}},k,\rho} \stackrel{\mathsf{def}}{=} \mu X. \| \triangleright P \|_{\mathcal{A}_{\mathsf{step}},k,\rho}$$

We then show that using this interpretation we obtain two different kinds of recursive types: guarded and non-guarded.

Guarded recursive types. With this definition of the interpretation of the recursive types, we obtain the following fold and unfold $\mathcal{A}_{\mathsf{step}}$-sound rules:

$$\frac{\mathcal{E}; \Gamma \vdash_1 v : (\triangleright(P[\mu X.P/X]), n)}{\mathcal{E}; \Gamma \vdash_1 v : (\mu X.P, n)} \qquad \frac{\mathcal{E}; \Gamma \vdash_1 v : (\mu X.P, n)}{\mathcal{E}; \Gamma \vdash_1 v : (\triangleright(P[\mu X.P/X]), n)}$$

This corresponds to the guarded recursive types *à la Nakano* [Nak00]. These guarded recursive types are known to preserve termination, unlike the unrestricted recursive types.

Unrestricted recursive types. In $\mathcal{A}_{\mathsf{step}}$, it is also possible to give an account of unrestricted recursive types. This is where the name step-indexing is justified. Indeed, this **MA** disposes of a particularly interesting $\mathcal{A}_{\mathsf{step}}$-monitor, given by the following pair $(\alpha_{\mathsf{step}}, f)$:

- $\alpha_{\mathsf{step}} \stackrel{\mathsf{def}}{=} \lambda x.\mathsf{case}\, x\, \mathsf{of}\, x.✠ \| x.\mathsf{ret}(x)$
- $f \stackrel{\mathsf{def}}{=} x \mapsto x+1$

By using the monitor α_{step}, we obtain the following rules:

$$\frac{\mathcal{E};\Gamma \vdash_1 v : (P[\mu X.P/X], n)}{\mathcal{E};\Gamma \vdash_1 v : (\mu X.P, n)}$$

$$\frac{\mathcal{E};\Gamma \vdash_1 v : (\mu X.P, n) \qquad \mathcal{E};\Delta, x : P[\mu X.P/X] \vdash_1 t : (N, m)}{\mathcal{E};\Gamma,\Delta \vdash_1 \mathsf{ret}(v)\ \mathsf{to}\ x.(\!|t|\!)_1^{\alpha_{\mathsf{step}}} : (N, \min(m, n))}$$

Since we know that unrestricted fixed-points break termination, why can we interpret them in a realizability semantics that is built around termination? Well, the presence of $(\!|.|\!)_1^{\alpha_{\mathsf{step}}}$ changes what is observed by the semantics. Indeed, it is *a priori* impossible to bound the number of times this observation will be made. But each time it arrives in head position, it decreases the counter, and triggers ✠ if it reaches 0. In that case, the execution stops, *even though* it could have continued otherwise (or even diverged). That's why we're not observing termination but safety: if we try every possible value in the memory cell, we observe that the execution never clashes.

Step-indexed MA. We finally give a condition on **MA**s, called the step-indexing condition, that allows us to transfer the properties of the step-indexing algebra to those **MA**s. In particular, doing a direct product or a simple iteration preserves the step-indexation, as well as most semi-direct iterations. This will be very useful when dealing with complex programming languages.

1.5.9.5 Higher-order references

The last basic block we are going to study in this thesis is the addition of higher-order references. The technique amounts to do the following steps:

- Choosing a step-indexed monitoring algebra $\mathcal{A}$, that is an algebra that keeps all the good properties of the step-indexing **MA** $\mathcal{A}_{\mathsf{step}}$.
- Identify a particular positive type P whose interpretation induces a contractive map. For example, in $\mathcal{A}_{\mathsf{step}}$ this can be any type of the form $\triangleright P$.
- Define a new $\mathcal{A}$-**MA**, that takes advantage of the existence of a fixed-point and define the simple iteration of $\mathcal{A}$ and $\mathcal{B}$.

One very important particularity of step-indexed **MA**s is that if a type is made out of non-expansive connectives, then its *forcing translation* induces a non-expansive map. Indeed, the forcing translation of a type P only uses the connectives used in P and the basic connectives such as $\multimap, \otimes, \Uparrow$ or $\Downarrow$. If we dispose of a contractive map, like the next modality $\triangleright$ of the step-indexing algebra, then the following map is contractive (for *any* level k) :

$$\begin{cases} \mathcal{I}(\mathbb{V}_{\mathcal{A}})^{\mathcal{M}} & \to & \mathcal{I}(\mathbb{V}_{\mathcal{A}})^{\mathcal{M}} \\ C & \mapsto & (p \mapsto \|(\triangleright P)^*(x)\|_{\mathcal{A},k,[x\leftarrow p,\mathsf{C}\leftarrow C]}) \end{cases}$$

Why is it interesting to obtain a fixed-point of such a map? Because it determines the content of the memory cell. The idea is to define the following $\mathcal{A}$-**MA** denoted $\mathcal{B}$:

- The forcing monoid $|\mathcal{B}|$ is the trivial one-element forcing monoid $\{\star\}$.
- We want $\mathcal{C}_{\mathcal{B}}$ to contain values of type P. But those values of type P must also be able to manipulate the store, hence a fixed-point is needed. We define the function F as

$$F : \begin{array}{rcl} (\mathbb{V}_{\mathcal{A}})^{\{\star\}} & \to & (\mathbb{V}_{\mathcal{A}})^{\{\star\}} \\ X & \mapsto & y \mapsto \|(P)^*(x)\|_{\mathcal{A}_0,n+1,[x\leftarrow y,\mathsf{C}\leftarrow X]} \end{array}$$

  We can then define $\mathcal{C}_{\mathcal{B}}$ as a fixpoint of F:

$$\mathcal{C}_{\mathcal{B}} = \mu F$$

We will make use of the simple iteration:

$$\mathcal{A}_{\mathsf{ref}(P)} \stackrel{\mathsf{def}}{=} \mathcal{A} \lhd \mathcal{B}$$

As the following property shows, the memory cell in $\mathcal{A}_{\mathsf{ref}(P)}$ contains realizers of the type P, in the *same monitoring algebra*, hence solving the circularity.

Property 30. *If $\phi(p) \in \mathbb{N}$, the following equivalence holds:*

$$(v, \overrightarrow{w}) \in \mathcal{C}_{\mathcal{A}_{\mathsf{ref}(P)}}(\star, p) \iff \exists r \in |\mathcal{A}|, \phi(r) < \infty \wedge \begin{cases} (v, (\star, r)) \in \|P\|_{\mathcal{A}_{\mathsf{ref}(P)},n+1,[]} \\ \overrightarrow{w} \in \mathcal{C}_{\mathcal{A}}(r \bullet p) \end{cases}$$

This allows us to introduce a primitive swap_n. Its behavior is given by the following reduction rule:

$$\langle \mathsf{swap}_{n+1}(v), \overrightarrow{\mathsf{a}(w')}.\mathsf{a}(w).E\rangle_{n+1} \quad \xrightarrow{n+1} \quad \langle \mathsf{ret}(w), \overrightarrow{\mathsf{a}(w')}.\mathsf{a}(v).E\rangle_{n+1}$$

This primitive linearly exchanges the content of the memory cell with its argument. We will show that in the context of $\mathcal{A}_{\mathsf{ref}(P)}$, the following theorem holds.

Theorem 31. *The following rule is $\mathcal{A}_{\mathsf{ref}(P)}$-sound.*

$$\frac{\mathcal{E}; \Gamma \vdash_{n+1} v : (P, (\star, p)) \qquad \phi(p) \in \mathbb{N}}{\mathcal{E}; \Gamma \vdash_{n+1} \mathsf{swap}_{n+1}(v) : (\Uparrow \triangleright P, (\star, p))}$$

By specializing this rule to different **MA**s, we can obtain different kind of higher-order references. For instance:

- Using the next modality in the step-indexing algebra, then we can always obtain guarded higher-order references, that do not break termination.
- By using the step-indexing monitoring α_{step}, we can then obtain unrestricted higher-order references. In that case we loose termination for the same reason as unrestricted recursive types.
- We can also trivially obtain first-order references in *any* **MA**, by using the trivial stratification $\phi(p) = 0$.

1.5.10 Chapter VII : Applications

In Chapter VII , we will explore three interesting examples of application of the monitoring algebra theory.

Linear naive set theory

We have seen in the Section 6.4 that by considering the step-indexing algebra $\mathcal{A}_{\mathsf{step}}$, it is possible to obtain two kinds recursive types:

- Guarded recursive types: in that case the realizability model still implies termination.
- Non-guarded recursive types: in which case we loose termination and only observe safety.

There is another kind of recursive type which, like guarded recursive types, do not harm termination: the **linear fixpoints**. It is well-known [Gir92] that in the context of linear logic, adding recursive types $\mu X.L$ for every *linear* type L (*i.e.* that does not contain any occurence of the exponential modality !) does not break strong normalization, even though they are not guarded.

Here, we turn our interest to an even more general and older question: the consistency of **linear naive set theory**. By naive set theory, we mean a set theory with *the principle of unrestricted comprehension*: for every predicate $P(x)$ there exists a set $\{\ x \mid P\ \}$ such that:

$$\forall t,\ (t \in \{\ x \mid P\ \} \text{ is logically equivalent to } P(t))$$

It is well-known that in intuitionistic or classical logic, this principle yields an inconsistency (for example considering the set $\{\ x \mid x \notin x\ \}$ implies Russel' Paradox). Grishin [Gri82] first introduced in 1982 a naive set theory based on a contraction-free logic and showed the consistency of such a theory. More expressive (and consistent) naive set theories have then been proposed, based on light logics [Gir98, Ter04]: these theories are based on logics that have a form of contraction that is not as powerful as the linear logic one but ensure the consistency. The consistency of these theories is given by the cut-elimination theorem, usually proved by a syntactic argument [Ter04, Shi94]. The question of finding a semantical proof of the consistency of these theories has been regularly raised [Kom89, Shi94, Ter02] but to our knowledge, no such semantical proof has been given. We propose such a proof in the case of linear naive set theory (*i.e.* , without any contraction), based on the combination of two 1-**MA**s:

- The quantitative 1-**MA** based on $\mathbb{N}$.
- The step-indexing **MA** $\mathcal{A}_{\mathsf{step}}$.

The termination argument is then obtained by a surprising interaction between the two monitors α_{time} and α_{step}, which compete to make their respective counter reach 0 first. We finally show evidence that this proof can be extended to more expressive naive set theories such as elementary set theory or light affine set theory [Ter02].

Polynomial-time programming language and recursive types

In this section, we consider a more concrete example: a programming language with unrestricted recursive types based on a substructural logic called soft affine logic (**SAL**) [BM04, Laf04]. It means that the usual exponential of linear logic ! is weakened: the contraction rule is restricted. As a result this type system ensures that all typable programs enjoy a polynomial-time termination property. This example is the occasion to showcase the methodology of monitoring algebras: we detail each step of the correctness proof of a concrete language. It is basically as follows:

- We define the syntax of a call-by-value language with an explicit modality and its type system based on **SAL**.
- We build step by step a monitoring algebra that integrates the features required to show the correctness of the language: a quantitative 1-**MA**, step-indexing to deal with recursive types, the addition of adjoint modalities to reduce under !. We then show a soundness theorem for this monitoring algebra, reusing many of the theorems and properties proved in this thesis.
- We finally define a translation of the language into the call-by-push-value, and show that the soundness of the monitoring algebra defined imply the polynomial-time termination property.

This language is inspired by [BM12].

Strong update and linear capability

We study a small language that features a higher-order reference, but has the ability to perform **strong updates**, very much inspired by the work of Ahmed et al. [AFM07]: in fact, modulo the addition of dynamic references, we could translate their core language into this one. Strong updates means that the type of the memory cell can *change* during the execution. This kind of feature is usually not sound, but here the access to the memory cell is restricted thanks to a linear discipline. We show that a simple monitoring algebra can be given by reusing a quantitative 1-**MA**, the step-indexing algebra and a variant of the higher-order references construction of Section 6.5, and a soundness theorem be proved such that it implies the termination of that language.

1.6 | Summary of the contributions

We give a brief summary of all the contributions of this Thesis, chapter by chapter.

Chapter II

The first chapter introduces a linear variant λ_{LCBPV} of the call-by-push-value calculus, whose type system features first-order quantifiers and inequational reasoning. An abstract machine

intended to execute the terms of this language is then defined. We then develop an extension of this calculus with special memory cells that are used to keep tracks of some informations of the execution. The corresponding abstract machine is called the Monitoring Abstract Machine. We finally give a (untyped) program transformation from $\lambda^{\mathbf{1}}_{\mathsf{Mon}}$ to λ_{LCBPV}.

Chapter III

The type part of the program transformation mentioned in Chapter II is the focus of Chapter III . We introduce the central notion of forcing monoid and forcing structure. We then define a forcing-based type system, that is an annotated version of the λ_{LCBPV} type system that we use throughout the thesis. Finally, we define the forcing type translation and use it to state and prove that the forcing program transformation also preserves typing in a certain way.

Chapter IV

This chapter is devoted to the introduction and the basic study of the Monitoring Algebras. We define the interpretation of λ_{LCBPV} and show that each n-**MA** induces a sound model of λ_{LCBPV}. We then turn our attention more closely to 1-**MA**s, by showing two important facts:

- A characterization of a subset of the $\mathcal{A}$-monitors.
- The proof of the Connection Lemma.

We finally give three basic examples of 1-**MA**s that illustrate the main use cases of the theory of monitoring algebras.

Chapter V

Chapter V introduces two important ways of building new $(n+1)$-**MA**s, starting with old ones. The first, the simple iteration, is shown to preserve several properties of the **MA**s, including monitors and a generalization of the connection lemma. The second, called the semi-direct iteration, is of a more exotic nature and we briefly study its limited preservation properties.

Chapter VI

Several basic, reusable technical blocks are given in Chapter VI .

- **Modalities:** we first define an algebraic notion of modality, and distinguish between blocking and call-by-value modalities. We define an algebraic operation on the monitoring algebra that can be used to turn a blocking modality into a call-by-value modality.
- **Stratification:** we define a class of **MA**s called stratified, that all can be endowed with a certain structure of complete ultrametric space. We then apply Banach fixed-point theorem to obtain a general purpose fixed-point theorem.

- **Step-indexing:** we define a particular stratified **MA** called the step-indexing algebra, that corresponds to the well-known technique of step-indexing in our framework. We study many of its properties and why it allows to obtain models of different kinds of recursive types: guarded à la Nakano and unrestricted recursive types. We finally show that these results can be transported to other **MA**s.
- **Bounded-time termination:** we study a class of 1-**MA**s called quantitative. We show that they provide a basis to study various kinds of bounded-resource execution.
- **Higher-order references:** starting with the notion of stratified and step-indexed **MA**, we show how higher-order references can be added to a language in a parametric way.

Chapter VII

We finally give in Chapter VII some more complex applications of the theory, that each use several of the basic blocks defined in Chapter VI .

- The first example is a proof of termination of linear naive set theory introduced by [Shi94]. It is a naive set theory with an unrestricted comprehension scheme, but which still enjoys consistency. We give the first semantics proof of this fact, by a surprising combination of a bounded-time monitoring algebra and the step-indexing algebra.
- The second example is a programming language based on soft linear logic [Laf04], that features recursive types. We show that our framework gives us a proof of the polynomial-time execution of typed programs.
- The third example is a programming language with linear references and strong updates, inspired by [AFM07]. We use a combination of the step-indexing **MA** and of a quantitative 1-**MA** to show that this language enjoys termination, even though it features higher-order references and strong update.

Chapter II

The monitoring abstract machine

Contents

Contributions

We introduce the basic languages that will be used in this thesis. We begin in Section 2.1 by introducing an affine variant of Levy's call-by-push-value called λ_{LCBPV}, whose type system is enriched with particular first-order quantifiers and inequational theories. Section 2.1 is devoted to the definition of affine versions of the call-by-name and call-by-value calculi, and their typed translation into λ_{LCBPV}. We then present an extension λ_{Mon} of λ_{LCBPV} and the associated **Monitoring Abstract Machine**, that constitutes the heart of this thesis. This is done in

Section 2.3. Finally, we give an untyped program transformation of λ_{Mon} into λ_{LCBPV}, based on the store-passing-style transformation.

2.1 | The call-by-push value calculus

We begin this chapter with the exposition of a variant of Levy's call-by-push-value. We remind the reader that call-by-push-value can be seen as a calculus generalizing both call-by-name and call-by-value calculi. It is a polarised calculus, where positive values and negative computations are two separated categories of the language.

2.1.1 Syntax

We start from an untyped λ-calculus featuring pairs, primitive integers, as well as conditional branching, that we call λ_{LCBPV} . Its syntax differentiates two syntactical categories: the values and the computations. Values correspond to already reduced terms, while computations correspond to programs that still have to compute. The syntax of this core language is parametrized by two sets:

- a set $\mathcal{W}$ of **value symbols**
- a set $\mathcal{K}$ of **primitive symbols**

The grammars of values and computations are defined by mutual induction as follows:

$$
\begin{array}{llll}
\text{Values} & v, w & ::= & x \mid (v, w) \mid * \mid \underline{0} \mid \underline{\mathsf{s}}(v) \mid \mathsf{thunk}(t) \mid \varpi \qquad (\text{where } \varpi \in \mathcal{W}) \\
\text{Computations} & t, u & ::= & \mathsf{force}(v) \mid \mathsf{let} * = v \,\mathsf{in}\, u \mid \mathsf{ret}(v) \mid \lambda x.t \mid \\
 & & & (t)v \mid t \,\mathsf{to}\, x.u \mid \mathsf{let}\,(x, y) = v \,\mathsf{in}\, t \mid \\
 & & & \mathsf{case}\, v \,\mathsf{of}\, x.t \parallel x.u \mid \maltese \mid \zeta \qquad (\text{where } \zeta \in \mathcal{K})
\end{array}
$$

This syntax features pairs, integers (including a successor and a case constructor). Here are some remarks about the more unusual parts of this syntax:

- The thunk() constructor transforms a computation into a value, hence freezing it into a thunk. Dually, the ret() constructor transforms a value into a computation.
- The application $(t)v$ is restricted to value on the right hand. The sequential composition is implemented by the constructor t to $x.u$: it reduces first t into a value, and then passes it to u.
- The special computation $\maltese$, called the **daimon**, is inherited from Girard's ludics [Gir01]. Once it is executed it stops immediately the computation and consider it a success. It is useful in the realizability semantics when one wants to test only some parts of a program while ignoring some others.

- Finally, the syntax is parametrized by two sets: $\mathcal{W}$ and $\mathcal{K}$. The first is a set of *additional values* one can consider, and the latter is a set of *additional primitives*. This will allow to add new primitives on the fly, and keep the definitions of semantics independent of any language extension.

We denote by the letters t, u computations and by v, w values. We denote by the letters a, b terms which can be computations or values. If $@$ is a term constructor (for example λ, the application of two terms, or case), we denote by $|t|_@$ the number of times this constructor appears in t.

Notation 32. *We define the following notations:*

- *We denote by* $\mathbb{V}_0$ *and* $\mathbb{P}_0$ *respecitively the sets of* closed *values and computations, including the additional primitives coming from the sets* $\mathcal{W}$ *and* $\mathcal{K}$.
- *We denote by* $\mathbb{V}_0^{\mathsf{Core}}$ *and* $\mathbb{P}_0^{\mathsf{Core}}$ *respecitively the sets of* closed *values and computations, without the additional primitives coming from the sets* $\mathcal{W}$ *and* $\mathcal{K}$.
- *We denote by* $\lambda_{\mathsf{LCBPV}}^{\mathsf{Core}}$ *the* core *language, i.e.* λ_{LCBPV} *without the additional primitives.*

2.1.2 An abstract machine

In order to execute λ_{LCBPV} programs, we define an abstract machine. We first define the syntactic classes of **environments** and **configurations**. The latter being the results of an interaction between a computation and an environment. We then give a reduction relation betwen such configurations. To be precise, we define a class of reduction relations that contain at least the core reduction steps. When we consider new primitives, this allows us to add new reduction steps as well.

Definition 33 (Environments and configurations)**.**

1. ***Environments*** *are the finite lists generated by the following grammar:*

$$E \quad ::= \quad \mathsf{nil} \mid \mathsf{a}(v).E \mid \mathsf{f}(x.t).E$$

   where v *is closed and* $\mathsf{FV}(t) \subseteq \{x\}$.

2. *A* ***configuration*** C *is a pair* $\langle t, E\rangle_0$ *of a closed computation* t *and an environment* E*, or the* ***daimon configuration*** ✠*, which represents an aborted execution.*

3. *A* ***co-configuration*** $\overline{C}$ *is a pair* $\langle \uparrow E, v\rangle_0$ *of a closed value* v *and an environment* E.

The concept of co-configuration will be useful when proving some results about the realizability models of Chapter IV . They correspond to the fact that the call-by-push-value is a fragment

of a much wider system where positive terms are not restricted to values and can be reduced. The meaning of such a notion will be made clearer in Chapters III and Chapter IV .

Notation 34. *We denote by* $\mathbb{C}_0$ *and* $\mathbb{E}_0$ *respectively the set of configurations and environments of* λ_{LCBPV}*. We denote by* $\mathbb{C}_0^{\mathsf{Core}}$ *and* $\mathbb{E}_0^{\mathsf{Core}}$ *respectively the set of configurations and environments of* $\lambda_{\mathsf{LCBPV}}^{\mathsf{Core}}$*.*

Remark 35. *Because of this definition, a configuration* C *never contains any free variable.*

We now define a *reduction relation* $\xrightarrow{0}$ on configurations. It is defined as the smallest relation between configurations containing the next rules. The first two rules correspond to the usual β-rule, that become simpler since the argument is always a value:

$$\begin{array}{rcl} \langle (t)v, E\rangle_0 & \xrightarrow{0} & \langle t, \mathsf{a}(v).E\rangle_0 \\ \langle \lambda x.t, v.E\rangle_0 & \xrightarrow{0} & \langle t[v/x], E\rangle_0 \end{array}$$

The next two rules correspond to the usual rules associated to a let constructor *à la Moggi*.

$$\begin{array}{rcl} \langle t \text{ to } x.u, E\rangle_0 & \xrightarrow{0} & \langle t, \mathsf{f}(x.u).E\rangle_0 \\ \langle \mathsf{ret}(v), \mathsf{f}(x.u).E\rangle_0 & \xrightarrow{0} & \langle u[v/x], E\rangle_0 \end{array}$$

The following reduction step corresponds to *unfreezing* a frozen computation:

$$\langle \mathsf{force}(\mathsf{thunk}(t)), E\rangle_0 \xrightarrow{0} \langle t, E\rangle_0$$

The rule concerning the unit is:

$$\langle \mathsf{let} * = * \mathsf{in}\, t, E\rangle_0 \xrightarrow{0} \langle t, E\rangle_0$$

The rule concerning the pairs is:

$$\langle \mathsf{let}\,(x,y)\ =\ (v,w)\,\mathsf{in}\,t, E\rangle_0 \quad \overset{0}{\to} \quad \langle t[v/x, w/y], E\rangle_0$$

The rules for integers and conditional branching:

$$\begin{aligned} \langle \mathsf{case}\,\underline{0}\,\mathsf{of}\,x.t_1 \parallel x.t_2, E\rangle_0 &\quad \overset{0}{\to} \quad \langle t_1[\underline{0}/x], E\rangle_0 \\ \langle \mathsf{case}\,\underline{\mathsf{s}}(v)\,\mathsf{of}\,x.t_1 \parallel x.t_2, E\rangle_0 &\quad \overset{0}{\to} \quad \langle t_2[v/x], E\rangle_0 \end{aligned}$$

Finally, we give a special rule for the daimon ✠:

$$\langle ✠, E\rangle_0 \quad \overset{0}{\to} \quad ✠$$

Remark 36. *Once the daimon ✠ arrives in head position, the execution stops on the daimon configuration. This is very similar to the behavior of the daimon in Girard's ludics.*

Notation 37. *Each reduction rule is determined by the term constructor in head position. If @ is such a constructor (for example λ or* case*), a @-step is any reduction step associated with a constructor @. For example, the following step is a λ-step:*

$$\langle \lambda x.t, v.E\rangle_0 \overset{0}{\to} \langle t[v/x], E\rangle_0$$

In addition to the reduction relation $\overset{0}{\to}$, we also define an equivalence relation $\cong_0$ between configurations and co-configurations:

$$\langle \uparrow E, v\rangle_0 \cong_0 \langle \mathsf{ret}(v), E\rangle_0$$

2.1.3 First-order signature and evaluation

In order to be sufficiently expressive to formalize the forcing in it, the type system we will consider for this language disposes of first-order quantifiers and inequational implication. We first define the language of first-order expressions on which we can quantify, and then define what an inequational theory is.

2.1.3.1 First-order signature

Definition 38 (First-order signature)**.** *We define several notions related to the first-order syntax:*

- *A **first-order signature** $\mathcal{S}$ is a set of preordered sets, usually denoted $(S, \preceq_S), (S', \preceq_{S'}), \ldots$.*
- *Given a signature $\mathcal{S}$, we call **arity** any finite tuple*

$$(S_1, \ldots, S_n) \in \mathcal{S}^n$$

We suppose that for each function $f : S_1 \times \cdots \times S_n \to S_{n+1}$ with $S_1, \ldots, S_{n+1} \in \mathcal{S}$, we have a corresponding **function symbol** $\dot{f}$.

Definition 39. *If $\dot{f}$ is a function symbol denoting a function $f : S_1 \times \ldots S_n \to S_{n+1}$, then its **arity**, denoted by $\mathsf{ar}(f)$ is*

$$\mathsf{ar}(\dot{f}) = (S_1, \ldots, S_n, S_{n+1})$$

For each arity S, we suppose disposing of an infinite number of **predicate variables** $X, Y, Z, \ldots$ of arity S. We denote the fact that X is of arity $(S_1, \ldots, S_n)$ by

$$X : (S_1, \ldots, S_n)$$

Finally, for each set $S \in \mathcal{S}$, we suppose disposing of an infinity of **first-order variables** denoted by either the roman letters $x^S, y^S, z^S, \ldots$, or the greek letters ι, κ, σ.

Definition 40 (First-order expressions)**.** *Given a signature $\mathbb{S}$ and a preordered set $S \in \mathcal{S}$, we define the set S-Exp of S-**expression** inductively using the two following rules:*

$$\frac{}{x^S \in S\text{-}Exp} \qquad \frac{e_1 \in S_1\text{-}Exp \quad \ldots \quad e_n \in S_n\text{-}Exp \quad \mathsf{ar}(\dot{f}) = (S_1 \times \cdots \times S_n, S_{n+1})}{\dot{f}(e_1, \ldots, e_n) \in S_{n+1}\text{-}Exp}$$

Definition 41 (Evaluation)**.**

- *A **first-order valuation** ρ is an assignment of every variable x^S to an actual element $\rho(x^S)$ of S.*
- *Given a first-order valuation ρ and an element e of S-Exp, we define the **evaluation of** e and denote it $[\![e]\!]_\rho$ by induction:*

$$\begin{aligned} [\![x^S]\!]_\rho &= \rho(x^S) \\ [\![\dot{f}(e_1, \ldots, e_n)]\!]_\rho &= f([\![e_1]\!]_\rho, \ldots, [\![e_n]\!]_\rho) \end{aligned}$$

Property 42. *If $e \in S$-Exp then for any first-order valuation ρ, $[\![e]\!]_\rho \in S$.*

2.1.4 The type system

We now have everything we need to describe the type system. Reflecting the distinction between values and computations, the types are separated into **positive types** denoted by capital letters $P, Q, \ldots$ and **negative types**, denoted by capital letters $N, M, \ldots$. Values will be typed by positive types, while computations correspond to negative types. The grammar is defined by mutual induction:

$$\begin{array}{rcl} P, Q & ::= & X(e_1, \ldots, e_n) \mid \Downarrow N \mid \mathsf{Nat} \mid P \otimes Q \mid \exists x \in S.P \mid \{e \preceq_S f\} \wedge P \\ N, M & ::= & \Uparrow P \mid P \multimap N \mid \forall x \in S.N \mid \{e \succeq_S f\} \mapsto N \end{array}$$

- The implication is *affine*. It means it represents the type of programs that use their argument *at most* once. Notice also that reflecting the application construction which is restricted to value argument, the implication is from positive to negatives.

- Two particular connectives are introduced because of the polarity: the **positive shift** $\Downarrow$ and the **negative shift** $\Uparrow$. They allow to pass from a negative to a positive, and *vice versa*.

- First-order quantifiers do not change the polarity, they are computationally transparent.

- Finally, we add an **inequational implication** and an **inequational conjunction**. They are a slight modification of the equational implication used by Miquel. The inequational implication intuitively means

$$\{e \succeq_S f\} \mapsto N \equiv \begin{cases} N & \text{if } f \preceq_S e \\ \top & \text{otherwise} \end{cases}$$

  where $\top$ is morally type of all computations. While the inequational conjunction intuition is

$$\{e \preceq_S f\} \wedge P \quad \equiv \quad e \preceq_S f \textbf{ and } P$$

  Like the quantifiers, they don't change the polarity of the type and they are computationally transparent.

Notation 43. *We use the capital letters $A, B, \ldots$ to denote either a positive type or a negative type.*

2.1.4.1 Predicates

Given a signature, we dispose of an infinity of predicate variables. Each of these predicate variable X has an arity $\mathsf{ar}(X)$. But it happens that types contain *free* first-order variables, hence denoting themselves a predicate. It hence makes sense to assign to each type an arity.

Definition 44. *Given a type A, the set of the **first-order free variables of** A denoted by $\mathsf{FV}(A)$ is defined inductively as follows:*

$$\begin{array}{rcl}
\mathsf{FV}(X(e_1,\dots,e_n)) & = & \bigcup_i \mathsf{FV}(e_i) \\
\mathsf{FV}(P \otimes Q) & = & \mathsf{FV}(P) \cup \mathsf{FV}(Q) \\
\mathsf{FV}(P \multimap N) & = & \mathsf{FV}(P) \cup \mathsf{FV}(N) \\
\mathsf{FV}(\Uparrow P) & = & \mathsf{FV}(P) \\
\mathsf{FV}(\Downarrow N) & = & \mathsf{FV}(N) \\
\mathsf{FV}(\forall x \in S.N) & = & \mathsf{FV}(N) \backslash \{x^S\} \\
\mathsf{FV}(\exists x \in S.P) & = & \mathsf{FV}(P) \backslash \{x^S\} \\
\mathsf{FV}(\{e \preceq_S f\} \wedge P) & = & \mathsf{FV}(P) \cup \mathsf{FV}(e) \cup \mathsf{FV}(f) \\
\mathsf{FV}(\{e \succeq_S f\} \mapsto P) & = & \mathsf{FV}(P) \cup \mathsf{FV}(e) \cup \mathsf{FV}(f)
\end{array}$$

Definition 45. *Let A be a type such that $\mathsf{FV}(A) = \{x_1^{S_1}, \dots, x_n^{S_n}\}$.*

- *An **interface** of A is a bijective function $\sigma : [\![1, n]\!] \to \mathsf{FV}(A)$, represented by the notation $(x_{\sigma(1)}, \dots, x_{\sigma(n)})$.*
- *A **predicate** is a type A together with an interface $(x_{\sigma(1)}, \dots, x_{\sigma(n)})$, and is denoted by $A(x_{\sigma(1)}, \dots, x_{\sigma(n)})$.*
- *The **arity** of such a predicate is $S_{\sigma(1)} \times \cdots \times S_{\sigma(n)}$.*
- *The **polarity** of a predicate is the polarity of its type.*

Definition 46 (Substitution)**.** *Given a predicate $A(x_1, \dots, x_n)$ of arity $S_1 \times \cdots \times S_n$, we define a notion of **first-order substitution**. If e is a S_k-expression, then*

$$A(x_1, \dots, x_{k-1}, e, x_{k+1}, \dots, x_n)$$

is the type A where all occurrences of $x_k^{S_k}$ have been replaced by the first-order term e.

Notation 47. *Given a predicate $A(x_1, \dots, x_n)$, we may sometimes distinguish only one variable x_i of interest. In that case, if there is no ambiguity, we only write $A(x_i)$. Similarly, we may sometimes identify a predicate $A(x_1, \dots, x_n)$ with the type A.*

2.1.4.2 Typing rules

Definition 48 (Inequational theory)**.** *Let $\mathbb{S}$ be a first-order signature. An **inequational theory** $\mathcal{E}$ (over $\mathbb{S}$) is a finite set of inequations written*

$$\mathcal{E} = e_1 \preceq_{S_1} f_1, \ldots, e_n \preceq_{S_n} f_n$$

where for $i \in [1, n]$, e_i, f_i are S_i-expressions.

We use the notation $\mathcal{E}, e \preceq_S f$ to denote the union of $\mathcal{E}$ and the singleton $\{e \preceq_S f\}$. Every such inequational theory $\mathcal{E}$ induces a subtyping relation between types.

Definition 49 ($\mathcal{E}$-valuation)**.** *If ρ is a first-order valuation and $\mathcal{E}$ is an inequational context, we say that ρ is a $\mathcal{E}$-valuation iff*

$$e \preceq_S f \in \mathcal{E} \Rightarrow [\![e]\!]_\rho \preceq_S [\![f]\!]_\rho$$

We then define the relation $\preceq_\mathcal{E}$ as follows

$$e \preceq_\mathcal{E} f \Leftrightarrow \text{ if } \rho \text{ is a } \mathcal{E}\text{-valuation, then } [\![e]\!]_\rho \preceq_S [\![f]\!]_\rho$$

Definition 50 (Subtyping)**.** *Given an inequational theory $\mathcal{E}$, we define by induction in Figure 1 the following relations :*

- *The **subtyping relation** $A \sqsubseteq_\mathcal{E} B$ between two types of the same polarity.*
- *If $A \sqsubseteq_\mathcal{E} B$ and $B \sqsubseteq_\mathcal{E} A$, we note it $A \cong_\mathcal{E} B$.*

Definition 51 (Typing contexts and judgments)**.**

- *A **typing context** is a set of positive types assigned to variables written*

  $$x_1 : P_1, \ldots, x_n : P_n$$

  and denoted by the capital greek letters Γ, Δ.
- *A **typing judgment** is of one of the two following forms:*

  $$\mathcal{E}; \Gamma \vdash_0 t : N$$

  $$\mathcal{E}; \Gamma \vdash_0 v : P$$

where $\mathcal{E}$ is an inequational context, and Γ is a typing context.

$$\frac{}{A \sqsubseteq_{\mathcal{E}} A} \qquad \frac{e_k \cong_S f}{X(e_1, \ldots, e_{k-1}, e_k, \ldots, e_n) \sqsubseteq_{\mathcal{E}} X(e_1, \ldots, e_{k-1}, f, \ldots, e_n)}$$

$$\frac{A \sqsubseteq_{\mathcal{E}} B \qquad B \sqsubseteq_{\mathcal{E}} C}{A \sqsubseteq_{\mathcal{E}} C} \qquad \frac{P \sqsubseteq_{\mathcal{E}} P' \qquad Q \sqsubseteq_{\mathcal{E}} Q'}{P \otimes Q \sqsubseteq_{\mathcal{E}} P' \otimes Q'} \qquad \frac{P \sqsubseteq_{\mathcal{E}} Q \qquad N \sqsubseteq_{\mathcal{E}} M}{Q \multimap N \sqsubseteq_{\mathcal{E}} P \multimap M}$$

$$\frac{N \sqsubseteq_{\mathcal{E}} M}{\forall x \in S. N \sqsubseteq_{\mathcal{E}} \forall x \in S. M} \qquad \frac{P \sqsubseteq_{\mathcal{E}} Q}{\exists x \in S. P \sqsubseteq_{\mathcal{E}} \exists x \in S. Q} \qquad \frac{P \sqsubseteq_{\mathcal{E}} Q}{\Uparrow P \sqsubseteq_{\mathcal{E}} \Uparrow Q} \qquad \frac{N \sqsubseteq_{\mathcal{E}} M}{\Downarrow N \sqsubseteq_{\mathcal{E}} \Downarrow M}$$

$$\frac{e' \preceq_{\mathcal{E}} e \qquad f \preceq_{\mathcal{E}} f' \qquad N \sqsubseteq_{\mathcal{E}, f \succeq_S e} M}{\{e \succeq_S f\} \mapsto N \sqsubseteq_{\mathcal{E}} \{e' \succeq_S f'\} \mapsto M} \qquad \frac{e \preceq_{\mathcal{E}} e' \qquad f' \preceq_{\mathcal{E}} f \qquad P \sqsubseteq_{\mathcal{E}, e \succeq_S f} Q}{\{f \preceq_S e\} \wedge P \sqsubseteq_{\mathcal{E}} \{f' \preceq_S e'\} \wedge Q}$$

Figure 1: Definition of $\sqsubseteq_{\mathcal{E}}$

In Figure 2 are described the core typing rules corresponding to the *logical rules*. In Figure 4 are presented the rules for primitive integers.

Multiplicative connectives

$$\frac{}{\mathcal{E}; \Gamma, x : P \vdash_0 x : P}$$

$$\frac{\mathcal{E}; \Gamma \vdash_0 v : P}{\mathcal{E}; \Gamma \vdash_0 \mathsf{ret}(v) : \Uparrow P} \qquad \frac{\mathcal{E}; \Gamma \vdash_0 t : \Uparrow P \qquad \mathcal{E}; \Delta, x : P \vdash_0 u : M}{\mathcal{E}; \Gamma, \Delta \vdash_0 t \text{ to } x.u : M}$$

$$\frac{\mathcal{E}; \Gamma \vdash_0 t : N}{\mathcal{E}; \Gamma \vdash_0 \mathsf{thunk}(t) : \Downarrow N} \qquad \frac{\mathcal{E}; \Gamma \vdash_0 v : \Downarrow N}{\mathcal{E}; \Gamma \vdash_0 \mathsf{force}(v) : N}$$

$$\frac{}{\mathcal{E}; \Gamma \vdash_0 * : \mathbb{1}} \qquad \frac{\mathcal{E}; \Gamma \vdash_0 v : \mathbb{1} \qquad \mathcal{E}; \Delta \vdash_0 t : N}{\mathcal{E}; \Gamma, \Delta \vdash_0 \mathsf{let} * = v \,\mathsf{in}\, t : N} \qquad \frac{\mathcal{E}; \Gamma, x : P \vdash_0 t : N}{\mathcal{E}; \Gamma \vdash_0 \lambda x.t : P \multimap N}$$

$$\frac{\mathcal{E}; \Gamma \vdash_0 t : P \multimap N \qquad \mathcal{E}; \Delta \vdash_0 v : P}{\mathcal{E}; \Gamma, \Delta \vdash_0 (t) v : N}$$

$$\frac{\mathcal{E}; \Gamma \vdash_0 v : P \qquad \mathcal{E}; \Delta \vdash_0 w : Q}{\mathcal{E}; \Gamma, \Delta \vdash_0 (v, w) : P \otimes Q} \qquad \frac{\mathcal{E}; \Gamma \vdash_0 v : P \otimes Q \qquad \mathcal{E}; \Delta, x : P, y : Q \vdash_0 t : N}{\mathcal{E}; \Gamma, \Delta \vdash_0 \mathsf{let}\,(x, y) = v \,\mathsf{in}\, t : N}$$

Figure 2: λ_{LCBPV} Typing rules

Remarks 52. *Here are some remarks on the type system.*

First-order quantifiers

$$\frac{\mathcal{E};\Gamma \vdash_0 t : N \qquad x\#\Gamma}{\mathcal{E};\Gamma \vdash_0 t : \forall x \in S.N} \qquad \frac{\mathcal{E};\Gamma \vdash_0 t : \forall x \in S.N \qquad e \in S\text{-Exp}}{\mathcal{E};\Gamma \vdash_0 t : N[e/x]}$$

$$\frac{\mathcal{E};\Gamma \vdash_0 v : P[e/x] \qquad e \in S\text{-Exp}}{\mathcal{E};\Gamma \vdash_0 v : \exists x \in S.P}$$

$$\frac{\mathcal{E};\Gamma \vdash_0 v : \exists x \in S.P \qquad \mathcal{E};\Gamma, z : P[y/x] \vdash_0 a : A \qquad y\#\Gamma}{\mathcal{E};\Gamma \vdash_0 a[v/z] : A}$$

Inequational implication and conjunction

$$\frac{\mathcal{E}, e \succeq_S f;\Gamma \vdash_0 t : N}{\mathcal{E};\Gamma \vdash_0 t : \{e \succeq_S f\} \mapsto N} \qquad \frac{\mathcal{E};\Gamma \vdash_0 t : \{e \succeq_S e\} \mapsto N}{\mathcal{E};\Gamma \vdash_0 t : N} \qquad \frac{\mathcal{E};\Gamma \vdash_0 v : P \qquad e \preceq_{\mathcal{E}} f}{\mathcal{E};\Gamma \vdash_0 v : \{e \preceq_S f\} \wedge P}$$

$$\frac{\mathcal{E};\Gamma \vdash_0 v : \{e \preceq_S e\} \wedge P}{\mathcal{E};\Gamma \vdash_0 v : P}$$

Subtyping

$$\frac{\mathcal{E};\Gamma \vdash_0 a : A \qquad A \sqsubseteq_{\mathcal{E}} B}{\mathcal{E};\Gamma \vdash_0 a : B}$$

Figure 3: λ_{LCBPV} Typing rules II

Primitive integers

$$\frac{}{\mathcal{E};\Gamma \vdash_0 \underline{0} : \mathsf{Nat}} \qquad \frac{\mathcal{E};\Gamma \vdash_0 v : \mathsf{Nat}}{\mathcal{E};\Gamma \vdash_0 \underline{\mathsf{s}}(v) : \mathsf{Nat}}$$

$$\frac{\mathcal{E};\Gamma \vdash_0 v : \mathsf{Nat} \qquad \mathcal{E};\Delta, x : \mathsf{Nat} \vdash_0 t : N \qquad \mathcal{E};\Delta, x : \mathsf{Nat} \vdash_0 u : N}{\mathcal{E};\Gamma, \Delta \vdash_0 \mathsf{case}\, v\, \mathsf{of}\, x.t \parallel x.u : N}$$

Figure 4: λ_{LCBPV} Typing rules III

1. *The type system is not* linear *but* affine. *That is, we allow unrestricted weakening. In particular a program typable by $P \multimap N$ uses its argument* at most once *(and not exactly once).*

2. *The rules concerning the inequational implication and conjunctions are perfectly trans-*

parent from the point of view of the terms: the term of the premise is not changed in the conclusion. This allows to optimize the terms associated to proofs containing some computationally-irrelevant reasoning.

Finally, we define different fragments of λ_{LCBPV} type system.

- The system containing only predicate variables, Nat, the tensor $\otimes$ and the affine implication $\multimap$ is reffered by:
$$\lambda_{\mathsf{LCBPV}}^{\otimes\mathsf{Nat}}$$
- The system containing in addition to Nat, $\otimes$, $\multimap$ the two universal and existential quantifiers is denoted by:
$$\lambda_{\mathsf{LCBPV}}^{\otimes\mathsf{Nat}\forall}$$

2.2 | Call-by-name and call-by-value translations

It is well-known that one can retrieve in CBPV both a call-by-name and a call-by-value as fragments. In fact, the purpose of CBPV is to unveil the elementary particles used to build these two calculi. In this section, we show that this is still true when we consider our affine variant λ_{LCBPV} of CBPV. We define an affinely typed λ-calculus called λ_{Aff}. We then endow it with two different reduction strategies, one being call-by-name, the other being call-by-value. We show how it is possible to give two different *typed translations* of λ_{Aff} in λ_{LCBPV}. These translations are shown to be both type and reduction preserving. The call-by-name calculus corresponds to the purely negative part of λ_{LCBPV}, while the call-by-value corresponds to the purely positive part of λ_{LCBPV}.

2.2.1 The λ-calculus with pairs and integers

We define a affinely typed λ-calculus with pairs and integers. The syntax of terms and types is given as follows:

$$\begin{array}{llll} \mathsf{Terms} & t,u & ::= & x \mid \underline{\mathsf{n}} \mid \underline{\mathsf{s}}(t) \mid \lambda x.t \mid (t)u \mid \mathsf{case}\, t\, \mathsf{of}\, x.u_1 \,\|\, x.u \mid (t,u) \mid \mathsf{let}\,(x,y) = t\,\mathsf{in}\,u \\ \mathsf{Types} & A,B & ::= & \mathsf{Nat} \mid A \multimap B \mid A \otimes B \end{array}$$

A typing judgment is of the form

$$x_1 : A, \ldots, x_n : A \vdash_{\mathsf{Aff}} t : B$$

The typing rules are given in Figure 5.

$$\frac{}{\Gamma, x : A \vdash_{\mathsf{Aff}} x : A} \qquad \frac{}{\Gamma \vdash_{\mathsf{Aff}} \underline{0} : \mathsf{Nat}} \qquad \frac{\Gamma \vdash_{\mathsf{Aff}} t : \mathsf{Nat}}{\Gamma \vdash_{\mathsf{Aff}} \underline{\mathsf{s}}(t) : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash_{\mathsf{Aff}} t : \mathsf{Nat} \qquad \Delta, x : \mathsf{Nat} \vdash_{\mathsf{Aff}} u_1 : A \qquad \Delta, x : \mathsf{Nat} \vdash_{\mathsf{Aff}} u_2 : A}{\Gamma, \Delta \vdash_{\mathsf{Aff}} \mathsf{case}\, t\, \mathsf{of}\, x.u_1 \parallel x.u_2 : A} \qquad \frac{\Gamma, x : A \vdash_{\mathsf{Aff}} t : B}{\Gamma \vdash_{\mathsf{Aff}} \lambda x.t : A \multimap B}$$

$$\frac{\Gamma \vdash_{\mathsf{Aff}} t : A \multimap B \qquad \Delta \vdash_{\mathsf{Aff}} u : A}{\Gamma, \Delta \vdash_{\mathsf{Aff}} (t)u : B} \qquad \frac{\Gamma \vdash_{\mathsf{Aff}} t : A \qquad \Delta \vdash_{\mathsf{Aff}} u : B}{\Gamma, \Delta \vdash_{\mathsf{Aff}} (t, u) : A \otimes B}$$

$$\frac{\Gamma \vdash_{\mathsf{Aff}} t : A \otimes B \qquad \Delta, x : A, y : B \vdash_{\mathsf{Aff}} u : C}{\Gamma, \Delta \vdash_{\mathsf{Aff}} \mathsf{let}\, (x, y) \;=\; t\, \mathsf{in}\, u : C}$$

Figure 5: λ_{Aff} typing rules

2.2.2 Call-by-name

We now endow λ_{Aff} with a call-by-name reduction relation $\to_{\mathsf{N}}$. It is described thanks to an abstract machine, which is a simple variant of Krivine Abstract Machine. We then define a translation of λ_{Aff} terms and types into λ_{LCBPV} and show that this translation preserves both typing and the reduction relation.

Reduction

An environment is an element of the following grammar:

$$E \quad ::= \quad \mathsf{nil} \mid t.E \mid \underline{\mathsf{s}}.E \mid (x.t \mid x.u).E \mid ((x, y).t).E$$

A configuration is a pair $\langle t, E \rangle_{\mathsf{N}}$ of a term and a call-by-name environment. We now define the reduction relation $\to_{\mathsf{N}}$ between such configurations as the union of the following reduction steps:

$$
\begin{array}{rcl}
\langle (t)u, E\rangle_{\mathsf{N}} & \to_{\mathsf{N}} & \langle t, u.E\rangle_{\mathsf{N}} \\
\langle \lambda x.t, u.E\rangle_{\mathsf{N}} & \to_{\mathsf{N}} & \langle t[u/x], E\rangle_{\mathsf{N}} \\
\langle \underline{\mathsf{s}}(t), E\rangle_{\mathsf{N}} & \to_{\mathsf{N}} & \langle t, \underline{\mathsf{s}}.E\rangle_{\mathsf{N}} \\
\langle \underline{\mathsf{n}}, \underline{\mathsf{s}}.E\rangle_{\mathsf{N}} & \to_{\mathsf{N}} & \langle \underline{\mathsf{n}+1}, E\rangle_{\mathsf{N}} \\
\langle \mathsf{case}\, t\, \mathsf{of}\, x.u_1 \parallel x.u_2, E\rangle_{\mathsf{N}} & \to_{\mathsf{N}} & \langle t, (x.u_1 \mid x.u_2).E\rangle_{\mathsf{N}} \\
\langle \underline{\mathsf{n}+1}, (x.t \mid x.u).E\rangle_{\mathsf{N}} & \to_{\mathsf{N}} & \langle u[\underline{\mathsf{n}}/x], E\rangle_{\mathsf{N}} \\
\langle \underline{\mathsf{0}}, (x.t \mid x.u).E\rangle_{\mathsf{N}} & \to_{\mathsf{N}} & \langle t[\underline{\mathsf{0}}/x], E\rangle_{\mathsf{N}} \\
\langle \mathsf{let}\,(x,y) = t\,\mathsf{in}\,u, E\rangle_{\mathsf{N}} & \to_{\mathsf{N}} & \langle t, ((x,y).u).E\rangle_{\mathsf{N}} \\
\langle (t,u), ((x,y).u).E\rangle_{\mathsf{N}} & \to_{\mathsf{N}} & \langle u[t/x, u/y], E\rangle_{\mathsf{N}}
\end{array}
$$

Translation

We now give a typed translation of λ_{Aff} to λ_{LCBPV}. This translation of terms and types is then shown to satisfy a preservation theorem. We first define a type translation map $(.)^{\mathsf{N}}$ that maps λ_{Aff} types to **negative** types of λ_{LCBPV}. It is defined by induction on types:

$$
\begin{array}{rcl}
\mathsf{Nat}^{\mathsf{N}} & = & \Uparrow\mathsf{Nat} \\
(A \multimap B)^{\mathsf{N}} & = & (\Downarrow A^{\mathsf{N}}) \multimap B^{\mathsf{N}} \\
(A \otimes B)^{\mathsf{N}} & = & \Uparrow((\Downarrow A^{\mathsf{N}}) \otimes (\Downarrow B^{\mathsf{N}}))
\end{array}
$$

We then provide a translation of λ_{Aff} terms to λ_{LCBPV} computations as follows:

$$
\begin{array}{rcl}
x^{\mathsf{N}} & = & \mathsf{force}(x) \\
\underline{\mathsf{n}}^{\mathsf{N}} & = & \mathsf{ret}(\underline{\mathsf{n}}) \\
(\underline{\mathsf{s}}(t))^{\mathsf{N}} & = & t\ \mathsf{to}\ x.\mathsf{ret}(\underline{\mathsf{s}}(x)) \\
(\mathsf{case}\, t\, \mathsf{of}\, x.t_1 \parallel x.t_2)^{\mathsf{N}} & = & t^{\mathsf{N}}\ \mathsf{to}\ z.\mathsf{case}\, z\, \mathsf{of}\, x.t_1^{\mathsf{N}} \parallel x.t_2^{\mathsf{N}} \\
(\lambda x.t)^{\mathsf{N}} & = & \lambda x.t^{\mathsf{N}} \\
((t)u)^{\mathsf{N}} & = & (t^{\mathsf{N}})\mathsf{thunk}(u^{\mathsf{N}}) \\
((t,u))^{\mathsf{N}} & = & \mathsf{ret}((\mathsf{thunk}(t^{\mathsf{N}}), \mathsf{thunk}(u^{\mathsf{N}}))) \\
(\mathsf{let}\,(x,y) = t\,\mathsf{in}\,u)^{\mathsf{N}} & = & t^{\mathsf{N}}\ \mathsf{to}\ z.\mathsf{let}\,(x,y) = z\,\mathsf{in}\,u^{\mathsf{N}}
\end{array}
$$

This translation of terms and types satisfies a type preservation theorem.

Theorem 53 (Call-by-name type preservation)**.** *If* $x_1 : A_1, \ldots, x_n : A_n \vdash_{\mathsf{Aff}} t : B$ *then* $x_1 : A_1^{\mathsf{N}}, \ldots, x_n : A_n^{\mathsf{N}} \vdash_{\mathsf{V}} t^{\mathsf{N}} : B^{\mathsf{N}}$.

We now want to show that the reduction is also preserved. To do that, we need to give first a translation of call-by-name environments into λ_{LCBPV} environments. It is defined by

induction as follows:

$$\begin{array}{rcl}
\mathsf{nil}^{\mathsf{N}} & = & \mathsf{nil} \\
(t.E)^{\mathsf{N}} & = & \mathsf{thunk}(t^{\mathsf{N}}).E^{\mathsf{N}} \\
(\underline{\mathsf{s}}.E)^{\mathsf{N}} & = & \mathsf{f}(x.\mathsf{ret}(\underline{\mathsf{s}}(x))).E^{\mathsf{N}} \\
((x.t \mid x.u).E)^{\mathsf{N}} & = & \mathsf{f}(z.\mathsf{case}\, z\, \mathsf{of}\, x.t_1^{\mathsf{N}} \parallel x.t_2^{\mathsf{N}}).E^{\mathsf{N}} \\
(((x,y).t).E)^{\mathsf{N}} & = & \mathsf{f}(z.\mathsf{let}\, (x,y) = z\, \mathsf{in}\, u^{\mathsf{N}}).E^{\mathsf{N}}
\end{array}$$

A call-by-name configuration $\langle t, E\rangle_{\mathsf{N}}$ is then mapped to a λ_{LCBPV} configuration

$$(\langle t, E\rangle_{\mathsf{N}})^{\mathsf{N}} = \langle t^{\mathsf{N}}, E^{\mathsf{N}}\rangle_0$$

Theorem 54 (Call-by-name reduction preservation)**.** *Let C, C' be two call-by-name configurations. Then*

$$C \to_{\mathsf{N}} C' \Rightarrow (C)^{\mathsf{N}} \overset{0}{\to}{}^{*}\ C'^{\mathsf{N}}$$

2.2.3 Call-by-value

We now endow λ_{Aff} with a call-by-value reduction relation $\to_{\mathsf{V}}$. It is described thanks to an abstract machine. We then define a translation of λ_{Aff} terms and types into λ_{LCBPV} and show that this translation preserves both typing and the call-by-value reduction relation.

Reduction

To define the reduction relation, we first need to define what values are:

$$v, w \quad ::= \quad \lambda x.t \mid (v, w) \mid \underline{\mathsf{n}}$$

A **call-by-value environment** is an element of the following grammar:

$$E \quad ::= \quad \mathsf{nil} \mid \mathsf{f}(v).E \mid \mathsf{a}(t).E \mid \underline{\mathsf{s}}.E \mid (x.t \mid x.u).E \mid (v, \_).E \mid (\_, t).E \mid ((x,y).t).E$$

A call-by-value configuration is a pair $\langle t, E\rangle_{\mathsf{V}}$ of a term and a call-by-value environment. We now define the reduction relation $\to_{\mathsf{V}}$ between such configurations as the union of the following reduction steps:

$$
\begin{array}{rcl}
\langle (t)u, E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle u, \mathsf{a}(t).E\rangle_{\mathsf{V}} \\
\langle \lambda x.t, \mathsf{f}(v).E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle t[v/x], E\rangle_{\mathsf{V}} \\
\langle \underline{\mathsf{s}}(t), E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle t, \underline{\mathsf{s}}.E\rangle_{\mathsf{V}} \\
\langle \underline{\mathsf{n}}, \underline{\mathsf{s}}.E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle \underline{\mathsf{n}+1}, E\rangle_{\mathsf{V}} \\
\langle \mathsf{case}\, t\, \mathsf{of}\, x.u_1 \parallel x.u_2, E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle t, (x.u_1 \mid x.u_2).E\rangle_{\mathsf{V}} \\
\langle \underline{\mathsf{n}+1}, (x.t \mid x.u).E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle u[\underline{\mathsf{n}}/x], E\rangle_{\mathsf{V}} \\
\langle \underline{\mathsf{0}}, (x.t \mid x.u).E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle t[\underline{\mathsf{0}}/x], E\rangle_{\mathsf{V}} \\
\langle \mathsf{let}\,(x,y) = t\,\mathsf{in}\,u, E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle t, ((x,y).u).E\rangle_{\mathsf{V}} \\
\langle (t,u), E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle t, (\_,u).E\rangle_{\mathsf{V}} \\
\langle v, (\_,t).E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle t, (v,\_).E\rangle_{\mathsf{V}} \\
\langle w, (v,\_).E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle (v,w), E\rangle_{\mathsf{V}} \\
\langle (v,w), ((x,y).u).E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle u[v/x, w/y], E\rangle_{\mathsf{V}}
\end{array}
$$

Translation

We now give a typed translation of λ_{Aff} to λ_{LCBPV}. This translation of values, terms and types is then shown to satisfy a type preservation theorem. We then show that call-by-value reduction is also preserved. We first define a type translation map $(.)^{\mathsf{V}}$ that maps λ_{Aff} types to *positive types* of λ_{LCBPV}. It is defined by induction on types:

$$
\begin{array}{rcl}
\mathsf{Nat}^V & = & \mathsf{Nat} \\
(A \multimap B)^V & = & \Downarrow(A^V \multimap (\Uparrow B^V)) \\
(A \otimes B)^V & = & A^V \otimes B^V
\end{array}
$$

We also give a translation of λ_{Aff} terms to computations of λ_{LCBPV}:

$$
\begin{array}{rcl}
x^V & = & \mathsf{ret}(x) \\
\underline{\mathsf{n}}^V & = & \mathsf{ret}(\underline{\mathsf{n}}) \\
(\underline{\mathsf{s}}(t))^V & = & t\ \mathsf{to}\ x.\mathsf{ret}(\underline{\mathsf{s}}(x)) \\
(\mathsf{case}\, t\, \mathsf{of}\, x.t_1 \parallel x.t_2)^V & = & t\ \mathsf{to}\ z.(\mathsf{case}\, z\, \mathsf{of}\, x.t_1^V \parallel x.t_2^V) \\
(\lambda x.t)^V & = & \mathsf{ret}(\mathsf{thunk}(\lambda x.t^V)) \\
((t)u)^V & = & u^V\ \mathsf{to}\ x.((t^V)\ \mathsf{to}\ y.((\mathsf{force}(y))x)) \\
((t,u))^V & = & t^V\ \mathsf{to}\ x.(u^V\ \mathsf{to}\ y.(\mathsf{ret}((x,y)))) \\
(\mathsf{let}\,(x,y) = t\,\mathsf{in}\,u)^V & = & t^V\ \mathsf{to}\ z.\mathsf{let}\,(x,y) = z\,\mathsf{in}\,u^V
\end{array}
$$

The translation satisfies the following preservation theorem.

Theorem 55 (Type preservation). *If* $x_1 : A_1, \dots, x_n : A_n \vdash_{\mathsf{Aff}} t : B$ *then* $x_1 : A_1^{\mathsf{V}}, \dots, x_n : A_n^{\mathsf{V}} \vdash_{\mathsf{V}} t^{\mathsf{V}} : \Uparrow B^{\mathsf{V}}$.

We now want to show that the reduction is also preserved. To do that, we need to give first a translation of call-by-value environments into λ_{LCBPV} environments. It is defined by induction as follows:

$$\begin{aligned}
\mathsf{nil}^{\mathsf{V}} &= \mathsf{nil} \\
(\mathsf{f}(t).E)^{\mathsf{V}} &= \mathsf{f}(x.(t^{\mathsf{V}} \text{ to } y.(\mathsf{force}(y))x).E^{\mathsf{V}} \\
(\mathsf{a}(v).E)^{\mathsf{V}} &= \mathsf{f}(x.\mathsf{force}(x)v^{\mathsf{V}}).(E)^{\mathsf{V}} \\
(\underline{\mathsf{s}}.E)^{\mathsf{V}} &= \mathsf{f}(x.\mathsf{ret}(\underline{\mathsf{s}}(x))).E^{\mathsf{V}} \\
((x.t \mid x.u).E)^{\mathsf{V}} &= \mathsf{f}(z.\mathsf{case}\, z \,\mathsf{of}\, x.t_1^{\mathsf{V}} \parallel x.t_2^{\mathsf{V}}).E^{\mathsf{V}} \\
(((x,y).t).E)^{\mathsf{V}} &= \mathsf{f}(z.\mathsf{let}\,(x,y) = z \,\mathsf{in}\, u^{\mathsf{V}}).E^{\mathsf{V}}
\end{aligned}$$

A call-by-value configuration $\langle t, E\rangle_{\mathsf{V}}$ is then mapped to a λ_{LCBPV} configuration

$$(\langle t, E\rangle_{\mathsf{V}})^{\mathsf{V}} = \langle t^{\mathsf{V}}, E^{\mathsf{V}}\rangle_0$$

Theorem 56 (Call-by-value reduction preservation). *Let* C, C' *be two call-by-value configurations. Then*

$$C \to_{\mathsf{V}} C' \Rightarrow C^{\mathsf{V}} \overset{0}{\to}{}^{*} C'^{\mathsf{V}}$$

2.3 | The monitoring abstract machine

We present an extension of the call-by-push-value that we call λ_{Mon}. λ_{Mon} is intended to be the target language of *various* translations of λ_{LCBPV}. As we will see, many translations are possible. Depending on the translation, observing the behavior of the translated programs will allow us to observe fine computational properties of the original source program. λ_{Mon} programs are executed in the **Monitoring Abstract Machine** (or **MAM**). It is an extension of the call-by-push-value machine defined in Section 2.1, with the addition of what we call a **monitoring state**, *i.e.* a state in which informations about the execution of λ-terms are accumlating during their reduction, and that can trigger different events depending on those informations.

2.3.1 λ_{Mon} syntax and reduction

Extended syntax

The syntax of computations, environments, configurations and co-configurations is extended as follows (here, $n \in \mathbb{N} \setminus \{0\}$):

$$\begin{array}{lcll} t & ::= & \ldots \mid (\!|t|\!)^{\alpha}_{n} & (\alpha \text{ is a closed computation}) \\ E & ::= & \ldots \mid \mathsf{m}_n(t).E & (t \text{ is a closed computation}) \\ C & ::= & \ldots \mid \langle t, E\rangle_n & (t \text{ is a closed computation}) \\ \overline{C} & ::= & \ldots \mid \langle \uparrow E, v\rangle_n & \end{array}$$

We will mostly be interested in configurations of the form:Âă

$$\langle t, \mathsf{a}(v_1).\ldots.\mathsf{a}(v_n).E\rangle_n$$

In such configurations the different values $v_1, \ldots, v_n$ represent the content of the memory cells. The constructor $(\!|.|\!)^{\alpha}_{n}$, which we call **observation**, will be responsible for the update of the n-th memory cell. This update is achieved through the program α called the **monitor**. This observation should be seen as an annotation in a program, that marks a point of control.

Notation 57. *For each $n \in \mathbb{N}$, we use $\lambda^{\mathbf{n}}_{\mathsf{Mon}}$ to refer to the restriction of this extended syntax to those constructors indexed up to n. As an example, $\lambda^{\mathbf{0}}_{\mathsf{Mon}}$ corresponds to λ_{LCBPV}. We use the notations $\mathbb{V}_n, \mathbb{P}_n, \mathbb{E}_n, \mathbb{C}_n$ and $\overline{\mathbb{C}}_n$ to denote respectively the closed values, closed computations, closed environments configurations and co-configurations of $\lambda^{\mathbf{n}}_{\mathsf{Mon}}$. We then have the following equalities:*

$$\mathbb{V} = \bigcup_n \mathbb{V}_n, \quad \mathbb{P} = \bigcup_n \mathbb{P}_n \quad \mathbb{E} = \bigcup_n \mathbb{E}_n \quad \mathbb{C} = \bigcup_n \mathbb{C}_n \text{ and } \overline{\mathbb{C}} = \bigcup_n \overline{\mathbb{C}}_n$$

Notation 58. *We will often use the vector notation as follows:*

- *$\overrightarrow{v}$ denotes a tuple of values $(v_1, (v_2, \ldots, v_n)\ldots)$.*
- *$\overrightarrow{\mathsf{a}(v)}.E$ denotes a stack $\mathsf{a}(v_1).\ldots.\mathsf{a}(v_n).E$.*

Monitoring reduction

We define a new reduction relation, denoted by $\xrightarrow{n}$. It is defined by induction on $n \in \mathbb{N}$, starting with $\xrightarrow{0}$ defined in Chapter II , using the following schemes of rules.

- The first scheme of reduction rules only says that $\xrightarrow{n+1}$ contains $\xrightarrow{n}$. It is defined as a rule, saying every $\xrightarrow{n}$ reduction step induces a corresponding $\xrightarrow{n+1}$ reduction step.

$$\frac{C \xrightarrow{n} C'}{C \xrightarrow{n+1} C'}$$

- The second scheme of reduction rules extends the relation $\xrightarrow{n}$ by adding exactly one memory cell. Each $\xrightarrow{n}$ reduction step induces a new reduction step that does not alter the value in that memory cell.

$$\frac{\langle t, \mathsf{a}(v_1).\ldots.\mathsf{a}(v_n).E\rangle_n \xrightarrow{n} \langle t', \mathsf{a}(v_1').\ldots.\mathsf{a}(v_n').E'\rangle_n}{\langle t, \mathsf{a}(v_1).\ldots.\mathsf{a}(v_n).\mathsf{a}(v).E\rangle_{n+1} \xrightarrow{n+1} \langle t', \mathsf{a}(v_1').\ldots.\mathsf{a}(v_n').\mathsf{a}(v).E'\rangle_{n+1}}$$

- We also distinguish the case when the daimon configuration is observed:

$$\frac{\langle t, \mathsf{a}(v_1).\ldots.\mathsf{a}(v_n).E\rangle_n \xrightarrow{n} \maltese}{\langle t, \mathsf{a}(v_1).\ldots.\mathsf{a}(v_{n+1}).E\rangle_{n+1} \xrightarrow{n+1} \maltese}$$

- The next rule allows to enter what we call the **monitoring mode**. It happens when an observation of the form $(\!|t|\!)^{\alpha}_n$ is in head position, and triggers the introduction of the monitor α, but at level n:

$$\langle (\!|t|\!)^{\alpha}_n, \mathsf{a}(v_1).\ldots.\mathsf{a}(v_{n+1}).E\rangle_{n+1} \xrightarrow{n+1} \langle \alpha, \mathsf{a}(v_1).\ldots.\mathsf{a}(v_{n+1}).\mathsf{m}_{n+1}(t).E\rangle_n$$

  Being at level n, α can then access the value v_{n+1} and modify it.

- We introduce a rule to exit the monitoring mode at level n. Once the computation triggered by a previous use of the monitor α has ended on a value, we can use the following rule to put that value into the monitoring state:

$$\langle \mathsf{ret}(v), \mathsf{a}(v_1).\ldots.\mathsf{a}(v_n).\mathsf{m}_t(n+1).E\rangle_n \xrightarrow{n+1} \langle t, \mathsf{a}(v_1).\ldots.\mathsf{a}(v_n).\mathsf{a}(v).E\rangle_{n+1}$$

- Finally, we also introduce the following equivalence rule:

$$\langle \mathsf{ret}(v), \mathsf{a}(v_1) \dots \mathsf{a}(v_{n+1}).E \rangle_{n+1} \cong_{n+1} \langle \uparrow E, (v, (v_{n+1}, \dots, v_1)) \rangle_{n+1}$$

Remark 59. *We can explain the meaning of the equivalence rule $\cong_n$. Consider the case $n = 1$:*

$$\langle \mathsf{ret}(v), \mathsf{a}(w).E \rangle_1 \cong_1 \langle \uparrow E, (v, w) \rangle_1$$

This will correspond to the fact that in the store-passing program translation, during the computation, the content of the state of type S is passed as an argument of the program, *and this corresponds to a type of the form*

$$S \multimap N$$

When the computation ends however, we return a pair of values *containing the value of the program and the final content of the store, corresponding to the type*

$$S \otimes P$$

To prove some results of Chapter IV (in particular the connection theorem), we will need to be able to switch between the two representations.

Notation 60. *We define the monitoring reduction relation $\to$ as the union of all $\overset{n}{\to}$:*

$$\to \quad = \quad \bigcup_{n \in \mathbb{N}} \overset{n}{\to}$$

We also define the monitoring equivalence relation $\cong$ as the union of all $\cong_n$:

$$\cong \quad = \quad \bigcup_{n \in \mathbb{N}} \cong_n$$

We finally define the notation $C \Uparrow$ that stands for "C diverges".

2.3.2 Examples of reduction at level ≤ 1

To give some intuitions on the monitoring abstract machine, we show several examples of reductions of a same configuration

$$\langle (\!| (\!| I |\!)^{\alpha}_1 \underline{0} |\!)^{\alpha}_1, \mathsf{a}(v).E \rangle_1$$

but for different choices of α and different choice of the value v initially put in the state. The reduction steps which are about the manipulation of the monitoring state have been highlighted in blue.

Trivial monitoring

The first monitor example is the **trivial monitor**

$$\alpha = I = \lambda x.\mathsf{ret}(x)$$

We consider any value v that we put in the state. Here is the sequence of reduction steps:

$$\begin{array}{rcl}
\langle (\!|(\!|I|\!)^{\alpha}_{1}\underline{0}|\!)^{\alpha}_{1}, \mathsf{a}(v).E\rangle_1 & \xrightarrow{1} & \langle \lambda x.\mathsf{ret}(x), \mathsf{a}(v).\mathsf{m}_1((\!|I|\!)^{\alpha}_{1}\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle \mathsf{ret}(v), \mathsf{m}_1((\!|I|\!)^{\alpha}_{1}\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle (\!|I|\!)^{\alpha}_{1}\underline{0}, \mathsf{a}(v).E\rangle_1 \\
& \xrightarrow{1} & \langle (\!|I|\!)^{\alpha}_{1}, \mathsf{a}(v).\mathsf{a}(\underline{0}).E\rangle_1 \\
& \xrightarrow{1} & \langle \lambda x.\mathsf{ret}(x), \mathsf{a}(v).\mathsf{m}_1(I).\mathsf{a}(\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle \mathsf{ret}(v), \mathsf{m}_1(I).\mathsf{a}(\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle I, \mathsf{a}(v).\mathsf{a}(\underline{0}).E\rangle_1 \\
& \xrightarrow{1} & \langle \mathsf{ret}(\underline{0}), \mathsf{a}(v).E\rangle_1
\end{array}$$

Clearly, the monitor is not interfering with the reduction. If we erase the highlighted reduction steps, we see that we retrieve the usual $\xrightarrow{0}$ reduction.

Clock

We now consider the **clock monitor**

$$\alpha = \lambda x.\mathsf{ret}(\underline{\mathsf{s}}(x))$$

We show the reduction sequence of $(\!|(\!|I|\!)^{\alpha}_{1}\underline{0}|\!)^{\alpha}_{1}$ and the term $\underline{3}$ in the monitoring state.

$$\begin{array}{rcl}
\langle (\!|(\!|I|\!)^{\alpha}_{1}\underline{0}|\!)^{\alpha}_{1}, \mathsf{a}(\underline{3}).E\rangle_1 & \xrightarrow{1} & \langle \lambda x.\mathsf{ret}(\underline{\mathsf{s}}(x)), \mathsf{a}(\underline{3}).\mathsf{m}_1((\!|I|\!)^{\alpha}_{1}\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle \mathsf{ret}(\underline{4}), \mathsf{m}_1((\!|I|\!)^{\alpha}_{1}\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle (\!|I|\!)^{\alpha}_{1}\underline{0}, \mathsf{a}(\underline{4}).E\rangle_1 \\
& \xrightarrow{1} & \langle (\!|I|\!)^{\alpha}_{1}, \mathsf{a}(\underline{4}).\mathsf{a}(\underline{0}).E\rangle_1 \\
& \xrightarrow{1} & \langle \lambda x.\mathsf{ret}(\underline{\mathsf{s}}(x)), \mathsf{a}(\underline{4}).\mathsf{m}_1(I).\mathsf{a}(\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle \mathsf{ret}(\underline{5}), \mathsf{m}_1(I).\mathsf{a}(\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle I, \mathsf{a}(\underline{5}).\mathsf{a}(\underline{0}).E\rangle_1 \\
& \xrightarrow{1} & \langle \mathsf{ret}(\underline{0}), \mathsf{a}(\underline{5}).E\rangle_1
\end{array}$$

This example witnesses one possible use of the monitoring state: a counter incremented when certain parts of a λ-term are executed. This can be used to implement a *clock*.

Countdown: divergence

Here is another example. We now choose the following monitor:

$$\alpha = \lambda x.\mathsf{case}\, x\, \mathsf{of}\, x.\Omega \parallel x.\mathsf{ret}(x)$$

Where Ω is the call-by-name translation of the term $(\lambda x.xx)(\lambda x.xx)$:

$$\Omega \stackrel{\mathsf{def}}{=} ((\lambda x.xx)(\lambda x.xx))^{\mathsf{N}}$$

Hence, when Ω comes into head position, it makes the configuration diverge. We show two different executions: when the state is initially set to $\underline{2}$ and $\underline{1}$.

$$\begin{array}{rcl}
\langle (\!|(\!|I|\!)^{\alpha}_1 \underline{0}|\!)^{\alpha}_1, \mathsf{a}(\underline{2}).E\rangle_1 & \xrightarrow{1} & \langle \alpha, \mathsf{a}(\underline{2}).\mathsf{m}_1((\!|I|\!)^{\alpha}_1 \underline{0}).E\rangle_0 \\
& \xrightarrow{1}{}^{*} & \langle \mathsf{ret}(\underline{1}), \mathsf{m}_1((\!|I|\!)^{\alpha}_1 \underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle (\!|I|\!)^{\alpha}_1 \underline{0}, \mathsf{a}(\underline{1}).E\rangle_1 \\
& \xrightarrow{1} & \langle (\!|I|\!)^{\alpha}_1, \mathsf{a}(\underline{1}).\mathsf{a}(\underline{0}).E\rangle_1 \\
& \xrightarrow{1} & \langle \alpha, \mathsf{a}(\underline{1}).\mathsf{m}_1(I).\mathsf{a}(\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle \mathsf{ret}(\underline{0}), \mathsf{m}_1(I).\mathsf{a}(\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle I, \mathsf{a}(\underline{0}).\mathsf{a}(\underline{0}).E\rangle_1 \\
& \xrightarrow{1} & \langle \mathsf{ret}(\underline{0}), \mathsf{a}(\underline{0}).E\rangle_1
\end{array}$$

Here the counter is decremented and the execution is somewhat similar to the previous example. Now here is what happens when we put $\underline{1}$ in the monitoring state.

$$\begin{array}{rcl}
\langle (\!|(\!|I|\!)^{\alpha}_1 \underline{0}|\!)^{\alpha}_1, \mathsf{a}(\underline{1}).E\rangle_1 & \xrightarrow{1} & \langle \alpha, \mathsf{a}(\underline{1}).\mathsf{m}_1((\!|I|\!)^{\alpha}_1 \underline{0}).E\rangle_0 \\
& \xrightarrow{1}{}^{*} & \langle \mathsf{ret}(\underline{0}), \mathsf{m}_1((\!|I|\!)^{\alpha}_1 \underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle (\!|I|\!)^{\alpha}_1 \underline{0}, \mathsf{a}(\underline{0}).E\rangle_1 \\
& \xrightarrow{1} & \langle (\!|I|\!)^{\alpha}_1, \mathsf{a}(\underline{0}).\mathsf{a}(\underline{0}).E\rangle_1 \\
& \xrightarrow{1} & \langle \alpha, \mathsf{a}(\underline{0}).\mathsf{m}_1(I).\mathsf{a}(\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle \Omega, \mathsf{m}_1(I).\mathsf{a}(\underline{0}).E\rangle_0 \\
& \Uparrow &
\end{array}$$

Here, the monitor plays the role of a countdown: it is a counter that makes the computation diverge once it reaches $\underline{0}$. This example shows that changing the value in the monitoring state can affect the termination of the resulting configuration.

Countdown: termination

If we now choose the following variation of the previous example:

$$\alpha = \lambda x.\mathsf{case}\, x\, \mathsf{of}\, x.✠ \parallel x.\mathsf{ret}(x)$$

We put $\underline{1}$ in the monitoring state and reduce the same term as before.

$$\begin{array}{rcl}
\langle (\!|(\!|I|\!)_1^\alpha \underline{0}|\!)_1^\alpha, \mathsf{a}(\underline{1}).E\rangle_1 & \xrightarrow{1} & \langle \alpha, \mathsf{a}(\underline{1}).\mathsf{m}_1((\!|I|\!)_1^\alpha \underline{0}).E\rangle_0 \\
& \xrightarrow{1}{}^* & \langle \mathsf{ret}(\underline{0}), \mathsf{m}_1((\!|I|\!)_1^\alpha \underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle (\!|I|\!)_1^\alpha \underline{0}, \mathsf{a}(\underline{0}).E\rangle_1 \\
& \xrightarrow{1} & \langle (\!|I|\!)_1^\alpha, \mathsf{a}(\underline{0}).\mathsf{a}(\underline{0}).E\rangle_1 \\
& \xrightarrow{1} & \langle \alpha, \mathsf{a}(\underline{0}).\mathsf{m}_1(I).\mathsf{a}(\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \langle \maltese, \mathsf{m}_1(I).\mathsf{a}(\underline{0}).E\rangle_0 \\
& \xrightarrow{1} & \maltese
\end{array}$$

Here again, we have a countdown. But when the counter reaches $\underline{0}$, we stop the execution using $\maltese$. Dually to the previous example where divergence can be caused by the value in the monitoring state, here it can prevent it from happening by triggering the execution of $\maltese$.

2.3.3 Reduction at level ≥ 2

The reduction at level ≥ 2 allows more complex behaviors.

Example 61. *One should observe that when executed at level* 2*, a program has no* direct *access to the memory cell of level* 1*. Indeed, changing the value of a memory cell is only possible by triggering an observation* $(\!|.|\!)_2$*, which is done on the memory cell of the current level.*

$$\langle (\!|t|\!)_2^\alpha, \mathsf{a}(v).\mathsf{a}(w).E\rangle_2 \to \langle \alpha, \mathsf{a}(v).\mathsf{a}(w).\mathsf{m}_2(t).E\rangle_1$$

It is however possible to change indirectly *the memory cell of level* 1 *through an appropriate monitor. Suppose* $\beta = \lambda x.\mathsf{ret}(\underline{\mathsf{s}}(x))$ *and* $\alpha = \lambda x.(\!|\mathsf{ret}(x)|\!)_1^\beta$*, then the configuration* $\langle (\!|t|\!)_2^\alpha, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{a}(v).\mathsf{nil}\rangle_2$ *reduces as follows:*

$$\begin{array}{rcl}
\langle (\!|t|\!)_2^\alpha, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{a}(v).\mathsf{nil}\rangle_2 & \to & \langle \lambda x.(\!|x|\!)_1^\beta, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{a}(v).\mathsf{m}_2(t).\mathsf{nil}\rangle_1 \\
& \to^* & \langle (\!|\mathsf{ret}(v)|\!)_1^\beta, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{m}_2(t).\mathsf{nil}\rangle_1 \\
& \to & \langle \lambda x.\mathsf{ret}(\underline{\mathsf{s}}(x)), \mathsf{a}(\underline{\mathsf{n}}).\mathsf{m}_1(\mathsf{ret}(v)).\mathsf{m}_2(t).\mathsf{nil}\rangle_0 \\
& \to^* & \langle \mathsf{ret}(\underline{\mathsf{n}+1}), \mathsf{m}_1(\mathsf{ret}(v)).\mathsf{m}_2(t).\mathsf{nil}\rangle_0 \\
& \to & \langle \mathsf{ret}(v), \mathsf{a}(\underline{\mathsf{n}+1}).\mathsf{m}_2(t).\mathsf{nil}\rangle_1 \\
& \to & \langle t, \mathsf{a}(\underline{\mathsf{n}+1}).\mathsf{a}(v).\mathsf{nil}\rangle_2
\end{array}$$

Example 62. *One can also make the update of the level* 1 *memory cell depend on the content of the* 2 *memory cell. Suppose* $\beta = \lambda x.\mathsf{ret}(\underline{\mathsf{s}}(x))$ *and* $\alpha = \lambda x.\mathsf{case}\, x\, \mathsf{of}\, x.(\!|\mathsf{ret}(\underline{0})|\!)_1^\beta \parallel x.\mathsf{ret}(x)$*. Here are two reductions for the configurations* $\langle (\!|t|\!)_2^\alpha, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{a}(\underline{\mathsf{k}}).\mathsf{nil}\rangle_2$ *where* $k \in \{0, 1\}$*. When*

$k = 1$ the reduction is:

$$\begin{array}{rcl} \langle (\!|t|\!)_2^\alpha, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{a}(\underline{1}).\mathsf{nil}\rangle_2 & \to & \langle \alpha, \mathsf{a}(\underline{1}).\mathsf{a}(\underline{\mathsf{n}}).\mathsf{m}_2(t).\mathsf{nil}\rangle_1 \\ & \to^* & \langle \mathsf{ret}(\underline{\mathsf{n}}), \mathsf{a}(\underline{0}).\mathsf{m}_2(t).\mathsf{nil}\rangle_1 \\ & \to^* & \langle t, \mathsf{a}(\underline{0}).\mathsf{a}(\underline{\mathsf{n}}).\mathsf{nil}\rangle_2 \end{array}$$

When $k = 0$, we have

$$\begin{array}{rcl} \langle (\!|t|\!)_2^\alpha, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{a}(\underline{0}).\mathsf{nil}\rangle_2 & \to^* & \langle \alpha, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{a}(\underline{0}).\mathsf{m}_2(t).\mathsf{nil}\rangle_1 \\ & \to^* & \langle (\!|\mathsf{ret}(\underline{0})|\!)_1^\beta, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{m}_2(t).\mathsf{nil}\rangle_1 \\ & \to^* & \langle \mathsf{ret}(\underline{\mathsf{n}+1}), \mathsf{m}_1(\mathsf{ret}(\underline{0})).\mathsf{m}_2(t).\mathsf{nil}\rangle_0 \\ & \to & \langle \mathsf{ret}(\underline{0}), \mathsf{a}(\underline{\mathsf{n}+1}).\mathsf{m}_2(t).\mathsf{nil}\rangle_1 \\ & \to & \langle t, \mathsf{a}(\underline{\mathsf{n}+1}).\mathsf{a}(\underline{0}).\mathsf{nil}\rangle_2 \end{array}$$

If it is possible to make the level 1 cell depends of the level 2 cell, it is impossible to reverse this dependency: the level 2 cell cannot depend a priori (without adding new primitives) of the the level 1 cell.

2.3.4 Program translations

We now give a first glimpse on typical use cases of the monitoring abstract machine. λ_{Mon} will often be the target of a certain translation from (an extension of) λ_{LCBPV} programs. These translations are typically *annotations* of the source program with well-chosen observations. By observing the behavior of these annotated programs in the **MAM**, we will be able to observe complex computational properties of the source program. We first define a generic annotation, parametric in the choice of the monitor.

Definition 63 (Annotation)**.** *Let α be a closed term and $n \in \mathbb{N}$. If t is a computation, we define $\{t\}_k^\alpha$ by induction on t. On λ constructors it is defined as:*

$$\{\lambda x.t\}_k^\alpha \quad = \quad (\!|\lambda x.\{t\}_k^\alpha|\!)_k^\alpha$$

and $\{.\}_k^\alpha$ commutes with all other constructors.

To put things simply, $\{.\}_k^\alpha$ annotates each λ constructor with an observation $(\!|.|\!)_\alpha^k$. This implies that before each β-reduction happening during the execution, this observation is triggered. We now look at particular instances of this annotation.

Bounded-time termination

The first example uses the following monitor, already seen in the examples:

$$\alpha_{\mathsf{time}} = \lambda x.\mathsf{case}\, x\, \mathsf{of}\, x.\Omega \parallel x.\mathsf{ret}((x))$$

The memory cell represents a counter that decreases each time an observation is made, or make the configuration diverge if the counter was already $\underline{0}$. The following theorem states that

observing the termination of annotated terms amounts to observe *bounded-time termination.*

Theorem 64. *Let $t \in \mathbb{P}_0$ be a level 0 computation. The following propositions are equivalent:*

1. *$\langle t, \mathsf{nil}\rangle_0$ terminates in less than n λ reduction steps.*
2. *$\langle \{t\}_1^{\alpha_{\mathsf{time}}}, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{nil}\rangle_1$ terminates.*

Proof. The reduction steps in the configuration at level 1 are the same as those of the original configuration at level 0, except that before each λ reduction step, the counter is decreased by one if it greater that $\underline{1}$ and diverge otherwise. Hence, the configuration $\langle \{t\}_1^{\alpha_{\mathsf{time}}}, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{nil}\rangle_1$ diverges if and only if strictly more than n λ reduction steps are performed during the execution of $\langle t, \mathsf{nil}\rangle_0$. □

Safety

The second example uses the monitor dual to α_{time}:

$$\alpha_{\mathsf{step}} = \lambda x.\mathsf{case}\, x\, \mathsf{of}\, x.✠ \parallel x.\mathsf{ret}((x))$$

When using this monitor, the memory cell intuitively represents a counter that decreases each time an observation is made, or make the configuration converge (by triggering the daimon ✠) if the counter was already $\underline{0}$. The following theorem states that observing the termination of annotated terms amounts to observe *safety.*

Theorem 65. *Let $t \in \mathbb{P}_0$ be a level 0 computation. The following propositions are equivalent:*

1. *$\langle t, \mathsf{nil}\rangle_0$ diverges or terminates on a value.*
2. *For every $n \in \mathbb{N}$, $\langle \{t\}_1^{\alpha_{\mathsf{step}}}, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{nil}\rangle_1$ terminates on a value or on ✠.*

Proof. The reduction steps in the configuration at level 1 are the same as those of the original configuration at level 0, except that before each λ reduction step, the counter is decreased by one if it greater that $\underline{1}$ and converges on ✠ otherwise. Hence, the configuration

$$C_n = \langle \{t\}_1^{\alpha_{\mathsf{step}}}, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{nil}\rangle_1$$

converges on a value or on ✠ if and only if the original configuration $\langle t, \mathsf{nil}\rangle_0$ uses at least $n + 1$ λ reduction steps (and in that case ✠ is triggered) or reduces on a value in less than n λ steps. Therefore C_n converges for all $n \in \mathbb{N}$ if and only if $\langle t, \mathsf{nil}\rangle_0$ terminates on a value or diverges. □

2.4 | Program transformation

We are interested in giving a typed program transformation of the monitoring calculus $\lambda^{\mathbf{1}}_{\mathsf{Mon}}$ into the base calculus λ_{LCBPV}. In this section, we define the syntactic, untyped part of this

transformation, while the more complicated type part is left for Chapter III . This program transformation is shown to preserve the dynamic semantics of the monitoring calculus. For the most part, it is an adaptation to the call-by-push-value of the usual store-passing-style program transformation, with a straightforward variation concerning the observation constructor. We could have defined a more general program transformation of $\lambda^{\mathbf{n+1}}_{\mathsf{Mon}}$ into $\lambda^{\mathbf{n}}_{\mathsf{Mon}}$, as it is essentially the *same* transformation.

Definition 66 (Program transformation)**.** *We define a translation function $\langle\!\langle . \rangle\!\rangle\kappa$ that maps $\lambda^{\mathbf{1}}_{\mathsf{Mon}}$ values to λ_{LCBPV} values, and $\lambda^{\mathbf{1}}_{\mathsf{Mon}}$ computations to λ_{LCBPV} computations. Let κ be a fresh variable. This variable appears free in every translated term $\langle\!\langle t \rangle\!\rangle\kappa$, whereas it only appears bound in $\langle\!\langle v \rangle\!\rangle$. These two maps are defined by mutual recursion.*

- *We first give the definition $\langle\!\langle . \rangle\!\rangle$ on* values *as follows:*

$$\begin{array}{rcl}
\langle\!\langle x \rangle\!\rangle & = & x \\
\langle\!\langle * \rangle\!\rangle & = & * \\
\langle\!\langle \underline{0} \rangle\!\rangle & = & \underline{0} \\
\langle\!\langle \underline{\mathsf{s}}(v) \rangle\!\rangle & = & \underline{\mathsf{s}}(\langle\!\langle v \rangle\!\rangle) \\
\langle\!\langle (v, w) \rangle\!\rangle & = & (\langle\!\langle v \rangle\!\rangle, \langle\!\langle w \rangle\!\rangle) \\
\langle\!\langle \mathsf{thunk}(t) \rangle\!\rangle & = & \mathsf{thunk}(\langle\!\langle t \rangle\!\rangle)
\end{array}$$

- *We finally unveil the definition of the $\langle\!\langle . \rangle\!\rangle$ map on computations:*

$$\begin{array}{rcl}
\langle\!\langle \mathsf{ret}(v) \rangle\!\rangle & = & \lambda\kappa.\mathsf{ret}((\langle\!\langle v \rangle\!\rangle, \kappa)) \\
\langle\!\langle \mathsf{force}(v) \rangle\!\rangle & = & \mathsf{force}(\langle\!\langle v \rangle\!\rangle) \\
\langle\!\langle \lambda x.t \rangle\!\rangle & = & \lambda\kappa.\lambda x.(\langle\!\langle t \rangle\!\rangle\kappa) \\
\langle\!\langle (t)v \rangle\!\rangle & = & \lambda\kappa.(\langle\!\langle t \rangle\!\rangle\kappa)\langle\!\langle v \rangle\!\rangle \\
\langle\!\langle t \text{ to } x.u \rangle\!\rangle & = & \lambda\kappa.\langle\!\langle t \rangle\!\rangle\kappa \text{ to } z.(\mathsf{let}\,(x, \kappa') = z \,\mathsf{in}\, \langle\!\langle u \rangle\!\rangle\kappa') \\
\langle\!\langle \mathsf{let}\,(x, y) = v \,\mathsf{in}\, t \rangle\!\rangle & = & \mathsf{let}\,(x, y) = \langle\!\langle v \rangle\!\rangle \,\mathsf{in}\, \langle\!\langle t \rangle\!\rangle \\
\langle\!\langle \mathsf{let} * = v \,\mathsf{in}\, t \rangle\!\rangle & = & \mathsf{let} * = \langle\!\langle v \rangle\!\rangle \,\mathsf{in}\, \langle\!\langle t \rangle\!\rangle \\
\langle\!\langle \mathsf{case}\, v \,\mathsf{of}\, x.t \parallel x.u \rangle\!\rangle & = & \mathsf{case}\, \langle\!\langle v \rangle\!\rangle \,\mathsf{of}\, x.\langle\!\langle t \rangle\!\rangle\kappa \parallel x.\langle\!\langle u \rangle\!\rangle \\
\langle\!\langle (\!| t |\!)^{\alpha}_{1} \rangle\!\rangle & = & \lambda\kappa.(\alpha)\kappa \text{ to } \kappa.\langle\!\langle t \rangle\!\rangle\kappa \\
\langle\!\langle \maltese \rangle\!\rangle & = & (\maltese)
\end{array}$$

The program transformation satisfies a substitution lemma.

Lemma 67 (Substitution lemma)**.** *Let $t \in \mathbb{P}$, $v, w \in \mathbb{V}$ and x a variable that does not appear bound in v or t. Then both propositions hold:*

$$\begin{array}{rcll}
\langle\!\langle v[w/x] \rangle\!\rangle & = & \langle\!\langle v \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x] & (1) \\
\langle\!\langle t[w/x] \rangle\!\rangle & = & \langle\!\langle t \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x] & (2)
\end{array}$$

Proof. These two propositions are proved by mutual induction on the v and t. Let's begin with the proof of (1) by inspecting all the cases for v.

- **Variable ::**

  - If v is the variable x, then $\langle\!\langle x[w/x] \rangle\!\rangle = \langle\!\langle w \rangle\!\rangle = \langle\!\langle x \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x]$.
  - If v is a variable $y \neq x$, then $\langle\!\langle y[w/x] \rangle\!\rangle = \langle\!\langle y \rangle\!\rangle = y = y[\langle\!\langle w \rangle\!\rangle/x] = \langle\!\langle y \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x]$.

- **Unit ::** It is immediate since $*$ has no free variable.
- **Zero ::** It is immediate since $\underline{0}$ has no free variable.
- **Successor ::** Suppose that we know by induction that (1) is true for v. Then

$$\langle\!\langle \underline{\mathsf{s}}(v)[w/x] \rangle\!\rangle = \langle\!\langle \underline{\mathsf{s}}(v[w/x]) \rangle\!\rangle = \underline{\mathsf{s}}(\langle\!\langle v[w/x] \rangle\!\rangle) = \underline{\mathsf{s}}(\langle\!\langle v \rangle\!\rangle[w/x]) = \langle\!\langle \underline{\mathsf{s}}(v) \rangle\!\rangle[w/x]$$

- **Pair ::** Suppose that we know by induction (IH) that

$$\begin{array}{rcl} \langle\!\langle v_1[w/x] \rangle\!\rangle & = & \langle\!\langle v_1 \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x] \\ \langle\!\langle v_2[w/x] \rangle\!\rangle & = & \langle\!\langle v_2 \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x] \end{array}$$

  Consider the pair (v_1, v_2). Then

$$\begin{array}{rcll} \langle\!\langle (v_1, v_2)[w/x] \rangle\!\rangle & = & (\langle\!\langle v_1[w/x] \rangle\!\rangle, \langle\!\langle v_2[w/x] \rangle\!\rangle) & \\ & = & (\langle\!\langle v_1 \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x], \langle\!\langle v_2 \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x]) & (IH) \\ & = & (\langle\!\langle v_1 \rangle\!\rangle, \langle\!\langle v_2 \rangle\!\rangle)[\langle\!\langle w \rangle\!\rangle/x] & \\ & = & \langle\!\langle (v_1, v_2) \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x] & \end{array}$$

- **Thunk ::** Suppose that (2) holds for t. Then similarly to the successor case

$$\begin{array}{rcll} \langle\!\langle \mathsf{thunk}(t)[w/x] \rangle\!\rangle & = & \langle\!\langle \mathsf{thunk}(t[w/x]) \rangle\!\rangle & \\ & = & \mathsf{thunk}(\langle\!\langle t[w/x] \rangle\!\rangle) & \\ & = & \mathsf{thunk}(\langle\!\langle t \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x]) & (2) \\ & = & \langle\!\langle \mathsf{thunk}(t) \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x] & \end{array}$$

Let's now consider all the cases for t and prove (2).

- **Return ::** Suppose that (1) holds for v (IH). We have:

$$\begin{array}{rcll} \langle\!\langle \mathsf{ret}(v)[w/x] \rangle\!\rangle & = & \langle\!\langle \mathsf{ret}(v[w/x]) \rangle\!\rangle & \\ & = & \lambda\kappa.\mathsf{ret}((\langle\!\langle v[w/x] \rangle\!\rangle, \kappa)) & \\ & = & \lambda\kappa.\mathsf{ret}((\langle\!\langle v \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x], \kappa)) & (\mathsf{IH}) \\ & = & (\lambda\kappa.\mathsf{ret}((\langle\!\langle v \rangle\!\rangle, \kappa)))[\langle\!\langle w \rangle\!\rangle/x] & \\ & = & \langle\!\langle \mathsf{ret}(v) \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x] & \end{array}$$

- **Force ::** This case is similar to the previous case.
- **Lambda ::** Suppose that (2) holds for t (IH). We have:

$$\begin{array}{rcll} \langle\!\langle (\lambda y.t)[w/x] \rangle\!\rangle & = & \langle\!\langle \lambda y.(t[w/x]) \rangle\!\rangle & \\ & = & \lambda\kappa.\lambda y.\langle\!\langle (t[w/x]) \rangle\!\rangle\kappa & \\ & = & \lambda\kappa.\lambda y.(\langle\!\langle t \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x])\kappa & (\mathsf{IH}) \\ & = & (\lambda\kappa.\lambda y.\langle\!\langle t \rangle\!\rangle\kappa)[\langle\!\langle w \rangle\!\rangle/x] & \text{by } \alpha\text{-conversion} \\ & = & \langle\!\langle \lambda y.t \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x] & \end{array}$$

- **Application ::** Suppose that (1) holds for v (IH1) and that (2) holds for t (IH2). Then:

$$\begin{array}{rcll} \langle\!\langle ((t)v)[w/x] \rangle\!\rangle & = & \langle\!\langle (t[w/x])v[w/x] \rangle\!\rangle & \\ & = & \lambda\kappa.(\langle\!\langle t[w/x] \rangle\!\rangle\kappa)\langle\!\langle v[w/x] \rangle\!\rangle & \\ & = & \lambda\kappa.(\langle\!\langle t \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x]\kappa)\langle\!\langle v \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x] & (\mathsf{IH1}), (\mathsf{IH2}) \\ & = & (\lambda\kappa.(\langle\!\langle t \rangle\!\rangle\kappa)\langle\!\langle v \rangle\!\rangle)[\langle\!\langle w \rangle\!\rangle/x] & \text{by } \alpha\text{-conversion} \\ & = & \langle\!\langle (t)v \rangle\!\rangle[\langle\!\langle w \rangle\!\rangle/x] & \end{array}$$

- **To ::** Suppose that (2) holds for t (IH1) and for u (IH2). Then:

$$\begin{array}{rcll}
\langle\!\langle t \text{ to } y.u[w/x]\rangle\!\rangle & = & \langle\!\langle t[w/x] \text{ to } y.u[w/x]\rangle\!\rangle & \\
& = & \lambda\kappa.\langle\!\langle t[w/x]\rangle\!\rangle\kappa \text{ to } z.(\mathsf{let}\,(y,\kappa') = z \,\mathsf{in}\, \langle\!\langle u[w/x]\rangle\!\rangle\kappa') & \\
& = & \lambda\kappa.\langle\!\langle t\rangle\!\rangle[\langle\!\langle w\rangle\!\rangle/x]\kappa \text{ to } z.(\mathsf{let}\,(y,\kappa') = z \,\mathsf{in}\, \langle\!\langle u\rangle\!\rangle[\langle\!\langle w\rangle\!\rangle/x]\kappa') & (\mathsf{IH1}),(\mathsf{IH2}) \\
& = & (\lambda\kappa.\langle\!\langle t\rangle\!\rangle\kappa \text{ to } z.(\mathsf{let}\,(y,\kappa') = z \,\mathsf{in}\, \langle\!\langle u\rangle\!\rangle\kappa'))[\langle\!\langle w\rangle\!\rangle/x] & \\
& = & \langle\!\langle t \text{ to } y.u\rangle\!\rangle[\langle\!\langle w\rangle\!\rangle/x] &
\end{array}$$

- **Pair elim. ::** Suppose that (1) holds for v (IH1) and (2) holds for t (IH2). Then:

$$\begin{array}{rcll}
\langle\!\langle(\mathsf{let}\,(z,y) = v \,\mathsf{in}\, t)[w/x]\rangle\!\rangle & = & \langle\!\langle \mathsf{let}\,(z,y) = v[w/x] \,\mathsf{in}\, t[w/x]\rangle\!\rangle & \\
& = & \mathsf{let}\,(z,y) = \langle\!\langle v[w/x]\rangle\!\rangle \,\mathsf{in}\, \langle\!\langle t[w/x]\rangle\!\rangle & \\
& = & \mathsf{let}\,(z,y) = \langle\!\langle v\rangle\!\rangle[\langle\!\langle w\rangle\!\rangle/x] \,\mathsf{in}\, \langle\!\langle t\rangle\!\rangle[\langle\!\langle w\rangle\!\rangle/x] & (\mathsf{IH1}),(\mathsf{IH2}) \\
& = & (\mathsf{let}\,(z,y) = \langle\!\langle v\rangle\!\rangle \,\mathsf{in}\, \langle\!\langle t\rangle\!\rangle)[\langle\!\langle w\rangle\!\rangle/x] & \alpha\text{-conversion} \\
& = & \langle\!\langle \mathsf{let}\,(z,y) = v \,\mathsf{in}\, t\rangle\!\rangle[\langle\!\langle w\rangle\!\rangle/x] &
\end{array}$$

- **Unit elim. ::** Suppose that (1) holds for v (IH1) and (2) holds for t (IH2). Then:

$$\begin{array}{rcll}
\langle\!\langle(\mathsf{let} * = v \,\mathsf{in}\, t)[w/x]\rangle\!\rangle & = & \langle\!\langle \mathsf{let} * = v[w/x] \,\mathsf{in}\, t[w/x]\rangle\!\rangle & \\
& = & \mathsf{let} * = \langle\!\langle v[w/x]\rangle\!\rangle \,\mathsf{in}\, \langle\!\langle t[w/x]\rangle\!\rangle & \\
& = & \mathsf{let} * = \langle\!\langle v\rangle\!\rangle[\langle\!\langle w\rangle\!\rangle/x] \,\mathsf{in}\, \langle\!\langle t\rangle\!\rangle[\langle\!\langle w\rangle\!\rangle/x] & (\mathsf{IH1}),(\mathsf{IH2}) \\
& = & (\mathsf{let} * = \langle\!\langle v\rangle\!\rangle \,\mathsf{in}\, \langle\!\langle t\rangle\!\rangle)[\langle\!\langle w\rangle\!\rangle/x] & \\
& = & \langle\!\langle \mathsf{let} * = v \,\mathsf{in}\, t\rangle\!\rangle[\langle\!\langle w\rangle\!\rangle/x] &
\end{array}$$

- **Case ::** this case is similar to the previous one.
- **Observation ::** Suppose that (2) is true for t (IH). Then:

$$\begin{array}{rcll}
\langle\!\langle(\!|t|\!)_1^\alpha[w/x]\rangle\!\rangle & = & \langle\!\langle(\!|t[w/x]|\!)_1^\alpha\rangle\!\rangle & \\
& = & \lambda\kappa.(\alpha)\kappa \text{ to } \kappa'.(\langle\!\langle t[w/x]\rangle\!\rangle\kappa') & \\
& = & \lambda\kappa.(\alpha)\kappa \text{ to } \kappa'.(\langle\!\langle t\rangle\!\rangle[\langle\!\langle w\rangle\!\rangle/x]\kappa') & (\mathsf{IH}) \\
& = & \langle\!\langle(\!|t|\!)_1^\alpha\rangle\!\rangle[\langle\!\langle w\rangle\!\rangle/x] &
\end{array}$$

- **Daimon ::** This case is easy since

$$\langle\!\langle \maltese[v/x]\rangle\!\rangle = \langle\!\langle \maltese\rangle\!\rangle = \maltese = \maltese[\langle\!\langle v\rangle\!\rangle/x]$$

□

Since we only have defined the execution of a program through the definition of the abstract machine, we need to extend the translation to environments and configurations in order to state the simulation theorem.

Definition 68 (Environments and configurations transformation)**.** *Since there are two modes of execution, we need two different maps, the first $\langle\!\langle . \rangle\!\rangle$ is used to translate environments used during executions in the normal mode.*

$$\begin{array}{rcl}
\langle\!\langle \mathsf{nil}\rangle\!\rangle & = & \mathsf{nil} \\
\langle\!\langle \mathsf{a}(v).E\rangle\!\rangle & = & \mathsf{a}(\langle\!\langle v\rangle\!\rangle).\langle\!\langle E\rangle\!\rangle \\
\langle\!\langle \mathsf{f}(x.t).E\rangle\!\rangle & = & \mathsf{f}(z.(\mathsf{let}\,(x,\kappa) = z \,\mathsf{in}\, \langle\!\langle t\rangle\!\rangle\kappa)).\langle\!\langle E\rangle\!\rangle
\end{array}$$

The second map $[\![.]\!]^\bullet$ corresponds to the monitoring mode.

$$\begin{array}{rcl} [\![\mathsf{nil}]\!] & = & \mathsf{nil} \\ [\![\mathsf{a}(v).E]\!] & = & \mathsf{a}(v).[\![E]\!] \\ [\![\mathsf{f}(x.t).E]\!] & = & \mathsf{f}(x.t).[\![E]\!] \\ [\![\mathsf{m}_1(t).E]\!] & = & \mathsf{f}(\kappa.\langle\!\langle t \rangle\!\rangle \kappa).\langle\!\langle E \rangle\!\rangle \end{array}$$

We finally translate the different configurations.

$$\begin{array}{rcl} \langle\!\langle \langle t, \mathsf{a}(v).E\rangle_1 \rangle\!\rangle & = & \langle \langle\!\langle t \rangle\!\rangle, \mathsf{a}(v).\langle\!\langle E \rangle\!\rangle\rangle_0 \\ \langle\!\langle \langle \uparrow E, (v, w)\rangle_1 \rangle\!\rangle & = & \langle \uparrow \langle\!\langle E \rangle\!\rangle, (\langle\!\langle v \rangle\!\rangle, w)\rangle_0 \\ \langle\!\langle \langle t, E\rangle_0 \rangle\!\rangle & = & \langle t, [\![E]\!]\rangle_0 \\ \langle\!\langle \langle \uparrow E, v\rangle_0 \rangle\!\rangle & = & \langle \uparrow [\![E]\!], v\rangle_0 \end{array}$$

The following simulation result holds for the program transformation. It basically says that the transformation preserves the reduction of $\lambda^{\mathbf{1}}_{\mathsf{Mon}}$.

Theorem 69 (Simulation). *The following propositions hold:*

$$C \xrightarrow{1} C' \text{ implies } \langle\!\langle C \rangle\!\rangle \, (\xrightarrow{0})^* \, \langle\!\langle C' \rangle\!\rangle$$

$$C \cong_1 \overline{C} \text{ implies } \langle\!\langle C \rangle\!\rangle \xrightarrow{0} C' \cong_0 \langle\!\langle \overline{C} \rangle\!\rangle$$

Proof. We have three different statements to prove. One for the reduction steps at level 1, one for the reduction steps at level 0 and finally for the equivalence.

1. Let $C = \langle t', \mathsf{a}(u).E\rangle_1$. We proceed by enumerating all the possible reduction steps:

   - **Application ::** In the case of $t' = (t)v$, then

$$\begin{array}{rcll} \langle\!\langle \langle (t)v, \mathsf{a}(u).E\rangle_1 \rangle\!\rangle & = & \langle \lambda\kappa.((\langle\!\langle t \rangle\!\rangle)\kappa)\langle\!\langle v \rangle\!\rangle, \mathsf{a}(u).\langle\!\langle E \rangle\!\rangle\rangle_0 & \\ & \xrightarrow{0} & \langle (\langle\!\langle t \rangle\!\rangle u)\langle\!\langle v \rangle\!\rangle, E\rangle_0 & \\ & \xrightarrow{0}{}^{2} & \langle \langle\!\langle t \rangle\!\rangle, \mathsf{a}(u).\mathsf{a}(\langle\!\langle v \rangle\!\rangle).\langle\!\langle E \rangle\!\rangle\rangle_0 & \\ & = & \langle \langle\!\langle t \rangle\!\rangle, \mathsf{a}(u).\langle\!\langle \mathsf{a}(v).E \rangle\!\rangle\rangle_0 & \\ & = & \langle\!\langle \langle t, \mathsf{a}(u).\mathsf{a}(v).E\rangle_1 \rangle\!\rangle & \end{array}$$

   - **Lambda ::** In the case of $t' = \lambda x.t$, then

$$\begin{array}{rcll} \langle\!\langle \langle \lambda x.t, \mathsf{a}(u).\mathsf{a}(v).E\rangle_1 \rangle\!\rangle & = & \langle \lambda\kappa.\lambda x.\langle\!\langle t \rangle\!\rangle \kappa, \mathsf{a}(u).\mathsf{a}(\langle\!\langle v \rangle\!\rangle).\langle\!\langle E \rangle\!\rangle\rangle_0 & \\ & \xrightarrow{0}{}^{2} & \langle \langle\!\langle t \rangle\!\rangle [\langle\!\langle v \rangle\!\rangle / x]u, \langle\!\langle E \rangle\!\rangle\rangle_0 & u \text{ is closed} \\ & \xrightarrow{0} & \langle \langle\!\langle t \rangle\!\rangle [\langle\!\langle v \rangle\!\rangle / x], \mathsf{a}(u).\langle\!\langle E \rangle\!\rangle\rangle_0 & \\ & = & \langle \langle\!\langle t[v/x] \rangle\!\rangle, \mathsf{a}(u).\langle\!\langle E \rangle\!\rangle\rangle_0 & \text{Lemma 67} \\ & = & \langle\!\langle \langle t[v/x], \mathsf{a}(u).E\rangle_1 \rangle\!\rangle & \end{array}$$

   - **Force ::** In the case of $t' = \mathsf{force}(\mathsf{thunk}(t))$, then

$$\begin{array}{rcl} \langle\!\langle \langle \mathsf{force}(\mathsf{thunk}(t)), \mathsf{a}(u).E\rangle_1 \rangle\!\rangle & = & \langle \mathsf{force}(\mathsf{thunk}(\langle\!\langle t \rangle\!\rangle)), \mathsf{a}(u).\langle\!\langle E \rangle\!\rangle\rangle_0 \\ & \xrightarrow{0}{}^{2} & \langle \langle\!\langle t \rangle\!\rangle, \mathsf{a}(u).\langle\!\langle E \rangle\!\rangle\rangle_0 \\ & = & \langle\!\langle \langle t, \mathsf{a}(u).E\rangle_1 \rangle\!\rangle \end{array}$$

- **To ::** Suppose $t' = t$ to $x.u$, then

$$\begin{array}{lcl} \langle\!\langle \langle t \text{ to } x.u, \mathsf{a}(u).E\rangle_1 \rangle\!\rangle & = & \langle \lambda\kappa.\langle\!\langle t\rangle\!\rangle\kappa \text{ to } z.(\mathsf{let}\,(x,\kappa') = z \,\mathsf{in}\, \langle\!\langle u\rangle\!\rangle\kappa'), \mathsf{a}(u).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & \overset{0}{\to}^{*} & \langle \langle\!\langle t\rangle\!\rangle, \mathsf{a}(u).\mathsf{f}(z.(\mathsf{let}\,(x,\kappa') = z \,\mathsf{in}\, \langle\!\langle u\rangle\!\rangle\kappa')).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & = & \langle\!\langle \langle t, \mathsf{a}(u).\mathsf{f}(x.u).E\rangle_1 \rangle\!\rangle \end{array}$$

- **Return ::** Suppose $t' = \mathsf{ret}(v)$, then

$$\begin{array}{lcl} \langle\!\langle \langle \mathsf{ret}(v), \mathsf{a}(u).\mathsf{f}(x.t).E\rangle_1 \rangle\!\rangle & = & \langle \lambda\kappa.\mathsf{ret}((\langle\!\langle v\rangle\!\rangle, \kappa)), \mathsf{a}(u).\mathsf{f}(z.\mathsf{let}\,(x,\kappa) = z \,\mathsf{in}\, \langle\!\langle t\rangle\!\rangle\kappa).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & \overset{0}{\to} & \langle \mathsf{ret}((\langle\!\langle v\rangle\!\rangle, u)), \mathsf{f}(z.\mathsf{let}\,(x,\kappa) = z \,\mathsf{in}\, \langle\!\langle t\rangle\!\rangle\kappa).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & \overset{0}{\to} & \langle \mathsf{let}\,(x,\kappa) = (\langle\!\langle v\rangle\!\rangle, u) \,\mathsf{in}\, \langle\!\langle t\rangle\!\rangle\kappa, \langle\!\langle E\rangle\!\rangle\rangle_0 \\ & \overset{0}{\to}^{2} & \langle \langle\!\langle t\rangle\!\rangle[\langle\!\langle v\rangle\!\rangle/x], \mathsf{a}(u).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & = & \langle \langle\!\langle t[v/x]\rangle\!\rangle, \mathsf{a}(u).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & = & \langle\!\langle \langle t[v/x], \mathsf{a}(u).E\rangle_1 \rangle\!\rangle \end{array}$$

- **Unit ::** Suppose $t' = \mathsf{let} * = * \,\mathsf{in}\, t$, then

$$\begin{array}{lcl} \langle\!\langle \langle \mathsf{let} * = * \,\mathsf{in}\, t, \mathsf{a}(u).E\rangle_1 \rangle\!\rangle & = & \langle \mathsf{let} * = * \,\mathsf{in}\, \langle\!\langle t\rangle\!\rangle, \mathsf{a}(u).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & \overset{0}{\to} & \langle \langle\!\langle t\rangle\!\rangle, \mathsf{a}(u).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & = & \langle\!\langle \langle t, \mathsf{a}(u).E\rangle_1 \rangle\!\rangle \end{array}$$

- **Pair ::** Suppose $t' = \mathsf{let}\,(x,y) = (v,w) \,\mathsf{in}\, t$, then

$$\begin{array}{lcl} \langle\!\langle \langle \mathsf{let}\,(x,y) = (v,w) \,\mathsf{in}\, t, \mathsf{a}(u).E\rangle_1 \rangle\!\rangle & = & \langle \mathsf{let}\,(x,y) = (\langle\!\langle v\rangle\!\rangle, \langle\!\langle w\rangle\!\rangle) \,\mathsf{in}\, \langle\!\langle t\rangle\!\rangle, \mathsf{a}(u).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & \overset{0}{\to} & \langle \langle\!\langle t\rangle\!\rangle[\langle\!\langle v\rangle\!\rangle/x, \langle\!\langle w\rangle\!\rangle/y], \mathsf{a}(u).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & = & \langle \langle\!\langle t[v/x, w/y]\rangle\!\rangle, \mathsf{a}(u).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & = & \langle\!\langle \langle t[v/x, w/y], \mathsf{a}(u).E\rangle_1 \rangle\!\rangle \end{array}$$

- **Case ::** Suppose $t' = \mathsf{case}\,\underline{0}\,\mathsf{of}\, x.t \parallel x.u$, then

$$\begin{array}{lcl} \langle\!\langle \langle \mathsf{case}\,\underline{0}\,\mathsf{of}\, x.t \parallel x.u, \mathsf{a}(u).E\rangle_1 \rangle\!\rangle & = & \langle \mathsf{case}\,\underline{0}\,\mathsf{of}\, x.\langle\!\langle t\rangle\!\rangle \parallel x.\langle\!\langle u\rangle\!\rangle, \mathsf{a}(u).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & \overset{0}{\to} & \langle \langle\!\langle t\rangle\!\rangle[\underline{0}/x], \mathsf{a}(u).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & = & \langle \langle\!\langle t[\underline{0}/x]\rangle\!\rangle, \mathsf{a}(u).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & = & \langle\!\langle \langle t[\underline{0}/x], u\rangle_0 E\rangle\!\rangle \end{array}$$

- **Successor ::** Suppose $t' = \mathsf{case}\,\underline{\mathsf{s}}(v)\,\mathsf{of}\, x.t \parallel x.u$, then

$$\begin{array}{lcl} \langle\!\langle \langle \mathsf{case}\,\underline{\mathsf{s}}(v)\,\mathsf{of}\, x.t \parallel x.u, \mathsf{a}(u).E\rangle_1 \rangle\!\rangle & = & \langle \mathsf{case}\,\underline{\mathsf{s}}(\langle\!\langle v\rangle\!\rangle)\,\mathsf{of}\, x.\langle\!\langle t\rangle\!\rangle \parallel x.\langle\!\langle u\rangle\!\rangle, \mathsf{a}(u).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & \overset{0}{\to} & \langle \langle\!\langle u\rangle\!\rangle[\langle\!\langle v\rangle\!\rangle/x], \mathsf{a}(u).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & = & \langle \langle\!\langle t[v/x]\rangle\!\rangle, \mathsf{a}(u).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & = & \langle\!\langle \langle t[v/x], u\rangle_0 E\rangle\!\rangle \end{array}$$

- **Observation ::** Suppose $t' = (\!|t|\!)^{\alpha}_1$, then

$$\begin{array}{lcl} \langle\!\langle \langle (\!|t|\!)^{\alpha}_1, \mathsf{a}(u).E\rangle_1 \rangle\!\rangle & = & \langle \lambda\kappa.(\alpha)\kappa \text{ to } \kappa'.\langle\!\langle t\rangle\!\rangle\kappa', \mathsf{a}(u).\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & \overset{0}{\to}^{2} & \langle (\alpha)u, \mathsf{f}(\kappa'.\langle\!\langle t\rangle\!\rangle\kappa').\langle\!\langle E\rangle\!\rangle\rangle_0 \\ & = & \langle\!\langle \langle (\alpha)u, \mathsf{m}_1(t).E\rangle_0 \rangle\!\rangle \end{array}$$

- **Daimon ::** Suppose $t' = \maltese$, then

$$\begin{array}{rcl} \langle\!\langle \langle \maltese, \mathsf{a}(u).E \rangle_1 \rangle\!\rangle & = & \langle \maltese, \mathsf{a}(u).\langle\!\langle E \rangle\!\rangle \rangle_0 \\ & \xrightarrow{0} & \maltese \\ & = & \langle\!\langle \maltese \rangle\!\rangle \end{array}$$

2. We now consider a configuration at level 0. All cases are straightforward but the following one:

$$\langle \mathsf{ret}(v), \mathsf{m}_1(t).E \rangle_0$$

$$\begin{array}{rcl} \langle\!\langle \langle \mathsf{ret}(v), \mathsf{m}_1(t).E \rangle_0 \rangle\!\rangle & = & \langle \mathsf{ret}(v), \mathsf{f}(\kappa.\langle\!\langle t \rangle\!\rangle \kappa).\langle\!\langle E \rangle\!\rangle \rangle_0 \\ & \xrightarrow{0} & \langle \langle\!\langle t \rangle\!\rangle v, \langle\!\langle E \rangle\!\rangle \rangle_0 \\ & \xrightarrow{0} & \langle \langle\!\langle t \rangle\!\rangle, \mathsf{a}(v).\langle\!\langle E \rangle\!\rangle \rangle_0 \\ & = & \langle\!\langle \langle t, \mathsf{a}(v).E \rangle_1 \rangle\!\rangle \end{array}$$

3. Let $C = \langle \mathsf{ret}(v), \mathsf{a}(w).E \rangle_1$. We have $C \cong_1 \langle \uparrow E, (v, w) \rangle_1$. But

$$\begin{array}{rcl} \langle\!\langle C \rangle\!\rangle & = & \langle \lambda\kappa.\mathsf{ret}((\langle\!\langle v \rangle\!\rangle, \kappa)), \mathsf{a}(w).\langle\!\langle E \rangle\!\rangle \rangle_0 \\ & \xrightarrow{0} & \langle \mathsf{ret}((\langle\!\langle v \rangle\!\rangle, w)), \langle\!\langle E \rangle\!\rangle \rangle_0 \\ & \cong_0 & \langle \uparrow \langle\!\langle E \rangle\!\rangle, (\langle\!\langle v \rangle\!\rangle, w) \rangle_0 \\ & = & \langle\!\langle \langle \uparrow E, (v, w) \rangle_1 \rangle\!\rangle \end{array}$$

□

Chapter III

Forcing monoids

Contents

Contributions

In Section 3.1, we define the basic theory of **forcing monoids**: we give the basic definitions, examples and properties as well as some useful constructions. Section 3.2 is devoted to the definition of an annotatd type system based on forcing monoids. Finally, Section 3.3 is devoted to a program transformation from the λ_{Mon} to LCBPV. We first define an untyped program transformation that is shown to preserve reduction. We then show how it is possible to type this program transformation using a particular internalization of forcing inside LCBPV.

3.1 | Forcing monoids

The core mathematical object of this thesis is the notion of **forcing monoid**. It is the main ingredient of the forcing program transformation. It generalizes the concept of *forcing poset* used in set theory. It can be seen as the basis of many works, including set theoretic forcing, Kripke semantics, quantitative realizability, phase spaces, among others. Its simplicity and generality will allow us to capture various existing and new techniques, that were previously seen as unrelated. In this section, we define what forcing monoids are, along with examples and basic definitions we will use throughout this thesis.

3.1.1 Definition

Definition 70 (Preordered commutative monoid)**.** *A **preordered commutative monoid** is a structure $(\mathcal{M}, +, \mathbf{0}, \leq)$ such that:*

- *$\mathcal{M}$ is a set*
- *$+ : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$ is a binary associative and commutative operation.*
- *$\mathbf{0} \in \mathcal{M}$ is neutral for $+$: $\forall p \in \mathcal{M}, p + \mathbf{0} = p$.*
- *$\leq$ is a preorder on $\mathcal{M}$ such that:*
  - *it is compatible with $+$: $\forall p, q, r \in \mathcal{M}, p \leq q \Rightarrow p + r \leq q + r$.*
  - *$\mathbf{0}$ is a minimum for $\leq$: $\forall p \in \mathcal{M}, \mathbf{0} \leq p$.*

Definition 71 (Left action)**.** *Let $(\mathcal{M}, +, \mathbf{0}, \leq)$ be a preordered commutative monoid. Let $(A, \leq)$ be a preordered set and $\delta : \mathcal{M} \times A \to A$. We say that δ is an **action of $\mathcal{M}$ on** A:*

- *$\forall m, n \in \mathcal{M}, \forall a \in A, \delta_{m+n}(a) = \delta_m(\delta_n(a))$*
- *$m \leq n$ implies that $\delta_m(a) \leq \delta_n(b)$*
- *$a \leq b$ implies that $\delta_m(a) \leq \delta_m(b)$*

We will alternatively use an infix notation. If $\star$ is a left action of $\mathcal{M}$ on A we use the notation $m \star a$ instead of $\star(m, a)$. For example the second axiom is rewritten as:

$$(m + n) \star a = m \star (n \star a)$$

In that case, we just write $m \star n \star a$ to denote $m \star (n \star a)$.

Definition 72. *A **forcing monoid** is a structure $(\mathcal{M}, +, \mathbf{0}, \bullet, \leq)$ where:*

- *$(\mathcal{M}, +, \mathbf{0}, \leq)$ is a preordered commutative monoid on $\mathcal{M}$.*
- *$\bullet$ is a left action of $(\mathcal{M}, \leq)$ on $(\mathcal{M}, +, \mathbf{0}, \leq)$.*

Notation 73. *If $\mathcal{M}$ is a forcing monoid, we will sometimes denote by $+_{\mathcal{M}}, \mathbf{0}_{\mathcal{M}}, \leq_{\mathcal{M}}$ and $\bullet_{\mathcal{M}}$ the different components of $\mathcal{M}$*

We often denote a forcing monoid by its underlying set $\mathcal{M}$. The elements of a forcing monoid will generally be denoted by letters $p, q, r, \ldots$. If $n \in \mathbb{N}$ and $p \in \mathcal{M}$, we use the notation $n.p$ to denote the element $\underbrace{p + p + \cdots + p}_{n \text{ times}}$.

Example 74. *Forcing posets are the core of the set theoretic forcing technique. A forcing poset is a triple* $(P, \leq, \mathbf{1}, \wedge)$ *which is a sup-semi lattice:*

- $\leq$ *is a preorder on* P*.*
- *For all* $p \in P$*,* $p \leq \mathbf{1}$*.*
- $\wedge$ *is a meet operation.*

They are in fact particular cases of forcing monoid. Indeed given a forcing poset $(P, \leq, \mathbf{1}, \wedge)$*, one can build the following forcing monoid on the set* P*:*

- $+ = \bullet = \wedge$
- *the neutral element is* $\mathbf{1}$
- $p \preceq q \Leftrightarrow q \leq p$

Definition 75 (Elementary properties)**.** *A forcing monoid* $\mathcal{M}$ *is said to be:*

- ***additive*** *iff* $+$ *and* $\bullet$ *coincide.*
- ***commutative*** *iff* $\bullet$ *is commutative.*
- ***idempotent*** *iff* $+$ *is idempotent.*

Remark 76. *An additive forcing monoid is just a preordered commutative monoid.*

Example 77. *Here are some examples of additive forcing monoids we will use throughout the next chapters.*

1. *The* ***trivial forcing monoid*** *on the the singleton* $\{0\}$*.*
2. *The monoid* $(\mathbb{N}, +, 0, \leq)$ *of natural numbers, endowed with the usual addition and order* $\leq$ *on natural numbers.*
3. *The monoid* $(\mathbb{N}, \max, 0, \leq)$ *of natural numbers, with* $\max$ *as the monoid operation, and the usual order* $\leq$ *on natural numbers.*
4. *The monoid* $(\overline{\mathbb{N}}, \min, \infty, \geq)$ *of extended natural numbers endowed with the operation* $\min$ *and the reverse order on natural numbers.*
5. *The boolean algebra* $(\mathbb{B}, \wedge, \top, \subseteq)$ *where* $\mathbb{B} = \{\bot, \top\}$*,* $\wedge$ *is the usual "and" operation, and*

$\subseteq$ *is the order such that* $\bot \subseteq \top$.

6. *The forcing monoid the vector space* $\mathbb{R}^n$. *Its elements are the vectors of* $\mathbb{R}^n$, *with the addition being the usual componentwise addition and the preorder being the componentwise preorder.*

7. *Separation logic related papers give us a lot of additive forcing structure examples. An example is the pointer model defined in [BBTS07]. They consider a set* H *of heaps, i.e. the set of finite partial functions from a set of locations* $\mathcal{L}$ *to a set of values* $\mathcal{V}$. *A forcing structure is then given by the pointer model* $(H_\bot, \star)$, *where* $H_\bot$ *is* H *augmented with a bottom element* $\bot$, *and* $\star$ *is the operation defined by:*

$$h_1 \star h_2 = \begin{cases} \bot & \text{if } h_1 = \bot \text{ or } h_2 = \bot \\ h_1 \cup h_2 & \text{if } h_1 \# h_2 \\ \bot & \text{otherwise} \end{cases}$$

The set is then ordered by the extension ordering:

$$h \preceq h' \Leftrightarrow \exists h'' \in H_\bot, h' = h'' \star h$$

3.1.2 Functions

An simple yet important notion is the *strength* of a function on $\mathcal{M}$.

Definition 78 (Strong function)**.** *If* $\mathcal{M}$ *is a forcing monoid, we say that a function* $f : \mathcal{M} \to \mathcal{M}$ *is* ***strong*** *if for all* $p, q \in \mathcal{M}$, *the following inequality holds:*

$$f(p \bullet q) \preceq f(p) \bullet q$$

Example 79. *The identity function is always strong.*

Property 80. *When* $\mathcal{M}$ *is additive, the following functions are always strong:*

- *The constant function* $x \mapsto p$ *for* $p \in \mathcal{M}$.
- *The adding function* $x \mapsto x + q$ *for any* $q \in \mathcal{M}$
- *For any* $n \in \mathbb{N}^*$, *the function* $x \mapsto n.x$.

Definition 81 (Sub-additive function)**.** *Let $\mathcal{M}$ and $\mathcal{N}$ be two forcing monoids. A function $\mathcal{M} \to \mathcal{N}$ is **sub-additive** iff*

$$f(p +_{\mathcal{M}} q) \preceq_{\mathcal{N}} f(p) +_{\mathcal{N}} f(q)$$

Example 82. *The following functions are sub-additive:*

1. *The identity function $x \mapsto x$.*
2. *The doubling function $x \mapsto x + x$.*
3. *Any scalar multiplication function $x \mapsto n.x$ with $n \in \mathbb{N}$.*

3.1.3 Algebraic constructions

Definition 83 (Direct product)**.** *Let $\mathcal{M}$ and $\mathcal{N}$ be two forcing monoids. We define their **direct product** $\mathcal{M} \times \mathcal{N}$ as follows:*

- *The underlying set is the cartesian product of the two sets $\mathcal{M} \times \mathcal{N}$.*
- *The addition $+$ is the componentwise addition:*

$$(m, p) + (n, q) = (m + n, p + q)$$

- *The additive identity element is*

$$\mathbf{0} = (\mathbf{0}, \mathbf{0})$$

- *The left action $\bullet$ is defined as:*

$$(m, p) \bullet (n, q) = (m \bullet n, p \bullet q)$$

- *The preorder is the product preorder:*

$$(m, p) \bullet (n, q) \Leftrightarrow m \preceq n \wedge p \preceq q$$

We now describe generalization of the direct product construction: the **semi-direct product**. It is inherited from the notion of semi-direct product of monoids. It is based on the same base set, that is the cartesian product, but it allows the right component to *act* on the first component by mean of the operation $\bullet$, unlike the direct product that clearly separate the two components. It relies on a more constrained notion of left action.

Definition 84 (Forcing monoid left action)**.** *Let $(\mathcal{M}, +, \mathbf{0}, \bullet, \leq)$ and $(\mathcal{N}, \ .\ , e, \star, \leq)$ be two forcing monoids. We say that δ is an action of $\mathcal{N}$ on $\mathcal{M}$ if:*

- *δ is an action of $(\mathcal{N}, \leq)$ on $(\mathcal{M}, +, \mathbf{0}, \leq)$.*
- *$\delta_{n\star m}(p) = \delta_m(p)$*
- *$\delta_n(p + q) = \delta_n(p) + \delta_m(q)$*

Remark 85. *The notion of forcing monoid left action is different from the notion of action of a preordered set on a forcing monoid.*

Example 86. *The identity action $\delta_n(p) = p$ is an example of forcing monoid left action.*

Definition 87 (Semi-direct product)**.** *Let $\mathcal{M}$ and $\mathcal{N}$ be two forcing monoids. Suppose that we have a left action $\delta : \mathcal{N} \times \mathcal{M} \to \mathcal{M}$ of $\mathcal{N}$ over $\mathcal{M}$. Then we can define the* ***semi-direct product*** *$\mathcal{M} \ltimes_\delta \mathcal{N}$ as follows:*

- *The underlying set is the cartesian product of the two sets $\mathcal{M} \times \mathcal{N}$.*
- *The addition $+$ is:*
$$(p, m) + (q, n) = (p + q, n + m)$$
- *The additive identity element is*
$$\mathbf{0} = (\mathbf{0}, \mathbf{0})$$
- *The left action $\bullet$ is defined as:*
$$(p, m) \bullet (q, n) = (\delta_n(p) \bullet q, n \bullet m)$$
- *The preorder is the product preorder:*
$$(p, m) \bullet (q, n) \Leftrightarrow m \leq n \wedge p \leq q$$

Property 88. *The semi-direct product is a forcing monoid.*

Proof.

- It is clear that $\mathcal{M} \times \mathcal{N}$ is a preordered commutative monoid.
- Let $(p, n), (q, m), (r, l) \in \mathcal{M} \times \mathcal{N}$. On the one hand, we have $((p, n) + (q, m)) \bullet (r, l) = (\delta_l(p+q) \bullet r, (n+m) \bullet l)$. On the other hand, we have $(p, n) \bullet (q, m) \bullet (r, l) = (\delta_{m \bullet l}(p) \bullet \delta_l(q) \bullet r, n \bullet m \bullet l)$. But we have $(n + m) \bullet l = n \bullet m \bullet l$. Hence we only need to show the equality of the first

component. We have $\delta_l(p+q) \bullet r = \delta_l(p) \bullet \delta_l(q) \bullet r$. But we also know that $\delta_{m \bullet l}(p) = \delta_l(p)$, hence the result.

- Suppose that $(p,n) \preceq (q,m)$, that is $p \preceq q$ and $n \preceq m$. If $(r,l) \in \mathcal{M} \times \mathcal{N}$, we want to show that $(p,n) \bullet (r,l) \preceq (q,m) \bullet (r,l)$. But $(p,n) \bullet (r,l) = (\delta_l(p) \bullet r, n \bullet l)$ and $(q,m) \bullet (r,l) = (\delta_l(q) \bullet r, m \bullet l)$. Since $n \preceq m$, we have $n \bullet l \preceq m \bullet l$. Moreover, $\delta_l(p) \preceq \delta_l(q)$ since $p \preceq q$, hence $\delta_l(p) \bullet r \preceq \delta_l(q) \bullet r$ and the result.
- Suppose that $(p,n) \preceq (q,m)$, that is $p \preceq q$ and $n \preceq m$. If $(r,l) \in \mathcal{M} \times \mathcal{N}$, we want to show that $(r,l) \bullet (p,n) \preceq (r,l) \bullet (q,m)$. But $(r,l) \bullet (p,n) = (\delta_n(r) \bullet p, l \bullet n)$ and $(r,l) \bullet (q,m) = (\delta_m(r) \bullet q, l \bullet m)$. Since $n \preceq m$, we have $l \bullet n \preceq l \bullet m$. Moreover, $\delta_m(r) \preceq \delta_n(r)$ since $m \preceq n$, hence $\delta_m(r) \bullet p \preceq \delta_n(r) \bullet p$. Finally since $p \preceq q$, we have $\delta_m(r) \bullet p \preceq \delta_n(r) \bullet p \preceq \delta_n(r) \bullet q$ and the result.

□

Remark 89. *The absorbing condition $\delta_{n \star m}(p) = \delta_m(p)$ is necessary because of the mixed associativity condition of forcing monoids, and because semi-direct product relies on a fundamental* non-commutativity. *It is very restrictive, but as we will see, the construction still has useful instances.*

3.2 | A forcing-based type system

We now introduce a typing system for a fragment of λ_{Mon}. More than an actual type system, it is an annotation of the type system of λ_{LCBPV} that makes use of the elements of a given forcing monoid $\mathcal{M}$. It also provides a typing rule for the constructor $(\!|t|\!)^{\alpha}_{n}$. We will use this annotated type system for two things:

- As a clean way of stating and proving the type preservation property of the forcing program transformation we will define in Section 3.3.
- As a way to extend the realizability relation induced by the structures we will define in Chapter IV to open terms, and to express the soundness of typing rules in the corresponding models.

The annotated type system is divided in two distinct parts:

- The first part contains the annotated versions of the λ_{LCBPV} typing rules, and is defined directly for every level $n \geq 1$, with annotations being elements of any forcing monoid $\mathcal{M}$.
- The second part consists in a single *new* typing rule, which is intented to type the observation $(\!|.|\!)^{\alpha}_{1}$. Operationnally, the observation makes a configuration at level 1 reduce to a configuration at level 0. Similarly, the typing rule at level 1 of the observation makes use of the typing rules at level 0, that is the typing relation $\vdash_0$.

Remark 90. *For now, we don't give any rule for the observation at level $n > 1$. This is possible, but requires to speak about forcing iteration. We postpone the introduction of such rules to Chapter V, where we study forcing iteration at the realizability level.*

The annotated typing relation is parametrized by what we call a **forcing structure**.

Definition 91 (Forcing structure)**.** *A **forcing structure** $\mathcal{F}$ is given by:*

- *A forcing monoid $(\mathcal{M}, +, \mathbf{0}, \bullet, \leq)$.*
- *A **forcing predicate**, i.e. a predicate C of arity $\mathcal{M}$ such that the following subtyping judgment is derivable:*

$$\frac{p \leq q}{\mathsf{C}(q) \sqsubseteq \mathsf{C}(p)}$$

Suppose that a forcing structure $\mathcal{F} = (\mathcal{M}, \mathsf{C})$ is fixed. We define an **annotated typing relation at level** n with $n \geq 1$, denoted $\vdash_n$. The corresponding rules manipulate judgments of one of the two following forms:

$$\mathcal{E}; \Gamma \vdash_n v : (P, p)$$
$$\mathcal{E}; \Gamma \vdash_n t : (N, p)$$

where Γ is a usual positive typing context, $\mathcal{E}$ is an inequational theory and $p \in \mathcal{M}$.

- The first part of the rules consists in annotated versions of the λ_{Mon} rules, which are defined in Figure 1 and Figure 2. We denote this set of rules by the name $\lambda^{\mathcal{M}}_{\mathsf{LCBPV}}$. The annotated type system induced by $\mathcal{M}$
- The second part consists in two new rules which are given in Figure 3. The first one says that we can always annotate a rule by a greater annotation (in the sense of $\leq$) and is true for every level n. The second one types the observation *at level* 1. Notice that one of the premises of this rule supposes a function f that is strong, in the sense of Definition 78.

Remark 92. *The observation rule is the only one that makes use of the forcing predicate C.*

3.3 | Forcing program transformation

We have defined in Section 2.4 a syntactic program transformation from $\lambda^{\mathbf{1}}_{\mathsf{Mon}}$ to λ_{LCBPV} that has been proved correct with respect to the reduction. We now turn our attention to the *typing* side of this transformation. To type the target of the transformation, we use what we call the **forcing type transformation**, which is an internalization of a forcing interpretation of λ_{LCBPV} *inside* λ_{LCBPV}. The source of the transformation (*i.e.* $\lambda^{\mathbf{1}}_{\mathsf{Mon}}$) is typed using the typing relation $\vdash_1$ defined in Section 3.2.

Multiplicative connectives

$$\frac{}{\mathcal{E};x:P\vdash_n x:(P,\mathbf{0})} \qquad \frac{\mathcal{E};\Xi\vdash_n t:(N,p)}{\mathcal{E};\Xi,x:P\vdash_n t:(N,p)} \qquad \frac{\mathcal{E};\Xi\vdash_n t:(N,p) \qquad p\preceq q}{\mathcal{E};\Xi\vdash_n t:(N,q)}$$

$$\frac{\mathcal{E};\Xi\vdash_n v:(P,p)}{\mathcal{E};\Xi\vdash_n \mathsf{ret}(v):(\Uparrow P,p)} \qquad \frac{\mathcal{E};\Xi\vdash_n t:(\Uparrow P,p) \qquad \mathcal{E};\Upsilon,x:P\vdash_n u:(N,q)}{\mathcal{E};\Xi,\Upsilon\vdash_n t\ \mathsf{to}\ x.u:(N,p+q)}$$

$$\frac{\mathcal{E};\Xi\vdash_n t:(N,p)}{\mathcal{E};\Xi\vdash_n \mathsf{thunk}(t):(\Downarrow N,p)} \qquad \frac{\mathcal{E};\Xi\vdash_n v:(\Downarrow N,p)}{\mathcal{E};\Xi,\Upsilon\vdash_n \mathsf{force}(v):(N,p)}$$

$$\frac{\mathcal{E};\Xi,x:P\vdash_n t:(N,p)}{\mathcal{E};\Xi\vdash_n \lambda x.t:(P\multimap N,p)} \qquad \frac{\mathcal{E};\Xi\vdash_n t:(P\multimap N,p) \qquad \mathcal{E};\Upsilon\vdash_n v:(P,q)}{\mathcal{E};\Xi,\Upsilon\vdash_n (t)v:(N,p+q)}$$

$$\frac{}{\mathcal{E};\Xi\vdash_n *:(\mathbb{1},\mathbf{0})} \qquad \frac{\mathcal{E};\Xi\vdash_n v:(\mathbb{1},q) \qquad \mathcal{E};\Upsilon\vdash_n t:(N,p)}{\mathcal{E};\Xi,\Upsilon\vdash_n \mathsf{let}\ *\ =\ v\ \mathsf{in}\ t:(N,p+q)}$$

$$\frac{\mathcal{E};\Xi\vdash_n v:(P,p) \qquad \mathcal{E};\Upsilon\vdash_n w:(Q,q)}{\mathcal{E};\Xi,\Upsilon\vdash_n (v,w):(P\otimes Q,p+q)}$$

$$\frac{\mathcal{E};\Xi\vdash_n v:(P\otimes Q,q) \qquad \mathcal{E};\Upsilon,x:P,y:Q\vdash_n t:(N,p)}{\mathcal{E};\Xi,\Upsilon\vdash_n \mathsf{let}\,(x,y)\ =\ v\ \mathsf{in}\ t:(N,p+q)}$$

$$\frac{}{\mathcal{E};\vdash_n \underline{0}:(\mathsf{Nat},\mathbf{0})} \qquad \frac{\mathcal{E};\Xi\vdash_n v:(\mathsf{Nat},p)}{\mathcal{E};\Xi\vdash_n \underline{\mathsf{s}}(v):(\mathsf{Nat},p)}$$

$$\frac{\mathcal{E};\Xi\vdash_n v:(\mathsf{Nat},q) \qquad \mathcal{E};\Upsilon,x:\mathsf{Nat}\vdash_n t:(N,p) \qquad \mathcal{E};\Upsilon,x:\mathsf{Nat}\vdash_n u:(N,p)}{\mathcal{E};\Xi,\Upsilon\vdash_n \mathsf{case}\,v\,\mathsf{of}\,x.t\parallel x.u:(N,p+q)}$$

Figure 1: λ_{Mon} Typing rules 1

Type translation

The type translation is parametrized by what we call a $\mathcal{F}$-model.

Definition 93 ($\mathcal{F}$-model)**.** *Let $\mathcal{F}=(\mathcal{M},\mathsf{C})$ be a forcing structure. A $\mathcal{F}$-**model** is a function that maps every predicate variable X of arity S to another predicate variable X^* of arity $S\times\mathcal{M}$.*

Suppose we have fixed a forcing structure $\mathcal{F}=(\mathcal{M},\mathsf{C})$ and a $\mathcal{F}$-model. The $\mathcal{F}$-model already associates to every predicate variable of arity S another predicate variable of arity $S\times\mathcal{M}$. The translation is an extension of the $\mathcal{F}$-model to all types, turning any predicate P

First-order quantifiers

$$\frac{\mathcal{E};\Xi \vdash_n t : (N,p) \qquad x\#\Xi}{\mathcal{E};\Xi \vdash_n t : (\forall x \in S.N,p)} \qquad \frac{\mathcal{E};\Xi \vdash_n t : (\forall x \in S.N,p) \qquad e \in S\text{-Exp}}{\mathcal{E};\Xi \vdash_n t : (N[e/x],p)}$$

$$\frac{\mathcal{E};\Xi \vdash_n v : (P[e/x],p) \qquad e \in S\text{-Exp}}{\mathcal{E};\Xi \vdash_n v : (\exists x \in S.P,p)}$$

$$\frac{\mathcal{E};\Xi \vdash_n v : (\exists x \in S.P,q) \qquad \mathcal{E};\Delta, z : P[y/x] \vdash_n t : (N,p) \qquad y\#\Gamma}{\mathcal{E};\Xi,\Delta \vdash_n t[v/z] : (N,p+q)}$$

Equational implication and conjunction

$$\frac{\mathcal{E}, e \succeq_S f;\Xi \vdash_n t : (N,p)}{\mathcal{E};\Xi \vdash_n t : (\{e \succeq_S f\} \mapsto N,p)} \qquad \frac{\mathcal{E};\Xi \vdash_n t : (\{e \succeq_S e\} \mapsto N,p)}{\mathcal{E};\Xi \vdash_n t : (N,p)}$$

$$\frac{\mathcal{E};\Xi \vdash_n v : (P,p) \qquad e \preceq_{\mathcal{E}} f}{\mathcal{E};\Xi \vdash_n v : (\{e \preceq_S f\} \wedge P,p)} \qquad \frac{\mathcal{E};\Xi \vdash_n v : (\{e \preceq_S e\} \wedge P,p)}{\mathcal{E};\Xi \vdash_n v : (P,p)}$$

$$\frac{\mathcal{E};\Xi \vdash_n c : (A,p) \qquad A \sqsubseteq_{\mathcal{E}} B}{\mathcal{E};\Xi \vdash_n c : (B,p)}$$

Figure 2: λ_{Mon} Typing rules 2

Additional rules

$$\frac{\mathcal{E};\Xi \vdash_n t : (N,p) \qquad p \preceq q}{\mathcal{E};\Xi \vdash_n t : (N,q)}$$

$$\frac{\mathcal{E};\Xi \vdash_1 t : (N,p) \qquad f \text{ is } \bullet\text{-strong} \qquad \mathcal{E};\Xi \vdash_0 \alpha : \forall x \in \mathcal{M}.\mathsf{C}(f(x)) \multimap \mathsf{C}(x)}{\mathcal{E};\Xi \vdash_1 (\!|t|\!)_1^{\alpha} : (N,f(p))}$$

Figure 3: λ_{Mon} Typing rules 3

(resp. negative predicate N) of arity S into a positive predicate $P^*(\kappa)$ (resp. negative predicate $N^*(\kappa)$ of arity $S \times \mathcal{M}$.

Notation 94. *In what follows, we use the letters $p, q, r, \ldots$ usually reserved for elements of $\mathcal{M}$ to denote $\mathcal{M}$-expressions (i.e. that possibly contain free variables of sort $\mathcal{M}$). By con-*

Positive interpretation

$$
\begin{array}{rcl}
X(e_1,\ldots,e_n)^*(p) & = & \exists q \in \mathcal{M}.\{q \preceq_{\mathcal{M}} p\} \wedge X^*(e_1,\ldots,e_n,q) \\
(\mathbb{1})^*(p) & = & \mathbb{1} \\
(\mathsf{Nat})^*(p) & = & \mathsf{Nat} \\
(P \otimes Q)^*(p) & = & \exists q_1 \in \mathcal{M}.\exists q_2 \in \mathcal{M}.\{q_1 + q_2 \preceq_{\mathcal{M}} p\} \wedge (P^*(q_1) \otimes Q^*(q_2)) \\
(\Downarrow N)^*(p) & = & \Downarrow(\forall r \in \mathcal{M}.\mathsf{C}(p \bullet r) \multimap N^\circ(r)) \\
(\exists x \in \mathcal{M}.P)^*(p) & = & \exists x \in \mathcal{M}.P^*(p) \\
(\{e \preceq_S f\} \wedge P)^*(p) & = & \{e \preceq_S f\} \wedge P^*(p)
\end{array}
$$

Negative environment interpretation

$$
\begin{array}{rcl}
(P \multimap M)^\circ(p) & = & \exists q \in \mathcal{M}.\exists r \in \mathcal{M}.\{p \succeq_{\mathcal{M}} q \bullet r\} \mapsto (P^*(q) \multimap M^\circ(r)) \\
(\Uparrow P)^\circ(p) & = & \Uparrow(\exists r \in \mathcal{M}.P^*(r) \otimes \mathsf{C}(r \bullet p)) \\
(\forall x \in \mathcal{M}.N)^\circ(p) & = & \forall x \in \mathcal{M}.N^\circ(p) \\
(\{e \succeq_S f\} \mapsto N)^\circ(p) & = & \{e \succeq_S f\} \mapsto N^\circ(p)
\end{array}
$$

Negative computation interpretation

$$N^*(p) \quad = \quad \forall r \in \mathcal{M}.\mathsf{C}(p \bullet r) \multimap N^\circ(r)$$

Figure 4: Forcing Type Translation

venience and because this is unambiguous, we will denote function symbols $\dot{f}$ simply by f. Hence we can write something like

$$\{p \succeq_{\mathcal{M}} r \bullet q\} \mapsto N$$

Definition 95 (Forcing translation of types)**.** *The forcing translation of types is defined inductively on the types in Figure 4.*

We now consider the annotated type system of Section 3.2 $\vdash_1$ at level 1 induced by the forcing structure $\mathcal{F}$. We show the following type preservation theorem:

Theorem 96 (Preservation theorem)**.** *The two following statements hold:*

1. *Suppose that the following judgment is derivable:*

$$\mathcal{E}; x_1 : P_1, \ldots, x_n : P_n \vdash_1 v : (P, p)$$

Then we also have for any distinct variables $\iota_1, \dots, \iota_n$:

$$\mathcal{E}; x_1 : P_1^*(\iota_1), \dots, x_n : P_n^*(\iota_n) \vdash_0 \langle\!\langle v \rangle\!\rangle : P^*(p + \sum_{1 \leq i \leq n} \iota_i)$$

2. *Suppose that the following judgment is derivable:*

$$\mathcal{E}; x_1 : P_1, \dots, x_n : P_n \vdash_1 t : (N, p)$$

Then we also have for any distinct variables $\iota_1, \dots, \iota_n$:

$$\mathcal{E}; x_1 : P_1^*(\iota_1), \dots, x_n : P_n^*(\iota_n) \vdash_0 \langle\!\langle t \rangle\!\rangle : N^*(p + \sum_{1 \leq i \leq n} \iota_i)$$

Lemma 97. *The following rule is derivable:*

$$\frac{\mathcal{E}; \Gamma, x : \mathsf{C}(p) \vdash_0 a : A^*(q) \qquad p \preceq_{\mathcal{E}} p' \qquad q \preceq_{\mathcal{E}} q'}{\mathcal{E}; \Gamma, x : \mathsf{C}(p') \vdash_0 a : A^*(q')}$$

Proof. The part concerning C is because of the definition of forcing structure (see Definition 91). We then only prove by induction on A that $A^*(p) \sqsubseteq_{\mathcal{E}} A^*(q)$ as soon as $p \preceq_{\mathcal{E}} q$.

- If $A = N$ then since we have $p \preceq_{\mathcal{E}} q$, we have by definition of the forcing structure and by subtyping, that $\mathsf{C}(q \bullet \iota) \sqsubseteq_{\mathcal{E}} \mathsf{C}(p \bullet \iota)$. Hence we obtain

  $$\mathsf{C}(p \bullet \iota) \multimap N^\circ(\iota) \sqsubseteq_{\mathcal{E}} \mathsf{C}(q \bullet \iota) \multimap N^\circ(\iota)$$

  And finally

  $$\forall \iota \in \mathcal{M}. \mathsf{C}(p \bullet \iota) \multimap N^\circ(\iota) \sqsubseteq_{\mathcal{E}} \forall \iota \in \mathcal{M}. \mathsf{C}(q \bullet \iota) \multimap N^\circ(\iota)$$

- If $A = P$, then we proceed by induction.
  - The cases $\mathsf{Nat}, \mathbb{1}$ are trivial.
  - The cases X and $\otimes$ are similar, so we prove only the latter. It is enough to show that for ι, κ fresh variables of sort $\mathcal{M}$, we have:

    $$\{p \preceq_{\mathcal{M}} \iota + \kappa\} \wedge P^*(\iota) \otimes Q^*(\kappa) \sqsubseteq_{\mathcal{E}} \{q \preceq_{\mathcal{M}} \iota + \kappa\} \wedge P^*(\iota) \otimes Q^*(\kappa)$$

    But this is obtained by application of the following rule:

    $$\frac{p \preceq_{\mathcal{E}} q \qquad \iota + \kappa \preceq_{\mathcal{E}} \iota + \kappa \qquad P^*(\iota) \otimes Q^*(\kappa) \sqsubseteq_{\mathcal{E}, p \succeq_{\mathcal{M}} \iota + \kappa} P^*(\iota) \otimes Q^*(\kappa)}{\{\iota + \kappa \preceq_{\mathcal{M}} p\} \wedge P^*(\iota) \otimes Q^*(\kappa) \sqsubseteq_{\mathcal{E}} \{\iota + \kappa \preceq_{\mathcal{M}} q\} \wedge P^*(\iota) \otimes Q^*(\kappa)}$$

  - For the positive shift, it is obtained by applying the $\Downarrow$ subtyping rule to the negative case already proved.
  - The existential quantifier and the inequational implication are proved by induction and using the corresponding subtyping rule.

□

Lemma 98. *Suppose that $A \sqsubseteq_{\mathcal{E}} B$. Then $A^*(\iota) \sqsubseteq_{\mathcal{E}} B^*(\iota)$.*

Proof. This is easily done by induction and using Lemma 97. □

We can now resume the proof of Theorem 96.

Proof. We prove these two statements by mutual induction on the typing derivation.

1. We only give the proofs of some interesting cases: the axiom, the introduction of $\Downarrow$, the introduction of $\mathbb{1}$, the introduction of Nat and the existential quantifier. The other cases are similar.
Axiom ::

$$\frac{}{\mathcal{E};\Gamma, x : P \vdash_1 x : (P, \mathbf{0})} \quad \rightsquigarrow \quad \frac{\dfrac{\dfrac{}{\mathcal{E};\Gamma^*(\overrightarrow{\iota'}), x : P^*(\iota) \vdash_0 x : P^*(\iota')}}{\mathcal{E};\Gamma^*(\overrightarrow{\iota'}), x : P^*(\iota) \vdash_0 x : P^*(\sum \overrightarrow{\tau} + \iota')}\text{Lemma 97}}{\mathcal{E};\Gamma^*(\overrightarrow{\tau}), x : P^*(\iota') \vdash_0 x : P^*(\mathbf{0} + \iota' + \sum \overrightarrow{\tau})} \cong_{\mathcal{E}}$$

Thunk ::

$$\frac{\mathcal{E};\Xi \vdash_1 t : (N, p)}{\mathcal{E};\Xi \vdash_1 \mathsf{thunk}(t) : (\Downarrow N, p)} \quad \rightsquigarrow \quad \frac{\mathcal{E};\Xi^*(\overrightarrow{\tau}) \vdash_0 \langle\!\langle t \rangle\!\rangle : N^*(p + \sum \overrightarrow{\tau})}{\mathcal{E};\Xi^*(\overrightarrow{\tau}) \vdash_0 \underbrace{\mathsf{thunk}(\langle\!\langle t \rangle\!\rangle)}_{=\langle\!\langle \mathsf{thunk}(t) \rangle\!\rangle} : \underbrace{\Downarrow N^*(p + \sum \overrightarrow{\tau})}_{=(\Downarrow N)^*(p + \sum \overrightarrow{\tau})}}$$

Tensor ::

$$\frac{\mathcal{E};\Xi \vdash_1 v : (P, p) \qquad \mathcal{E};\Upsilon \vdash_1 w : (Q, q)}{\mathcal{E};\Xi, \Upsilon \vdash_1 (v, w) : (P \otimes Q, p + q)} \quad \rightsquigarrow$$

$$\frac{\dfrac{\dfrac{\mathcal{E};\Xi^*(\overrightarrow{\tau}) \vdash_0 \langle\!\langle v \rangle\!\rangle : P^*(\overbrace{p + \sum \overrightarrow{\tau}}^{\text{noted } p'}) \qquad \mathcal{E};\Upsilon^*(\overrightarrow{\kappa}) \vdash_0 \langle\!\langle w \rangle\!\rangle : Q^*(\overbrace{q + \sum \overrightarrow{\kappa}}^{\text{noted } q'})}{\mathcal{E};\Xi^*(\overrightarrow{\tau}), \Upsilon^*(\overrightarrow{\kappa}) \vdash_0 (\langle\!\langle v \rangle\!\rangle, \langle\!\langle w \rangle\!\rangle) : P^*(p') \otimes Q^*(q') \qquad p' + q' \preceq_{\mathcal{E}} p' + q'}}{\mathcal{E};\Xi^*(\overrightarrow{\tau}), \Upsilon^*(\overrightarrow{\kappa}) \vdash_0 (\langle\!\langle v \rangle\!\rangle, \langle\!\langle w \rangle\!\rangle) : \{p' + q' \preceq_{\mathcal{M}} p' + q'\} \wedge (P^*(p') \otimes Q^*(q'))}}{\mathcal{E};\Xi^*(\overrightarrow{\tau}), \Upsilon^*(\overrightarrow{\kappa}) \vdash_0 \underbrace{(\langle\!\langle v \rangle\!\rangle, \langle\!\langle w \rangle\!\rangle)}_{=\langle\!\langle (v,w) \rangle\!\rangle} : \underbrace{\exists x \in \mathcal{M}. \exists y \in \mathcal{M}. \{x + y \preceq_{\mathcal{M}} p' + q'\} \wedge P^*(x) \otimes Q^*(y)}_{=(P \otimes Q)^*(p + q + \sum \overrightarrow{\tau} + \sum \overrightarrow{\kappa})}}$$

Existential quantifier ::

$$\frac{\mathcal{E};\Xi \vdash_1 v : (P[e/x^S], p)}{\mathcal{E};\Xi \vdash_1 v : (\exists x \in S.P, p)} \quad \rightsquigarrow \quad \frac{\mathcal{E};\Xi^*(\overrightarrow{\iota}) \vdash_0 \langle\!\langle v \rangle\!\rangle : P[e/x^S]^*(\overbrace{p + \sum \overrightarrow{\iota}}^{\text{noted } p'})}{\mathcal{E};\Xi^*(\overrightarrow{\iota}) \vdash_0 \langle\!\langle v \rangle\!\rangle : \underbrace{P[e/x^S]^*(p')}_{=P^*(p')[e/x^S]}}}{\mathcal{E};\Xi^*(\overrightarrow{\iota}) \vdash_0 \langle\!\langle v \rangle\!\rangle : \underbrace{\exists x \in S.P^*(p')}_{=(\exists x \in S.P)^*(p')}}$$

Unit ::

$$\frac{}{\mathcal{E};\Gamma \vdash_1 * : (\mathbb{1}, \mathbf{0})} \quad \rightsquigarrow \quad \frac{}{\mathcal{E};\Gamma^*(\overrightarrow{\iota}) \vdash_0 * : \underbrace{\mathbb{1}}_{=\mathbb{1}^*(\overrightarrow{\iota})}}$$

Zero ::

$$\frac{}{\mathcal{E};\vdash_1 \mathbf{0} : (\mathsf{Nat}, p)} \quad \rightsquigarrow \quad \frac{}{\mathcal{E};\vdash_0 \underbrace{\mathbf{0}}_{=\langle\!\langle \mathbf{0} \rangle\!\rangle} : \underbrace{\mathsf{Nat}}_{=\mathsf{Nat}^*(p)}}$$

Successor ::

$$\frac{\mathcal{E};\Xi \vdash_1 v : (\mathsf{Nat}, p)}{\mathcal{E};\Xi \vdash_1 \underline{\mathsf{s}}(v) : (\mathsf{Nat}, p)} \quad \rightsquigarrow \quad \frac{\mathcal{E};\Xi^*(\overrightarrow{\iota}) \vdash_0 \langle\!\langle v \rangle\!\rangle : \overbrace{\mathsf{Nat}^*(p + \sum \overrightarrow{\iota})}^{=\mathsf{Nat}}}{\mathcal{E};\Xi^*(\overrightarrow{\iota}) \vdash_0 \underbrace{\underline{\mathsf{s}}(\langle\!\langle v \rangle\!\rangle)}_{=\langle\!\langle \underline{\mathsf{s}}(v) \rangle\!\rangle} : \mathsf{Nat}}$$

2. We now prove the negative cases. We only give the proofs of some interesting cases: the introduction and elimination of $\multimap$, the introduction of $\Uparrow$ and the elimination of $\Downarrow$, and finally the observation rule. The other rules are handled in a similar way.

Lambda ::

$$\frac{\mathcal{E};\Xi, x : P \vdash_1 t : (N, p)}{\mathcal{E};\Xi \vdash_1 \lambda x.t : (P \multimap N, p)} \quad \rightsquigarrow$$

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{E};\Xi^*(\overrightarrow{\iota}), x : P^*(\kappa) \vdash_0 \langle\!\langle t \rangle\!\rangle : \forall r \in \mathcal{M}.\mathsf{C}((p + \sum \overrightarrow{\iota} + \kappa) \bullet r) \multimap N^\circ(r)}{\mathcal{E};\Xi^*(\overrightarrow{\iota}), \kappa : \mathsf{C}((p + \sum \overrightarrow{\iota} + \kappa) \bullet r), x : P^*(\kappa) \vdash_0 \langle\!\langle t \rangle\!\rangle \kappa : N^\circ(r)}}{\mathcal{E};\Xi^*(\overrightarrow{\iota}), \kappa : \mathsf{C}((p + \sum \overrightarrow{\iota} + \kappa) \bullet r) \vdash_0 \lambda x.(\langle\!\langle t \rangle\!\rangle)\kappa : P^*(\kappa) \multimap N^\circ(r)}}{\mathcal{E}, \sigma \geq_{\mathcal{M}} \kappa \bullet r;\Xi^*(\overrightarrow{\iota}), \kappa : \mathsf{C}((p + \sum \overrightarrow{\iota} + \kappa) \bullet r) \vdash_0 \lambda x.(\langle\!\langle t \rangle\!\rangle)\kappa : P^*(\kappa) \multimap N^\circ(r)}}{\mathcal{E}, \sigma \geq_{\mathcal{M}} \kappa \bullet r;\Xi^*(\overrightarrow{\iota}), \kappa : \mathsf{C}((p + \sum \overrightarrow{\iota}) \bullet (\kappa \bullet r)) \vdash_0 \lambda x.(\langle\!\langle t \rangle\!\rangle)\kappa : P^*(\kappa) \multimap N^\circ(r)}}{\mathcal{E}, \sigma \geq_{\mathcal{M}} \kappa \bullet r;\Xi^*(\overrightarrow{\iota}), \kappa : \mathsf{C}((p + \sum \overrightarrow{\iota}) \bullet \sigma) \vdash_0 \lambda x.(\langle\!\langle t \rangle\!\rangle)\kappa : P^*(\kappa) \multimap N^\circ(r)}}{\mathcal{E};\Xi^*(\overrightarrow{\iota}), \kappa : \mathsf{C}((p + \sum \overrightarrow{\iota}) \bullet \sigma) \vdash_0 \lambda x.(\langle\!\langle t \rangle\!\rangle)\kappa : \{\sigma \geq_{\mathcal{M}} \kappa \bullet r\} \mapsto (P^*(\kappa) \multimap N^\circ(r))}}{\mathcal{E};\Xi^*(\overrightarrow{\iota}), \kappa : \mathsf{C}((p + \sum \overrightarrow{\iota}) \bullet \sigma) \vdash_0 \lambda x.(\langle\!\langle t \rangle\!\rangle)\kappa : (P \multimap N)^\circ(\sigma)}}{\mathcal{E};\Xi^*(\overrightarrow{\iota}), \vdash_0 \lambda\kappa.\lambda x.(\langle\!\langle t \rangle\!\rangle)\kappa : \mathsf{C}((p + \sum \overrightarrow{\iota}) \bullet \sigma) \multimap (P \multimap N)^\circ(\sigma)}}{\mathcal{E};\Xi^*(\overrightarrow{\iota}), \vdash_0 \underbrace{\lambda\kappa.\lambda x.(\langle\!\langle t \rangle\!\rangle)\kappa}_{=\langle\!\langle \lambda x.t \rangle\!\rangle} : \underbrace{\forall \sigma \in \mathcal{M}.\mathsf{C}((p + \sum \overrightarrow{\iota}) \bullet \sigma) \multimap (P \multimap N)^\circ(\sigma)}_{=(P \multimap N)^*(p + \sum \overrightarrow{\iota})}}$$

Here, we have used the fact that $\kappa \bullet r \preceq \sigma$ and Lemma 97.

Application ::

$$\frac{\mathcal{E};\Xi \vdash_1 t : (P \multimap N, p) \qquad \mathcal{E};\Upsilon \vdash_1 P : (P, q)}{\mathcal{E};\Xi,\Upsilon \vdash_1 (t)v : (N, p+q)}$$

We pose:

$$\begin{array}{rcl} p' & = & p + \sum \overrightarrow{\iota} \\ q' & = & q + \sum \overrightarrow{\kappa} \end{array}$$

We have on the one hand

$$\frac{\dfrac{\mathcal{E};\Xi^*(\overrightarrow{\iota}) \vdash_0 \langle\!\langle t \rangle\!\rangle : \forall x \in \mathcal{M}.\mathsf{C}(p' \bullet x) \multimap (P \multimap N)^\circ(x)}{\mathcal{E};\Xi^*(\overrightarrow{\iota}),\kappa : \mathsf{C}(p' \bullet q' \bullet x') \vdash_0 \langle\!\langle t \rangle\!\rangle\kappa : (P \multimap N)^\circ(q' \bullet x')}}{\mathcal{E};\Xi^*(\overrightarrow{\iota}),\kappa : \mathsf{C}(p' \bullet q' \bullet x') \vdash_0 \langle\!\langle t \rangle\!\rangle\kappa : (P^*(q') \multimap N^\circ(x'))}$$

where x' is chosen fresh. On the other hand we have

$$\mathcal{E};\Upsilon^*(\overrightarrow{\kappa}) \vdash_0 \langle\!\langle v \rangle\!\rangle : P^*(q')$$

Hence, by combining both judgments we obtain

$$\mathcal{E};\Xi^*(\overrightarrow{\iota}),\Upsilon^*(\overrightarrow{\kappa}),\kappa : \mathsf{C}(p' \bullet q' \bullet x') \vdash_0 (\langle\!\langle t \rangle\!\rangle\kappa)\langle\!\langle v \rangle\!\rangle : N^\circ(x')$$

Which in turn gives

$$\mathcal{E};\Xi^*(\overrightarrow{\iota}),\Upsilon^*(\overrightarrow{\kappa}) \vdash_0 \lambda\kappa.(\langle\!\langle t \rangle\!\rangle\kappa)\langle\!\langle v \rangle\!\rangle : \mathsf{C}(p' \bullet q' \bullet x') \multimap N^\circ(x')$$

We conclude by using $\cong_{\mathcal{E}}$, because $p' \bullet q' \bullet x' = (p' + q') \bullet x'$ and because x' is fresh:

$$\mathcal{E};\Xi^*(\overrightarrow{\iota}),\Upsilon^*(\overrightarrow{\kappa}) \vdash_0 \lambda\kappa.(\langle\!\langle t \rangle\!\rangle\kappa)\langle\!\langle v \rangle\!\rangle : \forall x' \in \mathcal{M}.\mathsf{C}((p' + q') \bullet x') \multimap N^\circ(x')$$

To ::

$$\frac{\mathcal{E};\Xi, x : P \vdash_1 t : (N, p) \qquad \mathcal{E};\Upsilon \vdash_1 u : (\Uparrow P, q)}{\mathcal{E};\Xi,\Upsilon \vdash_1 u \ \mathsf{to}\ x.t : (N, p+q)}$$

First, we have

$$\frac{\dfrac{\mathcal{E};\Xi^*(\overrightarrow{\iota}), x : P^*(\iota') \vdash_0 \langle\!\langle t \rangle\!\rangle : \forall \sigma \in \mathcal{M}.\mathsf{C}((p + \sum \overrightarrow{\iota} + \iota') \bullet \sigma) \multimap N^\circ(\sigma)}{\mathcal{E};\Xi^*(\overrightarrow{\iota}), x : P^*(\iota'),\kappa' : \mathsf{C}(p' \bullet \iota' \bullet \sigma) \vdash_0 \langle\!\langle t \rangle\!\rangle\kappa' : N^\circ(\sigma)}}{\mathcal{E};\Xi^*(\overrightarrow{\iota}), z : P^*(\iota') \otimes \mathsf{C}(p' \bullet \iota' \bullet \sigma) \vdash_0 \mathsf{let}\,(x,\kappa') \ = \ z \ \mathsf{in}\ \langle\!\langle t \rangle\!\rangle\kappa' : N^\circ(\sigma)}$$

Secondly, if we pose $q' = q + \sum \overrightarrow{\kappa}$, we have:

$$\frac{\mathcal{E};\Upsilon^*(\overrightarrow{\kappa}) \vdash_0 \langle\!\langle u \rangle\!\rangle : \forall \sigma \in \mathcal{M}.\mathsf{C}(q' \bullet \sigma) \multimap \Uparrow\exists y \in \mathcal{M}.P^*(y) \otimes \mathsf{C}(y \bullet \sigma)}{\mathcal{E};\Upsilon^*(\overrightarrow{\kappa}),\kappa : \mathsf{C}(q' \bullet p' \bullet \sigma) \vdash_0 \langle\!\langle u \rangle\!\rangle\kappa : \Uparrow\exists y \in \mathcal{M}.P^*(y) \otimes \mathsf{C}(y \bullet p' \bullet \sigma)}$$

By using both the $\Uparrow$ and $\exists$ elimination rules, we obtain:

$$\mathcal{E};\Xi^*(\overrightarrow{\iota}),\Upsilon^*(\overrightarrow{\kappa}),\kappa : \mathsf{C}(q' \bullet p' \bullet \sigma) \vdash_0 \langle\!\langle u \rangle\!\rangle\kappa \ \mathsf{to}\ z.\mathsf{let}\,(x,\kappa') \ = \ z \ \mathsf{in}\ \langle\!\langle t \rangle\!\rangle\kappa' : N^\circ(\sigma)$$

Finally,

$$\mathcal{E};\Xi^*(\overrightarrow{\iota}),\Upsilon^*(\overrightarrow{\kappa})\vdash_0 \lambda\kappa.\langle\!\langle u\rangle\!\rangle\kappa \text{ to } z.\mathsf{let}\,(x,\kappa') \;=\; z\,\mathsf{in}\,\langle\!\langle t\rangle\!\rangle\kappa' : \forall\sigma\in\mathcal{M}.\mathsf{C}((p'+q')\bullet\sigma)\multimap N^\circ(\sigma)$$

Force ::

$$\frac{\mathcal{E};\Xi\vdash_1 v:(\Downarrow N,p)}{\mathcal{E};\Xi\vdash_1 \mathsf{force}(v):(N,p)} \quad\rightsquigarrow\quad \frac{\mathcal{E};\Xi^*(\overrightarrow{\iota})\vdash_0 \langle\!\langle v\rangle\!\rangle :: \overbrace{(\Downarrow N)^*(p+\overrightarrow{\iota})}^{=\Downarrow N^*(p+\overrightarrow{\iota})}}{\mathcal{E};\Xi^*(\overrightarrow{\iota})\vdash_0 \underbrace{\mathsf{force}(\langle\!\langle v\rangle\!\rangle)}_{=\langle\!\langle\mathsf{force}(v)\rangle\!\rangle} : N^*(p+\overrightarrow{\iota})}$$

Return ::

$$\frac{\mathcal{E};\Xi\vdash_1 v:(P,p)}{\mathcal{E};\Xi\vdash_1 \mathsf{ret}(v):(\Uparrow P,p)} \quad\rightsquigarrow$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{E};\Xi^*(\overrightarrow{\iota})\vdash_0 \langle\!\langle v\rangle\!\rangle :: P^*(\overbrace{p+\overrightarrow{\iota}}^{\text{noted } p'})}{\mathcal{E};\Xi^*(\overrightarrow{\iota}),\kappa:\mathsf{C}(y\bullet p')\vdash_0 (\langle\!\langle v\rangle\!\rangle,\kappa):P^*(p')\otimes\mathsf{C}(y\bullet p')}}{\mathcal{E};\Xi^*(\overrightarrow{\iota}),\kappa:\mathsf{C}(y\bullet p')\vdash_0 (\langle\!\langle v\rangle\!\rangle,\kappa):\exists x\in\mathcal{M}.(P^*(x)\otimes\mathsf{C}(y\bullet x))}}{\mathcal{E};\Xi^*(\overrightarrow{\iota}),\kappa:\mathsf{C}(y\bullet p')\vdash_0 (\langle\!\langle v\rangle\!\rangle,\kappa):\exists x\in\mathcal{M}.(P^*(x)\otimes\mathsf{C}(y\bullet x))}}{\mathcal{E};\Xi^*(\overrightarrow{\iota}),\kappa:\mathsf{C}(y\bullet p')\vdash_0 \mathsf{ret}((\langle\!\langle v\rangle\!\rangle,\kappa)):\Uparrow(\exists x\in\mathcal{M}.(P^*(x)\otimes\mathsf{C}(y\bullet x)))}}{\mathcal{E};\Xi^*(\overrightarrow{\iota})\vdash_0 \lambda\kappa.\mathsf{ret}((\langle\!\langle v\rangle\!\rangle,\kappa)):\mathsf{C}(y\bullet p')\multimap\Uparrow(\exists x\in\mathcal{M}.(P^*(x)\otimes\mathsf{C}(y\bullet x)))}}{\mathcal{E};\Xi^*(\overrightarrow{\iota})\vdash_0 \underbrace{\lambda\kappa.\mathsf{ret}((\langle\!\langle v\rangle\!\rangle,\kappa))}_{=\langle\!\langle\mathsf{ret}(v)\rangle\!\rangle} : \underbrace{\forall y\in\mathcal{M}.\mathsf{C}(y\bullet p')\multimap\Uparrow(\exists x\in\mathcal{M}.(P^*(x)\otimes\mathsf{C}(y\bullet x)))}_{=(\Uparrow P)^*(p')}}$$

Unit elim. ::

$$\frac{\mathcal{E};\Xi\vdash_1 v:(\mathbb{1},q)\qquad \mathcal{E};\Upsilon\vdash_1 t:(N,p)}{\mathcal{E};\Xi,\Upsilon\vdash_1 \mathsf{let}\,*\;=\;v\,\mathsf{in}\,t:(N,p+q)} \quad\rightsquigarrow$$

$$\frac{\dfrac{\mathcal{E};\Xi^*(\overrightarrow{\iota})\vdash_0 \langle\!\langle v\rangle\!\rangle:\mathbb{1}\qquad \mathcal{E};\Upsilon^*(\overrightarrow{\kappa})\vdash_0 \langle\!\langle t\rangle\!\rangle:N^*(p+\sum\overrightarrow{\kappa})}{\mathcal{E};\Xi^*(\overrightarrow{\iota}),\Upsilon^*(\overrightarrow{\kappa})\vdash_0 \mathsf{let}\,*\;=\;\langle\!\langle v\rangle\!\rangle\,\mathsf{in}\,\langle\!\langle t\rangle\!\rangle:N^*(p+\sum\overrightarrow{\kappa})}}{\mathcal{E};\Xi^*(\overrightarrow{\iota}),\Upsilon^*(\overrightarrow{\kappa})\vdash_0 \mathsf{let}\,*\;=\;\langle\!\langle v\rangle\!\rangle\,\mathsf{in}\,\langle\!\langle t\rangle\!\rangle:N^*(p+\sum\overrightarrow{\iota}+\sum\overrightarrow{\kappa})}\text{Lemma 97}$$

Observation :: Suppose we have:

$$\frac{\mathcal{E};\Xi\vdash_1 t:(N,p)\qquad f\text{ is }\bullet\text{-strong}\qquad \mathcal{E};\vdash_0\alpha:\forall x\in\mathcal{M}.\mathsf{C}(f(x))\multimap\Uparrow\mathsf{C}(x)}{\mathcal{E};\Xi\vdash_1 \langle\!|t|\!\rangle_1^\alpha:(N,f(p))}$$

Then, first we have by induction hypothesis:

$$\mathcal{E};\Xi^*(\overrightarrow{\iota})\vdash_0\langle\!\langle t\rangle\!\rangle:\forall x\in\mathcal{M}.\mathsf{C}((p+\sum\overrightarrow{\iota})\bullet x)\multimap N^\circ(x)$$

We note $p'=p+\sum\overrightarrow{\iota}$. Hence if y is a fresh variable, we obtain:

$$(1)\qquad \mathcal{E};\Xi^*(\overrightarrow{\iota}),\kappa:\mathsf{C}(p'\bullet y)\vdash_0\langle\!\langle t\rangle\!\rangle\kappa:N^\circ(y)$$

On the other hand we have by induction hypothesis on α that:

$$\mathcal{E}; \kappa' : \mathsf{C}(f(p' \bullet y)) \vdash_0 (\alpha)\kappa' : \Uparrow \mathsf{C}(p' \bullet y)$$

But since

$$\begin{array}{rcll} (f(p) + \iota_1 + \cdots + \iota_n) \bullet y & = & f(p) \bullet (\iota_1 \bullet \ldots \iota_n \bullet y) & \\ & \geq & f(p \bullet (\iota_1 \bullet \ldots \iota_n \bullet y)) & (f \text{ is } \bullet\text{-strong}) \\ & = & f(p' \bullet y) & \end{array}$$

We also have by Lemma 97:

$$(2) \qquad \mathcal{E}; \kappa' : \mathsf{C}((f(p) + \sum \overrightarrow{\iota}) \bullet y) \vdash_0 (\alpha)\kappa' : \Uparrow \mathsf{C}(p' \bullet y)$$

Therefore by combining (1) and (2) we obtain:

$$\mathcal{E}; \Xi^*(\overrightarrow{\iota}), \kappa' : \mathsf{C}((f(p) + \sum \overrightarrow{\iota}) \bullet y) \vdash_0 (\alpha)\kappa' \text{ to } \kappa.(\langle\!\langle t \rangle\!\rangle \kappa) : N^\circ(y)$$

By introducing a λ:

$$\mathcal{E}; \Xi^*(\overrightarrow{\iota}) \vdash_0 \lambda\kappa'.(\alpha)\kappa' \text{ to } \kappa.(\langle\!\langle t \rangle\!\rangle \kappa) : \mathsf{C}((f(p) + \sum \overrightarrow{\iota}) \bullet y) \multimap N^\circ(y)$$

Since y does not appear in $\mathcal{E}$ or $\Xi^*(\overrightarrow{\iota})$, we can introduce a universal quantifier:

$$\mathcal{E}; \Xi^*(\overrightarrow{\iota}) \vdash_0 \lambda\kappa'.(\alpha)\kappa' \text{ to } \kappa.(\langle\!\langle t \rangle\!\rangle \kappa) : \forall y \in \mathcal{M}.\mathsf{C}((f(p) + \sum \overrightarrow{\iota}) \bullet y) \multimap N^\circ(y)$$

We conclude by remarking that $\langle\!\langle \langle\!| t |\!\rangle_1^\alpha \rangle\!\rangle = \lambda\kappa'.(\alpha)\kappa' \text{ to } \kappa.(\langle\!\langle t \rangle\!\rangle \kappa)$ and $N^*(f(p) + \sum \overrightarrow{\iota}) = \forall y \in \mathcal{M}.\mathsf{C}((f(p) + \sum \overrightarrow{\iota}) \bullet y) \multimap N^\circ(y)$.

□

Chapter IV

Monitoring algebras

Contents

In this chapter, we introduce several realizability frameworks.

- The first one is the **unary realizability model**. It is a usual Krivine-style realizability framework, but based on the **MAM** (instead of the Krivine Abstract Machine), which makes this framework similar to Miquel's unary realizability based on the **KFAM** [Miq11]. A particularity of our framework is that we do not define one but an infinity of interpretations of λ_{LCBPV} types, for each level $n \in \mathbb{N}$ of the **MAM**. The interpretation at level 0 is then shown to be sound with respect to the rules of λ_{LCBPV}.

- The second one is based on the central notion of n-**Monitoring Algebra** (abbreviated n-**MA** in the rest of this thesis) . The definition of n-**MA**s crucially relies on both the monitoring abstract machine and on forcing monoids. Given a n-**MA**, we define a realizability framework which is an elaboration of the unary realizability model. One particularity of this model is that it is annotated by forcing conditions. The main results of this chapter concerning the n-**MA**s are:

  - The **soundness theorem**, which shows that given any n-**MA** $\mathcal{A}$, the annotated rules (at level n) of λ_{LCBPV} given in Section 3.2 are sound in the model induced by $\mathcal{A}$ (at level n).
  - The definition of the $\mathcal{A}$-**monitor** condition. A pair (α, f) that satisfies the $\mathcal{A}$-monitor condition is shown to make the following typing rule sound in $\mathcal{A}$:

$$\frac{\mathcal{E}; \Gamma \vdash_n t : (N, p)}{\mathcal{E}; \Gamma \vdash_n (\!|t|\!)^{\alpha}_n : (N, f(p))}$$

  - We devote a section to the explanation of what we mean by "adding a type" in our framework. We define an algebraic notion of $\mathcal{A}$-**connective**. Each such $\mathcal{A}$-connective admits a realizability interpretation (both in the unary and **MA** case) and an extension of the forcing translation.
  - We also study in more depth the remarkable properties of the 1-**MA**s. In particular, in a 1-**MA** $\mathcal{A}$, it is possible to characterize a large subset of the $\mathcal{A}$-monitors. We moreover prove the **connection theorem**, which says that each realizability model induced by a 1-**MA** can be uniquely decomposed as the *iteration* of a unary realizability model (based on the **MAM**) and of a forcing model (as defined in Chapter III). Another way of seeing things is to say that a monitoring algebra is obtained by a certain forcing program transformation of the unary realizability framework.

Contributions

Section 4.1 is devoted to the definition of a family of unary realizability interpretations of λ_{LCBPV} types indexed by the levels of the **MAM**, and the proof of a soundness theorem at level 0. In Section 4.2, we define the notion of n-monitoring algebra, together with the biorthogonality-based realizability interpretation they induce. We then prove that each n-**MA** induces a sound interpretation of the annotated type system of Section 3.2. We then explain in Section 4.3 how to extend the realizability interpretation and the forcing interpretation to new types using $\mathcal{A}$-connectives. We subsequently turn our attention more specifically to 1-**MA**s. We show

in Section 4.4 the main properties of 1-**MA**s: the connection theorem and the soundness of the monitoring rule. Finally, we give in Section 4.5 three important examples of 1-**MA**s: the linear-time algebra, an algebra for first-order references and finally the step-indexing algebra.

4.1 | Simple realizability

We now introduce a Krivine style realizability model based on the **MAM**. As we have already remarked, the **MAM** is similar in spirit to Miquel's **KFAM**. Miquel shows that his **KFAM** can be used to define a realizability semantics, just like the usual Krivine abstract machine. Similarly to Krivine's classical realizability, it is *unary*, meaning that we define a realizability relation of the form

$$t \Vdash A$$

It relates *one* program t to a type A. He then state and prove two different soundness results of this new realizability semantics:

1. A soundness theorem for the non-forcing mode.
2. A soundness theorem for the forcing mode.

We adopt a similar strategy, that is defining a realizability model based on the **MAM**, but the technical details differ because we in fact define an infinity of different realizability interpretations of types: one for each execution level k of the **MAM**. We then state two soundness results:

1. One for the level 0
2. One for the level 1

These two results are the counterparts in our framework of Miquel's results on the **KFAM**. In this section, we only prove the first one, while the second one will be proved as a corollary of the connection theorem of Section 4.4. This unary realizability model will be the **first stage** in the elaboration of the theory of monitoring algebras.

4.1.1 Orthogonality

As mentioned in Section 1.3, Krivine's realizability is built around the notion of **orthogonality**. It is a way of defining when a program and an environment *interact well*. It allows to define sets of programs (or sets of environments) that interact well with another set of well-chosen environments (or programs). This notion of *good interaction* is in fact a parameter of the construction, usually called the **pole**. Our realizability framework is built around a similar notion that we define here.

Definition 99 (Pole). *A **pole** is a set $\perp\!\!\!\perp$ of configurations and co-configurations in $\mathbb{C} \cup \overline{\mathbb{C}}$, which is $\to$-**saturated**:*

$$\forall C \in \perp\!\!\!\perp,\ C' \ \to\ C \Longrightarrow C' \in \perp\!\!\!\perp$$

and $\approx$-saturated:

$$\forall C \in \perp\!\!\!\perp,\ C' \ \cong\ C \Longrightarrow C' \in \perp\!\!\!\perp$$

The pole represents the notion of *computational correctness* we want to observe. It is a way of discriminating between *bad* and *good* configurations.

Remark 100. *A pole is in particular always $\xrightarrow{n}$-saturated for any $n \in \mathbb{N}$:*

$$\forall C \in \perp\!\!\!\perp,\ C' \ \xrightarrow{n}\ C \Longrightarrow C' \in \perp\!\!\!\perp$$

Example 101. *Here are some examples of valid poles:*

1. *The two following trivial poles:*
   - $\perp\!\!\!\perp \stackrel{\text{def}}{=} \varnothing$
   - $\perp\!\!\!\perp \stackrel{\text{def}}{=} \mathbb{C} \cup \overline{\mathbb{C}}$
2. *The pole of terminating configurations*

$$\perp\!\!\!\perp_{✠} \stackrel{\text{def}}{=} \{\ C \in \mathbb{C} \cup \overline{\mathbb{C}} \mid C \textit{ reduces on either } \begin{cases} \langle v, \mathsf{a}(v_n) \dots . \mathsf{a}(v_1).\mathsf{nil} \rangle_n \\ ✠ \end{cases} \ \}$$

3. *The pole of diverging configurations*

$$\perp\!\!\!\perp_{\Omega} \stackrel{\text{def}}{=} \{\ C \in \mathbb{C} \cup \overline{\mathbb{C}} \mid C \textit{ diverges}\ \}$$

If there are plenty of possible choices of poles, in this thesis we will build everything around the pole of terminating configurations defined in Example 101. But since most of the results are true for any pole, we keep it as a parameter in most parts of this thesis. Moreover, it may be the case that other poles are of interest in the future. As in Krivine's work, this pole induces an **orthogonality relation**. To be precise it induces a family of orthogonality relations indexed by $\mathbb{N}$ (one relation for each execution level of the **MAM**).

Definition 102 (Orthogonality). *Let $k \in \mathbb{N}$. We say that a computation t and an environment E are k-**orthogonal** and we note it $t \perp_k E$ iff the following holds:*

$$\langle t, E \rangle_k \in \perp\!\!\!\perp$$

*Similarly, we say that a value v and an environment E are k-**orthogonal** and we note it $v \perp_k E$ iff the following holds:*

$$\langle \uparrow E, v \rangle_k \in \perp\!\!\!\perp$$

Remark 103. *Because $\perp\!\!\!\perp$ is $\approx$-saturated, we have*

$$v \perp E \Leftrightarrow \mathsf{ret}(v) \perp_0 E$$

However, if $k \geq 1$, we do not have:

$$v \perp E \Leftrightarrow \mathsf{ret}(v) \perp_k E$$

The intuition behind this orthogonality relation is that a program t and an environment E are k-orthogonal if their combination yields a *good* interaction at level k.

Remark 104. *It is clear that the orthogonality relations at different levels are distincts. Indeed, the correctness of a configuration at level $n \in \mathbb{N}$ won't imply in general its correctness at level $m \neq n$. For example, if we consider the pole of diverging configurations, we have:*

$$\langle \Omega, \mathsf{nil} \rangle_0 \in \perp\!\!\!\perp_\Omega$$

But, since at level 1 it cannot reduce:

$$\langle \Omega, \mathsf{nil} \rangle_1 \notin \perp\!\!\!\perp_\Omega$$

Indeed, we need an argument to be pushed on the environment for the configuration to reduce at level 1. For example:

$$\langle \Omega, \mathsf{a}(\underline{0}).\mathsf{nil} \rangle_1 \in \perp\!\!\!\perp_\Omega$$

Definition 105 (Orthogonal of sets of values and environments)**.** *Each relation $\perp_n$ can be lifted to sets of closed values and sets of environments. Suppose that $X \subseteq \mathbb{V}$. Then we define its **orthogonal at level** n:*

$$X^{\perp_n} \stackrel{\mathsf{def}}{=} \{\ E \in \mathbb{E} \mid \forall v \in X, v \perp_n E\ \}$$

Similarly, if $Y \subseteq \mathbb{E}$, we define

$$Y^{\perp_n} \stackrel{\mathsf{def}}{=} \{\ t \in \mathbb{P} \mid \forall E \in Y, t \perp_n E\ \}$$

Property 106. *If $X \subseteq Y$, then $Y^{\perp_n} \subseteq X^{\perp_n}$.*

Proof. Let $E \in Y^{\perp_n}$. For any $v \in X$, we have $v \in Y$, hence $v \perp_n E$. That means $E \in X^{\perp_n}$. □

Property 107. *If $v \in X$, then $\mathsf{ret}(v) \in X^{\perp_1\perp_1}$.*

Proof. Let $E \in X^{\perp_1}$. Then we want to show that $\langle \mathsf{ret}(v), E\rangle_1 \in \perp\!\!\!\perp$. But this is equivalent by using $\cong_1$ to show that $\langle \uparrow E, v\rangle_1 \in \perp\!\!\!\perp$. This exactly means that $v \perp E$, which is true. □

4.1.2 Interpretation

We have now everything we need to give an interpretation of λ_{LCBPV} types. The idea is that:

- We associate to each *positive type* P a set of *values* $\|P\|$, called the **truth value of** P (following Miquel's convention).
- Similarly, we associate to each *negative type* N a set of *environments* $[\![N]\!]$, called the **falsity value of** N.
- Using the previously defined set of environments, we associate to each negative type N a set of *computations* $\|N\|$, called the **truth value of** N. This set is defined as the orthogonal of a set of environments $[\![N]\!]$, meaning that it contains the computations that *interact well* with the environments of this set.

We in fact build an infinity of interpretations for each level $k \in \mathbb{N}$. Those interpretations are all built following the same pattern, allowing us to only give one common definition.

Definition 108. *A **unary model** (or simply **model**) ρ is a function such that:*

- *It assigns to each first-order variable x^S an element $\rho(x) \in S$.*
- *It assigns to every predicate variable $\mathcal{P}$ of arity $S_1 \times \cdots \times S_n$ a function $S_1 \times \cdots \times S_n \to \mathcal{P}(\mathbb{V})$.*

Definition 109 (Interpretation)**.** *Let $k \in \mathbb{N}$ and ρ be a unary model. We define the **interpretation** $\|.\|_{k,\rho}$ of a type that associates:*

- *to every value type P a set $\|P\|_{k,\rho} \subseteq \mathbb{V}$ of values*
- *to every computation type N a set $[\![N]\!]_{k,\rho} \subseteq \mathbb{E}$ of environments.*
- *to every computation type N a set $\|N\|_{k,\rho} \subseteq \mathbb{P}$ of computations.*

This interpretation is defined by induction on the formula and given in Figure 1.

Positive interpretation

$$
\begin{array}{rcl}
\|X(e_1,\dots,e_n)\|_{n,\rho} & \stackrel{\mathsf{def}}{=} & \rho(X)([\![e_1]\!]_\rho,\dots,[\![e_n]\!]_\rho) \\
\|\mathbb{1}\|_{n,\rho} & \stackrel{\mathsf{def}}{=} & \{*\} \\
\|\mathsf{Nat}\|_{n,\rho} & \stackrel{\mathsf{def}}{=} & \{\ \underline{\mathsf{n}} \mid n \in \mathbb{N}\ \} \\
\|P \otimes Q\|_{n,\rho} & \stackrel{\mathsf{def}}{=} & \{\ (v,w) \mid v \in \|P\|_{n,\rho} \wedge w \in \|Q\|_{n,\rho}\ \} \\
\|\Downarrow N\|_{n,\rho} & \stackrel{\mathsf{def}}{=} & \{\ \mathsf{thunk}(t) \mid t \in [\![N]\!]_{n,\rho}{}^{\perp_n}\ \} \\
\|\exists x \in S.P\|_{n,\rho} & \stackrel{\mathsf{def}}{=} & \bigcup_{c\in S} \|P\|_{n,\rho[x\leftarrow c]} \\
\|\{e \preceq_S f\} \wedge P\|_{n,\rho} & \stackrel{\mathsf{def}}{=} & \begin{cases} \|P\|_{n,\rho} & \text{if } [\![e]\!]_\rho \preceq_S \rho(f) \\ \varnothing & \text{otherwise} \end{cases}
\end{array}
$$

Negative environment interpretation

$$
\begin{array}{rcl}
[\![P \multimap M]\!]_{n,\rho} & \stackrel{\mathsf{def}}{=} & \{\ \mathsf{a}(v).E \mid v \in \|P\|_{n,\rho}, E \in [\![M]\!]_{n,\rho}\ \} \\
[\![\Uparrow P]\!]_{n,\rho} & \stackrel{\mathsf{def}}{=} & \|P\|_{n,\rho}{}^{\perp_n} \\
[\![\forall x \in S.N]\!]_{n,\rho} & \stackrel{\mathsf{def}}{=} & \bigcup_{c\in S} [\![N]\!]_{n,\rho[x\leftarrow c]} \\
\|\{e \succeq_S f\} \mapsto N\|_{n,\rho} & \stackrel{\mathsf{def}}{=} & \begin{cases} \|N\|_{n,\rho} & \text{if } [\![e]\!]_\rho \succeq_S [\![f]\!]_\rho \\ \varnothing & \text{otherwise} \end{cases}
\end{array}
$$

Negative computation interpretation

$$
\|N\|_{n,\rho} \stackrel{\mathsf{def}}{=} [\![N]\!]_{n,\rho}{}^{\perp_n}
$$

Figure 1: Simple interpretation

If A is closed then the interpretation $\|A\|_{n,\rho}$ does not depend of the valuation ρ. In that case, we will often only write $\|A\|_n$. We finally define the unary realizability relation at level n, that relates closed terms and closed formulas.

$$t \Vdash_n N[\rho] \iff t \in \|N\|_{n,\rho}$$

Similarly,

$$v \Vdash_n P[\rho] \iff v \in \|P\|_{n,\rho}$$

4.1.3 Soundness

We now prove a soundness theorem for the unary realizability interpretation at level 0, which is in the style of the adequacy result of Krivine's realizability.

Definition 110 (Substitution). *A **substitution** σ is a partial function from the set of term variables to the set $\mathbb{V}$, whose domain $dom(\sigma)$ is finite. We note*

$$[x_1 \leftarrow v_1, \dots, x_n \leftarrow v_n]$$

the substitution σ such that $dom(\sigma) = \{x_1, \dots, x_n\}$ and such that $\sigma(x_i) = v_i$.

If σ is a substitution, we denote by $\sigma[x \leftarrow v]$ the substitution obtained from σ by rebinding x to v. If σ_1 is a subtitution and $\sigma_2 = [x_1 \leftarrow v_1, \dots, x_n \leftarrow v_n]$ is another substitution, we denote by

$$\sigma_1, \sigma_2 = (\dots(\sigma_1[x_1 \leftarrow v_1])\dots)[x_n \leftarrow v_n]$$

Now, take a substitution $\sigma = [x_1 \leftarrow v_1, \dots, x_n \leftarrow v_n]$. If t (resp. v) is a *possibly open* computation (resp. value) then we note

$$t[\sigma] = t[v_1/x_1, \dots, v_n/x_n]$$

$$(\text{resp. } v[\sigma] = v[v_1/x_1, \dots, v_n/x_n])$$

Remark 111. *If $\mathsf{FV}(t) \subseteq dom(\sigma)$, then $t[\sigma]$ is closed.*

Definition 112 (Adapted substitution). *Suppose $n \in \mathbb{N}$. Let $\Gamma = x_1 : P_1, \dots, x_n : P_n$ be a typing context and ρ a unary model. We define the set $\|\Gamma\|_{n,\rho}$ as the set of substitutions σ which are **adapted to** Γ **(at level** n**)**, that is:*

- $dom(\sigma) = \{x_1, \dots, x_n\}$
- $\forall i \in \{1, \dots, n\}, \sigma(x_i) \in \|P_i\|_{n,\rho}$

Definition 113 (Adapted model). *Given an inequational theory $\mathcal{E}$, we say that a model ρ is **adpated to** $\mathcal{E}$ and we note $\rho \Vdash \mathcal{E}$ iff the first-order part of ρ is a $\mathcal{E}$-valuation (as defined in Definition 49).*

Definition 114 (Sound judgment). *Suppose that:*

- *$\Gamma = x_1 : P_1, \dots, x_n : P_n$ is a typing context.*
- *$\mathcal{E}$ is a first-order inequational theory.*

*We say that a judgment of the form $\mathcal{E}; \Gamma \vdash t : N$ is **sound (at level n)** iff for every adapted model $\rho \Vdash \mathcal{E}$ and for every adapted substitution $\sigma \in \|\Gamma\|_{n,\rho}$, we have*

$$t[\sigma] \in \|N\|_{n,\rho}$$

*Similarly, we say that a judgment of the form $\mathcal{E}; \Gamma \vdash v : P$ is **sound (at level n)** iff for every adapted model $\rho \Vdash \mathcal{E}$ and for every n-adequate substitution $\sigma \in \|\Gamma\|_{n,\rho}$, we have*

$$v[\sigma] \in \|v\|_{n,\rho}$$

We define what it means for our interpretation to be sound with respect to a given typing rule. A typing rule is given by a sequence of premises J_i (which are typing judgments), side-conditions (SC) on these judgments, and a conclusion K:

$$\frac{J_1 \quad J_2 \quad \dots \quad J_n \qquad SC}{K}\,(rule)$$

Definition 115 (Sound rule)**.** *Suppose R is a typing rule, with $J_1, \dots, J_n$ being its premises judgments and K its conclusion. We say that R is **sound** iff whenever $J_1, \dots, J_n$ are sound and the side-conditions SC are met, the conclusion K is sound as well.*

Remark 116. *If a typing derivation π is built using only sound rules (at level n), then its conclusion is also sound (at level n).*

4.1.3.1 Soundness

We prove that all λ_{LCBPV} typing rules are sound.

Theorem 117. *Let $\mathcal{E}$ be an equational theory and ρ a model such that $\rho \Vdash \mathcal{E}$. Then given two types A, B and $n \in \mathbb{N}$, we have:*

$$\begin{array}{rcl} P \sqsubseteq_{\mathcal{E}} Q & \Longrightarrow & \|P\|_{n,\rho} \subseteq \|Q\|_{n,\rho} \\ N \sqsubseteq_{\mathcal{E}} M & \Longrightarrow & \|N\|_{n,\rho} \subseteq \|M\|_{n,\rho} \end{array}$$

Proof.

The proof is done by induction on the subtyping judgment defined in Figure 1. To prove the second statement, it is enough to prove that $N \sqsubseteq_{\mathcal{E}} M$ implies $[\![M]\!]_{n,\rho} \subseteq [\![N]\!]_{n,\rho}$ since $(.)^{\perp_n}$ is contravariant.

- **Reflexivity :** The base case $A \sqsubseteq_{\mathcal{E}} A$ is immediate.
- **Transitivity :** This is immediate by transitivity of the set inclusion $\subseteq$.
- **Predicate variables :** Suppose that $e \cong_S f$. Then it means that for any first-order valuation ρ we have $[\![e]\!]_\rho = [\![f]\!]_\rho$. It is then also true for any unary model ρ, and hence

$$\|X(e)\|_{n,\rho} = \|X(f)\|_{n,\rho}$$

- **Tensor :** Suppose that $P \sqsubseteq_{\mathcal{E}} P'$ and $Q \sqsubseteq_{\mathcal{E}} Q'$. Then, by induction we have that $\|P\|_{n,\rho} \subseteq \|P'\|_{n,\rho}$ and $\|Q\|_{n,\rho} \subseteq \|Q'\|_{n,\rho}$. It means that if $(v,w) \in \|P \otimes Q\|_{n,\rho}$, we have $v \in \|P'\|_{n,\rho}$ and $w \in \|Q'\|_{n,\rho}$. Therefore $(v,w) \in \|P' \otimes Q'\|_{n,rho}$.
- **Positive shift :** Suppose that $N \sqsubseteq_{\mathcal{E}} M$. Then we know that $[\![M]\!]_{n,\rho} \subseteq [\![N]\!]_{n,\rho}$. We have ${[\![N]\!]_{n,\rho}}^{\perp_n} \subseteq {[\![M]\!]_{n,\rho}}^{\perp_n}$ and hence the result $\|\Downarrow N\|_{n,\rho} \subseteq \|\Downarrow M\|_{n,\rho}$.
- **Positive shift :** Suppose that $P \sqsubseteq_{\mathcal{E}} Q$. Then $[\![\Uparrow Q]\!]_{n,\rho} = {\|Q\|_{n,\rho}}^{\perp_n} \subseteq {\|P\|_{n,\rho}}^{\perp_n} = [\![\Uparrow P]\!]_{n,\rho}$.
- **Linear implication :** Suppose that $P \sqsubseteq_{\mathcal{E}} Q$ and $N \sqsubseteq_{\mathcal{E}} M$. Then we have by induction that $\|P\|_{n,\rho} \subseteq \|Q\|_{n,\rho}$ and $[\![M]\!]_{n,\rho} \subseteq [\![N]\!]_{n,\rho}$. Hence we obtain that $[\![P \multimap M]\!]_{n,\rho} = \{\ \mathsf{a}(v).E \mid v \in \|P\|_{n,\rho} \wedge E \in [\![M]\!]_{n,\rho}\ \} \subseteq \{\ \mathsf{a}(v).E \mid v \in \|Q\|_{n,\rho} \wedge E \in [\![N]\!]_{n,\rho}\ \} = [\![Q \multimap N]\!]_{n,\rho}$.
- **Inequational implication :** Suppose that $e' \preceq_{\mathcal{E}} e$, $f \preceq_{\mathcal{E}} f'$ and $N \sqsubseteq_{\mathcal{E}, e \succeq_S f} M$. Then two cases are possible:
  - If $\neg([\![f']\!]_\rho \preceq_S [\![e']\!]_\rho)$, then $[\![\{e' \succeq_S f'\} \mapsto M]\!]_{n,\rho} = \varnothing \subseteq [\![\{e \succeq_S f\} \mapsto N]\!]_{n,\rho}$.
  - If $[\![f']\!]_\rho \preceq_S [\![e']\!]_\rho$ then $[\![\{e' \succeq_S f'\} \mapsto M]\!]_{n,\rho} = [\![M]\!]_{n,\rho}$. But since $[\![e']\!]_\rho \preceq_S [\![e]\!]_\rho$ and $[\![f]\!]_\rho \preceq_S [\![f']\!]_\rho$, we have $[\![\{e \succeq_S f\} \mapsto N]\!]_{n,\rho} = [\![N]\!]_{n,\rho}$. Finally, since $[\![f]\!]_\rho \preceq_S [\![e]\!]_\rho$, it means that ρ is adapted to $(\mathcal{E}, e \succeq_S f)$. Hence we conclude by induction hypothesis:
$$[\![M]\!]_{n,\rho} \subseteq [\![N]\!]_{n,\rho}$$
- **Inequational conjunction :** Suppose that $e' \preceq_{\mathcal{E}} e$, $f \preceq_{\mathcal{E}} f'$ and $P \sqsubseteq_{\mathcal{E}, e \preceq_S f} Q$. Then two cases are possible:
  - If $\neg([\![e]\!]_\rho \preceq_S [\![f]\!]_\rho)$, then $\|\{e \preceq_S f\} \wedge P\|_{n,\rho} = \varnothing \subseteq \|\{e' \preceq_S f'\} \wedge Q\|_{n,\rho}$.
  - If $[\![e]\!]_\rho \preceq_S [\![f]\!]_\rho$ then $\|\{e \preceq_S f\} \wedge P\|_{n,\rho} = \|P\|_{n,\rho}$. But since $[\![e']\!]_\rho \preceq_S [\![e]\!]_\rho$ and $[\![f]\!]_\rho \preceq_S [\![f']\!]_\rho$, we have $\|\{e' \preceq_S f'\} \wedge Q\|_{n,\rho} = \|Q\|_{n,\rho}$. Finally, since $[\![e]\!]_\rho \preceq_S [\![f]\!]_\rho$, it means that ρ is adapted to $(\mathcal{E}, e \preceq_S f)$. Hence we have by induction hypothesis:
$$\|P\|_{n,\rho} \subseteq \|Q\|_{n,\rho}$$
  We conclude that
$$\|\{e \preceq_S f\} \wedge P\|_{n,\rho} \subseteq \|\{e' \preceq_S f'\} \wedge Q\|_{n,\rho}$$
- **Quantifiers :** Here again we only prove the negative case, the positive one being similar. Suppose that $N \sqsubseteq_{\mathcal{E}} M$. We then have by applying multiple times the induction hypothesis:
$$\begin{array}{rcl} [\![\exists x \in S.M]\!]_{n,\rho} & = & \bigcup_{s \in S} \underbrace{[\![M]\!]_{n,\rho[x \leftarrow s]}}_{\subseteq [\![N]\!]_{n,\rho[x \leftarrow s]}} \\ & \subseteq & \bigcup_{s \in S} [\![N]\!]_{n,\rho[x \leftarrow s]} \end{array}$$

□

Theorem 118 (Soundness theorem)**.** *All the rules of* λ_{LCBPV} *are sound at level* 0.

Proof. The proof is done by examining each rule. Here $\mathcal{E}$ is fixed and ρ is a model such that $\rho \Vdash \mathcal{E}$.

- **(Var)** We want to show that $\mathcal{E}; \Gamma, x : P \vdash_0 x : P$ is sound. But it is immediate since if $\sigma \in \|\Gamma, x : P\|_{0,\rho}$ then $\sigma(x) \in \|P\|_{0,\rho}$, hence the result.

- **(Pos. shift)** We suppose that $\mathcal{E}; \Gamma \vdash_0 t : N$ is sound and want to conclude that $\mathcal{E}; \Gamma \vdash_0 \mathsf{thunk}(t) : \Downarrow N$ is. Let $\sigma \in \|\Gamma\|_{0,\rho}$. Then by hypothesis we know that

$$t[\sigma] \in \|N\|_{0,\rho}$$

Hence, by definition we immediately conclude that

$$\mathsf{thunk}(t[\sigma]) \in \|\Downarrow N\|_{0,\rho}$$

- **(Pos. shift elim.)** We suppose that $\mathcal{E}; \Gamma \vdash_0 v : \Downarrow N$ is sound and want to conclude that $\mathcal{E}; \Gamma \vdash_0 \mathsf{force}(v) : N$ is. Let $\sigma \in \|\Gamma\|_{0,\rho}$. Then by hypothesis we know that:

$$v[\sigma] \in \|\Downarrow N\|_{0,\rho}$$

First, by definition of $\|\Downarrow N\|_{0,\rho}$, we have $v[\sigma] = \mathsf{thunk}(t)$ with

$$t \in \|N\|_{0,\rho} \tag{1.1}$$

Let $E \in [\![N]\!]_{0,\rho}$. We need to show that

$$\mathsf{force}(\mathsf{thunk}(t)) = \mathsf{force}(v)[\sigma] \bot_0 E$$

Because of (2.21), we have immediately that $\langle E, \epsilon \rangle_t \perp\!\!\!\perp$. Since

$$\langle \mathsf{force}(\mathsf{thunk}(t)), E \rangle_0 \quad \to^{\mathsf{V}} \quad \langle t, E \rangle_0 \in \perp\!\!\!\perp$$

we conclude by saturation of the pole.

- **(Neg. shift)** We suppose that $\mathcal{E}; \Gamma \vdash_0 v : P$ is sound and want to conclude that $\mathcal{E}; \Gamma \vdash_0 \mathsf{ret}(v) : \Uparrow P$ is. Let $\sigma \in \|\Gamma\|_{0,\rho}$. Then by hypothesis we know that

$$v[\sigma] \in \|P\|_{0,\rho} \tag{1.2}$$

We need to show that $\mathsf{ret}(v[\sigma]) \in [\![\Uparrow P]\!]_{0,\rho}{}^{\bot_0}$. Let $E \in [\![\Uparrow P]\!]_{0,\rho}$. Because of (2.22) and by definition of $[\![\Uparrow P]\!]_{0,\rho}$ we conclude immediately.

- **(Neg. shift elim.)** We suppose that the two following judgments are sound:

$$\mathcal{E}; \Gamma \vdash_0 t : \Uparrow P \tag{1.3}$$

$$\mathcal{E}; \Delta, x : P \vdash_0 u : M \tag{1.4}$$

We want to show that $\mathcal{E}; \Gamma, \Delta \vdash_0 t \,\mathsf{to}\, x.u : M$ is sound. Let $\sigma \in \|\Gamma, \Delta\|_{0,\rho}$. Then because Γ and Δ are disjoint, it is clear that there exists two substitutions σ_1, σ_2 such that

  - $\sigma = \sigma_1, \sigma_2$
  - $\sigma_1 \in \|\Gamma\|_{0,\rho}$
  - $\sigma_2 \in \|\Delta\|_{0,\rho}$

Hence, by hypothesis we have

$$t[\sigma_1] \in \|\Uparrow P\|_{0,\rho} \tag{1.5}$$

$$\forall v \in \|P\|_{0,\rho}, u[\sigma_2][v/x] \in \|M\|_{0,\rho} \tag{1.6}$$

Let's show that

$$t[\sigma_1] \text{ to } x.u[\sigma_2] \in \|M\|_{0,\rho}$$

Suppose that $E \in [\![M]\!]_{0,\rho}$. Then, we can show the following intermediate result:

$$\mathsf{f}(x.u[\sigma_2]).E \in \|P\|_{0,\rho}{}^{\perp_0}$$

Indeed, if $v \in \|P\|_{0,\rho}$ we have

$$\langle \mathsf{ret}(v), \mathsf{f}(x.u[\sigma_2]).E\rangle_0 \to^{\mathsf{V}} \langle u[\sigma_2][v/x], E\rangle_0 \in \perp\!\!\!\perp$$

Hence, we obtain the result by saturation of $\perp\!\!\!\perp$. But since $[\![\Uparrow P]\!]_{0,\rho} = \|P\|_{0,\rho}{}^{\perp_0}$ and because

$$\langle t[\sigma_1] \text{ to } x.u[\sigma_2], E\rangle_0 \to^{\mathsf{V}} \langle t[\sigma_1], \mathsf{f}(x.u[\sigma_2]).E\rangle_0 \in \perp\!\!\!\perp$$

we deduce the conclusion by saturation of $\perp\!\!\!\perp$.

- **($\multimap$ intro.)** We suppose that $\mathcal{E}; \Gamma, x : P \vdash_0 t : N$ is sound and want to conclude that $\mathcal{E}; \Gamma \vdash_0 \lambda x.t : P \multimap N$ is. Let $\sigma \in \|\Gamma\|_{0,\rho}$. Let $v \in \|P\|_{0,\rho}$ and $E \in [\![N]\!]_{0,\rho}$. We need to show that $\lambda x.t[\sigma] \perp_0 \mathsf{a}(v).E$. We know that $\sigma[v/x] \in \|\Gamma, x : P\|_{0,\rho}$, hence by hypothesis that $t[\sigma][v/x] \in \|N\|_{0,\rho}$. This implies that
$$\langle t[\sigma][v/x], E\rangle_0 \in \perp\!\!\!\perp$$
On the other hand, we have
$$\langle \lambda x.t[\sigma], \mathsf{a}(v).E\rangle_0 \to^{\mathsf{V}} \langle t[\sigma][v/x], E\rangle_0$$
Therefore, by saturation of $\perp\!\!\!\perp$, we conclude that $\lambda x.t[\sigma] \perp \mathsf{a}(v).E$.
- **($\multimap$ elim.)** We suppose that the two following judgments are sound:
$$\mathcal{E}; \Gamma \vdash_0 t : P \multimap N \tag{1.7}$$
$$\mathcal{E}; \Delta \vdash_0 v : P \tag{1.8}$$
We want to show that $\mathcal{E}; \Gamma, \Delta \vdash_0 (t)v : N$ is sound. Let $\sigma \in \|\Gamma, \Delta\|_{0,\rho}$. Then because Γ and Δ are disjoint, it is clear that there exists two substitutions σ_1, σ_2 such that
  - $\sigma = \sigma_1, \sigma_2$
  - $\sigma_1 \in \|\Gamma\|_{0,\rho}$
  - $\sigma_2 \in \|\Delta\|_{0,\rho}$

  Hence, by hypothesis we have
$$t[\sigma_1] \in \|P \multimap N\|_{0,\rho} \tag{1.9}$$
$$v[\sigma_2] \in \|P\|_{0,\rho} \tag{1.10}$$
Let's show that
$$t[\sigma_1]v[\sigma_2] \in \|M\|_{0,\rho}$$
Suppose that $E \in [\![M]\!]_{0,\rho}$. Then because of (2.34),
$$\mathsf{a}(v[\sigma_2]).E \in [\![P \multimap N]\!]_{0,\rho}$$

Hence, we know by (2.29) that

$$\langle t[\sigma_1], \mathsf{a}(v[\sigma_2]).E\rangle_0 \in \perp\!\!\!\perp$$

Finally, we conclude by saturation of $\perp\!\!\!\perp$ that

$$\langle (t[\sigma_1])v[\sigma_2], E\rangle_0 \in \perp\!\!\!\perp$$

- **($\otimes$ intro.)** We suppose that $\mathcal{E}; \Gamma \vdash_0 v : P$ and $\mathcal{E}; \Delta \vdash_0 w : Q$ are sound and want to conclude that $\mathcal{E}; \Gamma \vdash_0 (v, w) : P \otimes Q$ is. Let $\sigma \in \|\Gamma, \Delta\|_{0,\rho}$. Then because Γ and Δ are disjoint, it is clear that there exists two substitutions σ_1, σ_2 such that
  - $\sigma = \sigma_1, \sigma_2$
  - $\sigma_1 \in \|\Gamma\|_{0,\rho}$
  - $\sigma_2 \in \|\Delta\|_{0,\rho}$

  Hence, by hypothesis we have
  $$v[\sigma_1] \in \|P\|_{0,\rho}$$
  and
  $$w[\sigma_2] \in \|Q\|_{0,\rho}$$
  Hence, clearly because $(v, w)[\sigma] = (v[\sigma_1], w[\sigma_2])$ we have
  $$(v, w)[\sigma] \in \|P \otimes Q\|_{0,\rho}$$

- **($\otimes$ elim.)** We suppose that the two following judgments are sound:
  $$\mathcal{E}; \Gamma, x : P, y : Q \vdash_0 t : N \tag{1.11}$$
  $$\mathcal{E}; \Delta \vdash_0 v : P \otimes Q \tag{1.12}$$
  We want to show that $\mathcal{E}; \Gamma, \Delta \vdash_0 \mathsf{let}\,(x, y) = v\,\mathsf{in}\,t : N$ is sound. Let $\sigma \in \|\Gamma, \Delta\|_{0,\rho}$. Then because Γ and Δ are disjoint, it is clear that there exists two substitutions σ_1, σ_2 such that
  - $\sigma = \sigma_1, \sigma_2$
  - $\sigma_1 \in \|\Gamma\|_{0,\rho}$
  - $\sigma_2 \in \|\Delta\|_{0,\rho}$

  Hence, by hypothesis we have
  $$\forall v \in \|P\|_{0,\rho}, \forall w \in \|Q\|_{0,\rho}, t[\sigma_1][v/x, w/y] \in \|N\|_{0,\rho} \tag{1.13}$$
  and
  $$v[\sigma_2] \in \|P \otimes Q\|_{0,\rho} \tag{1.14}$$
  By definition of the interpretation of $P \otimes Q$, we have $v[\sigma_2] = (w, w')$ such that $w \in \|P\|_{0,\rho}$ and $w' \in \|Q\|_{0,\rho}$. Hence, by (2.33) we have
  $$t[\sigma_1][w/x, w'/y] \in \|N\|_{0,\rho}$$
  Now, for any $E \in [\![N]\!]_{0,\rho}$ we have
  $$\langle \mathsf{let}\,(x, y) = v[\sigma_2]\,\mathsf{in}\,t[\sigma_1], E\rangle_0 \to^{\mathsf{V}} \langle t[\sigma_1][w/x, w'/y], E\rangle_0$$
  But since $t[\sigma_1][w/x, w'/y] \perp E$, we have by saturation that
  $$\langle \mathsf{let}\,(x, y) = v[\sigma_2]\,\mathsf{in}\,t[\sigma_1], E\rangle_0 \in \perp\!\!\!\perp$$
  Which exactly means that $\mathsf{let}\,(x, y) = v[\sigma_2]\,\mathsf{in}\,t[\sigma_1] \in \|N\|_{0,\rho}$.

- **(unit)** We want to show that $\mathcal{E}; \Gamma \vdash_0 * : \mathbb{1}$ is sound. This is immediate by definition of $\|\mathbb{1}\|_{0,\rho}$.
- **(unit elim.)** We want to show that $\mathcal{E}; \Gamma, \Delta \vdash_0 \mathsf{let} * = v \,\mathsf{in}\, t : N$ is sound provided that $\mathcal{E}; \Gamma \vdash_0 v : \mathbb{1}$ and $\mathcal{E}; \Delta \vdash_0 t : N$ are sound. Let ρ be an adapted model and let $\sigma \in \|\Gamma, \Delta\|_{0,\rho}$. Then because Γ and Δ are disjoint, it is clear that there exists two substitutions σ_1, σ_2 such that
  - $\sigma = \sigma_1, \sigma_2$
  - $\sigma_1 \in \|\Gamma\|_{0,\rho}$
  - $\sigma_2 \in \|\Delta\|_{0,\rho}$

  Hence, by hypothesis we have
  $$v[\sigma_1] \in \|P\|_{0,\rho} \tag{1.15}$$
  and
  $$t[\sigma_2] \in \|N\|_{0,\rho} \tag{1.16}$$
  But by definition we know that $v[\sigma_1] = *$. Let $E \in [\![N]\!]_{0,\rho}$. We want to show that
  $$\langle \mathsf{let} * = * \,\mathsf{in}\, t[\sigma_2], E\rangle_0 \in \perp\!\!\!\perp$$
  This is immediate by $\to$-saturation of $\perp\!\!\!\perp$ since
  $$\langle t[\sigma_2], E\rangle_0 \in \perp\!\!\!\perp$$
- **(prim. integers)** We want to show that $\mathcal{E}; \Gamma \vdash_0 \underline{\mathsf{n}} : \mathsf{Nat}$ is sound. This is immediate by definition of $\|\mathsf{Nat}\|_{0,\rho}$.
- **(succ)** We want to show that $\mathcal{E}; \Gamma \vdash_0 \underline{\mathsf{s}}(v) : \mathsf{Nat}$ is sound provided that $\mathcal{E}; \Gamma \vdash_0 v : \mathsf{Nat}$ is sound. Let $\sigma \in \|\Gamma\|_{0,\rho}$. We have by hypothesis that
  $$v[\sigma] \in \|\mathsf{Nat}\|_{0,\rho}$$
  Hence, by definition of the interpretation, there is $n \in \mathbb{N}$ such that $v[\sigma] = \underline{\mathsf{n}}$. Therefore, since $\underline{\mathsf{s}}(\underline{\mathsf{n}}) = \underline{\mathsf{n+1}}$, we have
  $$\underline{\mathsf{s}}(v)[\sigma] = \underline{\mathsf{n+1}} \in \|\mathsf{Nat}\|_{0,\rho}$$
- **(Case)** We suppose that $\mathcal{E}; \Gamma \vdash_0 v : \mathsf{Nat}$, $\mathcal{E}; \Delta, x : \mathsf{Nat} \vdash_0 t_1 : N$ and $\mathcal{E}; \Delta, x : \mathsf{Nat} \vdash_0 t_2 : N$ are sound. We want to prove that the following judgment is sound too:
  $$\mathcal{E}; \Gamma, \Delta \vdash_0 \mathsf{case}\, v \,\mathsf{of}\, x.t_1 \,\|\, x.t_2 : N$$
  Let $\sigma_1 \in \|\Gamma\|_{0,\rho}$ and $\sigma_2 \in \|\Delta\|_{0,\rho}$. We first know that
  $$v[\sigma_1] \in \|\mathsf{Nat}\|_{0,\rho}$$
  Hence, there is $n \in \mathbb{N}$ such that $v[\sigma_1] = \underline{\mathsf{n}}$. Since $\sigma_2[x \leftarrow v[\sigma_1]] \in \|\Delta, x : \mathsf{Nat}\|_{0,\rho}$ we also know that
  $$t_1[\sigma_2][v[\sigma_1]/x] \in \|N\|_{0,\rho} \tag{1.17}$$
  $$t_2[\sigma_2][v[\sigma_1]/x] \in \|N\|_{0,\rho} \tag{1.18}$$
  Let $E \in [\![N]\!]_{0,\rho}$. Depending of wether $n = 0$ or $n > 0$ we have
  $$\langle \mathsf{case}\, v[\sigma_1] \,\mathsf{of}\, x.t_1[\sigma_2] \,\|\, x.t_2[\sigma_2], E\rangle_0 \to^{\mathsf{V}} \langle t_i[\sigma_2][v[\sigma]/x], E\rangle_0$$
  We have $\langle t_i[\sigma_2][v[\sigma_1]], E\rangle_0 \in \perp\!\!\!\perp$ because of (2.35) and (2.36). Hence by saturation of $\perp\!\!\!\perp$ we obtain
  $$\langle \mathsf{case}\, v[\sigma_1] \,\mathsf{of}\, x.t_1[\sigma_2] \,\|\, x.t_2[\sigma_2], E\rangle_0 \in \perp\!\!\!\perp$$

- **($\forall$ intro.)** We want to show that given that ι^S does not appear free in Γ nor in $\mathcal{E}$ and that $\mathcal{E};\Gamma \vdash_0 t : N$ is sound we obtain that $\mathcal{E};\Gamma \vdash_0 v : \forall \iota \in S.N$ is sound too. Let $\rho \Vdash \mathcal{E}$ and $\sigma \in \|\Gamma\|_{0,\rho}$. Let $E \in [\![\forall \iota \in \sigma.N]\!]_{0,\rho} = \bigcup_{a\in S} [\![N]\!]_{0,\rho[\iota\leftarrow a]}$. We want to show that $t[\sigma] \perp E$. We know there exists $a \in S$ such that $E \in [\![N]\!]_{0,\rho[\iota^S\leftarrow a]}$. We have $\rho[\iota^S \leftarrow a] \Vdash \mathcal{E}$ since ι^S does not appear free in $\mathcal{E}$. Moreover $\sigma \in \|\Gamma\|_{0,\rho[\iota\leftarrow a]}$ since ι^S does not appear free in Γ. Therefore, we know by hypothesis that
$$t[\sigma] \in [\![N]\!]_{0,\rho[\iota\leftarrow a]}{}^{\perp_0}$$
That concludes the proof.

- **($\forall$ elim.)** We want to show that given that $\mathcal{E};\Gamma \vdash_0 t : \forall \iota \in S.N$ is sound then for any $a \in S$, $\mathcal{E};\Gamma \vdash_0 t : N[a/\iota^S]$ is sound too. Let $\rho \Vdash \mathcal{E}$ and $\sigma \in \|\Gamma\|_{0,\rho}$ and $a \in S$. We know by hypothesis that
$$\begin{array}{rcl} t & \in & (\bigcup_{b\in S} [\![N]\!]_{0,\rho[\iota^S\leftarrow b]})^{\perp_0} \\ & = & \bigcap_{b\in S} ([\![N]\!]_{0,\rho[\iota^S\leftarrow b]})^{\perp_0} \\ & \subseteq & [\![N]\!]_{0,\rho[\iota^S\leftarrow a]}{}^{\perp_0} \\ & = & [\![N[a/\iota^S]]\!]_{0,\rho}{}^{\perp_0} \end{array}$$
This concludes the proof.

- **($\exists$ intro.)** We want to show that given that $\mathcal{E};\Gamma \vdash_0 v : P[e/\iota^S]$ is sound we obtain that $\mathcal{E};\Gamma \vdash_0 v : \exists \iota \in S.P$ is sound too. Let $\rho \Vdash \mathcal{E}$ and $\sigma \in \|\Gamma\|_{0,\rho}$. We know by hypothesis that
$$v[\sigma] \in \|P[e/\iota^S]\|_{0,\rho}$$
But $\|P[e/\iota^S]\|_{0,\rho} = \|P\|_{0,\rho[\iota^S\leftarrow\rho(e)]}$ Hence we obtain immediately that:
$$v[\sigma] \in \bigcup_{a\in S} \|P\|_{0,\rho[\iota^S\leftarrow a]} = \|\exists \iota \in S.P\|_{0,\rho}$$

- **($\exists$ elim.)** We suppose that $\mathcal{E};\Gamma \vdash_0 v : \exists \iota \in S.P$ and $\mathcal{E};\Delta, x : P[\kappa/\iota] \vdash_0 t : N$ are sound, with κ not appearing free in $\mathcal{E}$, Δ or N. We want to show that
$$\mathcal{E};\Gamma,\Delta \vdash_0 t[v/x] : N$$
Let $\rho \Vdash \mathcal{E}$ and let $\sigma \in \|\Gamma,\Delta\|_{0,\rho}$. We have $\sigma = \sigma_1,\sigma_2$ such that $\sigma_1 \in \|\Gamma\|_{0,\rho}$ and $\sigma_2 \in \|\Delta\|_{0,\rho}$. We know by hypothesis that
$$v[\sigma_1] \in \bigcup_{a\in S} \|P\|_{0,\rho[\iota^S\leftarrow a]}$$
So there exists $a \in S$ such that
$$v[\sigma_1] \in \|P\|_{0,\rho[\iota^S\leftarrow a]} \tag{1.19}$$
On the other hand we observe that since κ^S does not appear free in $\mathcal{E}$ and because $\rho \Vdash \mathcal{E}$, we have:
$$\rho[\kappa^S \leftarrow a] \Vdash \mathcal{E}$$
Moreover, since κ^S does not appear free in Γ we also have
$$\sigma_2 \in \|\Delta\|_{0,\rho[\kappa^S\leftarrow a]}$$
Finally, (2.37) implies that
$$v[\sigma_1] \in \|P[\kappa/\iota]\|_{0,\rho[\kappa^S\leftarrow a]}$$

So, by the second hypothesis we obtain that

$$t[\sigma_2][v[\sigma_1]/x] \in \|N\|_{0,\rho[\kappa^S \leftarrow a]}$$

But since κ does not appear free in N, then

$$\|N\|_{0,\rho[\kappa^S \leftarrow a]} = \|N\|_{0,\rho}$$

- **(Inequ. conj. intro.)** We want to show that provided that $\mathcal{E};\Gamma \vdash_0 v : P$ is sound and that $f \preceq_{\mathcal{E}} e$, we have $\mathcal{E};\Gamma \vdash_0 v : \{f \preceq_S e\} \wedge P$ is sound too. Let $\rho \Vdash \mathcal{E}$ and let $\sigma \in \|\Gamma\|_{0,\rho}$. We know that $[\![f]\!]_\rho \preceq_S [\![e]\!]_\rho$ by hypothesis. Hence, by the other hypothesis we have that $v[\sigma] \in \|P\|_{0,\rho} = \|\{f \preceq_S e\} \wedge P\|_{0,\rho}$.
- **(Inequ. imp. intro.)** We want to show that provided that $\mathcal{E}, e \succeq_S f;\Gamma \vdash_0 t : N$ is sound, we have $\mathcal{E};\Gamma \vdash_0 t : \{e \succeq_S f\} \mapsto N$ is sound too. Let $\rho \Vdash \mathcal{E}$ and let $\sigma \in \|\Gamma\|_{0,\rho}$. There are two possibilities:
  - If $[\![e]\!]_\rho \succeq_S [\![f]\!]_\rho$ then $\rho \Vdash \mathcal{E}, e \succeq_S f$. Hence, by hypothesis we have that $t[\sigma] \in [\![N]\!]_{0,\rho}{}^{\perp_0} = [\![\{e \succeq_S f\} \mapsto N]\!]_{0,\rho}{}^{\perp_0}$.
  - If $\neg([\![e]\!]_\rho \succeq_S [\![f]\!]_\rho)$ then by hypothesis we have that $t[\sigma] \in [\![\{e \succeq_S f\} \mapsto N]\!]_{0,\rho}{}^{\perp_0} = \varnothing^{\perp_0} = \mathbb{P}$.
- **(Inequ. conj. elim.)** We want to show that provided that $\mathcal{E};\Gamma \vdash_0 v : \{e \preceq_S e\} \wedge P$ is sound, $\mathcal{E};\Gamma \vdash_0 v : P$ is sound too. Let $\rho \Vdash \mathcal{E}$ and let $\sigma \in \|\Gamma\|_{0,\rho}$. We conclude easily because $\|\{e \preceq_S e\} \wedge P\|_{0,\rho} = \|P\|_{0,\rho}$.
- **(Inequ. imp. elim.)** We want to show that provided that $\mathcal{E};\Gamma \vdash_0 t : \{e \succeq_S e\} \mapsto N$ is sound, $\mathcal{E};\Gamma \vdash_0 t : N$ is sound too. Let $\rho \Vdash \mathcal{E}$ and let $\sigma \in \|\Gamma\|_{0,\rho}$. We conclude easily because $[\![\{e \succeq_S e\} \mapsto N]\!]_{0,\rho} = [\![N]\!]_{0,\rho}$.
- **(Subtyping)** Finally, we want to show that if $A \sqsubseteq_{\mathcal{E}} B$ and $\mathcal{E};\Gamma \vdash_0 a : A$ is sound then so is $\mathcal{E};\Gamma \vdash_0 a : B$. This is a direct consequence of Theorem 117.

□

4.2 | n-Monitoring algebras

Now that we have defined the unary realizability model based on the Monitoring Abstract Machine, we can turn our attention to the central notion of this thesis, namely the **Monitoring Algebra** or **MA**. A monitoring algebra is a structure that combine the forcing program transformation with the unary realizability of the previous section. The algebraic nature of the forcing layer will allow for complex combinations and constructions on monitoring algebras, thus giving birth to models of elaborated programming languages. In the rest of this thesis, we suppose having a fixed a unary pole $\perp\!\!\!\perp$ as defined in Section 4.1.

4.2.1 Definition

Definition 119 (n-Monitoring Algebra)**.** *Let $n \in \mathbb{N}$. A n-**monitoring algebra** $\mathcal{A} = (|\mathcal{A}|, \mathcal{C}_\mathcal{A})$ is a structure such that:*

- *$|\mathcal{A}|$ is a forcing monoid, called the **carrier** of the algebra.*
- *$\mathcal{C}_\mathcal{A} : |\mathcal{A}| \to \mathcal{P}(\mathbb{V}^n)$ is the **test function**, which maps an element p of the forcing monoid $|\mathcal{A}|$ to a set $\mathcal{C}_\mathcal{A}(p)$ of n-uples of values. The function $\mathcal{C}_\mathcal{A}$ is moreover required to be decreasing, that is:*

$$\forall p, q \in |\mathcal{A}|, p \preceq q \text{ implies } \mathcal{C}_\mathcal{A}(q) \subseteq \mathcal{C}_\mathcal{A}(p)$$

*The natural number n is called **the level of** $\mathcal{A}$.*

Remark 120. *The test function $\mathcal{C}_\mathcal{A}$ is somehow similar to what is called the **erasure map** in the Views framework [DYBG+ 13] and in recent separation logic models such as [SBP13].*

In what follows, we will denote n-monitoring algebras (or n-**MA**s for short) with capital letters $\mathcal{A}, \mathcal{B}, \mathcal{D}$, etc.

4.2.2 $\mathcal{A}$-orthogonality

In what follows, we suppose that $\mathcal{A}$ is a fixed n-**MA**. In the simple realizability model given in Section 4.1, the realizers are: values, computations or environments. In monitoring algebras, the actors of the realizability are **annotated terms**: *i.e.* a term associated to an element of the forcing monoid $|\mathcal{A}|$. This element is here to give an information on the execution of the term: the time complexity, a quantity of resources the term can use, security credentials, etc.

Definition 121 ($\mathcal{A}$-terms)**.**

- *A $\mathcal{A}$-**value** is an element (v, p) of $\mathbb{V} \times |\mathcal{A}|$. The set of $\mathcal{A}$ values is denoted by $\mathbb{V}_\mathcal{A}$.*
- *A $\mathcal{A}$-**environment** is an element (E, p) of $\mathbb{E} \times |\mathcal{A}|$. The set of $\mathcal{A}$-environments is denoted by $\mathbb{E}_\mathcal{A}$.*
- *Finally, a $\mathcal{A}$-**computation** is an element (t, p) of $\mathbb{P} \times |\mathcal{A}|$. The set of $\mathcal{A}$-computations is denoted by $\mathbb{P}_\mathcal{A}$.*

In the unary realizability case, we considered a family of orthogonality relations that relate computations (resp. values) to environments. Each n-**MA** $\mathcal{A}$ induces a similar family of orthogonality relations. But instead, they relate $\mathcal{A}$-computations (resp. $\mathcal{A}$-values) and $\mathcal{A}$-environments. These relations are obtained by extending the simple orthogonality inherited from the pole $\perp\!\!\!\perp$, and by incorporating the forcing conditions using the test function $\mathcal{C}_\mathcal{A}$. These forcing conditions refine the notion of *good interaction* defined by $\perp\!\!\!\perp$.

Definition 122 ($\mathcal{A}$-orthogonality)**.** *Let $\mathcal{A}$ be a n-**MA**. We define two families of orthogonality relations indexed by $\mathbb{N}$:*

- *For each $k \in \mathbb{N}$, a relation $\bot_{\mathcal{A},k} \subseteq \mathbb{P}_{\mathcal{A}} \times \mathbb{E}_{\mathcal{A}}$ defined as:*

$$(t,p)\bot_{\mathcal{A},k}(E,q) \iff \forall (v_1,\dots,v_n) \in \mathcal{C}_{\mathcal{A}}(p+q), \langle t, \mathsf{a}(v_1).\dots.\mathsf{a}(v_n).E\rangle_k \in \perp\!\!\!\perp$$

- *For each $k \in \mathbb{N}$, a relation $\bot_{\mathcal{A},k} \subseteq \mathbb{E}_{\mathcal{A}} \times \mathbb{V}_{\mathcal{A}}$ defined as:*

$$(E,q)\bot_{\mathcal{A},k}(v,p) \iff \forall (v_1,\dots,v_n) \in \mathcal{C}_{\mathcal{A}}(p+q), \langle \uparrow E, (v,(v_n,\dots,v_1))\rangle_k \in \perp\!\!\!\perp$$

*We denote these two relations by the same symbol $\bot_{\mathcal{A},k}$: their domain being disjoint, there is no possible confusion. We call both these relations $\mathcal{A}$-**orthogonality at level** k.*

This definition makes clear the role of the forcing condition: it parametrises the orthogonality with a set of tests (the values given by the test function $\mathcal{C}_{\mathcal{A}}$) that the configuration has to pass. Of course, the presence of those values only affects configurations that will manipulate the store during its execution, using the observation $(\!|.|\!)^{\alpha}_{k}$ or new primitives.

Notation 123. *In addition, we define the following set that we will sometimes find convenient to use:*

$$\perp\!\!\!\perp_{\mathcal{A},k} \stackrel{\text{def}}{=} \{\ (t,E,p) \mid \forall (v_1,\dots,v_n) \in \mathcal{C}_{\mathcal{A}}(p), \langle t, \mathsf{a}(v_1).\dots.\mathsf{a}(v_n).E\rangle_k \in \perp\!\!\!\perp\ \}$$

*This new set plays the same role as the pole in the simple realizability model and we call it the $\mathcal{A}$-**pole at level** k. Using it, we can reformulate the orthogonality between computations and environments as follows:*

$$(t,p)\bot_{\mathcal{A},k}(E,q) \Leftrightarrow (t,E,p+q) \in \perp\!\!\!\perp_{\mathcal{A},k}$$

Remark 124. *It is possible to express the $\mathcal{A}$-orthogonality using the simple orthogonality as follows:*

$$(t,p)\bot_{\mathcal{A},k}(E,q) \Leftrightarrow \forall (v_1,\dots,v_n) \in \mathcal{C}_{\mathcal{A}}(p+q), t\bot_k \mathsf{a}(v_1).\dots.\mathsf{a}(v_n).E$$

These two orthogonality relations can in turn be lifted to two operations on sets of $\mathcal{A}$-values and $\mathcal{A}$-environments respectively. If $X \subseteq \mathbb{V}_{\mathcal{A}}$, we define

$$X^{\bot_{\mathcal{A},k}} \stackrel{\text{def}}{=} \{\ (E,q) \in \mathbb{E}_{\mathcal{A}} \mid \forall (v,p) \in X, (E,q)\bot_{\mathcal{A},k}(v,p)\ \}$$

Similarly if we take $Y \subseteq \mathbb{E}_{\mathcal{A}}$, we define

$$Y^{\bot_{\mathcal{A},k}} \stackrel{\text{def}}{=} \{\ (t,p) \in \mathbb{P}_{\mathcal{A}} \mid \forall (E,q) \in Y, (t,p)\bot_{\mathcal{A},k}(E,q)\ \}$$

Notation 125. *If $\mathcal{A}$ is a n-**MA**, we denote by $\perp_{\mathcal{A}}$ the relation $\perp_{\mathcal{A},n}$.*

4.2.3 Interpretation of multiplicatives

We now show how each n-**MA** $\mathcal{A}$ induces a realizability model of λ_{LCBPV}. We need to give an interpretation of types, similarly to what has been done for the unary realizability model of Section 4.1. A positive (resp. negative) type will be interpreted by a set of $\mathcal{A}$-values (resp. $\mathcal{A}$-computations) which is upward closed (for the preorder $\preceq_{|\mathcal{A}|}$).

Definition 126. *We denote by:*

- *$\mathcal{I}(\mathbb{V}_{\mathcal{A}})$ the set of all subsets $X \subseteq \mathbb{V}_{\mathcal{A}}$ that are upward-closed:*

$$\forall (v,p) \in X, p \preceq q \Rightarrow (v,q) \in X$$

- *$\mathcal{I}(\mathbb{E}_{\mathcal{A}})$ the set of all subsets $X \subseteq \mathbb{E}_{\mathcal{A}}$ that are upward-closed:*

$$\forall (E,p) \in X, p \preceq q \Rightarrow (E,q) \in X$$

- *$\mathcal{I}(\mathbb{P}_{\mathcal{A}})$ the set of all subsets $X \subseteq \mathbb{P}_{\mathcal{A}}$ that are upward-closed:*

$$\forall (t,p) \in X, p \preceq q \Rightarrow (t,q) \in X$$

Definition 127 ($\mathcal{A}$-model)**.** *A $\mathcal{A}$-**model** is a mapping ρ such that:*

- *ρ associates to every predicate variable X of arity $S_1 \times \cdots \times S_n$ a function*

$$\rho_{\mathcal{A}}(X) : S_1 \times \cdots \times S_n \longrightarrow \mathcal{I}(\mathbb{V}_{\mathcal{A}})$$

- *It associates to every first-order variable of sort S an element $\rho(x^S) \in S$.*

Property 128. *Let $k \in \mathbb{N}$. If $X \subseteq \mathbb{V}_{\mathcal{A}}$ (resp. $Y \subseteq \mathbb{E}_{\mathcal{A}}$) then $X^{\perp_{\mathcal{A},k}} \in \mathcal{I}(\mathbb{E}_{\mathcal{A}})$ (resp. $Y^{\perp_{\mathcal{A},k}} \in \mathcal{I}(\mathbb{P}_{\mathcal{A}})$).*

Proof.

- We prove that given $X \subseteq \mathbb{V}_{\mathcal{A}}$, $X^{\perp_{\mathcal{A},k}} \in \mathcal{I}(\mathbb{E}_{\mathcal{A}})$, the other case can be proved by the same reasoning. Let $(E,p) \in X^{\perp_{\mathcal{A},k}}$ and q such that $p \preceq q$. We want to show that $(E,q) \in X^{\perp_{\mathcal{A},k}}$. Let $(v,r) \in X$ and $(w_1,\dots,w_n) \in \mathcal{C}_{\mathcal{A}}(r \bullet q)$. We need to prove that

$$\langle \uparrow E, (v,(w_n,\dots,w_1)) \rangle_n \in \perp\!\!\!\perp \tag{2.20}$$

But $p \preceq q$ implies that $r \bullet p \preceq r \bullet q$, hence because $\mathcal{C}_{\mathcal{A}}$ is non-increasing, we have $(w_1, \dots, w_n) \in \mathcal{C}_{\mathcal{A}}(r \bullet p)$. Since $(E, p) \in X^{\perp_{\mathcal{A},k}}$ we deduce (2.20).

- The case where $X \subseteq \mathbb{E}_{\mathcal{A}}$ is proved in a similar way, using the fact that $p \preceq q$ implies $p \bullet r \preceq q \bullet r$.

□

Definition 129 (Interpretation). *Let $n \in \mathbb{N}$ and ρ be a $\mathcal{A}$-model. The interpretation of types is given as follows:*

- *We associate to every value type P a set $\|P\|_{\mathcal{A},k,\rho} \subseteq \mathbb{V}_{\mathcal{A}}$ of $\mathcal{A}$-values, called the $\mathcal{A}$-**truth value of** P (at level k).*
- *We associate to every computation type N:*
  - *a set $[\![N]\!]_{\mathcal{A},k,\rho} \subseteq \mathbb{E}_{\mathcal{A}}$ of $\mathcal{A}$-environments, called the $\mathcal{A}$-**falsity value of** N (at level k).*
  - *a set $\|N\|_{\mathcal{A},k,\rho} \subseteq \mathbb{P}_{\mathcal{A}}$ of $\mathcal{A}$-computations, called the $\mathcal{A}$-**truth value of** N (at level k).*

These sets are defined by mutual induction on the type and given in Figure 2.

Property 130. *Let P be a positive type and N a negative type. Let $\mathcal{A}$ be a **MA**, $n \in \mathbb{N}$ and ρ a $\mathcal{A}$-model. Then:*

- $\|P\|_{\mathcal{A},n,\rho} \in \mathcal{I}(\mathbb{V}_{\mathcal{A}})$
- $[\![N]\!]_{\mathcal{A},n,\rho} \in \mathcal{I}(\mathbb{E}_{\mathcal{A}})$
- $\|N\|_{\mathcal{A},n,\rho} \in \mathcal{I}(\mathbb{P}_{\mathcal{A}})$

Proof. The proof is easily done by induction on the formula and by using Property 128. □

If a type A (resp. N) is closed, then we will often just write $\|A\|_{\mathcal{A},n}$ (resp. $[\![N]\!]_{\mathcal{A},n}$) to denote its truth value (resp. falsity value). Moreover, if $\mathcal{A}$ and n are clear from the context, we will sometimes omit them.

Definition 131 (Realizability relation). *Let $\mathcal{A}$ be a n-monitoring algebra and $k \in \mathbb{N}$. We define the **realizability relation** $\Vdash_{\mathcal{A},k} \_[\rho]$ at level k that relates:*

- *A $\mathcal{A}$-computation $(t, p) \in \mathbb{P}_{\mathcal{A}}$*
- *A closed negative type N*

Positive interpretation

$$
\begin{array}{rcl}
\|X(e_1,\dots,e_k)\|_{\mathcal{A},n,\rho} & \stackrel{\mathsf{def}}{=} & \rho(X)([\![e_1]\!]_\rho,\dots,[\![e_k]\!]_\rho) \\
\|\mathsf{Nat}\|_{\mathcal{A},n,\rho} & \stackrel{\mathsf{def}}{=} & \{\ (\underline{k},p) \mid k\in\mathbb{N}\wedge p\in|\mathcal{A}|\ \} \\
\|\mathbb{1}\|_{\mathcal{A},n,\rho} & \stackrel{\mathsf{def}}{=} & \{\ (*,p) \mid p\in|\mathcal{A}|\ \} \\
\|P\otimes Q\|_{\mathcal{A},n,\rho} & \stackrel{\mathsf{def}}{=} & \{\ ((v,w),r) \mid p+q\preceq r\wedge (v,p)\in\|P\|_{\mathcal{A},n,\rho}\wedge (w,q)\in\|Q\|_{\mathcal{A},n,\rho}\ \} \\
\|\Downarrow N\|_{\mathcal{A},n,\rho} & \stackrel{\mathsf{def}}{=} & \{\ (\mathsf{thunk}(t),p) \mid (t,p)\in [\![N]\!]_{\mathcal{A},n,\rho}{}^{\perp_{\mathcal{A},n}}\ \} \\
\|\exists x\in S.P\|_{\mathcal{A},n,\rho} & \stackrel{\mathsf{def}}{=} & \bigcup_{c\in S}\|P\|_{\mathcal{A},n,\rho[x\leftarrow c]} \\
\|\{e\preceq_S f\}\wedge P\|_{\mathcal{A},n,\rho} & \stackrel{\mathsf{def}}{=} & \begin{cases} \|P\|_{\mathcal{A},n,\rho} & \text{if } [\![e]\!]_\rho\preceq_S[\![f]\!]_\rho \\ \varnothing & \text{otherwise}\end{cases}
\end{array}
$$

Negative environment interpretation

$$
\begin{array}{rcl}
[\![P\multimap M]\!]_{\mathcal{A},n,\rho} & \stackrel{\mathsf{def}}{=} & \{\ (\mathsf{a}(v).E,r) \mid p\bullet q\preceq r\wedge (v,p)\in\|P\|_{\mathcal{A},n,\rho}\wedge (E,q)\in [\![Q]\!]_{\mathcal{A},n,\rho}{}^{\perp_{\mathcal{A},n}}\ \} \\
[\![\Uparrow P]\!]_{\mathcal{A},n,\rho} & \stackrel{\mathsf{def}}{=} & \|P\|_{\mathcal{A},n,\rho}{}^{\perp_{\mathcal{A},n}} \\
[\![\forall x\in S.N]\!]_{\mathcal{A},n,\rho} & \stackrel{\mathsf{def}}{=} & \bigcup_{c\in S}[\![N]\!]_{\mathcal{A},n,\rho[x\leftarrow c]} \\
[\![\{e\succeq_S f\}\mapsto N]\!]_{\mathcal{A},n,\rho} & \stackrel{\mathsf{def}}{=} & \begin{cases} [\![N]\!]_{\mathcal{A},n,\rho} & \text{if } [\![e]\!]_\rho\succeq_S[\![f]\!]_\rho \\ \varnothing & \text{otherwise}\end{cases}
\end{array}
$$

Negative computation interpretation

$$\|N\|_{\mathcal{A},n,\rho} \stackrel{\mathsf{def}}{=} [\![N]\!]_{\mathcal{A},n,\rho}{}^{\perp_{\mathcal{A},n}}$$

Figure 2: Unary monitoring interpretation

It is defined as follows:

$$(t,p)\Vdash_{\mathcal{A},k} N[\rho] \stackrel{\mathsf{def}}{=} (t,p)\in\|N\|_{\mathcal{A},k,\rho}$$

We finish by proving a useful technical lemma.

Lemma 132. *Suppose that ρ is a $\mathcal{A}$-model such that for all predicate variable X of arity S, for any $s\in S$ there is $v\in\mathbb{V}$ such that $(v,\mathbf{0})\in\rho(X)(s)$. Then the two following statements hold:*

1. *For every positive type P of $\lambda_{\mathsf{LCBPV}}^{\otimes\mathsf{Nat}\forall}$, there exists $v\in\mathbb{V}$ such that $(v,\mathbf{0})\in\|P\|_{\mathcal{A},1,\rho}$.*

2. *For every negative type N of $\lambda_{\mathsf{LCBPV}}^{\otimes\mathsf{Nat}\forall}$, there exists $E \in \mathbb{E}$ such that*

$$(E, \mathbf{0}) \in [\![N]\!]_{\mathcal{A},1,\rho}$$

Proof.

We prove the two by induction on the negative types.

1. We prove that assertation by induction on the type.
   - If $P = X$ with X a predicate variable then it is the base hypothesis.
   - If $P = \Downarrow N$: then we have $(✠, \mathbf{0}) \in \|N\|_{\mathcal{A},1,\rho}$. Hence $(\mathsf{thunk}(✠), \mathbf{0}) \in \|\Downarrow N\|_{\mathcal{A},1,\rho}$.
   - If $P = Q_1 \otimes Q_2$, then by induction we have $(v, \mathbf{0}) \in \|Q_1\|_{\mathcal{A},1,\rho}$ and $(w, \mathbf{0}) \in \|Q_2\|_{\mathcal{A},1,\rho}$. Therefore $((v, w), \mathbf{0}) \in \|Q_1 \otimes Q_2\|_{\mathcal{A},1,\rho}$.
   - If $P = \mathsf{Nat}$, then $(\underline{0}, \mathbf{0}) \in \|\mathsf{Nat}\|_{\mathcal{A},1,\rho}$.
   - If $P = \exists x \in S.Q$ then it is easy since S is not empty we have at least $(v, \mathbf{0}) \in \|Q\|_{\mathcal{A},1,\rho[x \leftarrow c]}$ for some $c \in S$.
2. We now prove the second assertation, still by induction on N.
   - If $N = \Uparrow P$, then we know that for any value $(v, p) \in \|P\|_{\mathcal{A},1,\rho}$, $(\mathsf{ret}(v), \mathsf{nil}, p) \in \perp\!\!\!\perp_{\mathcal{A},1}$, since the resulting configuration will never reduce for $\rightarrow$.
   - If $N = \forall x \in S.N$, then it is immediate since $[\![\forall x \in S.N]\!]_{\mathcal{A},1,\rho}$ is an union of interpretations of negative types which satisfy the property.
   - If $N = P \multimap M$, then by the property 1., there exists $v \in \mathbb{V}$ such that $(v, \mathbf{0}) \in \|P\|_{\mathcal{A},1,\rho}$. Moreover we have $(E, \mathbf{0}) \in [\![N]\!]_{\mathcal{A},1,\rho}$ for some $E \in \mathbb{E}$. Hence $(\mathsf{a}(v).E, \mathbf{0}) \in [\![P \multimap N]\!]_{\mathcal{A},1,\rho}$.

□

Remark 133. *This lemma will be used to prove the computational correctness using our realizability framework. Notice that it would not hold if one includes the inequational implication. That's why we restrict it to $\lambda_{\mathsf{LCBPV}}^{\otimes\mathsf{Nat}\forall}$ types.*

4.2.4 Parametric soundness

We now prove a **soundness theorem** for **MA**s. This result is similar to the forcing soundness result of [Miq11]. In particular, we prove that each n-**MA** $\mathcal{A}$ induces a realizability model of λ_{LCBPV}. That means that for every negative type N and every positive type P, we have:

$$\begin{array}{rcl} \mathcal{E}; \Gamma \vdash_0 v : P & \Rightarrow & \exists p \in |\mathcal{A}|, (v, p) \in \|P\|_{\mathcal{A},n} \\ \mathcal{E}; \Gamma \vdash_0 t : N & \Rightarrow & \exists p \in |\mathcal{A}|, (t, p) \in \|N\|_{\mathcal{A},n} \end{array}$$

Notice that the level at which we prove soundness is n, which is the level of $\mathcal{A}$. The element $p \in |\mathcal{A}|$ that accompanies the value v (resp. the computation t) can directly be calculated from the type system defined in Section 3.2. In fact, we use this type system to define what it means for a rule to be $\mathcal{A}$-**sound** and then state the soundness theorem. We begin by defining some useful notions.

Definition 134 (Substitution)**.** *A* $\mathcal{A}$***-substitution*** σ *is a partial application from the set of term variables* Var *to the set* $\mathbb{V}_{\mathcal{A}}$*, whose domain* $dom(\sigma)$ *is finite. We note*

$$[x_1 \leftarrow (v_1, p_1), \ldots, x_n \leftarrow (v_n, p_n)]$$

the substitution σ *such that* $dom(\sigma) = \{x_1, \ldots, x_n\}$ *and such that* $\sigma(x_i) = (v_i, p_i)$*.*

If σ is a $\mathcal{A}$-substitution, we denote by $\sigma[x \leftarrow (v, p)]$ the substitution obtained from σ by binding x to (v, p). If σ_1 is a subtitution and $\sigma_2 = [x_1 \leftarrow (v_1, p_1), \ldots, x_n \leftarrow (v_n, p_n)]$ is another substitution (disjoint from σ_1), we denote by

$$\sigma_1, \sigma_2 = (\ldots(\sigma_1[x_1 \leftarrow (v_1, p_1)])\ldots)[x_n \leftarrow (v_n, p_n)]$$

Now, take a substitution $\sigma = [x_1 \leftarrow (v_1, p_1), \ldots, x_n \leftarrow (v_n, p_n)]$. Let $(t, q) \in \mathbb{P}_{\mathcal{A}}$ (hence t is *possibly open*). Then we note

$$\begin{array}{rcl} t[\sigma] & = & t[v_1/x_1, \ldots, v_n/x_n] \\ q[\sigma] & = & q + \sum_i p_i \\ (t, q)[\sigma] & = & (t[\sigma], q[\sigma]) \end{array}$$

Similarly if $(w, q) \in \mathbb{V}_{\mathcal{A}}$, we note:

$$\begin{array}{rcl} w[\sigma] & = & w[v_1/x_1, \ldots, v_n/x_n] \\ (w, q)[\sigma] & = & (w[\sigma], q[\sigma]) \end{array}$$

Remark 135. *If* $\mathsf{FV}(t) \subseteq dom(\sigma)$ *(resp.* $\mathsf{FV}(v) \subseteq dom(\sigma)$*) then* $(t, q)[\sigma]$ *(resp.* $(v, q)[\sigma]$*) is closed.*

We now want to define what it means for a typing rule to be **sound** with respect to $\mathcal{A}$. We consider typing rules of the type system $\lambda^{|\mathcal{A}|}_{\mathsf{LCBPV}}$ whose judgments are of the form:

$$\mathcal{E}; \Gamma \vdash_k a : (A, p) \quad \text{where } p \in |\mathcal{A}|$$

We first define the soundness of a judgment, and then of a typing rule. We will then prove the soundness of every typing rule.

Definition 136 (Adequate substitution)**.** *Let* $\Gamma = x_1 : P_1, \ldots, x_n : P_n$ *be a context and* ρ *a* $\mathcal{A}$*-model. We say that a* $\mathcal{A}$*-substitution* σ *is* ***adequate to*** $\Gamma[\rho]$ *and we note* $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}$ *iff*

- $dom(\sigma) = \{x_1, \ldots, x_n\}$
- $\forall i \in \{1, \ldots, n\}, \sigma(x_i) \in \|P_i\|_{\mathcal{A},k,\rho}$

Definition 137 ($\mathcal{A}$-sound judgment)**.** *Suppose that:*

- $\mathcal{A}$ *is a* k-***MA***.
- $\Gamma = x_1 : P_1, \ldots, x_n : P_n$ *is a typing context.*
- $\mathcal{E}$ *is a first-order inequational theory.*
- $p \in |\mathcal{A}|$.
- $a \in \mathbb{P} \cup \mathbb{V}$.
- A *is a negative or positive type.*

We say that a judgment of the form $\mathcal{E}; \Gamma \vdash_k a : (A, p)$ *is* $\mathcal{A}$-***sound*** *iff for every* $\mathcal{A}$*-model* ρ *such that* $\rho \Vdash \mathcal{E}$ *and for every adequate substitution* $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}$*, we have*

$$(a, p)[\sigma] \in \|A\|_{\mathcal{A},k,\rho}$$

We can define what it means for our interpretation to be sound with respect to to a given typing rule. A typing rule is given by a sequence of premises J_i (which are typing judgments), side-conditions (SC) on these judgments, and a conclusion K:

$$\frac{J_1 \quad J_2 \quad \ldots \quad J_n \qquad SC}{K}\,(rule)$$

Definition 138 ($\mathcal{A}$-sound rule)**.** *Let* $\mathcal{A}$ *be a* k-***MA***. *Suppose* R *is a typing rule, with* $J_1, \ldots, J_n$ *being its premises judgments and* K *its conclusion. We say that:*

R *is* $\mathcal{A}$-***sound*** *iff the* $\mathcal{A}$*-soundness of all premises* J_i *and the side-conditions* SC *implie the soundness of the conclusion* K.

Theorem 139 (Subtyping soundness)**.** *Let* $\mathcal{A}$ *be a* k-***MA***. *Suppose that* $\mathcal{E}$ *is an inequational theory and that* $A \sqsubseteq_{\mathcal{E}} B$. *For every* $\mathcal{A}$*-model* ρ *such that* $\rho \Vdash \mathcal{E}$*, we have*

$$\|A\|_{\mathcal{A},k,\rho} \subseteq \|B\|_{\mathcal{A},k,\rho}$$

Proof. The proof is very similar to the proof Theorem 117, hence we omit it. □

Theorem 140 (Soundness)**.** *Let* $\mathcal{A}$ *be a* k-***MA***. *All* $\lambda^{|\mathcal{A}|}_{\mathsf{LCBPV}}$ *typing rules at level* k *are* $\mathcal{A}$*-sound.*

Proof. The proof is done by examining each rule. Here $\mathcal{E}$ is fixed and ρ is always a $\mathcal{A}$-model such that $\rho \Vdash \mathcal{E}$.

- **(Var)** We want to show that $\mathcal{E}; \Gamma, x : P \vdash_k x : (P, \mathbf{0})$ is sound. But it is immediate since if $\sigma \in \|\Gamma, x : P\|_{\mathcal{A},k,\rho}$ then $\sigma(x) \in \|P\|_{\mathcal{A},k,\rho}$, hence the result.
- **(Unit)** We want to show that $\mathcal{E}; \Gamma \vdash_k * : (\mathbb{1}, \mathbf{0})$ is sound. But it is immediate since if $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}$ then $(*, \mathbf{0})[\sigma] \in \|\mathbb{1}\|_{\mathcal{A},k,\rho}$ by $\preceq$-saturation.
- **(Pos. shift)** We suppose that $\mathcal{E}; \Gamma \vdash_n t : (N, p)$ is sound and want to conclude that $\mathcal{E}; \Gamma \vdash_n \mathsf{thunk}(t) : (\Downarrow N, p)$ is. Let $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}$. Then by hypothesis we know that

$$(t, p)[\sigma] \in \|N\|_{\mathcal{A},k,\rho}$$

  Hence, by definition we immediately conclude that

$$\underbrace{(\mathsf{thunk}(t[\sigma]), p[\sigma])}_{=(\mathsf{thunk}(t),p)[\sigma]} \in \|\Downarrow N\|_{\mathcal{A},k,\rho}$$

- **(Pos. shift elim.)** We suppose that $\mathcal{E}; \Gamma \vdash_n v : (\Downarrow N, p)$ is sound and want to conclude that $\mathcal{E}; \Gamma \vdash_n \mathsf{force}(v) : (N, p)$ is. Let $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}$. Then by hypothesis we know that:

$$(v, p)[\sigma] \in \|\Downarrow N\|_{\mathcal{A},k,\rho}$$

  First, by definition of $\|\Downarrow N\|_{\mathcal{A},k,\rho}$, we have $(v, p)[\sigma] = (\mathsf{thunk}(t), p)$ with

$$(t, p[\sigma]) \in \|N\|_{\mathcal{A},k,\rho} \tag{2.21}$$

  Let $(E, q) \in [\![N]\!]_{\mathcal{A},k,\rho}$. We need to show that

$$(\mathsf{force}(\mathsf{thunk}(t)), p[\sigma]) = (\mathsf{force}(v)[\sigma], p[\sigma]) \perp_{\mathcal{A}} (E, q)$$

  Let $\overrightarrow{w} \in \mathcal{C}_{\mathcal{A}}(p[\sigma] \bullet q)$. Because of (2.21), we have immediately that $\langle t, \overrightarrow{\mathsf{a}(w)}.E\rangle_k \in \perp\!\!\!\perp$. Since

$$\langle \mathsf{force}(\mathsf{thunk}(t)), \overrightarrow{\mathsf{a}(w)}.E\rangle_k \xrightarrow{k} \langle t, \overrightarrow{\mathsf{a}(w)}.E\rangle_k$$

  we conclude by saturation of the pole.

- **(Neg. shift)** We suppose that $\mathcal{E}; \Gamma \vdash_n v : (P, p)$ is sound and want to conclude that $\mathcal{E}; \Gamma \vdash_n \mathsf{ret}(v) : (\Uparrow P, p)$ is. Let $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}$. Then by hypothesis we know that

$$(v, p)[\sigma] \in \|P\|_{\mathcal{A},k,\rho} \tag{2.22}$$

  We need to show that $(\mathsf{ret}(v[\sigma]), p[\sigma]) \in [\![\Uparrow P]\!]_{\mathcal{A},k,\rho}{}^{\perp_{\mathcal{A}}}$. Let $(E, q) \in [\![\Uparrow P]\!]_{\mathcal{A},k,\rho}$. Because of (2.22) and by definition of $\|\Uparrow P\|_{\mathcal{A},k,\rho}$ we conclude immediately.
- **(Neg. shift elim.)** We suppose that the two following judgments are $\mathcal{A}$-sound:

$$\mathcal{E}; \Gamma \vdash_n t : (\Uparrow P, p_1) \tag{2.23}$$

$$\mathcal{E}; \Delta, x : P \vdash_n u : (M, p_2) \tag{2.24}$$

  We want to show that $\mathcal{E}; \Gamma, \Delta \vdash_n t \text{ to } x.u : (M, p_1 + p_2)$ is $\mathcal{A}$-sound. Let $\sigma \in \|\Gamma, \Delta\|_{\mathcal{A},k,\rho}$. Then because Γ and Δ are disjoint, it is clear that there exists two substitutions σ_1, σ_2 such that

  – $\sigma = \sigma_1, \sigma_2$

- $\sigma_1 \in \|\Gamma\|_{\mathcal{A},k,\rho}$
- $\sigma_2 \in \|\Delta\|_{\mathcal{A},k,\rho}$

Hence, by hypothesis we have

$$(t, p_1)[\sigma_1] \in \|\Uparrow P\|_{\mathcal{A},k,\rho} \tag{2.25}$$

$$\forall (v, q) \in \|P\|_{\mathcal{A},k,\rho}, (u[\sigma_2, v/x], p_2[\sigma_2] + q) \in \|M\|_{\mathcal{A},k,\rho} \tag{2.26}$$

Let's show that

$$(t[\sigma_1] \text{ to } x.u[\sigma_2], p_1[\sigma_1] + p_2[\sigma_2]) \in \|M\|_{\mathcal{A},k,\rho}$$

Suppose that $(E, r) \in [\![M]\!]_{\mathcal{A},k,\rho}$. Then, we can show the following intermediate result:

$$(\mathsf{f}(x.u[\sigma_2]).E, p_2[\sigma_2] \bullet r) \in \|P\|_{\mathcal{A},k,\rho}{}^{\perp_{\mathcal{A}}}$$

- Indeed, if $(v, p') \in \|P\|_{\mathcal{A},k,\rho}$ and $\overrightarrow{w} \in \mathcal{C}_{\mathcal{A}}(p' \bullet p_2[\sigma] \bullet r)$, we have

$$\langle \mathsf{ret}(v), \overrightarrow{\mathsf{a}(w)}.\mathsf{f}(x.u[\sigma_2]).E\rangle_k \xrightarrow{k} \langle u[\sigma_2][v/x], \overrightarrow{\mathsf{a}(w)}.E\rangle_k$$

  The latter is in $\perp\!\!\!\perp$ by (2.26). Hence, we obtain the result by saturation of $\perp\!\!\!\perp$.

Let $\overrightarrow{w} \in \mathcal{C}_{\mathcal{A}}((p_1[\sigma_1] + p_2[\sigma_2]) \bullet r)$. Then $\overrightarrow{w} \in \mathcal{C}_{\mathcal{A}}(p_1[\sigma_1] \bullet p_2[\sigma_2] \bullet r)$. Since $[\![\Uparrow P]\!]_{\mathcal{A},k,\rho} = \|P\|_{\mathcal{A},k,\rho}{}^{\perp_{\mathcal{A}}}$ and because

$$\langle t[\sigma_1] \text{ to } x.u[\sigma_2], \overrightarrow{\mathsf{a}(w)}.E\rangle_k \xrightarrow{k} \langle t[\sigma_1], \overrightarrow{\mathsf{a}(w)}.\mathsf{f}(x.u[\sigma_2]).E\rangle_k \in \perp\!\!\!\perp$$

we deduce the conclusion by saturation of $\perp\!\!\!\perp$.

- **($\multimap$ intro.)** We suppose that $\mathcal{E}; \Gamma, x : P \vdash_n t : (N, p)$ is $\mathcal{A}$-sound and want to conclude that $\mathcal{E}; \Gamma \vdash_n \lambda x.t : (P \multimap M, p)$ is. Let $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}$. Let $(v, q) \in \|P\|_{\mathcal{A},k,\rho}$ and $(E, r) \in [\![N]\!]_{\mathcal{A},k,\rho}$. We need to show that $(\lambda x.t[\sigma], p)\perp_{\mathcal{A}}(\mathsf{a}(v).E, q \bullet r)$. Let $\overrightarrow{w} \in \mathcal{C}_{\mathcal{A}}(p \bullet q \bullet r)$. We know that $\sigma[(v, q)/x] \in \|\Gamma, x : P\|_{\mathcal{A},k,\rho}$, hence by hypothesis that $(t[\sigma][v/x], p[\sigma] + q) \in \|N\|_{\mathcal{A},k,\rho}$. By associativity, this implies that

$$\langle t[\sigma][v/x], \overrightarrow{\mathsf{a}(w)}.E\rangle_k \in \perp\!\!\!\perp$$

  On the other hand, we have

$$\langle \lambda x.t[\sigma], \overrightarrow{\mathsf{a}(w)}.\mathsf{a}(v).E\rangle_k \xrightarrow{k} \langle t[\sigma][v/x], \overrightarrow{\mathsf{a}(w)}.E\rangle_k$$

  Therefore, by saturation of $\perp\!\!\!\perp$, we conclude that $(\lambda x.t, p)[\sigma]\perp_{\mathcal{A}}(\mathsf{a}(v).E, q \bullet r)$.

- **($\multimap$ elim.)** We suppose that the two following judgments are $\mathcal{A}$-sound:

$$\mathcal{E}; \Gamma \vdash_n t : (P \multimap N, p) \tag{2.27}$$

$$\mathcal{E}; \Delta \vdash_n v : (P, q) \tag{2.28}$$

  We want to show that $\mathcal{E}; \Gamma, \Delta \vdash_n (t)v : (N, p+q)$ is $\mathcal{A}$-sound. Let $\sigma \in \|\Gamma, \Delta\|_{\mathcal{A},k,\rho}$. Then because Γ and Δ are disjoint, it is clear that there exists two substitutions σ_1, σ_2 such that

  - $\sigma = \sigma_1, \sigma_2$
  - $\sigma_1 \in \|\Gamma\|_{\mathcal{A},k,\rho}$

- $\sigma_2 \in \|\Delta\|_{\mathcal{A},k,\rho}$

Hence, by hypothesis we have

$$(t[\sigma_1], p[\sigma_1]) \in \|P \multimap N\|_{\mathcal{A},k,\rho} \tag{2.29}$$

$$(v[\sigma_2], q[\sigma_2]) \in \|P\|_{\mathcal{A},k,\rho} \tag{2.30}$$

Let's show that

$$(t[\sigma_1]v[\sigma_2], p[\sigma_1] + q[\sigma_2]) \in \|M\|_{\mathcal{A},k,\rho}$$

Suppose that $(E, r) \in [\![M]\!]_{\mathcal{A},k,\rho}$ and $\overrightarrow{w} \in \mathcal{C}_{\mathcal{A}}((p[\sigma_1] + q[\sigma_2]) \bullet r)$. Then because of (2.34),

$$(\mathsf{a}(v[\sigma_2]).E, q[\sigma_2] \bullet r) \in [\![P \multimap N]\!]_{\mathcal{A},k,\rho}$$

Hence, we know by (2.29) that

$$\langle t[\sigma_1], \overrightarrow{\mathsf{a}(w)}.\mathsf{a}(v[\sigma_2]).E\rangle_k \in \perp\!\!\!\perp$$

Finally, we conclude by saturation of $\perp\!\!\!\perp$ that

$$\langle (t[\sigma_1])v[\sigma_2], \overrightarrow{\mathsf{a}(w)}.E\rangle_k \in \perp\!\!\!\perp$$

- **($\otimes$ intro.)** We suppose that $\mathcal{E}; \Gamma \vdash_n v : (P, p)$ and $\mathcal{E}; \Delta \vdash_n w : (Q, q)$ are $\mathcal{A}$-sound and want to conclude that $\mathcal{E}; \Gamma, \Delta \vdash_n (v, w) : (P \otimes Q, p + q)$ is. Let $\sigma \in \|\Gamma, \Delta\|_{\mathcal{A},k,\rho}$. Then because Γ and Δ are disjoint, it is clear that there exists two substitutions σ_1, σ_2 such that
  - $\sigma = \sigma_1, \sigma_2$
  - $\sigma_1 \in \|\Gamma\|_{\mathcal{A},k,\rho}$
  - $\sigma_2 \in \|\Delta\|_{\mathcal{A},k,\rho}$

  Hence, by hypothesis we have
  $$(v, p)[\sigma_1] \in \|P\|_{\mathcal{A},k,\rho}$$
  and
  $$(w, q)[\sigma_2] \in \|Q\|_{\mathcal{A},k,\rho}$$
  Hence, clearly because $((v, w), p + q)[\sigma] = ((v[\sigma_1], w[\sigma_2]), p[\sigma_1] + q[\sigma_2])$ we have
  $$((v, w), p + q)[\sigma] \in \|P \otimes Q\|_{\mathcal{A},k,\rho}$$

- **($\otimes$ elim.)** We suppose that the two following judgments are $\mathcal{A}$-sound:

$$\mathcal{E}; \Gamma, x : P, y : Q \vdash_n t : (N, p) \tag{2.31}$$

$$\mathcal{E}; \Delta \vdash_n v : (P \otimes Q, q) \tag{2.32}$$

We want to show that $\mathcal{E}; \Gamma, \Delta \vdash_n \mathsf{let}\,(x, y) = v\,\mathsf{in}\,t : (N, p + q)$ is $\mathcal{A}$-sound. Let $\sigma \in \|\Gamma, \Delta\|_{\mathcal{A},k,\rho}$. Then because Γ and Δ are disjoint, it is clear that there exists two substitutions σ_1, σ_2 such that

- $\sigma = \sigma_1, \sigma_2$
- $\sigma_1 \in \|\Gamma\|_{\mathcal{A},k,\rho}$

– $\sigma_2 \in \|\Delta\|_{\mathcal{A},k,\rho}$

Hence, by hypothesis we have

$$\forall (w_1, q_1) \in \|P\|_{\mathcal{A},k,\rho}, \forall (w_2, q_2) \in \|Q\|_{\mathcal{A},k,\rho}, (t[\sigma_1][w_1/x, w_2/y], p[\sigma_1] + q_1 + q_2) \in \|N\|_{\mathcal{A},k,\rho} \tag{2.33}$$

and

$$(v, q)[\sigma_2] \in \|P \otimes Q\|_{\mathcal{A},k,\rho} \tag{2.34}$$

By definition of the interpretation of $P \otimes Q$, we have $(v, q)[\sigma_2] = ((w_1, w_2), q_1 + q_2)$ such that $(w, q_1) \in \|P\|_{\mathcal{A},k,\rho}$ and $(w, q_2) \in \|Q\|_{\mathcal{A},k,\rho}$. Hence, by (2.33) we have

$$(t[\sigma_1][w_1/x, w_2/y], p[\sigma_1] + q[\sigma_2]) \in \|N\|_{\mathcal{A},k,\rho}$$

Now, for any $(E, r) \in [\![N]\!]_{\mathcal{A},k,\rho}$ and $\overrightarrow{w} \in \mathcal{C}_{\mathcal{A}}((p[\sigma_1] + q[\sigma_2]) \bullet r)$ we have

$$\langle \mathsf{let}\,(x, y) \;=\; v[\sigma_2]\,\mathsf{in}\,t[\sigma_1], \overrightarrow{\mathsf{a}(w)}.E\rangle_k \xrightarrow{k} \langle t[\sigma_1][w_1/x, w_2/y], \overrightarrow{\mathsf{a}(w)}.E\rangle_k$$

But since $(t[\sigma_1][w_1/x, w_2/y], (p + q)[\sigma])\bot E$, we have by saturation that

$$\langle \mathsf{let}\,(x, y) \;=\; v[\sigma_2]\,\mathsf{in}\,t[\sigma_1], \overrightarrow{\mathsf{a}(w)}.E\rangle_k \in \perp\!\!\!\perp$$

Which exactly means that $(\mathsf{let}\,(x, y) \;=\; v[\sigma_2]\,\mathsf{in}\,t[\sigma_1], p[\sigma_1] + q[\sigma_2]) \in \|N\|_{\mathcal{A},k,\rho}$.

- **(prim. integers)** We want to show that $\mathcal{E}; \Gamma \vdash_n \underline{\mathsf{n}} : (\mathsf{Nat}, \mathbf{0})$ is $\mathcal{A}$-sound. This is immediate by definition of $\|\mathsf{Nat}\|_{\mathcal{A},k,\rho}$.
- **(succ)** We want to show that $\mathcal{E}; \Gamma \vdash_n \underline{\mathsf{s}}(v) : (\mathsf{Nat}, p)$ is $\mathcal{A}$-sound provided that $\mathcal{E}; \Gamma \vdash_n v : (\mathsf{Nat}, p)$ is $\mathcal{A}$-sound. Let $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}$. We have by hypothesis that

$$(v, p)[\sigma] \in \|\mathsf{Nat}\|_{\mathcal{A},k,\rho}$$

Hence, by definition of the interpretation, there is $n \in \mathbb{N}$ such that $(v, p)[\sigma] = (\underline{\mathsf{n}}, p[\sigma])$. Therefore, since $\underline{\mathsf{s}}(\underline{\mathsf{n}}) = \underline{\mathsf{n+1}}$, we have

$$(\underline{\mathsf{s}}(v), p)[\sigma] = (\underline{\mathsf{n+1}}, p[\sigma]) \in \|\mathsf{Nat}\|_{\mathcal{A},k,\rho}$$

- **(Case)** We suppose that $\mathcal{E}; \Gamma \vdash_n v : (\mathsf{Nat}, p)$, $\mathcal{E}; \Delta, x : \mathsf{Nat} \vdash_n t_1 : (N, q)$ and $\mathcal{E}; \Delta, x : \mathsf{Nat} \vdash_n t_2 : (N, q)$ are $\mathcal{A}$-sound. We want to prove that the following judgment is $\mathcal{A}$-sound too:

$$\mathcal{E}; \Gamma, \Delta \vdash_n \mathsf{case}\, v\, \mathsf{of}\, x.t_1 \parallel x.t_2 : (N, p + q)$$

Let $\sigma_1 \in \|\Gamma\|_{\mathcal{A},k,\rho}$ and $\sigma_2 \in \|\Delta\|_{\mathcal{A},k,\rho}$. We first know that

$$(v, p)[\sigma_1] \in \|\mathsf{Nat}\|_{\mathcal{A},k,\rho}$$

Hence, there is $n \in \mathbb{N}$ such that $v[\sigma_1] = \underline{\mathsf{n}}$. Since $\sigma_2[x \leftarrow (v[\sigma_1], p[\sigma_1])] \in \|\Delta, x : \mathsf{Nat}\|_{\mathcal{A},k,\rho}$ we also know that

$$(t_1[\sigma_2][v[\sigma_1]/x], q[\sigma_2] + p[\sigma_1)] \in \|N\|_{\mathcal{A},k,\rho} \tag{2.35}$$

$$(t_2[\sigma_2][v[\sigma_1]/x], q[\sigma_2] + p[\sigma_1]) \in \|N\|_{\mathcal{A},k,\rho} \tag{2.36}$$

Let $(E,r) \in [\![N]\!]_{\mathcal{A},k,\rho}$ and $\overrightarrow{w} \in \mathcal{C}_{\mathcal{A}}((p[\sigma_1]+q[\sigma_2]) \bullet r)$. Depending of wether $n=0$ or $n>0$ we have

$$\langle \mathsf{case}\, v[\sigma_1]\, \mathsf{of}\, x.t_1[\sigma_2] \parallel x.t_2[\sigma_2], \overrightarrow{\mathsf{a}(w)}.E\rangle_k \xrightarrow{k} \langle t_i[\sigma_2][v[\sigma]/x], \overrightarrow{\mathsf{a}(w)}.E\rangle_k$$

We have $\langle t_i[\sigma_2][v[\sigma_1]], \overrightarrow{\mathsf{a}(w)}.E\rangle_k \in \perp\!\!\!\perp$ because of (2.35) and (2.36). Hence by saturation of $\perp\!\!\!\perp$ we obtain

$$\langle \mathsf{case}\, v[\sigma_1]\, \mathsf{of}\, x.t_1[\sigma_2] \parallel x.t_2[\sigma_2], \overrightarrow{\mathsf{a}(w)}.E\rangle_k \in \perp\!\!\!\perp$$

- **($\forall$ intro.)** We want to show that given that ι^S does not appear free in Γ nor in $\mathcal{E}$ and that $\mathcal{E};\Gamma \vdash_n t : (N,p)$ is $\mathcal{A}$-sound we obtain that $\mathcal{E};\Gamma \vdash_n t : (\forall \iota \in S.N, p)$ is sound too. Let $\rho \Vdash \mathcal{E}$ and $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}$. Let $(E,r) \in [\![\forall \iota \in \sigma.N]\!]_{\mathcal{A},k,\rho} = \bigcup_{a\in S} [\![N]\!]_{\mathcal{A},k,\rho[\iota\leftarrow a]}$. We want to show that $(t,p)[\sigma]\perp_{\mathcal{A}}(E,r)$. We know there exists $a \in S$ such that $(E,r) \in [\![N]\!]_{\mathcal{A},k,\rho[\iota^S\leftarrow a]}$. We have $\rho[\iota^S \leftarrow a] \Vdash \mathcal{E}$ since ι^S does not appear free in $\mathcal{E}$. Moreover $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}\rho[\iota \leftarrow a]$ since ι^S does not appear free in Γ. Therefore, we know by hypothesis that

  $$(t,p)[\sigma] \in [\![N]\!]_{\mathcal{A},k,\rho[\iota\leftarrow a]}{}^{\perp_{\mathcal{A}}}$$

  That concludes the proof.

- **($\forall$ elim.)** We want to show that given that $\mathcal{E};\Gamma \vdash_n t : (\forall \iota \in S.N,p)$ is $\mathcal{A}$-sound then for any $a \in S$, $\mathcal{E};\Gamma \vdash_n t : (N[a/\iota^S],p)$ is $\mathcal{A}$-sound too. Let $\rho \Vdash \mathcal{E}$ and $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}$ and $a \in S$. We know by hypothesis that

  $$\begin{array}{rcl} (t,p)[\sigma] & \in & (\bigcup_{b\in S} \|N\|_{\mathcal{A},k,\rho[\iota^S\leftarrow b]})^{\perp_{\mathcal{A}}} \\ & = & \bigcap_{b\in S} ([\![N]\!]_{\mathcal{A},k,\rho[\iota^S\leftarrow b]})^{\perp_{\mathcal{A}}} \\ & \subseteq & [\![N]\!]_{\mathcal{A},k,\rho[\iota^S\leftarrow a]}{}^{\perp_{\mathcal{A}}} \\ & = & [\![N[a/\iota^S]]\!]_{\mathcal{A},k,\rho}{}^{\perp_{\mathcal{A}}} \end{array}$$

  This concludes the proof.

- **($\exists$ intro.)** We want to show that given that $\mathcal{E};\Gamma \vdash_n v : (P[e/\iota^S],p)$ is $\mathcal{A}$-sound we obtain that $\mathcal{E};\Gamma \vdash_n v : (\exists \iota \in S.P,p)$ is $\mathcal{A}$-sound too. Let $\rho \Vdash \mathcal{E}$ and $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}$. We know by hypothesis that

  $$(v,p)[\sigma] \in \|P[e/\iota^S]\|_{\mathcal{A},k,\rho}$$

  But $\|P[e/\iota^S]\|_{\mathcal{A},k,\rho} = \|P\|_{\mathcal{A},k,\rho[\iota^S\leftarrow\rho(e)]}$ Hence we obtain immediately that:

  $$(v,p)[\sigma] \in \bigcup_{a\in S} \|P\|_{\mathcal{A},k,\rho[\iota^S\leftarrow a]} = \|\exists \iota \in S.P\|_{\mathcal{A},k,\rho}$$

- **($\exists$ elim.)** We suppose that $\mathcal{E};\Gamma \vdash_n v : (\exists \iota \in S.P,p)$ and $\mathcal{E};\Delta,x : P[\kappa/\iota] \vdash_n t : (N,q)$ are $\mathcal{A}$-sound, with κ not appearing free in $\mathcal{E}$, Δ or N. We want to show that

  $$\mathcal{E};\Gamma,\Delta \vdash_n \mathcal{A} : (k, t[v/x])Np+q$$

  Let $\rho \Vdash \mathcal{E}$ and let $\sigma \in \|\Gamma,\Delta\|_{\mathcal{A},k,\rho}$. We have $\sigma = \sigma_1,\sigma_2$ such that $\sigma_1 \in \|\Gamma\|_{\mathcal{A},k,\rho}$ and $\sigma_2 \in \|\Delta\|_{\mathcal{A},k,\rho}$. We know by hypothesis that

  $$(v,p)[\sigma_1] \in \bigcup_{a\in S} \|P\|_{\mathcal{A},k,\rho[\iota^S\leftarrow a]}$$

  So there exists $a \in S$ such that

  $$(v,p)[\sigma_1] \in \|P\|_{\mathcal{A},k,\rho[\iota^S\leftarrow a]} \tag{2.37}$$

On the other hand we observe that since κ^S does not appear free in $\mathcal{E}$ and because $\rho \Vdash \mathcal{E}$, we have:

$$\rho[\kappa^S \leftarrow a] \Vdash \mathcal{E}$$

Moreover, since κ^S does not appear free in Γ we also have

$$\sigma_2 \in \|\Delta\|_{\mathcal{A},k,\rho[\kappa^S \leftarrow a]}$$

Finally, (2.37) implies that

$$(v,p)[\sigma_1] \in \|P[\kappa/\iota]\|_{\mathcal{A},k,\rho[\kappa^S \leftarrow a]}$$

So, by the second hypothesis we obtain that

$$(t[\sigma_2][v[\sigma_1]/x], q[\sigma_2] + p[\sigma_1]) \in \|N\|_{\mathcal{A},k,\rho[\kappa^S \leftarrow a]}$$

But since κ does not appear free in N, then

$$\|N\|_{\mathcal{A},k,\rho[\kappa^S \leftarrow a]} = \|N\|_{\mathcal{A},k,\rho}$$

- **(Inequ. conj. intro.)** We want to show that provided that $\mathcal{E};\Gamma \vdash_n v : (P,p)$ is $\mathcal{A}$-sound and that $f \preceq_{\mathcal{E}} e$, we have $\mathcal{E};\Gamma \vdash_n v : (\{f \preceq_S e\} \wedge P, p)$ is $\mathcal{A}$-sound too. Let $\rho \Vdash \mathcal{E}$ and let $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}$. We know that $[\![f]\!]_\rho \preceq_S [\![e]\!]_\rho$ and then, $\rho \Vdash \mathcal{E}, e \succeq_S f$. Hence, by hypothesis we have that $(v,p)[\sigma] \in \|P\|_{\mathcal{A},k,\rho} = \|\{f \preceq_S e\} \wedge P\|_{\mathcal{A},k,\rho}$.
- **(Inequ. imp. intro.)** We want to show that provided that $\mathcal{E}, e \succeq_S f; \Gamma \vdash_n t : (N,p)$ is $\mathcal{A}$-sound, we have $\mathcal{E};\Gamma \vdash_n t : (\{e \succeq_S f\} \mapsto N, p)$ is $\mathcal{A}$-sound too. Let $\rho \Vdash \mathcal{E}$ and let $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}$. There are two possibilities:
  - If $[\![e]\!]_\rho \succeq_S [\![f]\!]_\rho$ then $\rho \Vdash \mathcal{E}, e \succeq_S f$. Hence, by hypothesis we have that $(t,p)[\sigma] \in [\![N]\!]_{\perp_{\mathcal{A}},=}[\![\{e \succeq_S f\} \mapsto N]\!]_{\mathcal{A},k,\rho}^{\perp_{\mathcal{A}}}$.
  - If $\neg([\![e]\!]_\rho \succeq_S [\![f]\!]_\rho)$ then by hypothesis we have that $(t,p)[\sigma] \in [\![\{e \succeq_S f\} \mapsto N]\!]_{\mathcal{A},k,\rho}^{\perp_{\mathcal{A}}} = \varnothing^{\perp_{\mathcal{A}}} = \mathbb{P}_{\mathcal{A}}$.
- **(Inequ. conj. elim.)** We want to show that provided that $\mathcal{E};\Gamma \vdash_n v : (\{e \preceq_S e\} \wedge P, p)$ is $\mathcal{A}$-sound, $\mathcal{E};\Gamma \vdash_n v : (P,p)$ is $\mathcal{A}$-sound too. Let $\rho \Vdash \mathcal{E}$ and let $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}$. We conclude easily because $\|\{e \preceq_S e\} \wedge P\|_{\mathcal{A},k,\rho} = \|P\|_{\mathcal{A},k,\rho}$.
- **(Inequ. imp. elim)** We want to show that provided that $\mathcal{E};\Gamma \vdash_n t : (\{e \succeq_S e\} \mapsto N, p)$ is sound, $\mathcal{E};\Gamma \vdash_n t : (N,p)$ is sound too. Let $\rho \Vdash \mathcal{E}$ and let $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}$. We conclude easily because $[\![\{e \succeq_S e\} \mapsto N]\!]_{\mathcal{A},k,\rho} = [\![N]\!]_{\mathcal{A},k,\rho}$.
- **(Subtyping)** Finally, we want to show that if $A \sqsubseteq_{\mathcal{E}} B$ and $\mathcal{E};\Gamma \vdash_n a : (A,p)$ is $\mathcal{A}$-sound then $\mathcal{E};\Gamma \vdash_n a : (B,p)$. This is a direct consequence of Theorem 139.

□

4.2.5 Monitors

For now, all typing rules we have proved to be $\mathcal{A}$-sound are either just using the forcing condition **0** or the addition +. Hence, there is no rule that really modifies the forcing condition. That means the programs we build using these rules are not really dependent of what values

are in the memory cells. And indeed, the programs built using these typing rules never change or use the memory cells, since it is the role of the observation $(\!|.|\!)^{\alpha}_{k}$ that has not been considered yet. We have seen in Section 3.2 one kind of typing rule for $(\!|.|\!)^{\alpha}_{k}$. It was however specific to the level 1. There is in fact no evident way to generalize this rule to any level n. However one can see that the part of the rule that is not specific to the level 1 (that is the predicate C) is as follows:

$$\frac{\mathcal{E};\Gamma \vdash_k t : (N,p)}{\mathcal{E};\Gamma \vdash_k (\!|t|\!)^{\alpha}_{k} : (N, f(p))}$$

Where f is a certain function on the forcing monoid. This is clearly where the forcing condition associated with a program is changed. This f expresses the cost we have to pay to trigger an observation. The good news is that given a n-**MA** $\mathcal{A}$, we can give a *sufficient condition* on $\alpha \in \mathbb{P}$ and $f : |\mathcal{A}| \to |\mathcal{A}|$ so that this typing rule becomes $\mathcal{A}$-sound. Every (α, f) satisfying that condition will be called a $\mathcal{A}$-**monitor**. In the next sections and chapters, we will identify certain situations where we can characterize a subset of those monitors.

Definition 141 (Monitor). *Let $\mathcal{A}$ be a n-**MA**. A $\mathcal{A}$-**monitor** is a pair (α, f) where $f : |\mathcal{A}| \to |\mathcal{A}|$, is a strong function and $\alpha \in \mathbb{P}$ are such that:*

$$\forall t \in \mathbb{P}, \forall E \in \mathbb{P}, \forall p \in |\mathcal{A}|, (t, E, p \bullet q) \in \perp\!\!\!\perp_{\mathcal{A},k} \Rightarrow ((\!|t|\!)^{\alpha}_{n}, E, f(p) \bullet q) \in \perp\!\!\!\perp_{\mathcal{A},k}$$

Lemma 142. *If (α, f) is a $\mathcal{A}$-monitor, then we have*

$$(t, p + p') \perp_{\mathcal{A},k} (E, q) \Rightarrow \quad ((\!|t|\!)^{\alpha}_{k}, f(p) + p') \quad \perp_{\mathcal{A},k} \quad (E, q)$$

Proof. Suppose that $(t, p+p') \perp_{\mathcal{A},k} (E, q)$. Then it means that $(t, E, (p+p') \bullet q) \in \perp\!\!\!\perp_{\mathcal{A},k}$. Since (α, f) is a monitor we have

$$((\!|t|\!)^{\alpha}_{k}, E, f((p + p') \bullet q)) \in \perp\!\!\!\perp_{\mathcal{A},k}$$

But since $f((p + p') \bullet q) \preceq (f(p) + p') \bullet q$ (because f is strong), by $\preceq$-saturation of $\perp\!\!\!\perp_{\mathcal{A},k}$ we obtain

$$((\!|t|\!)^{\alpha}_{k}, E, (f(p) + p') \bullet q) \in \perp\!\!\!\perp_{\mathcal{A},k}$$

□

Lemma 143. *To show that (α, f) is a $\mathcal{A}$-monitor, it is enough to show that*

$$\forall t \in \mathbb{P}, \forall E \in \mathbb{P}, \forall p \in |\mathcal{A}|, (t, E, p) \in \perp\!\!\!\perp_{\mathcal{A},k} \Rightarrow ((\!|t|\!)^{\alpha}_{n}, E, f(p)) \in \perp\!\!\!\perp_{\mathcal{A},k}$$

Proof. This is because $f(p \bullet q) \preceq f(p) \bullet q$ (f is strong) and because $\mathcal{C}_{\mathcal{A}}$ is decreasing. □

We now show that for each $\mathcal{A}$-monitor the announced typing rule is indeed $\mathcal{A}$-sound.

Property 144. *Let (α, f) be a $\mathcal{A}$-monitor, where $\mathcal{A}$ is a n-**MA**. Then the following typing rule is $\mathcal{A}$-sound:*

$$\frac{\mathcal{E}; \Gamma \vdash_n t : (N, p)}{\mathcal{E}; \Gamma \vdash_n (\!|t|\!)^{\alpha}_n : (N, f(p))}$$

Proof. Let ρ be a $\mathcal{A}$-model and $\sigma \in \|\Gamma\|_{\mathcal{A},n,\rho}$. Then we know by induction that $(t, p)[\sigma] \in \|N\|_{\mathcal{A},n,\rho}$. Let $(E, q) \in [\![N]\!]_{\mathcal{A},n,\rho}$. Since we have $(t, p)[\sigma] \perp_{\mathcal{A},n} (E, q)$ and because (α, f) is a $\mathcal{A}$-monitor and by Lemma 142, we obtain that

$$(t[\sigma], f(p)[\sigma]) \perp_{\mathcal{A},n} (E, q)$$

This permits us to conclude that $(t, f(p))[\sigma] \in \|N\|_{\mathcal{A},n,\rho}$, hence the $\mathcal{A}$-soundness of the conclusion. □

4.3 | Adding types

We have shown that each k-**MA** gives rise to a sound realizability interpretation of the affine core type system λ_{LCBPV}, or from another point of view of the annotated type system $\lambda^{|\mathcal{A}|}_{\mathsf{LCBPV}}$ at level k. As a programming language, this core is very weak from the point of view of expressivity, as it does not even allow sharing of variables. In order to obtain models of reasonable programming languages, we need to add new types and new typing rules. In this section we explain what we mean by *adding types* to the programming language associated with a **MA**.

In general, adding a type amounts to extend the grammar of positive or negative types and define the corresponding interpretation. However, if we want to automatically extend some of the important properties of the monitoring algebra theory to these new types, we need to structure them. In this thesis, we will only consider adding types through what we call **simple connectives**. Even though there are other ways to build new types, types built using simple connectives are extremely well-behaved.

4.3.1 Simple connectives

We begin by defining what **simple connectives** are in the unary realizability context.

Definition 145 (Connective arity)**.** *A **connective arity** is a function $\sigma : \{+, -\} \to \mathbb{N}$.*

Definition 146 (Simple connective)**.** *We define the notions of positive and negative simple connectives:*

1. *A **positive connective** (of arity σ) $\odot$ is a function $v_\odot : \mathbb{V}^{\sigma(+)} \times \mathbb{E}^{\sigma(-)} \to \mathbb{V}$*
2. *A **negative connective** (of arity σ) $\odot$ is a function $e_\odot : \mathbb{V}^{\sigma(+)} \times \mathbb{E}^{\sigma(-)} \to \mathbb{E}$.*

This notion of simple connective admits a simple realizability interpretation.

Definition 147 (Connective interpretation)**.** *Its interpretation is defined as follows:*

1. *For the positive connectives:*

$$\begin{array}{l} \odot(X_1, \ldots, X_{\sigma(+)}, Y_1, \ldots, Y_{\sigma(-)}) \\ \stackrel{\mathsf{def}}{=} \{\ v_\odot(v_1, \ldots, v_{\sigma(+)}, E_1, \ldots, E_{\sigma(-)}) \mid v_i \in X_i, E_j \in Y_j\ \} \end{array}$$

2. *For the negative connectives:*

$$\begin{array}{l} \odot(X_1, \ldots, X_{\sigma(+)}, Y_1, \ldots, Y_{\sigma(-)}) \\ \stackrel{\mathsf{def}}{=} \{\ e_\odot(v_1, \ldots, v_{\sigma(+)}, E_1, \ldots, E_{\sigma(-)}) \mid v_i \in X_i, E_j \in Y_j\ \} \end{array}$$

Example 148. *The type constructors $\otimes$ are respective examples of such positive simple connective, while $\multimap$ is an example of a negative simple connective. Indeed, here are the corresponding functions:*

$$\begin{array}{rcl} v_\otimes(v, w) & \stackrel{\mathsf{def}}{=} & (v, w) \\ e_\multimap(v, E) & \stackrel{\mathsf{def}}{=} & \mathsf{a}(v).E \end{array}$$

Example 149. *Let $w \in \mathbb{V}$. Consider the following simple constant positive connective*

$$v_\odot(v) \stackrel{\mathsf{def}}{=} w$$

Then when applied to any non empty set X, one obtain a singleton type*:*

$$\odot(X) = \{w\}$$

4.3.2 Simple $\mathcal{A}$-connectives

We now define what simple connectives are in the context of a n-**MA**. We fix such a **MA** denoted $\mathcal{A}$.

Definition 150 (Simple $\mathcal{A}$-connective)**.** *We define the notions of positive and negative simple $\mathcal{A}$-connectives:*

1. *A **positive simple $\mathcal{A}$-connective** (of arity σ) $\odot$ is a pair given by*
   - *A function $v_\odot : \mathbb{V}^{\sigma(+)} \times \mathbb{E}^{\sigma(-)} \to \mathbb{V}$*
   - *A function $\phi_\odot : |\mathcal{A}|^{\sigma(+)+\sigma(-)} \to |\mathcal{A}|$*
2. *A **negative simple $\mathcal{A}$-connective** (of arity n) $\odot$ is given by:*
   - *A function $e_\odot : \mathbb{V}^{\sigma(+)} \times \mathbb{E}^{\sigma(-)} \to \mathbb{E}$*
   - *A function $\phi_\odot : |\mathcal{A}|^{\sigma(+)+\sigma(-)} \to |\mathcal{A}|$*

Definition 151 (Simple connective interpretation)**.** *Its interpretation is defined as follows:*

1. *For the positive simple connectives:*

$$\begin{aligned}&\odot(X_1, \ldots, X_{\sigma(+)}, Y_1, \ldots, Y_{\sigma(-)})\\ &\stackrel{\text{def}}{=} \{\ (v_\odot(\overrightarrow{v_i}, \overrightarrow{E_j}), \phi_\odot(\overrightarrow{p_i}, \overrightarrow{q_j})) \mid (v_i, p_i) \in X_i, (E_j, q_j) \in Y_j\ \}\end{aligned}$$

2. *For the negative simple connectives:*

$$\begin{aligned}&\odot(X_1, \ldots, X_{\sigma(+)}, Y_1, \ldots, Y_{\sigma(-)})\\ &\stackrel{\text{def}}{=} \{\ (e_\odot(\overrightarrow{v_i}, \overrightarrow{E_j}), \phi_\odot(\overrightarrow{p_i}, \overrightarrow{q_j})) \mid (v_i, p_i) \in X_i, (E_j, q_j) \in Y_j\ \}\end{aligned}$$

To clarify the definition, we have used the notation $(\overrightarrow{v_i}, \overrightarrow{E_j})$ to denote the tuple

$$(v_1, \ldots, v_{\sigma(+)}, E_1, \ldots, E_{\sigma(-)})$$

And $(\overrightarrow{p_i}, \overrightarrow{q_j})$ to denote the tuple

$$(p_1, \ldots, p_{\sigma(+)}, q_1, \ldots, q_{\sigma(-)})$$

Remark 152. *Every simple $\mathcal{A}$-connective also defines a simple connective by keeping only the function $v_\odot$ (resp. $e_\odot$). In some sense it is a pair of connectives on both components.*

Example 153. *$\multimap$, $\otimes$ are examples of simple $\mathcal{A}$-connectives, with the following associated functions.*

$$\begin{aligned}\phi_\otimes(p, q) &\stackrel{\text{def}}{=} p + q\\ \phi_\multimap(p, q) &\stackrel{\text{def}}{=} p \bullet q\end{aligned}$$

Example 154. *Let $w \in \mathbb{V}$ and $r \in |\mathcal{A}|$. Consider the following simple constant positive connective*

$$\begin{aligned} v_\odot(v) &\stackrel{\text{def}}{=} w \\ \phi_\odot(p) &\stackrel{\text{def}}{=} r \end{aligned}$$

Then when applied to any non empty set X, one obtain the upward closure of a singleton type*:*

$$\odot(X) = \{\ (w, p) \mid r \preceq p\ \}$$

Notation 155. *We will often take the liberty to use the same notation $\odot$ for all the components of a simple $\mathcal{A}$-connective, if there is no possible ambiguity.*

4.3.3 Forcing transformation

One of the main interests of simple $\mathcal{A}$-connective is that they admit a *forcing interpretation*, *i.e.* the forcing type translation can be extended to those new connectives in a canonical way.

Positive

Let $(v_\odot, \phi_\odot)$ be a positive $\mathcal{A}$-connective. Given $X_1, \dots, X_{\sigma(+)}$ be positive predicates of arity $|\mathcal{A}|$, and $Y_1, \dots, Y_{\sigma(-)}$ be negative predicates of arity $|\mathcal{A}|$, $\odot$ induces a positive predicate of arity $|\mathcal{A}|$ denoted by

$$(\odot(X_1, \dots, X_{\sigma(+)}, Y_1, \dots, Y_{\sigma(-)}))^*(\iota)$$

which is defined as follows (ι is a fresh variable of sort $|\mathcal{A}|$):

$$\begin{aligned} &\exists p_1, \dots, p_{\sigma(+)}, q_1, \dots, q_{\sigma(-)} \in |\mathcal{A}|, \\ &\quad \{\phi_\odot(\overrightarrow{p_i}, \overrightarrow{q_j}) \preceq_{|\mathcal{A}|} \iota\} \wedge (\odot(X_1(p_1), \dots, X_{\sigma(+)}(p_{\sigma(+)}), Y_1(q_1), \dots, Y_{\sigma(-)}(q_{\sigma(-)}))) \end{aligned}$$

Negative

Similarly, if $(e_\odot, \phi_\odot)$ is a negative $\mathcal{A}$-connective. Given $X_1, \dots, X_{\sigma(+)}$ be positive predicates of arity $|\mathcal{A}|$, and $Y_1, \dots, Y_{\sigma(-)}$ be negative predicates of arity $|\mathcal{A}|$, $\odot$ induces a negative predicate of arity $|\mathcal{A}|$ denoted by

$$(\odot(X_1, \dots, X_{\sigma(+)}, Y_1, \dots, Y_{\sigma(-)}))^\circ(\iota)$$

which is defined as follows:

$$\begin{aligned} &\exists p_1, \dots, p_{\sigma(+)}, q_1, \dots, q_{\sigma(-)} \in |\mathcal{A}|, \\ &\quad \{\iota \succeq_{|\mathcal{A}|} \phi_\odot(\overrightarrow{p_i}, \overrightarrow{q_j})\} \mapsto (\odot(X_1(p_1), \dots, X_{\sigma(+)}(p_{\sigma(+)}), Y_1(q_1), \dots, Y_{\sigma(-)}(q_{\sigma(-)}))) \end{aligned}$$

Example 156. *Here is an example on how the inductive forcing interpretation of types would be extended when considering a negative binary simple $\mathcal{A}$-connective $\odot$, of arity $\sigma(+) = 1 = \sigma(-)$:*

$$(\odot(P,N))^\circ(p) \quad = \quad \exists q, r \in |\mathcal{A}|.\{p \succeq_{|\mathcal{A}|} \phi_\odot(q,r)\} \mapsto \odot(P^*(q), N^\circ(q))$$

It is clear that if we take the example of the $\multimap$ connective, we get back its original forcing interpretation.

4.4 | Properties of 1-Monitoring Algebras

We now turn our attention to 1-**MA**s, which are particularly well-behaved **MA**s. We show that they are particularly interesting on the two following aspects:

- First, for each 1-**MA**, we are able to identify a substantial class of monitors in 1-**MA**s using typing, which corresponds to the typing rule at level 1 given in Section 3.2 for the observation constructor.
- We then prove a crucial result: the **connection theorem**. It formally shows how each 1-**MA** comes from the composition of a unary realizability with a forcing transformation. Or said otherwise, each $\mathcal{A}$-model comes from a forcing model chosen inside a unary realizability model. This theorem is not only conceptually interesting but also important in practice, as we will see in the next chapters, as it will make possible to use the characteristics of a given **MA** to build another one *inside it*.

These two points do not hold in general for n-**MA**s, but they are not restricted to 1-**MA**s either. We will show in the next chapter how it possible to build n-**MA**s that enjoy generalized form of those two good properties.

4.4.1 Monitors

We first show the soundness of the typing rule for the observation constructor $(\!|.|\!)_1^\alpha$ already given in Section 3.2. We in fact give a sufficient condition for a pair (α, f) to be a $\mathcal{A}$-monitor, by *only* considering the unary realizability model of Section 4.1. This reflects the fact that the observation makes the level of the configuration temporarily pass from 1 to 0.

Property 157. *Let $\mathcal{A}$ be a 1-**MA**. Suppose we have a pair (α, f) where:*

- *$\alpha \in \mathbb{T}$ is a closed term.*
- *$f : |\mathcal{A}| \to |\mathcal{A}|$ is strong.*

Then (α, f) is a $\mathcal{A}$-monitor if the following holds:

$$\alpha \Vdash_0 \forall x \in |\mathcal{A}|.\mathcal{C}_{\mathcal{A}}(f(x)) \multimap \Uparrow \mathcal{C}_{\mathcal{A}}(x)$$

Proof. By Lemma 143, it is enough to show that given $(t, E, p) \in \perp\!\!\!\perp_{\mathcal{A},k}$, we have $(\!|t|\!)_1^{\alpha}, E, f(p)) \in \perp\!\!\!\perp_{\mathcal{A},k}$. So let $v \in \mathcal{C}_{\mathcal{A}}(f(p))$. We then want to prove that

$$\langle (\!|t|\!)_1^{\alpha}, \mathsf{a}(v).E\rangle_1 \in \perp\!\!\!\perp$$

But this configuration reduces to $\langle \alpha, \mathsf{a}(v).\mathsf{m}_1(t).E\rangle_0$. To conclude by $\rightarrow$-saturation of $\perp\!\!\!\perp$, we only need to show that $\alpha \perp_1 \mathsf{a}(v).\mathsf{m}_1(t).E$, but since $\alpha \Vdash_0 \forall x \in |\mathcal{A}|.\mathcal{C}_{\mathcal{A}}(f(x)) \multimap \Uparrow \mathcal{C}_{\mathcal{A}}(x)$, it is enough to show:

$$\mathsf{a}(v).\mathsf{m}_1(t).E \in [\![\mathcal{C}_{\mathcal{A}}(f(p)) \multimap \Uparrow \mathcal{C}_{\mathcal{A}}(p)]\!]_{0,[]}$$

Since $v \in \mathcal{C}_{\mathcal{A}}(f(p))$, it is enough to show that $\mathsf{m}_1(t).E \in \mathcal{C}_{\mathcal{A}}(p)^{\perp_0}$. Let $w \in \mathcal{C}_{\mathcal{A}}(p)$. We have

$$\langle w, \mathsf{m}_1(t).E\rangle_0 \rightarrow \langle t, \mathsf{a}(w).E\rangle_1$$

The latter is in $\perp\!\!\!\perp$ since $(t, E, p) \in \perp\!\!\!\perp_{\mathcal{A},k}$, hence the conclusion by $\rightarrow$-saturation of $\perp\!\!\!\perp$.

□

As an immediate corollary of Proposition 157, we get the following extension of the Soundness theorem.

Corollary 158. *Let $\mathcal{A}$ be a 1-**MA**. Then the following rule is $\mathcal{A}$-sound:*

$$\frac{\mathcal{E}; \Gamma \vdash_1 t : (N, p) \qquad \mathcal{E}; \vdash_0 \alpha : \forall x \in |\mathcal{A}|.\mathcal{C}_{\mathcal{A}}(f(x)) \multimap \mathcal{C}_{\mathcal{A}}(x) \qquad f \text{ is strong}}{\mathcal{E}; \Gamma \vdash_1 (\!|t|\!)_1^{\alpha} : (N, f(p))}$$

It is clear that this reasoning does not make sense in the general case of a $(n+1)$-**MA** $\mathcal{A}$: $\mathcal{A}$ is not necessarily built using a n-**MA**, whereas a 1-**MA** is built on top of the simple realizability (which is morally a 0-**MA**). We would not know what to use as a replacement of the side condition:

$$\alpha : \forall x \in |\mathcal{A}|.\mathcal{C}_{\mathcal{A}}(f(x)) \multimap \mathcal{C}_{\mathcal{A}}(x)$$

4.4.2 Connection theorem

The **connection theorem** is a crucial technical result about 1-**MA**s. It formally shows that each 1-**MA** is the composition of the simple realizability model at level 1 described in Section 4.1 and of a forcing structure as described in Chapter III . This result is conceptually and technically important, as it will be used in the next chapters to build models of complex programming languages. To establish the connection theorem, we start with:

- A 1-**MA** $\mathcal{A}$.
- A $\mathcal{A}$-model ρ.

We want to show that our $\mathcal{A}$-model ρ can be seen as a forcing model inside a certain unary realizability model $\overline{\rho}$. We begin by choosing the following forcing structure induced by $\mathcal{A}$ and noted $\mathcal{F}(\mathcal{A})$:

- The forcing monoid is $|\mathcal{A}|$.
- The predicate is a fixed predicate variable C.

And we suppose disposing of a $\mathcal{F}(\mathcal{A})$-model, that is a function mapping all predicate variables X of arity $T_1 \times \cdots \times T_k$ to another variable $X^* : T_1 \times \cdots \times T_k \times |\mathcal{A}|$.

Remark 159. *$\mathcal{F}(\mathcal{A})$ is not a forcing structure* stricto sensu*, as C is not provably decreasing. But inside the unary model $\overline{\rho}$, it will be.*

We now define the desired unary model $\overline{\rho}$, which depends on ρ.

- The valuation $\overline{\rho}$ is defined as follows:
  - On first-order expressions e, $\overline{\rho}(e) = \rho(e)$.
  - The valuation of the predicate C of arity $|\mathcal{A}|$ is given by
  $$\overline{\rho}(\mathsf{C})(p) = \mathcal{C}_{\mathcal{A}}(p)$$
  - If X is a predicate arity of arity $T_1 \times \cdots \times T_k$ we define the valuation $\overline{\rho}(X^*)$ of X^* as:
  $$\overline{\rho}(X^*)(e_1, \ldots, e_k, n) = \{\ v \in \mathbb{V} \mid (v, p) \in \rho(e_1, \ldots, e_k)\ \}$$

With these definitions we can show how the realizability model ρ induced by $\mathcal{A}$ and the forcing relation defined in the unary realizability model $\overline{\rho}$ are connected. We moreover prove it not only for the basic types of λ_{LCBPV}, but for an extension of those types with *any* simple $\mathcal{A}$-connective. To do this, we state and prove the following Connection Lemma:

Lemma 160. *Suppose that ρ is a $\mathcal{A}$-model, $p \in \mathcal{M}$, $v \in \mathbb{V}$, $t \in \mathbb{P}$ and $E \in \mathbb{E}$. Suppose that P and N are types built using the core grammar of λ_{LCBPV} augmented with any number of simple $\mathcal{A}$-connectives. Then the three following equivalences hold:*

$$\begin{array}{lllll} (1) & (v,p) \in \|P\|_{\mathcal{A},n,\rho} & \iff & v \in \|P^*(p)\|_{n,\overline{\rho}} \\ (2) & (E,p) \in [\![N]\!]_{\mathcal{A},n,\rho} & \iff & E \in [\![N^\circ(p)]\!]_{n,\overline{\rho}} \\ (3) & (t,p) \in \|N\|_{\mathcal{A},n,\rho} & \iff & t \in \|N^*(p)\|_{n,\overline{\rho}} \end{array}$$

Proof. We prove the three statements by mutual induction on the type.

1. We begin by proving (1), by looking at all the possible cases. We fix $k \in \mathbb{N}$, ρ a simple valuation.

- **Predicate variable.** Given a predicate variable X of signature $S_1 \times \cdots \times S_n$ and $e_1, \ldots, e_n$ respectively in S_1-Exp, $\ldots$, S_n-Exp, we have:

$$\begin{array}{rcl}(v,p) \in \|X(e_1,\ldots,e_n)\|_{\mathcal{A},k,\rho} & \Leftrightarrow & \exists q \in |\mathcal{A}|, q \preceq p \wedge (v,q) \in \rho(X)([\![e_1]\!]_\rho, \ldots, [\![e_n]\!]_\rho) \\ & \Leftrightarrow & \exists q \in |\mathcal{A}|, q \preceq p \wedge v \in \overline{\rho}(X^*)([\![e_1]\!]_{\overline{\rho}}, \ldots, [\![e_n]\!]_{\overline{\rho}}, \overline{\rho}(q)) \\ & \Leftrightarrow & \exists q \in |\mathcal{A}|, q \preceq p \wedge v \in \overline{\rho}(X^*)([\![e_1]\!]_{\overline{\rho}}, \ldots, [\![e_n]\!]_{\overline{\rho}}, q) \\ & \Leftrightarrow & v \in \|\exists q \in |\mathcal{A}|.\{q \preceq_{|\mathcal{A}|} p\} \wedge \overline{\rho}(X^*)([\![e_1]\!]_{\overline{\rho}}, \ldots, [\![e_n]\!]_{\overline{\rho}}, q)\|_{k,\overline{\rho}} \\ & \Leftrightarrow & v \in \|(X(e_1,\ldots,e_n))^*(p)\|_{k,\overline{\rho}}\end{array}$$

- **Primitive integers.** Let $v \in \mathbb{V}$ and $p \in \mathcal{M}$. We have the following equivalences:

$$\begin{array}{rcl}(v,p) \in \|\mathsf{Nat}\|_{\mathcal{A},k,\rho} & \Leftrightarrow & \exists n \in \mathbb{N}, v = \underline{\mathsf{n}} \\ & \Leftrightarrow & v \in \|\mathsf{Nat}\|_{k,\overline{\rho}} \\ & \Leftrightarrow & v \in \|\mathsf{Nat}^*(p)\|_{k,\overline{\rho}}\end{array}$$

- **Unit.** Let $v \in \mathbb{V}$ and $p \in \mathcal{M}$. We have the following equivalences:

$$\begin{array}{rcl}(v,p) \in \|\mathbb{1}\|_{\mathcal{A},k,\rho} & \Leftrightarrow & v = *\\ & \Leftrightarrow & v \in \|\mathbb{1}\|_{k,\overline{\rho}} \\ & \Leftrightarrow & v \in \|\mathbb{1}^*(p)\|_{k,\overline{\rho}}\end{array}$$

- **Positive $\mathcal{A}$-connective**. Suppose that $\odot$ is a $\mathcal{A}$-connective. We suppose to simplify the presentation, but without any loss of generality that its signature is such that $\sigma(+) = 1 = \sigma(-)$, hence we consider the type $\odot(P,N)$ for some positive type P and some negative type N. We suppose that by induction, (1) is true for P $(H1)$ and (2) is true for N $(H2)$. Then we have:

$$\begin{array}{rcll} & (v,p) \in \|\odot(P,N)\|_{\mathcal{A},k,\rho} & & \\ \Leftrightarrow & \exists q_1, q_2 \in |\mathcal{A}|, \odot(q_1,q_2) \preceq_{|\mathcal{A}|} p, \exists (w,E) \in \mathbb{V} \times \mathbb{E}, v = \odot(w,E) & & \\ & \wedge (w,q_1) \in \|P\|_{\mathcal{A},k,\rho} \wedge (E,q_2) \in [\![N]\!]_{\mathcal{A},k,\rho} & & \\ \Leftrightarrow & \exists q_1, q_2 \in |\mathcal{A}|, \odot(q_1,q_2) \preceq_{|\mathcal{A}|} p, \exists (w,E) \in \mathbb{V} \times \mathbb{E}, v = \odot(w,E) & & \\ & \wedge w \in \|P^*(q_1)\|_{k,\overline{\rho}} \wedge E \in [\![N^\circ(q_2)]\!]_{k,\overline{\rho}} & & (H1),(H2) \\ \Leftrightarrow & \exists q_1, q_2 \in |\mathcal{A}|, \odot(q_1,q_2) \preceq_{|\mathcal{A}|} p, v \in \|\odot(P^*(q_1), N^\circ(q_2))\|_{k,\overline{\rho}} & & \\ \Leftrightarrow & v \in \|\exists q_1, q_2 \in |\mathcal{A}|, \{\odot(q_1,q_2) \preceq_{|\mathcal{A}|} p\} \wedge \odot(P^*(q_1), N^\circ(q_2))\|_{k,\overline{\rho}} & & \\ \Leftrightarrow & v \in \|(\odot(P,N))^*(p)\|_{k,\overline{\rho}} & & (\text{Def of } .^*)\end{array}$$

- **Tensor.** The tensor is an example of positive $\mathcal{A}$-connective, hence it is already proved.
- **Positive shift.** Let $v \in \mathbb{V}$ and $p \in \mathcal{M}$. We suppose that (3) is true for the computation type N (IH). We know that $(v,p) \in \|\Downarrow N\|_{\mathcal{A},k,\rho}$ is equivalent to

$$\exists t \in \mathbb{P}, v = \mathsf{thunk}(t) \wedge (t,p) \in ([\![N]\!]_{\mathcal{A},k,\rho})^{\perp_{\mathcal{A},k}}$$

But because of (IH) this is equivalent to

$$\exists t \in \mathbb{P}, v = \mathsf{thunk}(t) \wedge t \in \|\forall x \in \mathcal{M}.\mathsf{C}(p \bullet x) \multimap N^*(x)\|_{k,\overline{\rho}}$$

This is in turn equivalent to

$$v \in \|\Downarrow(\forall x \in \mathcal{M}.\mathsf{C}(p \bullet x) \multimap N^*(x))\|_{k,\overline{\rho}}$$

And finally, by definition of $(\Downarrow N)^*(p)$ this last proposition is equivalent to

$$v \in \|(\Downarrow N)^*(p)\|_{k,\overline{\rho}}$$

- **Existential quantifier.** Let $v \in \mathbb{V}$ and $p \in \mathcal{M}$. We suppose that (1) is true for P (IH).

$$\begin{array}{rcll}(v,p) \in \|\exists x \in T.P\|_{\mathcal{A},k,\rho} & \Leftrightarrow & \exists r \in T, (v,p) \in \|P\|_{\mathcal{A},k,\rho[x\leftarrow r]} & \\ & \Leftrightarrow & \exists r \in T, v \in \|P^*(p)\|_{k,\overline{\rho}[x\leftarrow r]} & (IH) \\ & \Leftrightarrow & v \in \|\exists x \in T.P^*(p)\|_{k,\overline{\rho}} & \\ & \Leftrightarrow & v \in \|(\exists x \in T.P)^*(p)\|_{k,\overline{\rho}} & (\text{def of } (.)^*)\end{array}$$

- **Inequational conjunction.** Let $v \in \mathbb{V}$ and $p \in \mathcal{M}$. Now suppose that (1) is true for P (IH).

$$\begin{array}{rcl}(v,p) \in \|\{s \preceq_T r\} \wedge P\|_{\mathcal{A},k,\rho} & \Leftrightarrow & [\![r]\!]_\rho \succeq_T [\![s]\!]_\rho \wedge (v,p) \in \|P\|_{\mathcal{A},k,\rho} \\ & \Leftrightarrow & [\![r]\!]_{\overline{\rho}} \succeq_T [\![s]\!]_{\overline{\rho}} \wedge v \in \|P^*(p)\|_{k,\overline{\rho}} \\ & \Leftrightarrow & v \in \|\{s \preceq_T r\} \wedge P^*(p)\|_{k,\overline{\rho}} \\ & \Leftrightarrow & v \in \|(\{s \preceq_T r\} \wedge P)^*(p)\|_{k,\overline{\rho}}\end{array}$$

2. We now prove the proposition (2) by looking at all possible cases for negative types. In all cases we fix $k \in \mathbb{N}$ and ρ a simple valuation.

   - **Negative $\mathcal{A}$-connective**. Suppose that $\odot$ is a negative $\mathcal{A}$-connective. We suppose to simplify the presentation, but without any loss of generality that its signature is such that $\sigma(+) = 1 = \sigma(-)$, hence we consider the type $\odot(P,N)$ for some positive type P and some negative type N. We suppose that by induction, (1) is true for P $(H1)$ and (2) is true for N $(H2)$. Then we have:

$$\begin{array}{rcll} & (E,p) \in [\![\odot(P,N)]\!]_{\mathcal{A},k,\rho} & & \\ \Leftrightarrow & \exists q_1,q_2 \in |\mathcal{A}|, \odot(q_1,q_2) \preceq_{|\mathcal{A}|} p, \exists (w,E') \in \mathbb{V}\times\mathbb{E}, E = \odot(w,E') & & \\ & \wedge (w,q_1) \in \|P\|_{\mathcal{A},k,\rho} \wedge (E',q_2) \in [\![N]\!]_{\mathcal{A},k,\rho} & & \\ \Leftrightarrow & \exists q_1,q_2 \in |\mathcal{A}|, \odot(q_1,q_2) \preceq_{|\mathcal{A}|} p, \exists (w,E') \in \mathbb{V}\times\mathbb{E}, E = \odot(w,E') & & \\ & \wedge w \in \|P^*(q_1)\|_{k,\overline{\rho}} \wedge E' \in [\![N^\circ(q_2)]\!]_{k,\overline{\rho}} & (H1),(H2) & \\ \Leftrightarrow & \exists q_1,q_2 \in |\mathcal{A}|, \odot(q_1,q_2) \preceq_{|\mathcal{A}|} p, E \in [\![\odot(P^*(q_1),N^\circ(q_2))]\!]_{k,\overline{\rho}} & & \\ \Leftrightarrow & E \in [\![\exists q_1,q_2 \in |\mathcal{A}|, \{p \succeq_{|\mathcal{A}|} q_1+q_2\} \mapsto \odot(P^*(q_1),N^\circ(q_2))]\!]_{k,\overline{\rho}} & & \\ \Leftrightarrow & E \in [\![(\odot(P,N))^\circ(p)]\!]_{k,\overline{\rho}} & (\text{Def of } .^*) & \end{array}$$

   - **Linear implication.** The linear implication is particular case of negative $\mathcal{A}$-connective. Hence it is already proved.
   - **Negative shift.** Suppose that (1) is true for P (IH). Take $E \in \mathbb{E}$ and $q \in \mathcal{M}$. We prove the following equivalences:

$$\begin{array}{rcll} & (E,q) \in [\![\Uparrow P]\!]_{\mathcal{A},k,\rho} & & \\ \Leftrightarrow & \forall (w,p) \in \|P\|_{\mathcal{A},k,\rho}, \forall u \in \mathcal{C}_{\mathcal{A}}(p \bullet q), \langle (w,u), \uparrow E\rangle_k \in \perp\!\!\!\perp & & \\ \Leftrightarrow & \forall p \in \mathcal{M}, \forall w \in \|P^*(p)\|_{k,\overline{\rho}}, \forall u \in \mathcal{C}_{\mathcal{A}}(p \bullet q), \langle (w,u), \uparrow E\rangle_k \in \perp\!\!\!\perp & (IH) & \\ \Leftrightarrow & \forall p \in \mathcal{M}, E \in [\![\Uparrow(P^*(p) \otimes \mathsf{C}(p \bullet q))]\!]_{k,\overline{\rho}} & & \\ \Leftrightarrow & E \in [\![\Uparrow(\exists x \in \mathcal{M}.P^*(x) \otimes \mathsf{C}(x \bullet q))]\!]_{k,\overline{\rho}} & & \\ \Leftrightarrow & E \in [\![(\Uparrow P)^\circ(q)]\!]_{k,\overline{\rho}} & (\text{Def of } (.)^*) & \end{array}$$

   - **Universal quantification.** Suppose that (2) is true for N (IH). For all $E \in \mathbb{E}$ and $p \in \mathcal{M}$ we have the following:

$$\begin{array}{rcll}(E,p) \in [\![\forall x \in T.N]\!]_{\mathcal{A},k,\rho} & \Leftrightarrow & \exists r \in T, (E,p) \in [\![N]\!]_{\mathcal{A},k,\rho[x\leftarrow r]} & \\ & \Leftrightarrow & \exists r \in T, E \in [\![N^\circ(p)]\!]_{k,\overline{\rho}[x\leftarrow r]} & (IH) \\ & \Leftrightarrow & E \in [\![\forall x \in T.N^\circ(p)]\!]_{k,\overline{\rho}} & \\ & \Leftrightarrow & E \in [\![(\forall x \in T.N)^\circ(p)]\!]_{k,\overline{\rho}} & (\text{Def of } (.)^*)\end{array}$$

- **Inequational implication.** Let $E \in \mathbb{E}$ and $p \in \mathcal{M}$. Now suppose that (2) is true for N (IH).

$$\begin{array}{rcl}(E,p) \in [\![\{r \succeq_T s\} \mapsto N]\!]_{\mathcal{A},k,\rho} & \Leftrightarrow & [\![r]\!]_\rho \succeq_T [\![s]\!]_\rho \wedge (E,p) \in [\![N]\!]_{\mathcal{A},k,\rho} \\ & \Leftrightarrow & [\![r]\!]_{\overline{\rho}} \succeq_T [\![s]\!]_{\overline{\rho}} \wedge E \in [\![N^\circ(p)]\!]_{k,\overline{\rho}} \\ & \Leftrightarrow & E \in [\![\{r \succeq_T s\} \mapsto N^\circ(p)]\!]_{k,\overline{\rho}} \\ & \Leftrightarrow & E \in [\![(\{r \succeq_T s\} \mapsto N)^\circ(p)]\!]_{k,\overline{\rho}}\end{array}$$

3. We finally prove (3) by showing that it is a direct consequence of (2). Suppose that (2) is true for N (H). Let $t \in \mathbb{P}$ and $p \in \mathcal{M}$. We have the following equivalences:

$$\begin{array}{rl} & (t,p) \in \|N\|_{\mathcal{A},k,\rho} \\ \Leftrightarrow & \forall (E,q) \in [\![N]\!]_{\mathcal{A},k,\rho}, \forall u \in \mathcal{C}_{\mathcal{A}}(p \bullet q), \langle t, \mathsf{a}(u).E\rangle_k \in \perp\!\!\!\perp \\ \Leftrightarrow & \forall q \in \mathcal{M}, \forall E \in [\![N^\circ(q)]\!]_{k,\overline{\rho}}, \forall u \in \mathcal{C}_{\mathcal{A}}(p \bullet q), \langle t, \mathsf{a}(u).E\rangle_k \in \perp\!\!\!\perp \quad (H) \\ \Leftrightarrow & \forall q \in \mathcal{M}, t \in \|\mathsf{C}(p \bullet q) \multimap N^\circ(q)\|_{k,\overline{\rho}} \\ \Leftrightarrow & t \in \|\forall x \in \mathcal{M}.\mathsf{C}(p \bullet x) \multimap N^\circ(x)\|_{k,\overline{\rho}} \\ \Leftrightarrow & t \in \|N^*(p)\|_{k,\overline{\rho}}\end{array}$$

□

As a corollary, we obtain the more graphical **Connection theorem**.

Theorem 161 (Connection theorem)**.** *Let $(t,p) \in \mathbb{P}_{\mathcal{A}}$ and N be a negative type, and ρ a $\mathcal{A}$-model. Then the following equivalence holds:*

$$(t,p) \Vdash_{\mathcal{A},k} N[\rho] \Longleftrightarrow t \Vdash_k (p \ \ \mathbb{F}_{F(\mathcal{A})} \ N)[\overline{\rho}]$$

4.5 | Basic 1-**MA**s examples

To make sense of the definition of Monitoring Algebra, we now give some examples of **MA**s. Each of these examples illustrate a different aspect and use-case of **MA**s, hence covering most of the future uses we will encounter in this thesis. We just give the main definitions and intuitions and leave most proofs of the results mentionned here for the next chapters, where they are extensively developped. The examples detailed in this section are as follows:

- **Time monitoring Algebra:** this example illustrates how different **MA**s can be used to *observe refined computational properties of programs.* As we have already remarked, when we use no monitor, we basically observe *termination* of programs. However, for a choice of **MA** and when programs are decorated with observations $(\!|.|\!)^1_\alpha$, we can observe much more involved properties. Here we illustrate this point of view by defining an algebra that allows to observe **bounded-time termination**, as we have seen in Subsection 2.3.4. As a corollary of the soundness theorem applied to this **MA**, we obtain a linear-time termination for the programs typable in λ_{LCBPV}.

- **First-order references Algebra:** we show how we can use **MA**s to *add new primitives* to our language. Indeed, remember that our language is parametric in the set $\mathcal{K}$ of primitives. Hence, we can very well add new programming features that way, but we have no guarantee that we can find associated *typing rules* that can be shown to be adequate with respect to any **MA**. However, considering a specific **MA** (or a class of **MA**s) may allow to find such a typing rule. Here, we add a new primitive $\mathsf{swap}(v)$ that takes the content of the memory cell, returns it and replace it by v. We also give a typing rule for such a primitive when applied only to primitive integers. We show that the first-order references algebra is sound with respect to this typing rule. This example makes no use of monitors.
- **Step-indexing Algebra: MA**s can also be used to *add new types* to our type system. Indeed in some **MA**s, one can define new interpretations associated to meaningful new types. In Subsection 4.5.3, we give the example of the **step-indexing algebra**. It has the particularity of allowing an interpretation of various kinds of recursive types. It moreover allows to handle program fixed-point combinators.

4.5.1 Time monitoring

We now define a particular algebra denoted $\mathcal{A}_{\mathsf{time}}$ which has several particularities.

- This algebra is based on a forcing monoid that is fundamentally linear. It gives a first example of a **MA** such that the following contraction principle is not sound in general:
$$A \multimap A \otimes A$$
- This algebra is resource-conscious. Indeed, there is a $\mathcal{A}_{\mathsf{time}}$-monitor that allows us to track the number of reduction steps of programs. We in fact use the monitor based on the combinator α_{time} already defined in Subsection 2.3.4. As already noted, observing the termination of $\{t\}^{\alpha_{\mathsf{time}}}$ at level 1 amounts to observe the bounded-time termination of t at level 0. As we will see, this allows us to derive a linear time execution property of typable programs as a consequence of Theorem 140 applied to this particular algebra.

Definition 162 (Time monitoring Algebra)**.** *The **time monitoring algebra**, denoted by $\mathcal{A}_{\mathsf{time}}$ is defined as follows:*

- *$|\mathcal{A}_{\mathsf{time}}|$ is the additive forcing monoid is $(\mathbb{N}, +, 0, \leq)$.*
- *The test function is defined by*
$$\mathcal{C}_{\mathcal{A}_{\mathsf{time}}}(n) \stackrel{\mathsf{def}}{=} \{\ \underline{\mathsf{k}} \mid k \geq n\ \}$$

We remind the definition of α_{time}:

$$\alpha_{\mathsf{time}} \stackrel{\mathsf{def}}{=} \lambda x.\mathsf{case}\, x\, \mathsf{of}\, x.\Omega \parallel x.\mathsf{ret}(x)$$

It gives the first concrete example of a $\mathcal{A}_{\mathsf{time}}$-monitor.

Property 163. *The pair* $(\alpha_{\mathsf{time}}, x \mapsto x+1)$ *is a* $\mathcal{A}_{\mathsf{time}}$*-monitor.*

The soundness theorem for **MA**s has a nice corollary: the **linear-time termination theorem**. Each computation t typable in λ_{LCBPV} has the property that the number of λ-steps needed for the configuration $\langle t, \mathsf{nil}\rangle_0$ to terminate is bounded by the number $|t|_\lambda$ of λ in t.

Theorem 164 (Linear-time execution)**.** *Suppose that* $\vdash t : N$*. Then:*

$$\langle t, \mathsf{nil}\rangle_0 \to^* \langle \mathsf{ret}(v), \mathsf{nil}\rangle_0$$

Moreover, the number of λ*-steps* k *is such that* $k \leq |t|_\lambda$*.*

We will prove this theorem in Chapter VI , Section 6.2, where we generalize this construction by defining the class of **quantitative monitoring algebras**.

4.5.2 First-order references

As we have already remarked several times, the forcing program transformation is an indexed version of the usual state monad program transformation. Hence, by making the indexed part trivial, it seems reasonable to obtain a **MA** that can deal with simple first-order references.

Definition 165. *We define* $\mathcal{A}_{\mathsf{ref}[1]}$ *as the* ***MA*** *given by:*

- *The trivial forcing monoid on the singleton* $\{\star\}$*.*
- *The test function* $\mathcal{C}_{\mathcal{A}_{\mathsf{ref}[1]}}(\star) \stackrel{\mathsf{def}}{=} \{\ \underline{n} \mid n \in \mathbb{N}\ \}$*.*

We can show that it is possible to add and type new primitives in our language. Consider the primitive $\mathsf{swap}(.)$ with the following reduction rule:

$$\langle \mathsf{swap}(v), \mathsf{a}(w).E\rangle_1 \xrightarrow{1} \langle \mathsf{ret}(w), \mathsf{a}(v).E\rangle_1$$

We suppose that this primitive is in the set $\mathcal{K}$ of additional primitives and that the reduction contains the previous reduction step. This primitive linearly exchanges the value in the memory cell with the one given in argument.

Property 166. *The following typing rule is* $\mathcal{A}_{\mathsf{ref}[1]}$*-sound.*

$$\frac{\mathcal{E}; \Gamma \vdash_1 v : (\mathsf{Nat}, \star)}{\mathcal{E}; \Gamma \vdash_1 \mathsf{swap}(v) : (\Uparrow\mathsf{Nat}, \star)}$$

Hence, using $\mathcal{A}_{\mathsf{ref}[1]}$ we are able to obtain a model of a programming language with linear references. We can in fact obtain full first-order references. Indeed, the following property shows that we have unrestricted contraction. In fact, every idempotent **MA** has unrestricted contraction. This applies to $\mathcal{A}_{\mathsf{ref}[1]}$, which is trivially idempotent.

Property 167. *Suppose that $\mathcal{A}$ is idempotent. Then the following rule is $\mathcal{A}$-sound.*

$$\frac{\mathcal{E};\Gamma, x : P, y : P \vdash_k t : (N, \star)}{\mathcal{E};\Gamma, x : P \vdash_k t[x/y] : (N, \star)}$$

Proof. Suppose that $\mathcal{E};\Gamma, x : P, y : P \vdash_k t : (N, p)$ is $\mathcal{A}$-sound. Let $\rho \Vdash \mathcal{E}$ and $\sigma, [x \mapsto (v, q)] \in \|\Gamma, x : P\|_{\mathcal{A},k,\rho}$. Then $\sigma, [x \mapsto (v, q), x \mapsto (v, q)] \in \|\Gamma, x : P, y : P\|_{\mathcal{A},k,\rho}$. Therefore, by hypothesis we have

$$(t[v/x, v/y], p + q + q)[\sigma] \in \|N\|_{\mathcal{A},k,\rho}$$

Since $p + q + q = p + q$, we obtain

$$(t[x/y][v/x], p)[\sigma] \in \|N\|_{\mathcal{A},k,\rho}$$

Hence the conclusion. □

Using these two typing rules, we can derive the usual primitives used to access references, namely set. and get. Those are defined from swap. as follows:

$$\begin{array}{rcl} \mathsf{set}(v) & = & \mathsf{swap}(v) \text{ to } x.\mathsf{ret}(\star) \\ \mathsf{get} & = & \mathsf{swap}(\underline{0}) \text{ to } x.(\mathsf{swap}(x) \text{ to } y.x) \end{array}$$

The next property shows how we can type them.

Property 168. *These two primitives are typable as follows in $\mathcal{A}_{\mathsf{ref}[1]}$:*

$$\frac{\mathcal{E};\Gamma \vdash_1 v : (\mathsf{Nat}, \star)}{\mathcal{E};\Gamma \vdash_1 \mathsf{set}(v) : (\mathbb{1}, \star)} \qquad \frac{}{\mathcal{E};\Gamma \vdash_1 \mathsf{get} : (\mathsf{Nat}, \star)}$$

We will see in Section 6.5 some generalizations of this constructions, including higher-order references.

4.5.3 Step-indexing

In this example, we define an idempotent 1-**MA** that induces a realizability model enjoying several interesting properties:

- First, we can interpret various kinds of *recursive types* in it. That is, types of the form $\mu X.P$, where X is a type variable, which are in some way equivalent to $P[\mu X.P/X]$.
- Secondly, by considering a particular monitor in this algebra, the observation made at level 0 of *termination* becomes *safety* at level 1. As a by-product we can type a program fixed-point combinator.

This monitoring algebra is the counterpart in our setting of the well-known technique of step-indexing.

Definition 169 (Step-indexing **MA**). *The **step-indexing monitoring algebra** $\mathcal{A}_{\mathsf{step}}$ is given by the following components:*

- *The forcing monoid is an additive forcing monoid defined as follows:*
  - *The carrier is* $\mathbb{N} \cup \{\infty\}$
  - *The operation* $+$ *is the* $\min$ *on* $\mathbb{N}$ *and the neutral element is* ∞.
  - *The preorder is the reverse order* $\geq$ *on* $\mathbb{N}$
- *The test function is* $\mathcal{C}_{\mathcal{A}_{\mathsf{step}}}(n) \stackrel{\mathsf{def}}{=} \{\ \underline{\mathsf{m}} \mid m \leq n\ \}$

We remind the definition of α_{step}:

$$\alpha_{\mathsf{step}} \stackrel{\mathsf{def}}{=} \lambda x.\mathsf{case}\, x\, \mathsf{of}\, x.\maltese \parallel x.\mathsf{ret}(x)$$

It gives us a $\mathcal{A}_{\mathsf{step}}$-monitor.

Proposition 170. *The pair* $(\alpha_{\mathsf{step}}, x \mapsto x+1)$ *is a* $\mathcal{A}_{\mathsf{step}}$*-monitor.*

Proof. It is clear that f is strong since $|\mathcal{A}_{\mathsf{step}}|$ is commutative. Let's just check that for any $n \in \overline{\mathbb{N}}$, we have:

$$\alpha \Vdash_0 \mathcal{C}_{\mathcal{A}_{\mathsf{step}}}(n+1) \multimap \Uparrow \mathcal{C}_{\mathcal{A}_{\mathsf{step}}}(n)$$

Let $n \in \overline{\mathbb{N}}$ and $v \in \mathcal{C}_{\mathcal{A}_{\mathsf{step}}}(n+1)$. Then $v = \underline{\mathsf{m}}$ with $m \in \mathbb{N}$ such that $m \leq n+1$. Let $E \in \mathcal{C}_{\mathcal{A}_{\mathsf{step}}}(n)^{\perp_0}$. We want to prove that

$$\langle \alpha, \mathsf{a}(\underline{\mathsf{m}}).E\rangle_0 \in \perp\!\!\!\perp$$

Let's consider the following cases:

- If $m = 0$, then
  $$\langle \alpha, \mathsf{a}(\underline{0}).E\rangle_0 \to \langle \maltese, E\rangle_0 \to \maltese \in \perp\!\!\!\perp$$
  Hence, we conclude by $\to$-saturation of $\perp\!\!\!\perp$ that $\langle \alpha, \mathsf{a}(\underline{\mathsf{m}}).E\rangle_0 \in \perp\!\!\!\perp$.
- If $m = m' + 1$, then because $m' + 1 \leq n + 1$ we have then $m' \leq n$ (this is true even if $n = \infty$) and $\underline{\mathsf{m}'} \perp E$. On the other hand,
  $$\langle \alpha, \mathsf{a}(\underline{\mathsf{m}' + 1}).E\rangle_0 \to \langle \mathsf{ret}(\underline{\mathsf{m}'}), E\rangle_0 \in \perp\!\!\!\perp$$
  Hence, we conclude by $\to$-saturation that $\langle \alpha, \mathsf{a}(\underline{\mathsf{m}' + 1}).E\rangle_0 \in \perp\!\!\!\perp$.

□

Recursive types

Let's extend the grammar of types with unrestricted recursive types as follows:

$$P \quad ::= \quad \ldots \mid \mu X.P$$

where X is a predicate variable. Then, it is possible to interpret $\mu X.P$ in $\mathcal{A}_{\mathsf{step}}$ in such a way that the following rules are $\mathcal{A}_{\mathsf{step}}$-sound:

$$\frac{\mathcal{E};\Gamma \vdash_1 v : (P[\mu X.P/X], n)}{\mathcal{E};\Gamma \vdash_1 v : (\mu X.P, n)}$$

$$\frac{\mathcal{E};\Gamma \vdash_1 v : (\mu X.P, n) \qquad \mathcal{E};\Delta, x : P[\mu X.P/X] \vdash_1 t : (N, m)}{\mathcal{E};\Gamma,\Delta \vdash_1 \mathsf{ret}(v)\ \mathsf{to}\ x.(\!|t|\!)_1^{\alpha_{\mathsf{step}}} : (N, \min(m, n))}$$

Recursive types are known to break termination. It does not contradict the fact that we can interpret them in our realizability model based on termination. Indeed, notice that we make use of the monitor α_{step}. As we have seen in Subsection 2.3.4, observing termination in presence of this monitor amounts to observe *safety*, which is not in contradiction with the presence of recursive types. This will be proved in details, in a more general context, in Chapter VI , Section 6.4.

Divergence

When we consider the translation $t \mapsto \{t\}^{\alpha_{\mathsf{step}}}$ of Subsection 2.3.4, we have seen that observing termination amounts to observe *safety*. This allows to type fixed-point combinators (without using recursive types). Indeed, consider the following term, which is a modification of the usual Y call-by-name fixed-point combinator (defined through the call-by-name translation of Section 2.2):

$$\mathsf{Y}^{\mathcal{A}_{\mathsf{step}}} = \{(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))^{\mathsf{N}}\}^{\alpha_{\mathsf{step}}}$$

It does the same thing as the Y combinator, except that it makes the counter represented by the memory cell decrease:

$$\langle (\mathsf{Y}^{\mathcal{A}_{\mathsf{step}}})\mathsf{thunk}(t), \mathsf{a}(\underline{\mathsf{k}+1}).E\rangle_1 \to^* \langle t, \mathsf{a}(\underline{\mathsf{k}}).(\mathsf{Y}^{\mathcal{A}_{\mathsf{step}}})\mathsf{thunk}(t).E\rangle_1$$

It is then possible to show directly that the following rule is $\mathcal{A}_{\mathsf{step}}$-sound for every $n \in \overline{\mathbb{N}}$:

$$\frac{}{\mathcal{E};\Gamma \vdash_1 \mathsf{Y}^{\mathcal{A}_{\mathsf{step}}} : (\Downarrow(\Downarrow N \multimap N) \multimap N, n)}$$

The main ingredient is that we can prove this by induction on the annotation $n \in |\mathcal{A}_{\mathsf{step}}|$. This stratification technique is essentially what is also known as step-indexing.

Chapter V

Iteration

Contents

In the previous chapter, we have defined the general notion of n-**MA** and proved that each of them induces a sound realizability interpretation of the forcing annotated version of λ_{LCBPV} defined in Section 3.2. If we have seen some interesting examples of 1-**MA**s, it remains to see how it is possible to *combine* **MA**s to form new ones. Indeed, when proving the correctness of a complex language, we don't want to come upÂădirectly with the right **MA**, but reuse basic semantical blocks already defined. In this chapter, we propose to study two different constructions that allows one to build new n-**MA**s out of previously defined ones. These two constructions are called respectively the **MA simple iteration** and **MA semi-direct iteration**:

- The **simple iteration** consists in picking a 1-**MA** *inside* another n-**MA** $\mathcal{A}$. As we have seen, 1-**MA**s admit a characterization of a large subset of the monitors, but it is not always the case in general n-**MA**s. We will see that each simple iteration admits such a characterization. Moreover, **MA**s built using the simple iteration satisfy a generalized version of the connection theorem of Section 4.4.
- The **semi-direct iteration** is a generalization of the simple iteration, which replaces the use of the direct product of forcing monoids by a semi-direct product. This construction will be useful, but is not as well-behaved as the simple iteration. For example, the connection theorem does not make sense for a semi-direct iteration. However we will see that it preserves monitors, under certain conditions.

Contributions

We first define in Section 5.1 the **simple iteration** and prove some of its remarkable properties, including a monitor preservation property and the identification of a subset of monitors, but also a generalized connection theorem that we prove in Section 5.2. We then define and briefly study the more involved construction called the **semi-direct iteration** in Section 5.3.

5.1 | Simple iteration

We present a particular construction on Monitoring Algebras called **simple iteration**. This operation will be the main ingredient used to build new **MA**s. This operation consists in doing the following:

- Choosing a n-Monitoring Algebra $\mathcal{A}$ (in the sense of Chapter IV).
- Picking *inside the realizability interpretation induced by* $\mathcal{A}$ a new 1-Monitoring Algebra $\mathcal{B}$.

As we will see, $\mathcal{B}$ is not really a 1-**MA** but can be seen as a 1-**MA** relatively to $\mathcal{A}$. $\mathcal{B}$ indeed lives inside the model induced by $\mathcal{A}$. The iteration leads to the definition of a new $(n+1)$-**MA** denoted by $\mathcal{A} \lhd \mathcal{B}$.

Let $\mathcal{A}$ be a n-**MA**. $\mathcal{A}$ naturally gives rise to a realizability model, as defined in Chapter IV . We now define formally the notion of $\mathcal{A}$-**MA**, *i.e.* a 1-**MA** chosen *inside* $\mathcal{A}$. We then define the simple iteration $\mathcal{A} \lhd \mathcal{B}$.

Definition 171 ($\mathcal{A}$-**MA**). *Let $\mathcal{A}$ be a n-**MA** with $n \geq 1$. A $\mathcal{A}$-**MA** is a structure $\mathcal{B}$ given by:*

- *A forcing monoid $|\mathcal{B}|$.*
- *A function $\mathcal{C}_{\mathcal{B}} : |\mathcal{B}| \to \mathcal{I}(\mathbb{V}_{\mathcal{A}})$ which is decreasing:*

$$p \preceq_{|\mathcal{B}|} q \Rightarrow \mathcal{C}_{\mathcal{B}}(q) \subseteq \mathcal{C}_{\mathcal{B}}(p)$$

Remark 172. *Clearly, the notion of $\mathcal{A}$-**MA** resembles that of 1-**MA**. But it is not a 1-**MA** since the test predicate is* relativized *to $\mathcal{A}$.*

Definition 173 (Simple iteration)**.** *If $\mathcal{A}$ is a n-**MA** and $\mathcal{B}$ is a $\mathcal{A}$-**MA**, we denote by $\mathcal{A} \lhd \mathcal{B}$ the n-**MA** such that:*

- *The carrier $|\mathcal{A} \lhd \mathcal{B}| \stackrel{\text{def}}{=} |\mathcal{A}| \times |\mathcal{B}|$ is the direct product forcing monoid.*

- *$\mathcal{C}_{\mathcal{A}\lhd\mathcal{B}}$ is the function defined by*

$$\mathcal{C}_{\mathcal{A}\lhd\mathcal{B}}(p,m) \stackrel{\text{def}}{=} \{\ (\overrightarrow{v},w) \mid \exists r \in |\mathcal{A}|, (w,r) \in \mathcal{C}_{\mathcal{B}}(m) \wedge \overrightarrow{v} \in \mathcal{C}_{\mathcal{A}}(r \bullet p)\ \}$$

*$\mathcal{A} \lhd \mathcal{B}$ is called the **simple iteration of $\mathcal{B}$ over $\mathcal{A}$**.*

Property 174. *Any simple iteration $\mathcal{A} \lhd \mathcal{B}$ with $\mathcal{A}$ a n-**MA** is a $(n+1)$-**MA**.*

Proof. We only need to check that $\mathcal{C}_{\mathcal{A}\lhd\mathcal{B}}$ is a test function. Suppose that $(p,n) \preceq (q,m)$, *i.e.* $p \preceq_{|\mathcal{A}|} q$ and $n \preceq_{|\mathcal{B}|} m$. Take $(\overrightarrow{v},w) \in \mathcal{C}_{\mathcal{A}\lhd\mathcal{B}}(q,m)$. Then we have some $r \in |\mathcal{A}|$ such that $(w,r) \in \mathcal{C}_{\mathcal{B}}(m)$ and $\overrightarrow{v} \in \mathcal{C}_{\mathcal{A}}(r \bullet q)$. But since $n \preceq_{|\mathcal{B}|} m$ we have $(w,r) \in \mathcal{C}_{\mathcal{B}}(n)$. Moreover, since $r \bullet p \preceq r \bullet q$ we have $\overrightarrow{v} \in \mathcal{C}_{\mathcal{A}}(r \bullet p)$. Hence we have $(\overrightarrow{v},w) \in \mathcal{C}_{\mathcal{A}\lhd\mathcal{B}}(p,n)$. □

Notation 175. *If $\mathcal{D} = \mathcal{A} \lhd \mathcal{B}$ we say that $\mathcal{A}$ and $\mathcal{B}$ are **factors** of $\mathcal{D}$. We extend this definition by adding that if $\mathcal{A}'$ is factor of $\mathcal{A}$ or $\mathcal{B}$, then it is also a factor of $\mathcal{D}$.*

Definition 176 (Canonical extension of a $\mathcal{A}$-connective)**.** *Suppose that $\mathcal{D} = \mathcal{A} \lhd \mathcal{B}$. If $\odot$ is a $\mathcal{A}$-connective (resp. a $\mathcal{B}$-connective) then its **canonical extension to** $\mathcal{D}$ is a $\mathcal{D}$-connective defined by:*

- *It is extended on the product forcing monoid as*

$$\odot((p,n)) \stackrel{\text{def}}{=} (\odot(p),n)$$

$$\textit{(resp. } \odot((p,n)) \stackrel{\text{def}}{=} (p,\odot(n))\textit{)}$$

- *Identical to $\odot$ on the term part.*

We use the same symbol $\odot$ to denote its canonical extension.

5.1.1 Properties of the simple iteration

Simple iteration has remarkable properties, that were already proved for 1-**MA**s. The first important property concerns the monitors. If $\mathcal{D} = \mathcal{A} \lhd \mathcal{B}$, we have mainly two ways of obtaining $\mathcal{D}$-monitors:

- By taking a $\mathcal{A}$-monitor and lifting it to $\mathcal{D}$. This is a preservation property.
- By choosing a new one using the structure coming from $\mathcal{B}$, just as it was done in 1-**MA**s.

We first prove that $\mathcal{A}$-monitors can be lifted to $\mathcal{D}$-monitors.

Property 177. *Let $\mathcal{D} = \mathcal{A} \lhd \mathcal{B}$ be a simple iteration, (α, f) a $\mathcal{A}$-monitor. Then*

$$(\lambda x.(\!|\mathsf{ret}(x)|\!)^{\alpha}_{k}, (a,b) \mapsto (f(a), b))$$

is a $\mathcal{D}$-monitor.

Proof. We pose $F : (a,b) \mapsto (f(a), b)$.

- We first show that F is strong. Let $(a,b),(a',b') \in |\mathcal{A}| \times |\mathcal{B}|$. We have $F((a,b) \bullet (a',b')) = (f(a \bullet a'), b \bullet b')$. By strength of f, we have $f(a \bullet a') \preceq_{|\mathcal{A}|} f(a) \bullet a'$. Therefore we have

$$F((a,b) \bullet (a',b')) \preceq_{|\mathcal{A}\times\mathcal{B}|} (f(a) \bullet a', b \bullet b') = F(a,b) \bullet (a',b')$$

- We pose $\alpha' = \lambda x.(\!|\mathsf{ret}(x)|\!)^{\alpha}_{k}$. Suppose that $(t, E, (p \bullet q, n \bullet m)) \in \perp\!\!\!\perp_{\mathcal{D},k+1}$. We want to show the two following assertion:

$$((\!|t|\!)^{\alpha'}_{k+1}, E, (f(p) \bullet q, n \bullet m)) \in \perp\!\!\!\perp_{\mathcal{D},k+1}$$

Let $(\overrightarrow{w}, v) \in \mathcal{C}_{\mathcal{D}}(f(p), n)$. It means that there exists $r \in |\mathcal{A}|$ such that

$$\begin{cases} (v,r) & \in & \mathcal{C}_{\mathcal{B}}(n \bullet m) \\ \overrightarrow{w} & \in & \mathcal{C}_{\mathcal{A}}(r \bullet f(p) \bullet q) \end{cases}$$

We have $r \bullet f(p) \bullet q = f(p) \bullet r \bullet q$, but since f is strong, we also have $f(p \bullet r) \preceq f(p) \bullet r$ and hence $f(p \bullet r) \bullet q \preceq r \bullet f(p) \bullet q$. That implies that $\overrightarrow{w} \in \mathcal{C}_{\mathcal{A}}(f(p \bullet r) \bullet q)$. We want to show that

$$\langle (\!|t|\!)^{;}_{k+1} \mathsf{a}(v).\overrightarrow{\mathsf{a}(w)}.E \rangle_{k+1} \in \perp\!\!\!\perp$$

By $\rightarrow$-saturation of $\perp\!\!\!\perp$, it is enough to show that

$$\langle (\!|\mathsf{ret}(v)|\!)^{\alpha}_{k}, \overrightarrow{\mathsf{a}(w)}.\mathsf{m}_{k+1}(t).E \rangle_{k} \in \perp\!\!\!\perp$$

We show the following stronger statement:

$$((\!|\mathsf{ret}(v)|\!)^{\alpha}_{k}, \mathsf{m}_{k+1}(t).E, f(p \bullet r) \bullet q) \in \perp\!\!\!\perp_{\mathcal{A},k}$$

Since (α, f) is a $\mathcal{A}$-monitor, by definition it is enough to show that

$$(\mathsf{ret}(v), \mathsf{m}_{k+1}(t).E, \underbrace{p \bullet r \bullet q}_{=r \bullet p \bullet q}) \in \perp\!\!\!\perp_{\mathcal{A},k}$$

Let $\overrightarrow{u} \in \mathcal{C}_{\mathcal{A}}(r \bullet p \bullet q)$. We have

$$\langle \mathsf{ret}(v), \overrightarrow{\mathsf{a}(u)}.\mathsf{m}_{k+1}(t).E \rangle_{k} \rightarrow \langle t, \mathsf{a}(v).\overrightarrow{\mathsf{a}(u)}.E \rangle_{k+1}$$

But the last configuration is in $\perp\!\!\!\perp$ since:

– $(t, E, (p \bullet q, n \bullet m)) \in \perp\!\!\!\perp_{\mathcal{D},k+1}$

– $(v, \overrightarrow{u}) \in \mathcal{C}_{\mathcal{D}}(p \bullet q, n \bullet m)$

Hence we conclude by $\rightarrow$-saturation of $\perp\!\!\!\perp$.

□

To simplify the proofs and notations, when α is part of such a $\mathcal{A}$-monitor and $\lambda x.(\!|\mathsf{ret}(x)|\!)^{\alpha}_{k}$ is its lifting to the level $k+1$, we will denote the term $(\!|t|\!)^{\lambda x.(\!|\mathsf{ret}(x)|\!)^{\alpha}_{k}}_{k+1}$ simply by $(\!|t|\!)^{\alpha}_{k}$. We then have the following reduction steps when we are at level $k+1$:

$$\langle (\!|t|\!)^{\alpha}_{k}, \overrightarrow{\mathsf{a}(w)}.\mathsf{a}(v).E\rangle_{k+1} \quad \rightarrow^{*} \quad \langle (\!|t|\!)^{\alpha}_{k}, \overrightarrow{\mathsf{a}(w)}.\mathsf{m}_{k+1}(\mathsf{ret}(v)).E\rangle_{k}$$

In general, when we will consider $(k+n)$-**MA**s of which $\mathcal{A}$ is a factor, we have the following reduction steps:

$$\begin{array}{rl} & \langle (\!|t|\!)^{\alpha}_{k}, \overrightarrow{\mathsf{a}(v)}.\ldots.\mathsf{a}(v_{k+1}).\ldots.\mathsf{a}(v_{k+n}).E\rangle_{n} \\ \rightarrow^{*} & \langle (\!|t|\!)^{\alpha}_{k}, \overrightarrow{\mathsf{a}(v)}.\mathsf{m}_{k+1}(\mathsf{ret}(v_{k+1})).\ldots.\mathsf{m}_{k+n}(\mathsf{ret}(v_{k+n})).E\rangle_{k} \end{array}$$

Since the methodology we follow to build models for complex programming languages is to build more and more involved **MA**s using simple iteration, Property 177 essentially tells us that all the good work is not lost. In Section 4.4 we gave a sufficient condition for (α, f) to be a monitor, in the context of 1-**MA**s. As we said, 1-**MA**s can be intuitively seen as a form of simple iteration. It is then no surprise that we can give a similar sufficient condition for $\mathcal{D}$-monitors.

Property 178. *Suppose we have a closed computations α, a strong function $f : |\mathcal{B}| \rightarrow |\mathcal{B}|$ and $q \in |\mathcal{A}|$. Then $(\alpha, (x, y) \mapsto (q \bullet x, f(y)))$ is a $\mathcal{D}$-monitor if the following holds:*

$$(\alpha, q) \Vdash_{\mathcal{A},k} \forall x \in |\mathcal{B}|.\mathcal{C}_{\mathcal{B}}(f(x)) \multimap \Uparrow \mathcal{C}_{\mathcal{B}}(x)$$

Proof.

- We first show that $F = (x, y) \mapsto (q \bullet x, f(y))$ is strong. On the other hand, we have

$$F((a, b) \bullet (a', b')) = (q \bullet (a \bullet a'), f(b \bullet b')) \preceq ((q \bullet a) \bullet a', f(b) \bullet b') = F(a, b) \bullet (a', b')$$

- By Lemma 143, it is enough to show that given $(t, E, (p, n)) \in \perp\!\!\!\perp_{\mathcal{D},k+1}$, we have $((\!|t|\!)^{\alpha}_{k+1}, E, (q \bullet p, f(n))) \in \perp\!\!\!\perp_{\mathcal{A},k+1}$. Let $(\overrightarrow{w}, v) \in \mathcal{C}_{\mathcal{D}}(q \bullet p, f(n))$. It means that there exists $r \in |\mathcal{A}|$ such that

$$\begin{cases} (v, r) & \in & \mathcal{C}_{\mathcal{B}}(f(n)) \\ \overrightarrow{w} & \in & \mathcal{C}_{\mathcal{A}}(r \bullet (q \bullet p)) \end{cases}$$

We want to show that

$$\langle (\!|t|\!)^{\alpha}_{k+1}, \mathsf{a}(v).\overrightarrow{\mathsf{a}(w)}.E\rangle_{k+1} \in \perp\!\!\!\perp$$

But it is enough by $\rightarrow$-saturation to show that

$$\langle \alpha, \overrightarrow{\mathsf{a}(w)}.\mathsf{a}(v).\mathsf{m}_{k+1}(t).E\rangle_{k} \in \perp\!\!\!\perp$$

Since we know that

$$(\alpha, q) \Vdash_{\mathcal{A},k} \forall x \in |\mathcal{B}|.\mathcal{C}_{\mathcal{B}}(f(x)) \multimap \Uparrow \mathcal{C}_{\mathcal{B}}(x)$$

We only need to show that

$$\begin{cases} (1) & \overrightarrow{w} & \in & \mathcal{C}_{\mathcal{A}}(r \bullet (q \bullet p)) \\ (2) & (v, r) & \in & \mathcal{C}_{\mathcal{B}}(f(n)) \\ (3) & (\mathsf{m}_{k+1}(t).E, p) & \in & \mathcal{C}_{\mathcal{B}}(n)^{\perp_{\mathcal{A},k}} \end{cases}$$

We already have (1) and (2) by hypothesis, so we only have to prove (3). Let $(u, q_0) \in \mathcal{C}_{\mathcal{B}}(n)$ and $\overrightarrow{v'} \in \mathcal{C}_{\mathcal{A}}(q_0 \bullet p)$. We then have

$$C_0 = \langle \mathsf{ret}(u), \overrightarrow{\mathsf{a}(v')}.\mathsf{m}_{k+1}(t).E \rangle_k \to \langle t, \mathsf{a}(u).\overrightarrow{\mathsf{a}(v')}.E \rangle_{k+1} = C_1$$

Since $(t, E, (p, n)) \in \perp\!\!\!\perp_{\mathcal{D},k+1}$ and because it is easy to check that $(\overrightarrow{v'}, u) \in \mathcal{C}_{\mathcal{D}}(p, n)$, we have $C_1 \in \perp\!\!\!\perp$ and hence $C_0 \in \perp\!\!\!\perp$, which concludes the proof.

□

5.1.2 Direct product

One important particular case of simple iteration is the **direct product of monitoring algebras**. We begin by defining what the direct product is and then show why it is an example of a simple iteration.

Definition 179 (Direct product)**.** *Let $\mathcal{A}$ be a n-**MA**, and $\mathcal{B}$ be a 1-**MA**. Then we define the **direct product of $\mathcal{A}$ and $\mathcal{B}$**, denoted by $\mathcal{A} \times \mathcal{B}$ as the $(n+1)$-**MA** such that:*

- *The carrier $|\mathcal{A} \times \mathcal{B}|$ is the direct product forcing monoid $|\mathcal{A}| \times |\mathcal{B}|$.*
- $\mathcal{C}_{\mathcal{A}\times\mathcal{B}} \stackrel{\mathsf{def}}{=} (p, k) \in |\mathcal{A}| \times |\mathcal{B}| \mapsto \{\ (\overrightarrow{v}, w) \mid \overrightarrow{v} \in \mathcal{C}_{\mathcal{A}}(p) \wedge w \in \mathcal{C}_{\mathcal{B}}(k)\ \}$

Theorem 180. *The direct product of a 1-**MA** with a n-**MA** is a simple iteration.*

Proof. Let $\mathcal{A}$ be a n-**MA**, and $\mathcal{B}$ be a 1-**MA**. Then we can define a $\mathcal{A}$-**MA** denoted $\mathcal{B}'$ as follows:

- $|\mathcal{B}'| = |\mathcal{B}|$
- $\mathcal{C}_{\mathcal{B}'}(n) = \mathcal{C}_{\mathcal{B}} \times |\mathcal{A}|$

We only need to prove that $\mathcal{A} \triangleleft \mathcal{B}' = \mathcal{A} \times \mathcal{B}$.

- The carrier is the same product forcing monoid.
- We now show that $\mathcal{C}_{\mathcal{A}\triangleleft\mathcal{B}'}(p, n) = \mathcal{C}_{\mathcal{A}\times\mathcal{B}}(p, n)$, by double inclusion.
  - If $(\overrightarrow{v}, w) \in \mathcal{C}_{\mathcal{A}\times\mathcal{B}}(p, n)$ then $w \in \mathcal{C}_{\mathcal{B}}(n)$ and $\overrightarrow{v} \in \mathcal{C}_{\mathcal{A}}(p)$. By definition, we have $(w, \mathbf{0}) \in \mathcal{C}_{\mathcal{B}'}(n)$ and $\overrightarrow{v} \in \mathcal{C}_{\mathcal{A}}(\mathbf{0} \bullet p)$, hence $(\overrightarrow{v}, w) \in \mathcal{C}_{\mathcal{A}\triangleleft\mathcal{B}'}(p, n)$.

– If $(\overrightarrow{v}, w) \in \mathcal{C}_{\mathcal{A} \lhd \mathcal{B}'}(p, n)$ then there is $r \in |\mathcal{A}|$ such that $(w, r) \in \mathcal{C}_{\mathcal{B}'}(n)$ and $\overrightarrow{v} \in \mathcal{C}_{\mathcal{B}}(r \bullet p)$. But by definition of $\mathcal{C}_{\mathcal{B}'}(n)$, we also have $(w, \mathbf{0}) \in \mathcal{C}_{\mathcal{B}'}(n)$ and because $p \preceq_{|\mathcal{B}|} r \bullet p$, we also have $\overrightarrow{v} \in \mathcal{C}_{\mathcal{B}}(p)$. Hence $(\overrightarrow{v}, w) \in \mathcal{C}_{\mathcal{A} \times \mathcal{B}}(p, n)$.

□

Property 181. *Let $\mathcal{D} = \mathcal{A} \times \mathcal{B}$ be a direct product. Suppose that (α, f) is a $\mathcal{A}$-monitor and (β, g) is a $\mathcal{B}$-monitor. Then the two following pairs are $\mathcal{D}$-monitors:*

- $(\lambda x.(\!|\mathsf{ret}(x)|\!)^{\alpha}_{k}, (a, b) \mapsto (f(a), b))$
- $(\beta, (a, b) \mapsto (a, g(b)))$

Proof. The proof is an easy adaptation of the proof of Property 177. □

5.2 | Generalized connection theorem

The connection theorem proved in Chapter IV in the case of 1-**MA**s is not adaptable as it is to general n-**MA**s. We remind the reader that the connection theorem intuitively means that given a 1-**MA** $\mathcal{A}$, each $\mathcal{A}$-model can be obtained by the choice of a forcing model *inside* a unary realizability model. The principle of the simple iteration we have defined in the previous section is similar: we define a $(n+1)$-**MA** by choosing a 1-**MA** *inside* another n-**MA**. It is then not surprising that in the context of simple iterations, a variant of the connection theorem can be proved. This section is devoted to this **generalized connection theorem**.

Preliminaries

We start with:

- A n-**MA** $\mathcal{A}$ and a $\mathcal{A}$-**MA** $\mathcal{B}$.
- A simple iteration $\mathcal{D} = \mathcal{A} \lhd \mathcal{B}$.
- A $\mathcal{D}$-model ρ.

In the proof of the connection Lemma 160 for 1-**MA**s, we considered types built using any simple connectives. Similarly to the impossibility of proving a connection theorem for general n-**MA**s, it is also impossible to consider general simple $\mathcal{D}$-connectives. We need to consider specific $\mathcal{D}$-connectives that are given by their action on $\mathcal{A}$ and $\mathcal{B}$ separately. Those connectives are called **separable**.

Definition 182 (Separable $\mathcal{A} \triangleleft \mathcal{B}$-connective)**.** *Suppose that $\mathcal{D} = \mathcal{A} \triangleleft \mathcal{B}$ is a simple iteration. Let $\odot$ be a $\mathcal{D}$-connective. We say that $(\odot, \phi_\odot)$ is $\mathcal{D}$-**separable** if*

$$\begin{cases} \phi_\odot = (p, n) \mapsto (\phi_{\odot\mathcal{A}}(p), \phi_{\odot\mathcal{B}}(n)) \\ (\odot, \phi_{\odot\mathcal{A}}) \text{ is a } \mathcal{A}\text{-connective} \\ (\odot, \phi_{\odot\mathcal{B}}) \text{ is a } \mathcal{B}\text{-connective} \end{cases}$$

In that case, to simplify, we will denote all these connectives (in $\mathcal{A}, \mathcal{B}$ and $\mathcal{D}$) by the same symbol $\odot$.

Remark 183. *In a certain sense, simple connectives in 1-**MA**s satisfy a condition similar to the separability condition: they are characterized by two functions, one on terms and one on the elements of the forcing monoid.*

In this section, we prove a connection lemma for any type built using the core grammar of λ_{LCBPV} augmented with any number of $\mathcal{D}$-separable connectives.

Connection theorem

We want to show that our $\mathcal{D}$-model ρ can be seen as a forcing model inside a certain $\mathcal{A}$-model. We begin by choosing the following forcing model $\mathcal{F}(\mathcal{B})$:

- The forcing monoid $|\mathcal{B}|$.
- The orthogonality predicate is a predicate variable C.

We now define the desired $\mathcal{A}$-model, which is function of ρ.

- The valuation $\overline{\rho}$ is defined as follows:
  - On first-order variables, $\rho(x) \overset{\text{def}}{=} \overline{\rho}(x)$.
  - The valuation of the predicate C of arity $|\mathcal{B}|$ is given by
  $$\overline{\rho}(\mathsf{C})(p) \overset{\text{def}}{=} \mathcal{C}_{\mathcal{B}}(p)$$
  - To each predicate X of arity $T_1 \times \cdots \times T_k$ corresponds a forcing predicate $X^* : T_1 \times \cdots \times T_k \times |\mathcal{B}|$. We define its valuation $\overline{\rho}(X^*)$ by:
  $$\overline{\rho}(X^*)(e_1, \ldots, e_k, n) \overset{\text{def}}{=} \{\ (v, p) \in \mathbb{V}_{\mathcal{A}} \mid (v, (p, n)) \in \rho_{\mathcal{A}}(e_1, \ldots, e_k)\ \}$$

With these definitions we can show how the realizability model induced by $\mathcal{A} \triangleleft \mathcal{B}$ and the forcing relation defined in the realizability model induced by $\mathcal{A}$ are connected. To do this, we state and prove the following Connection Lemma:

Lemma 184. *Suppose that ρ is a valuation, $p \in |\mathcal{A}|$, $m \in |\mathcal{B}|$, $v \in \mathbb{V}$, $t \in \mathbb{T}$ and $E \in \mathbb{E}$. Suppose that P and N are types built using the core grammar of λ_{LCBPV} augmented with any number of separable $\mathcal{A} \lhd \mathcal{B}$-connectives. Then the three following equivalences hold:*

$$\begin{array}{lll} (1) & (v,(p,m)) \in \|P\|_{\mathcal{A}\lhd\mathcal{B},k,\rho} & \iff (v,p) \in \|P^*(m)\|_{\mathcal{A},k,\overline{\rho}} \\ (2) & (E,(p,m)) \in [\![N]\!]_{\mathcal{A}\lhd\mathcal{B},k,\rho} & \iff (E,p) \in [\![N^\circ(m)]\!]_{\mathcal{A},k,\overline{\rho}} \\ (3) & (t,(p,m)) \in \|N\|_{\mathcal{A}\lhd\mathcal{B},k,\rho} & \iff (t,p) \in \|N^*(m)\|_{\mathcal{A},k,\overline{\rho}} \end{array}$$

Proof. We prove the three statements by mutual induction on the type.

1. We begin by proving (1), by looking at all the possible cases. We fix $k \in \mathbb{N}$, ρ a simple valuation.

   - **Predicate variable.** Given a predicate variable X of signature $S_1 \times \cdots \times S_m$ and $e_1, \dots, e_m$ respectively in S_1-Exp, $\dots$, S_m-Exp, we have:

   $$\begin{array}{rll} (v,(p,n)) \in \|X(e_1,\dots,e_m)\|_{\mathcal{D},k,\rho} & \Leftrightarrow & (v,(p,n)) \in \rho(X)([\![e_1]\!]_\rho,\dots,[\![e_m]\!]_\rho) \\ & \Leftrightarrow & (v,p) \in \overline{\rho}(X^*)([\![e_1]\!]_{\overline{\rho}},\dots,[\![e_m]\!]_{\overline{\rho}},n) \\ & \Leftrightarrow & (v,p) \in \overline{\rho}(X^*)([\![e_1]\!]_{\overline{\rho}},\dots,[\![e_m]\!]_{\overline{\rho}},n) \\ & \Leftrightarrow & (v,p) \in \|(X(e_1,\dots,e_m))^*(n)\|_{k,\overline{\rho}} \end{array}$$

   - **Primitive integers.** Let $v \in \mathbb{V}$ and $p \in \mathcal{M}$. We have the following equivalences:

   $$\begin{array}{rll} (v,(p,n)) \in \|\mathsf{Nat}\|_{\mathcal{D},k,\rho} & \Leftrightarrow & \exists k \in \mathbb{N}, v = \underline{k} \\ & \Leftrightarrow & (v,p) \in \|\mathsf{Nat}\|_{\mathcal{A},k,\overline{\rho}} \\ & \Leftrightarrow & (v,p) \in \|\mathsf{Nat}^*(n)\|_{\mathcal{A},k,\overline{\rho}} \end{array}$$

   - **Tensor unit.**

   $$\begin{array}{rll} (v,(p,n)) \in \|\mathbb{1}\|_{\mathcal{D},k,\rho} & \Leftrightarrow & v = * \\ & \Leftrightarrow & (v,p) \in \|\mathbb{1}\|_{\mathcal{A},k,\overline{\rho}} \\ & \Leftrightarrow & (v,p) \in \|\mathbb{1}^*(n)\|_{\mathcal{A},k,\overline{\rho}} \end{array}$$

   - **Positive separable $\mathcal{A}$-connective**. Suppose that $\odot$ is a positive separable $\mathcal{A}$-connective. We suppose to simplify the presentation, but without any loss of generality that its signature is such that $\sigma(+) = 1 = \sigma(-)$, hence we consider the type $\odot(P,N)$ for some positive type P and some negative type N. We suppose that by induction, (1) is true for P $(H1)$ and (2) is true for N $(H2)$. Then we have:

   $$\begin{array}{rll} & (v,(p,n)) \in [\![\odot(P,N)]\!]_{\mathcal{D},k,\rho} & \\ \Leftrightarrow & \exists (q_1,m_1),(q_2,m_2) \in |\mathcal{D}|, \phi_\odot((q_1,m_1),(q_2,m_2)) \preceq_{|\mathcal{D}|} (p,n), & \\ & \quad \exists (w,E) \in \mathbb{V}\times\mathbb{E}, v = \odot(w,E) \wedge & \\ & \quad (w,(q_1,m_1)) \in \|P\|_{\mathcal{D},k,\rho} \wedge (E,(q_2,m_2)) \in [\![N]\!]_{\mathcal{D},k,\rho} & \\ \Leftrightarrow & \exists (q_1,q_2) \in |\mathcal{A}|, \exists (m_1,m_2) \in |\mathcal{B}|, \underbrace{\phi_\odot((q_1,m_1),(q_2,m_2))}_{=(\phi_\odot(q_1,q_2),\phi_\odot(m_1,m_2))} \preceq_{|\mathcal{D}|} (p,n), & \odot \text{ is separable} \\ & \quad \exists (w,E) \in \mathbb{V}\times\mathbb{E}, v = \odot(w,E) \wedge & \\ & \quad (w,q_1) \in \|P^*(m_1)\|_{\mathcal{A},k,\overline{\rho}} \wedge (E,q_2) \in \|N^\circ(m_2)\|_{\mathcal{A},k,\overline{\rho}} & (H1),(H2) \\ \Leftrightarrow & \exists q_1,q_2 \in |\mathcal{A}|, \phi_\odot(q_1,q_2) \preceq_{|\mathcal{A}|} p, \exists m_1,m_2 \in |\mathcal{B}|, \phi_\odot(m_1,m_2) \preceq_{|\mathcal{B}|} n, & \\ & \quad \exists (w,E) \in \mathbb{V}\times\mathbb{E}, v = \odot(w,E) \wedge & \\ & \quad (w,q_1) \in \|P^*(m_1)\|_{\mathcal{A},k,\overline{\rho}} \wedge (E,q_2) \in \|N^\circ(m_2)\|_{\mathcal{A},k,\overline{\rho}} & (H1),(H2) \\ \Leftrightarrow & \exists m_1,m_2 \in |\mathcal{B}|, \phi_\odot(m_1,m_2) \preceq_{|\mathcal{B}|} n, (v,p) \in \|\odot(P^*(m_1),N^\circ(m_2))\|_{\mathcal{A},k,\overline{\rho}} & \\ \Leftrightarrow & (v,p) \in \|(\odot(P,N))^*(n)\|_{\mathcal{A},k,\overline{\rho}} & (\text{Def of } .^*) \end{array}$$

- **Tensor.** The tensor is positive separable $\mathcal{D}$-connective.

- **Positive shift.** Let $(v,p) \in \mathbb{V}_{\mathcal{A}}$ and $n \in |\mathcal{B}|$. We suppose that (3) is true for the computation type N (IH). We know that $(v,(p,n)) \in \|\Downarrow N\|_{\mathcal{D},k,\rho}$ is equivalent to

$$\exists t \in \mathbb{P}, v = \mathsf{thunk}(t) \wedge (t,(p,n)) \in (\llbracket N \rrbracket_{\mathcal{D},k,\rho})^{\perp_{\mathcal{D},k}}$$

  But because of (IH) this is equivalent to

$$\exists t \in \mathbb{P}, v = \mathsf{thunk}(t) \wedge (t,p) \in \|\forall x \in |\mathcal{B}|.\mathcal{C}_{\mathcal{B}}(n \bullet x) \multimap N^*(x)\|_{\mathcal{A},k,\overline{\rho}}$$

  This is in turn equivalent to

$$(v,p) \in \|\Downarrow(\forall x \in |\mathcal{B}|.\mathcal{C}_{\mathcal{B}}(n \bullet x) \multimap N^*(x))\|_{\mathcal{A},k,\overline{\rho}}$$

  And finally, by definition of $(\Downarrow N)^*(n)$ this last proposition is equivalent to

$$(v,p) \in \|(\Downarrow N)^*(n)\|_{\mathcal{A},k,\overline{\rho}}$$

- **Existential quantifier.** Let $v \in \mathbb{V}$ and $p \in \mathcal{M}$. We suppose that (1) is true for P (IH).

$$\begin{array}{rcll} (v,(p,n)) \in \|\exists x \in T.P\|_{\mathcal{D},k,\rho} & \Leftrightarrow & \exists r \in T, (v,(p,n)) \in \|P\|_{\mathcal{D},k,\rho[x \leftarrow r]} & \\ & \Leftrightarrow & \exists r \in T, (v,p) \in \|P^*(n)\|_{\mathcal{A},k,\overline{\rho}[x \leftarrow r]} & (IH) \\ & \Leftrightarrow & (v,p) \in \|\exists x \in T.P^*(n)\|_{\mathcal{A},k,\overline{\rho}} & \\ & \Leftrightarrow & (v,p) \in \|(\exists x \in T.P)^*(n)\|_{\mathcal{A},k,\overline{\rho}} & (\text{def of } (.)^*) \end{array}$$

- **Inequational conjunction.** Let $v \in \mathbb{V}$ and $p \in \mathcal{M}$. Now suppose that (1) is true for P (IH).

$$\begin{array}{rcl} (v,(p,n)) \in \|\{s \preceq_T r\} \wedge P\|_{\mathcal{D},k,\rho} & \Leftrightarrow & \llbracket r \rrbracket_{\rho} = \llbracket s \rrbracket_{\rho} \wedge (v,(p,n)) \in \|P\|_{\mathcal{D},k,\rho} \\ & \Leftrightarrow & \llbracket r \rrbracket_{\overline{\rho}} = \llbracket s \rrbracket_{\overline{\rho}} \wedge (v,p) \in \|P^*(n)\|_{\mathcal{A},k,\overline{\rho}} \\ & \Leftrightarrow & (v,p) \in \|\{s \preceq_T r\} \wedge P^*(n)\|_{\mathcal{A},k,\overline{\rho}} \\ & \Leftrightarrow & (v,p) \in \|(\{s \preceq_T r\} \wedge P)^*(n)\|_{\mathcal{A},k,\overline{\rho}} \end{array}$$

2. We now prove the proposition (2) by looking at all possible cases for negative types. In all cases we fix $k \in \mathbb{N}$ and ρ a simple valuation.

- **Negative separable $\mathcal{A}$-connective**. Suppose that $\odot$ is a Negative separable $\mathcal{A}$-connective. We suppose to simplify the presentation, but without any loss of generality that its signature is such that $\sigma(+) = 1 = \sigma(-)$, hence we consider the type $\odot(P,N)$ for some positive type P and some negative type N. We suppose that by induction, (1) is true for P $(H1)$

and (2) is true for N $(H2)$. Then we have:

$$
\begin{array}{lll}
& (E,(p,n)) \in [\![\odot(P,N)]\!]_{\mathcal{D},k,\rho} & \\
\Leftrightarrow & \exists (q_1,m_1),(q_2,m_2) \in |\mathcal{D}|, \phi_\odot((q_1,m_1),(q_2,m_2)) \preceq_{|\mathcal{D}|} (p,n), & \\
& \quad \exists (w,E') \in \mathbb{V} \times \mathbb{E}, E = \odot(w,E') \wedge & \\
& \quad (w,(q_1,m_1)) \in \|P\|_{\mathcal{D},k,\rho} \wedge (E',(q_2,m_2)) \in [\![N]\!]_{\mathcal{D},k,\rho} & \\
\Leftrightarrow & \exists (q_1,q_2) \in |\mathcal{A}|, \exists (m_1,m_2) \in |\mathcal{B}|, \underbrace{\phi_\odot((q_1,m_1),(q_2,m_2))}_{=(\phi_\odot(q_1,q_2),\phi_\odot(m_1,m_2))} \preceq_{|\mathcal{D}|} (p,n), & \odot \text{ is separable} \\
& & \\
& \quad \exists (w,E') \in \mathbb{V} \times \mathbb{E}, E = \odot(w,E') \wedge & \\
& \quad (w,q_1) \in \|P^*(m_1)\|_{\mathcal{A},k,\overline{\rho}} \wedge (E',q_2) \in \|N^\circ(m_2)\|_{\mathcal{A},k,\overline{\rho}} & (H1),(H2) \\
\Leftrightarrow & \exists q_1,q_2 \in |\mathcal{A}|, \phi_\odot(q_1,q_2) \preceq_{|\mathcal{A}|} p, \exists m_1,m_2 \in |\mathcal{B}|, \phi_\odot(m_1,m_2) \preceq_{|\mathcal{B}|} n, & \\
& \quad \exists (w,E') \in \mathbb{V} \times \mathbb{E}, E = \odot(w,E') \wedge & \\
& \quad (w,q_1) \in \|P^*(m_1)\|_{\mathcal{A},k,\overline{\rho}} \wedge (E',q_2) \in \|N^\circ(m_2)\|_{\mathcal{A},k,\overline{\rho}} & (H1),(H2) \\
\Leftrightarrow & \exists m_1,m_2 \in |\mathcal{B}|, \phi_\odot(m_1,m_2) \preceq_{|\mathcal{B}|} n, & \\
& (E,p) \in [\![\odot(P^*(m_1),N^\circ(m_2))]\!]_{\mathcal{A},k,\overline{\rho}} & \\
\Leftrightarrow & (E,p) \in [\![\exists m_1,m_2 \in |\mathcal{B}|, \{n \succeq_{|\mathcal{B}|} \phi_\odot(m_1,m_2)\} \mapsto \odot(P^*(m_1),N^\circ(m_2))]\!]_{\mathcal{A},k,\overline{\rho}} & \\
\Leftrightarrow & (E,p) \in [\![(\odot(P,N))^*(n)]\!]_{\mathcal{A},k,\overline{\rho}} & \text{(Def of } .^*\text{)}
\end{array}
$$

- **Linear implication.** The linear implication is a negative separable $\mathcal{D}$-connective.
- **Negative shift.** Suppose that (1) is true for P (IH). Take $(E,q) \in \mathbb{E}_\mathcal{A}$ and $m \in |\mathcal{B}|$. We prove the following equivalences:

$$
\begin{array}{lll}
& (E,(q,m)) \in [\![\Uparrow P]\!]_{\mathcal{D},k,\rho} & \\
\Leftrightarrow & \forall (w,(p,n)) \in \|P\|_{\mathcal{D},k,\rho}, \forall (\overrightarrow{u},v) \in \mathcal{C}_\mathcal{D}(p \bullet q, n \bullet m), \langle((w,v),\overrightarrow{u}),\uparrow E\rangle_k \in \perp\!\!\!\perp & \\
\Leftrightarrow & \forall n \in |\mathcal{B}|, \forall (w,p) \in \|P^*(n)\|_{\mathcal{A},k,\overline{\rho}}, \forall (v,r) \in \mathcal{C}_\mathcal{B}(n \bullet m), & \\
& \quad \forall \overrightarrow{u} \in \mathcal{C}_\mathcal{A}(\underbrace{r \bullet p \bullet q}_{=(p+r)\bullet q}), \langle((w,v),\overrightarrow{u}),\uparrow E\rangle_k \in \perp\!\!\!\perp & (IH) \\
\Leftrightarrow & \forall n \in |\mathcal{B}|, (E,q) \in [\![\Uparrow(P^*(n) \otimes \mathcal{C}_\mathcal{B}(n \bullet m))]\!]_{\mathcal{A},k,\overline{\rho}} & \\
\Leftrightarrow & (E,q) \in [\![\Uparrow(\exists x \in |\mathcal{B}|.P^*(x) \otimes \mathcal{C}_\mathcal{B}(x \bullet m))]\!]_{\mathcal{A},k,\overline{\rho}} & \\
\Leftrightarrow & (E,q) \in [\![(\Uparrow P)^\circ(m)]\!]_{\mathcal{A},k,\overline{\rho}} & \text{(Def of } (.)^*\text{)}
\end{array}
$$

- **Universal quantification.** Suppose that (2) is true for N (IH). For all $E \in \mathbb{E}$ and $p \in \mathcal{M}$ we have the following:

$$
\begin{array}{lll}
(E,(p,n)) \in [\![\forall x \in T.N]\!]_{\mathcal{D},k,\rho} & \Leftrightarrow & \exists r \in T, (E,(p,n)) \in [\![N]\!]_{\mathcal{D},k,\rho[x \leftarrow r]} \\
& \Leftrightarrow & \exists r \in T, (E,p) \in [\![N^\circ(n)]\!]_{\mathcal{A},k,\overline{\rho}[x \leftarrow r]} \qquad (IH) \\
& \Leftrightarrow & (E,p) \in [\![\forall x \in T.N^\circ(n)]\!]_{\mathcal{A},k,\overline{\rho}} \\
& \Leftrightarrow & (E,p) \in [\![(\forall x \in T.N)^\circ(n)]\!]_{\mathcal{A},k,\overline{\rho}} \qquad \text{(Def of } (.)^*\text{)}
\end{array}
$$

- **Inequational implication.** Let $E \in \mathbb{E}$ and $p \in \mathcal{M}$. Now suppose that (2) is true for N (IH).

$$
\begin{array}{lll}
(E,(p,n)) \in [\![\{r \succeq_T s\} \mapsto N]\!]_{\mathcal{D},k,\rho} & \Leftrightarrow & [\![r]\!]_\rho = [\![s]\!]_\rho \wedge (E,(p,n)) \in [\![N]\!]_{\mathcal{D},k,\rho} \\
& \Leftrightarrow & [\![r]\!]_{\overline{\rho}} = [\![s]\!]_{\overline{\rho}} \wedge (E,p) \in [\![N^\circ(n)]\!]_{\mathcal{A},k,\overline{\rho}} \\
& \Leftrightarrow & (E,p) \in [\![\{r \succeq_T s\} \mapsto N^\circ(n)]\!]_{\mathcal{A},k,\overline{\rho}} \\
& \Leftrightarrow & (E,p) \in [\![(\{r \succeq_T s\} \mapsto N)^\circ(n)]\!]_{\mathcal{A},k,\overline{\rho}}
\end{array}
$$

3. We finally prove (3) by showing that it is a direct consequence of (2). Suppose that (2) is true for N (H). Let $t \in \mathbb{P}$ and $p \in \mathcal{M}$. We have the following equivalences:

$$\begin{array}{lll} & (t,(p,n)) \in \|N\|_{\mathcal{D},k,\rho} & \\ \Leftrightarrow & \forall (E,(q,m)) \in [\![N]\!]_{\mathcal{D},k,\rho}, \forall (\overrightarrow{u},w) \in \mathcal{C}_{\mathcal{D}}(p \bullet q, n \bullet m), \langle t, \overrightarrow{\mathsf{a}(u)}.\mathsf{a}(w).E\rangle_k \in \perp\!\!\!\perp & \\ \Leftrightarrow & \forall m \in |\mathcal{B}|, \forall (E,q) \in [\![N^\circ(m)]\!]_{\mathcal{A},k,\overline{\rho}}, \forall (w,r) \in \mathcal{C}_{\mathcal{B}}(n \bullet m), \forall \overrightarrow{u} \in \mathcal{C}_{\mathcal{A}}(r \bullet p \bullet q), & \\ & \langle t, \overrightarrow{\mathsf{a}(u)}.\mathsf{a}(w).E\rangle_k \in \perp\!\!\!\perp & (H) \\ \Leftrightarrow & \forall m \in |\mathcal{B}|, (t,p) \in \|\mathcal{C}_{\mathcal{B}}(n \bullet m) \multimap N^\circ(m)\|_{\mathcal{A},k,\overline{\rho}} & \\ \Leftrightarrow & (t,p) \in \|\forall x \in |\mathcal{B}|.\mathcal{C}_{\mathcal{B}}(n \bullet x) \multimap N^\circ(x)\|_{\mathcal{A},k,\overline{\rho}} & \\ \Leftrightarrow & (t,p) \in \|N^*(n)\|_{\mathcal{A},k,\overline{\rho}} & \end{array}$$

□

As a corollary, we obtain the more graphical **Connection theorem**.

Theorem 185 (Connection theorem)**.** *Let $(t,(p,n)) \in \mathbb{P} \times |\mathcal{A} \lhd \mathcal{B}|$ and N be a negative type, and ρ a $\mathcal{A} \lhd \mathcal{B}$-model. Then the following equivalence holds:*

$$(t,(p,n)) \Vdash_{\mathcal{A}\lhd\mathcal{B},k} N[\rho] \Longleftrightarrow (t,p) \Vdash_{\mathcal{A},k} (n \ \Vdash_{\mathcal{F}(\mathcal{B})} N)[\overline{\rho}]$$

5.3 | Semi-direct iteration

We now turn our attention to the **semi-direct iteration**. It is a generalization of the simple iteration, where the semi-direct product of forcing monoids is used instead of the direct product. This construction is not as well-behaved as the simple iteration, but still has some form of monitor preservation while being more general.

Definition 186 (Semi-direct iteration)**.** *Let $\mathcal{A}$ be a n-**MA** and $\mathcal{B}$ a $\mathcal{A}$-**MA**. Suppose that $|\mathcal{B}|$ acts on $|\mathcal{A}|$ via a left action δ. Then we denote by $\mathcal{A} \ltimes_\delta \mathcal{B}$ the n-**MA** such that:*

- *The carrier $|\mathcal{A} \ltimes_\delta \mathcal{B}| \stackrel{\text{def}}{=} |\mathcal{A}| \ltimes_\delta |\mathcal{B}|$ is the semi-direct product forcing monoid.*
- *$\mathcal{C}_{\mathcal{A}\ltimes_\delta\mathcal{B}}$ is the function defined by*

$$\mathcal{C}_{\mathcal{A}\ltimes_\delta\mathcal{B}}(p,m) \stackrel{\text{def}}{=} \{\ (\overrightarrow{v},w) \mid \exists r \in |\mathcal{A}|, (w,r) \in \mathcal{C}_{\mathcal{B}}(m) \wedge \overrightarrow{v} \in \mathcal{C}_{\mathcal{A}}(r \bullet p)\ \}$$

*$\mathcal{A} \ltimes_\delta \mathcal{B}$ is called the **semi-direct iteration of $\mathcal{B}$ over $\mathcal{A}$**.*

It is easy to check that $\mathcal{A} \ltimes_\delta \mathcal{B}$ indeed defines a $(n+1)$-**MA**.

Notation 187. *We extend the meaning of **factor** to semi-direct products in the natural way.*

This construction does not have most of the good properties that the simple iteration has: the connection property for instance is not provable with the current definition of forcing transformation. However, it does preserves monitors under certain conditions.

Property 188. *Let $\mathcal{D} = \mathcal{A} \ltimes_\delta \mathcal{B}$ be a semi-direct iteration, (α, f) a $\mathcal{A}$-monitor. Suppose moreover that f weakly commutes with δ, i.e. :*

$$\forall p \in |\mathcal{A}|, f(\delta_n(p)) \preceq_{|\mathcal{A}|} \delta_n(f(p))$$

Then

$$(\lambda x.(\!|\mathsf{ret}(x)|\!)^\alpha_k, (a,b) \mapsto (f(a), b))$$

is a $\mathcal{D}$-monitor.

Proof. We need to check two things: that $(a,b) \mapsto (f(a),b)$ is strong and that the monitor condition of Definition 141 is met.

- We have to show that for any $(a,b),(a',b') \in |\mathcal{A}| \times |\mathcal{B}|$, we have

  $$(f(\delta_{b'}(a) \bullet a'), b \bullet b') \preceq (f(a),b) \bullet (a',b')$$

  Since f is strong, we have

  $$f(\delta_{b'}(a) \bullet a') \preceq_{|\mathcal{A}|} f(\delta_{b'}(a)) \bullet a'$$

  But $f(\delta_{b'}(a)) \preceq \delta_{b'}(f(a))$. Hence,

  $$(f(\delta_{b'}(a) \bullet a'), b \bullet b') \preceq \underbrace{(\delta_{b'}(f(a)) \bullet a', b \bullet b')}_{=(f(a),b)\bullet(a',b')}$$

- The proof that (α, f) satisfies the monitoring condition is exactly the same as for Property 177. Indeed, this proof makes no use of the $\bullet$ operation over the product forcing monoid, and hence is independant of the choice of operation.

□

If we want to consider an analoguous of Property 178 in the context of semi-direct iteration, there is more difficulty. The main reason is that a function of the form

$$F(x,y) = (q \bullet x, f(y))$$

is not strong in general. Indeed,

$$F((x,y) \bullet (x',y')) = (q \bullet (\delta_{y'}(x) \bullet x'), f(y \bullet y')) \preceq (q \bullet (\delta_{y'}(x) \bullet x'), f(y) \bullet y')$$

But on the other hand, we have

$$F(x,y) \bullet (x',y') = (\delta_{y'}(q \bullet x) \bullet x', f(y) \bullet y)$$

But we can't a priori reconcile the $\delta_{y'}(q \bullet x)$ with $q \bullet \delta_{y'}(x)$, except if $q = \mathbf{0}$ or $\delta_{y'}$ is the identity.

Chapter VI

Basic semantical blocks

Contents

In this chapter, we exhibit several applications of the theory of monitoring algebras. We show how the different constructions and theorems proved on monitoring algebras can be used to build modularly realizability models for complex programming languages. We first focus on a serie of basic constructions on monitoring algebras that make use of operations like the simple or semi-direct iteration:

- **Modalities -** we show how it is possible to algebraically add a modality to the programming language. Examples of modalities contain Girard's linear logic modality ! [Gir87], or Nakano's recursion modality $\triangleright$ [Nak00]. Given a **MA** $\mathcal{A}$, we define an algebraic notion of $\mathcal{A}$-**modality**, and a stronger notion of **strong** $\mathcal{A}$-**modality** that allows to encode call-by-value modalities. We then show how, under certain conditions, we can generically transform a modality into a strong modality by an algebraic transformation based on the semi-direct iteration.

- **Bounded-time monitoring -** we generalize the linear-time monitoring example described in Section 4.5 by defining a class of 1-**MA**s called **quantitative**. They all share the common particularity that the program α_{time} is a monitor, which, together with the soundness theorem implies a bounded-time termination property.

- **Stratification -** a central application of our theory is the definition of realizability models of recursive types. We explore a class of **MA**s called **stratified**. We prove that in such a **MA**, the set of semantic types ($\mathcal{I}(\mathbb{V}_{\mathcal{A}})$ and $\mathcal{I}(\mathbb{E}_{\mathcal{A}})$) can be endowed with a structure of complete ultrametric space, hence satisfying a fixed-point theorem. This method is inspired by previous works [BST09, BSS10]. We then study a particular stratified 1-**MA**: the step-indexing algebra defined in Section 4.5. We show that several recursive types can be interpreted in this algebra: the **guarded recursive types** *à la Nakano* [Nak00], which don't break termination, and the usual non-guarded recursive types which break termination. In particular we identify the difference between the two kind of recursive types as the result of the use the monitor α_{step}. We finally define a subclass of the stratified **MA**s called **step-indexed** that inherit many good properties of the step-indexing algebra.

- **Higher-order references -** we show how simple iteration can be used to obtain a realizability model of a simple language with higher-order references. We in fact explain a generic construction based on the simple iteration which consists in:

  1. Taking a monitoring algebra $\mathcal{A}$ satisfying the step-indexing condition, hence admitting the fixed-point theorem and many good properties.
  2. Picking inside $\mathcal{A}$ an algebra similar to the algebra of first-order references already presented in Section 4.5, but making use of the step-indexed structure of $\mathcal{A}$ to add a reference for any positive type inducing a fixed-point in $\mathcal{A}$.

  This fairly generic technique is then instanciated in several interesting particular cases:

  – In the most simple case, by choosing a trivial stratification of the algebra, we get back first-order references.

– We show that in the step-indexing algebra, we obtain both guarded and non-guarded higher-order references for any positive type P.

6.1 | Adding a modality

We have identified in Section 4.3 a notion of simple $\mathcal{A}$-connective that can be used to extend the interpretation as well as the connection property to new type constructors. In this section, we explore a particular case of simple $\mathcal{A}$-connective: the **modalities**. Modalities appear everywhere in logic and programming languages, and we will use them in all the concrete applications presented in Chapter VII . Our modalities are defined in a purely algebraic way. We then extend this notion of modality into the more involved **adjoint modality** that correspond to **call-by-value modalities** (that is, modalities that don't block the reduction). We finally give a purely algebraic transformation on monitoring algebras to transform any modality into an adjoint modality.

6.1.1 Preliminaries

Modalities are particular 1-ary connectives. They are of specific interest as they will constitute one of the preferred way to extend the expressivity of the language. What we informally mean by modality is any connective $\Box$ that comes with at least the following promotion rule:

$$\frac{\mathcal{E}; x_1 : P_1, \ldots, x_n : P_n \vdash_0 v : Q}{\mathcal{E}; x_1 : \Box P_1, \ldots, x_n : \Box P_n \vdash_0 v : \Box Q}$$

Some examples of such modalities are the exponential modality ! of linear logic [Gir87] or Nakano's recursion modality [Nak00], which have both the promotion rule plus some additional logical principles. These modalities often control some structural aspects of the type system: the linear logic exponential permits to introduce variable sharing, while Nakano's recursion modality can be used to tame recursive types. At the operational level, we distinguish two kinds of modalities: the **blocking modalities** and the **call-by-value modalities**.

Blocking modalities

If we turn to the linear call-by-value λ-calculus defined in Section 2.2, it can be extended with a modality using this promotion rule as described in Figure 1. We have added a modality constructor $\Box$ and the accompanying $\mathsf{let}\,\Box x \;=\; \_\,\mathsf{in}\,\_$ construction. This extension is justified by the corresponding extension of the translation $(.)^{\mathsf{V}}$ that can be proved to preserve the new reduction and typing rules. Notice that since $\Box t$ is a value, we *cannot* reduce under the modality $\Box$. This is linked to the fact that to extend the modality to negatives, we have to use $\mathsf{thunk}(.)$.

Extension of types

$$A, B \quad ::= \quad \ldots \mid \Box A$$

$$\frac{\Gamma \vdash_{\mathsf{Aff}} t : A}{\Box\Gamma \vdash_{\mathsf{Aff}} \Box t : \Box A} \qquad \frac{\Gamma, x : \Box A \vdash_{\mathsf{Aff}} u : B \qquad \Delta \vdash_{\mathsf{Aff}} t : \Box A}{\Gamma, \Delta \vdash_{\mathsf{Aff}} \mathsf{let}\, \Box x \;=\; t \,\mathsf{in}\, u : B}$$

Syntax and reduction

$$\begin{array}{rcl} A, B & ::= & \ldots \mid \Box A \\ v, w & ::= & \ldots \mid \Box t \\ t, u & ::= & \ldots \mid \Box t \mid \mathsf{let}\, \Box x \;=\; t \,\mathsf{in}\, u \\ E & ::= & \ldots \mid (\Box x.u).E \end{array}$$

$$\begin{array}{rcl} \langle \mathsf{let}\, \Box x \;=\; t \,\mathsf{in}\, u, E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle t, \mathsf{a}((\Box x.u)).E\rangle_{\mathsf{V}} \\ \langle \Box t, \mathsf{a}((\Box x.u)).E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle u[t/x], E\rangle_{\mathsf{V}} \end{array}$$

Call-by-value translation

$$\begin{array}{rcl} (\Box A)^{\mathsf{V}} & \stackrel{\mathsf{def}}{=} & \Box(\Downarrow (A)^{\mathsf{V}}) \\ (\Box t)^{\mathsf{V}} & \stackrel{\mathsf{def}}{=} & \mathsf{ret}(\mathsf{thunk}((t)^{\mathsf{V}})) \\ (\mathsf{let}\, \Box x \;=\; t \,\mathsf{in}\, u)^{\mathsf{V}} & \stackrel{\mathsf{def}}{=} & (t)^{\mathsf{V}} \,\mathsf{to}\, x.(u)^{\mathsf{V}} \\ ((\Box x.u).E)^{\mathsf{V}} & \stackrel{\mathsf{def}}{=} & \mathsf{f}(x.(u)^{\mathsf{V}}).(E)^{\mathsf{V}} \end{array}$$

Figure 1: Extension of the call-by-value λ-calculus with a modality

Call-by-value modalities

One can wonder what we need in order to be able to *reduce under the modality*. It means having the following definition for values and environment instead of the one defined in Figure 1:

$$v, w \quad ::= \quad \ldots \mid \Box v$$

We also have the following new environment:

$$E \quad ::= \quad \ldots \mid \Box.E$$

And the following new reduction rules, to reduce under the modality:

$$\begin{array}{rcl} \langle \Box t, E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle t, \Box.E\rangle_{\mathsf{V}} \\ \langle v, \Box.E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle \Box v, E\rangle_{\mathsf{V}} \end{array}$$

If one wanted to implement this in λ_{LCBPV}, he would have to define the following translation of the $\Box$ constructor:

$$\begin{array}{rcl} (\Box t)^{\mathsf{V}} & \stackrel{\mathsf{def}}{=} & (t)^{\mathsf{V}} \,\mathsf{to}\, x.\mathsf{ret}(x) \\ (\Box.E)^{\mathsf{V}} & \stackrel{\mathsf{def}}{=} & \mathsf{f}((x.\mathsf{ret}(x))).(E)^{\mathsf{V}} \end{array}$$

While this syntactic translation can be shown to preserve the reduction we have just defined, it does not preserve typing. The faulty rule being the promotion rule described in Figure 1. To reobtain the promotion rule while being able to reduce under the modality, we need a slightly stronger rule:

$$\frac{\mathcal{E}; x_1 : P_1, \dots, x_n : P_n \vdash_0 t : \Uparrow Q}{\mathcal{E}; x_1 : \Box P_1, \dots, x_n : \Box P_n \vdash_0 t : \Uparrow \Box Q}$$

We will examine under which conditions it is possible to obtain the soundness of such a rule.

6.1.2 $\mathcal{A}$-modalities

Definition 189 ($\mathcal{A}$-modality)**.** *A $\mathcal{A}$-**modality** is a 1-ary simple $\mathcal{A}$-connective $(v \mapsto v, \Box)$ where $\Box : |\mathcal{A}| \to |\mathcal{A}|$ is a $|\mathcal{A}|$-morphism.*

Since a $\mathcal{A}$-modality is entirely defined by the function $\Box$ on the forcing monoid, we will from now on use the latter to denote the former.

Property 190. *Let $\mathcal{A}$ be a **MA**. If $\Box$ is a $\mathcal{A}$-modality then the following rule is sound:*

$$\frac{\mathcal{E}; x_1 : P_1, \dots, x_n : P_n \vdash_n v : (Q, p)}{\mathcal{E}; x_1 : \Box P_1, \dots, x_n : \Box P_n \vdash_n v : (\Box Q, \Box(p))}$$

Proof. Suppose that the premise is $\mathcal{A}$-sound. Take ρ a $\mathcal{A}$-valuation, and $[x_1 \leftarrow (v_1, p_1), \dots, x_n \leftarrow (v_n, p_n)] \in \|\Box\Gamma\|_{\mathcal{A},k,\rho}$. Then we have for each $i \in [1, n]$, $\Box(q_i) \preceq p_i$ and $(v_i, q_i) \in \|P_i\|_{\mathcal{A},k,\rho}$. Hence $\sigma = [x_1 \leftarrow (v_1, q_1), \dots, x_n \leftarrow (v_n, q_n)] \in \|\Gamma\|_{\mathcal{A},k,\rho}$. By $\mathcal{A}$-soundness of the premise, we have $(v[\sigma], p[\sigma]) \in \|Q\|_{\mathcal{A},k,\rho}$. Hence, $(v[\sigma], \Box(p + \sum_i q_i)) \in \|\Box Q\|_{\mathcal{A},k,\rho}$. But since $\Box$ is a $\mathcal{A}$-modality, we have $\Box(p + \sum_i q_i) \preceq \Box(p) + \sum_i \Box(q_i) \preceq \Box(p) + \sum_i p_i$. Finally $(v[\sigma], \Box(p) + \sum_i p_i) \in \|\Box Q\|_{\mathcal{A},k,\rho}$, by $\preceq$-saturation. □

We prove a property that will be useful later.

Lemma 191. *Let $\mathcal{A}$ be a k-**MA**. Suppose that $\Box$ is a $\mathcal{A}$-modality and that $(\alpha, \Box)$ is a $\mathcal{A}$-monitor. Then*

$$\frac{\mathcal{E}; \Gamma \vdash_k t : (\Uparrow(\Box P), p)}{\mathcal{E}; \Gamma \vdash_k t \text{ to } x.(\!|\mathsf{ret}(x)|\!)^k_\alpha : (\Uparrow P, p)}$$

As we have seen in the preliminary, this form of modality is not enough if one wants to model call-by-value modalities, that is to reduce under the modality. At the operational level, it means having a new environment constructor $\Box.E$ and a reduction rule such that

$$\begin{array}{rcl} \langle \Box t, E \rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle t, \Box.E \rangle_{\mathsf{V}} \\ \langle v, \Box.E \rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle \Box v, E \rangle_{\mathsf{V}} \end{array}$$

These two reduction rules are reminiscent of a notion of operational adjoint. And indeed, to obtain such modalities, we require the existence of what we call a $\mathcal{A}$-**adjoint**.

Definition 192 (Left and right $\mathcal{A}$-adjoint)**.** *Given two functions $F, G : |\mathcal{A}| \to |\mathcal{A}|$, we say that G is a **right $\mathcal{A}$-adjoint of** F (and that F is a **left $\mathcal{A}$-adjoint of** G) iff*

$$\mathcal{C}_{\mathcal{A}}(p \bullet G(q)) = \mathcal{C}_{\mathcal{A}}(F(p) \bullet q)$$

Remark 193. *Let $\mathcal{A}$ be a **MA** and $F, G : |\mathcal{A}| \to |\mathcal{A}|$ be two functions. It is enough for F and G to be adjoints that the two following conditions are met:*

$$p \bullet G(q) \ \preceq_{|\mathcal{A}|} \ F(p) \bullet q$$

$$F(p) \bullet q \ \preceq_{|\mathcal{A}|} \ p \bullet G(q)$$

Definition 194 (Adjoint $\mathcal{A}$-modality)**.** *Let $\mathcal{A}$ be a n-**MA**. We say that $\Box : |\mathcal{A}| \to |\mathcal{A}|$ is an **adjoint $\mathcal{A}$-modality** if it is a $\mathcal{A}$-modality and has a right $\mathcal{A}$-adjoint, denoted $\overline{\Box}$.*

The notion of adjoint modality is exactly what we need to model what we have called call-by-value modality. This is all due to the following typing rule.

Property 195. *Let $\mathcal{A}$ be a **MA**. If $\Box$ is an adjoint $\mathcal{A}$-modality then the following rule is sound:*

$$\frac{\mathcal{E}; \Gamma \vdash_n t : (\Uparrow P, p)}{\mathcal{E}; \Box(\Gamma) \vdash_n t : (\Uparrow(\Box P), \Box(p))}$$

Proof. Suppose that the premise is $\mathcal{A}$-sound. Take ρ a $\mathcal{A}$-model, and $[x_1 \leftarrow (v_1, p_1), \ldots, x_n \leftarrow (v_n, p_n)] \in \|\Box(\Gamma)\|_{\mathcal{A},k,\rho}$. Then we have for each $i \in [1, n]$, $\Box(q_i) \preceq p_i$ and $(v_i, q_i) \in \|P_i\|_{\mathcal{A},k,\rho}$. HEnce $\sigma = [x_1 \leftarrow (v_1, q_1), \ldots, x_n \leftarrow (v_n, q_n)] \in \|\Gamma\|_{\mathcal{A},k,\rho}$. By $\mathcal{A}$-soundness of the premise, we have

$$(\star) \quad (t[\sigma], p[\sigma]) \in \|\Uparrow P\|_{\mathcal{A},k,\rho}$$

If we can prove that $(t[\sigma], \Box(p[\sigma])) \in \|\Uparrow \Box P\|_{\mathcal{A},k,\rho}$, then we could conclude using the fact that $\Box$ is a $\mathcal{A}$-modality and by $\preceq$-saturation. We pose $p' = p[\sigma]$ and $t' = t[\sigma]$. Let $(E, q) \in [\![\Uparrow \Box P]\!]_{\mathcal{A},k,\rho} = \|\Box P\|_{\mathcal{A},k,\rho}{}^{\perp_{\mathcal{A},k}}$. We want to prove that

$$(t', \Box(p')) \perp_{\mathcal{A},k} (E, q)$$

But since $\Box$ has an $\mathcal{A}$-adjoint $\overline{\Box}$, this is equivalent to

$$(t', p') \perp_{\mathcal{A},k} (E, \overline{\Box}(q))$$

But to prove this, by using $(\star)$, it is enough to show that

$$(E, \overline{\Box}(q)) \in [\![\Uparrow P]\!]_{\mathcal{A},k,\rho}$$

Take $(w, r) \in \|P\|_{\mathcal{A},k,\rho}$. We want to prove that $(w, r) \perp_{\mathcal{A},k} (E, \overline{\Box}(q))$, which is equivalent to

$$(w, \Box(r)) \perp_{\mathcal{A},k} (E, q)$$

But since $(E, q) \in \|\Box P\|_{\mathcal{A},k,\rho}{}^{\perp_{\mathcal{A},k}}$ and $(w, \Box(r)) \in \|\Box P\|_{\mathcal{A},k,\rho}$, we have our result. □

Remark 196. *Not all modalities have an adjoint. For example take the linear-time algebra $\mathcal{A}_{\mathsf{time}}$ defined in Section 4.5. In this algebra, we have:*

$$1 \not\leq 0$$

Now consider the null constant function $x \mapsto 0$. This map is clearly a $\mathcal{A}$-modality. We have:

$$1 \not\leq 0 = \Box(1)$$

Hence for any morphism G we have

$$1 + G(0) \not\leq \Box(1) + 0 = 0$$

Therefore, in this algebra, $\mathcal{C}_{\mathcal{A}}(1 + G(0)) \neq \mathcal{C}_{\mathcal{A}}(\Box(1) + 0)$ for any G: $\Box$ has no adjoint.

6.1.3 Preservation theorem

We have shown in Chapter V that given a n-**MA** $\mathcal{A}$, $\mathcal{A}$-connectives can be canonically extended to any simple iteration $\mathcal{A} \lhd \mathcal{B}$ in $\mathcal{A}$. The same is true for semi-direct iteration. We extend these properties to modalities, by showing that the structure of $\mathcal{A}$-modality is preserved after an iteration.

Theorem 197 (Modality preservation)**.** *Let $\mathcal{A}$ be a n-**MA** and $\Box$ be a $\mathcal{A}$-modality. Let $\mathcal{B}$ be a $\mathcal{A}$-**MA**. Then the following assertions hold:*

1. *If $\mathcal{D} = \mathcal{A} \lhd \mathcal{B}$ is a simple iteration, then the canonical extension of $\Box$ to $\mathcal{D}$ is a $\mathcal{D}$-modality.*
2. *Suppose that δ is an action of $\mathcal{B}$ over $\mathcal{A}$ such that δ weakly commutes with $\Box$, i.e. :*

$$\forall n \in |\mathcal{B}|, \forall p \in |\mathcal{A}|, \Box(\delta_n(p)) \preceq_{|\mathcal{A}|} \delta_n(\Box(p))$$

Consider the semi-direct iteration $\mathcal{D} = \mathcal{A} \ltimes^{\delta} \mathcal{B}$. Then the canonical extension of $\Box$ to $\mathcal{D}$ is a $\mathcal{D}$-modality.

Proof. Since the simple iteration is a special case of semi-direct iteration, with δ being the identity (and hence always commuting with $\Box$), we only prove the second statement. It is immediate that the canonical extension of $\Box$ is order preserving. We just have to show it is also a $\mathcal{D}$-modality. Indeed, if

$(a,b),(a',b') \in \mathcal{A} \lhd \mathcal{B}$, we have

$$\begin{array}{rcl} \Box((a,b)+(a',b')) & = & (\Box(\delta_{b'}(a)+\delta_b(a')), b+b') \\ & \leq & (\Box(\delta_{b'}(a))+\Box(\delta_b(a')), b+b') \\ & \leq & (\delta_{b'}(\Box(a))+\delta_b(\Box(a')), b+b') \\ & = & \Box(a,b)+\Box(a',b') \end{array}$$

□

In general, **adjoint** modalities are preserved by direct product without any condition and by semi-direct product under some reasonable assumptions.

Theorem 198 (adjoint modality preservation)**.** *Let $\mathcal{A}$ be a n-**MA** and $\Box$ be an adjoint $\mathcal{A}$-modality. Let $\mathcal{B}$ be a $\mathcal{A}$-**MA**. Then the following assertions hold:*

1. *If $\mathcal{D} = \mathcal{A} \times \mathcal{B}$ is a direct product, then the canonical extension of $\Box$ to $\mathcal{D}$ is an adjoint $\mathcal{D}$-modality,*
2. *Suppose that δ is an action of $\mathcal{B}$ over $\mathcal{A}$ such that δ commutes with $\Box$ and $\overline{\Box}$:*

$$\forall n \in |\mathcal{B}|, \forall p \in |\mathcal{A}|, \Box(\delta_n(p)) = \delta_n(\Box(p))$$

$$\forall n \in |\mathcal{B}|, \forall p \in |\mathcal{A}|, \overline{\Box}(\delta_n(p)) = \delta_n(\overline{\Box}(p))$$

Consider the semi-direct product $\mathcal{D} = \mathcal{A} \ltimes^{\delta} \mathcal{B}$. Then the canonical extension of $\Box$ to $\mathcal{D}$ is an adjoint $\mathcal{D}$-modality.

Proof. Second, suppose $\overline{\Box}$ is an $\mathcal{A}$-adjoint of $\Box$. Then we show that the canonical extension of $\overline{\Box}$ is an $\mathcal{D}$-adjoint of the canonical extension of $\Box$. We have

$$\mathcal{C}_{\mathcal{D}}((a,b) \bullet \overline{\Box}(a',b')) = \mathcal{C}_{\mathcal{D}}(\delta_{b'}(a) \bullet \delta_b(\overline{\Box}(a')), b \bullet b')$$

Take $(\overrightarrow{v}, w) \in \mathcal{C}_{\mathcal{D}}(\delta_{b'}(a) \bullet \delta_b(\overline{\Box}(a')), b \bullet b')$. Then we have

$$\begin{cases} w & \in & \mathcal{C}_{\mathcal{B}}(b \bullet b') \\ \overrightarrow{v} & \in & \mathcal{C}_{\mathcal{A}}(\delta_{b'}(a) \bullet \delta_b(\overline{\Box}(a'))) \end{cases}$$

But since $\overline{\Box}(\delta_b(a')) = \delta_b(\overline{\Box}(a'))$, we have

$$\begin{cases} w & \in & \mathcal{C}_{\mathcal{B}}(b \bullet b') \\ \overrightarrow{v} & \in & \mathcal{C}_{\mathcal{A}}(\delta_{b'}(a) \bullet \overline{\Box}(\delta_b(a'))) \end{cases}$$

Finally, since $\overline{\Box}$ and $\Box$ are adjoint, we have

$$\overrightarrow{v} \in \mathcal{C}_{\mathcal{A}}(\Box(\delta_{b'}(a)) \bullet \delta_b(a'))$$

Again, by commutativity,

$$\overrightarrow{v} \in \mathcal{C}_{\mathcal{A}}(\delta_{b'}(\Box(a)) \bullet \delta_b(a'))$$

Therefore, $(\overrightarrow{v}, w) \in \mathcal{C}_{\mathcal{D}}(\Box(a,b) \bullet (a',b'))$. □

If adjoint modalities are not preserved by simple or centered semi-direct iteration, we can give some additional conditions that ensure that they are.

Definition 199 (Monoidal modality). *A $\mathcal{A}$-modality $\Box$ is* monoidal *if:*

$$\forall r, p \in |\mathcal{A}|, \Box(r) + \Box(p) \preceq \Box(r + p)$$

Theorem 200 (Adjoint modality preservation by iteration). *Let $\mathcal{A}$ be a n-**MA**. Suppose that:*

- *$\Box_1$ is an adjoint $\mathcal{A}$-modality, that satisfies the monoidality condition.*
- *$\Box_2$ is an adjoint $\mathcal{B}$-modality such that*

$$\forall p \in |\mathcal{B}|, \mathcal{C}_{\mathcal{B}}(\Box_2(n)) = \Box_1(\mathcal{C}_{\mathcal{B}}(n))$$

  and its adjoint is the identity.

If $\mathcal{D} = \mathcal{A} \lhd \mathcal{B}$ then $(x, y) \mapsto (\Box_1(p), \Box_2(n))$ is an adjoint $\mathcal{D}$-modality.

Proof. We prove that it has an adjoint

$$\overline{\Box}(a, b) = (\overline{\Box}_1(a), b)$$

We have

$$\mathcal{C}_{\mathcal{D}}(\overline{\Box}(a, b) \bullet (a', b')) = \mathcal{C}_{\mathcal{D}}(\overline{\Box}_1(a) \bullet a', \overline{\Box}_2(b) \bullet b')$$

Take $(\overrightarrow{v}, w) \in \mathcal{C}_{\mathcal{D}}(\overline{\Box}_1(a) \bullet a', \overline{\Box}_2(b) \bullet b')$. Then we have

$$\begin{cases} (w, r) & \in & \mathcal{C}_{\mathcal{B}}(\overline{\Box}_2(b) \bullet b') \\ \overrightarrow{v} & \in & \mathcal{C}_{\mathcal{A}}(r \bullet (\overline{\Box}_1(a) \bullet a')) \end{cases}$$

But $(w, r) \in \mathcal{C}_{\mathcal{B}}(\overline{\Box}_2(b \bullet b')) = \overline{\Box}_1(\mathcal{C}_{\mathcal{B}}(b \bullet b'))$ Hence $\overline{\Box}_1(r') \preceq_{|\mathcal{A}|} r$ and $(w, r') \in \mathcal{C}_{\mathcal{B}}(b \bullet b')$. We then have

$$\begin{cases} (w, r') & \in & \mathcal{C}_{\mathcal{B}}(b \bullet b') \\ \overrightarrow{v} & \in & \mathcal{C}_{\mathcal{A}}(\overline{\Box}_1(r' + a) \bullet a') \end{cases}$$

Finally, since $\overline{\Box}_1$ and $\Box_1$ are adjoint, we have

$$\overrightarrow{v} \in \mathcal{C}_{\mathcal{A}}(r' \bullet a \bullet \overline{\Box}_1(a'))$$

Hence

$$(\overrightarrow{v}, v) \in \mathcal{C}_{\mathcal{D}}(a \bullet \overline{\Box}_1(a'), b \bullet b')$$

Therefore

$$(\overrightarrow{v}, v) \in \mathcal{C}_{\mathcal{D}}((a, b) \bullet \overline{\Box}(a', b'))$$

□

6.1.4 Example of modality: the linear logic exponential

As an example, we give a possible axiomatization of the linear logic exponential modality, in the form of a set of inequalities in the forcing monoid. By considering different subsets of those inequalities, we obtain substructural modalities. Let $\mathcal{M}$ be a forcing monoid and $! : \mathcal{M} \to \mathcal{M}$. We enumerate three conditions on !, which are all parametrized by an element $c \in \mathcal{M}$:

$$\begin{array}{rrcl} (c\text{-Contraction}) & \forall p \in \mathcal{M}, !p + !p & \preceq & c + !p \\ (c\text{-Digging}) & \forall p \in \mathcal{M}, !!p & \preceq & c + !p \\ (c\text{-Dereliction}) & \forall p \in \mathcal{M}, p & \preceq & c + !p \end{array}$$

When $c = \mathbf{0}$, we just denote these conditions by Contraction, Digging and Dereliction. Each of these conditions implies the soundness of a certain rule.

Property 201. *Suppose that $\mathcal{A}$ is a n-**MA** and $! : |\mathcal{A}| \to |\mathcal{A}|$ is a $\mathcal{A}$-modality.*

- *If $!$ satisfies the c-Contraction condition, then the following typing rule is $\mathcal{A}$-sound:*

$$\frac{\mathcal{E}; \Gamma, x : !P, y : !P \vdash_n a : (A, p)}{\mathcal{E}; \Gamma, x : !P \vdash_n a[x/y] : (A, p + c)}$$

- *If $!$ satisfies the c-Digging condition, then the following typing rule is $\mathcal{A}$-sound:*

$$\frac{\mathcal{E}; !\Gamma \vdash_n v : (P, p)}{\mathcal{E}; !\Gamma \vdash_n v : (!P, !p + c)}$$

- *If $!$ satisfies the c-Dereliction condition, then the following typing rule is $\mathcal{A}$-sound:*

$$\frac{\mathcal{E}; \Gamma, x : P \vdash_n a : (A, p)}{\mathcal{E}; \Gamma, x : !P \vdash_n a[x/y] : (A, p + c)}$$

These three rules constitute the rules of the linear logic exponential !. Some restrictions of this notion also have a specific interest. For instance, if one only consider the c-Contraction condition (and abandon the c-Digging and c-Dereliction conditions), we obtain the exponential of the elementary linear logic [DJ03]: a substructural logic that ensures the elementary time execution of the typable programs.

Definition 202 (Exponential modalities)**.** *Suppose that $\mathcal{A}$ is a n-**MA** and $! : |\mathcal{A}| \to |\mathcal{A}|$ is a $\mathcal{A}$-modality.*

- *If there are $c_1, c_2, c_3 \in |\mathcal{A}|$ such that $!$ satisfies the c_1-Contraction condition, the c_2-Digging and the c_3-Dereliction condition, then we say that $(!, c_1, c_2, c_3)$ is an **exponential**.*

- *If there is $c \in |\mathcal{A}|$ such that $!$ satisfies the c-*Contraction *condition, then $(!, c)$ is an* ***elementary exponential****.*

Example 203. *Here are some examples of such modalities:*

- *Consider a* ***MA*** *whose carrier is the additive forcing monoid $(\overline{\mathbb{N}}, +, 0, \leq)$. We define $!$ as follows:*
$$!p \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } p = 0 \\ \infty & \text{if } p > 0 \end{cases}$$
Then $(!, 0, 0, 0)$ is an exponential modality.
- *Consider a* ***MA*** *whose carrier is the additive* ***elementary monoid*** *$\mathcal{M}_{\text{elem}}$ (which is adapted from the elementary resource monoid of [DLH05]) given by:*
  - *The set of triples $(n, m, f) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}^{\mathbb{N}}$.*
  - $(n, m, f) + (l, k, g) \stackrel{\text{def}}{=} (n + l, m + k, \max(f, g))$
  - $(n, m, f) \leq (l, k, g)$ *iff*
    - $n \leq l$
    - $2^n m \leq 2^l k$
    - $f(x) \leq g(x)$ *for any* $x \in \mathbb{N}$

  We pose:
$$!(n, m, f) \stackrel{\text{def}}{=} (1, n + m, x \mapsto x f(x 2^x))$$
Then $(!, (2, 0, x \mapsto 0))$ is an elementary modality.

Property 204. *If $!_1$ is a $\mathcal{M}_1$-modality and $!_2$ is a $\mathcal{M}_2$-modality, then $!(p_1, p_2) \stackrel{\text{def}}{=} (!p_1, !p_2)$ is a $(\mathcal{M}_1 \times \mathcal{M}_2)$-modality.*

Proof. Straightforward. □

Lemma 205. *If $\mathcal{M}$ is idempotent then the identity is an exponential $\mathcal{M}$-modality.*

Proof. Straightforward. □

6.1.5 Adding adjoint modalities

We have shown that in general a $\mathcal{A}$-modality is not an adjoint modality since it does not always admit an adjoint. However, under certain conditions it is possible to *extend* the n-**MA** in order

to build an adjoint. We now exhibit a construction based on the semi-direct product which does the following:

- Start with a monitoring algebra $\mathcal{A}$, and a $\mathcal{A}$-modality $\Box$ that satisfies additional conditions.
- Define a new monitoring algebra $\mathcal{A}^{\{\Box\}}$ in which the canonical extension of $\Box$ to $\mathcal{A}^{\{\Box\}}$ has an adjoint.

In what follows, we suppose that $\Box$ is a $\mathcal{A}$-modality such that:

- $\Box(p+q) = \Box(p) + \Box(q)$
- $p \preceq q$ implies $\Box(p) \preceq \Box(q)$
- $p \preceq \Box(p)$

Example 206. *The exponential modality on $\overline{\mathbb{N}}$ of Example 203 satisfies these conditions.*

Definition 207. *We define the 1-**MA** $\mathbb{N}_{\mathsf{adj}}$ defined by the following components:*

- *The carrier is the forcing monoid $(\mathbb{N}, \max, 0, \bullet, \leq)$ where:*
  - *$n \bullet m \stackrel{\mathsf{def}}{=} m$ is the right identity.*
  - *$\leq$ is the usual order on $\mathbb{N}$.*
- *$\mathcal{C}_{\mathbb{N}_{\mathsf{adj}}} \stackrel{\mathsf{def}}{=} n \mapsto \{\star\}$ is the constant function that maps any natural number to the singleton containing the unit value.*

We now define an action of $\mathbb{N}_{\mathsf{adj}}$ over $\mathcal{A}$, which is basically the n-th iteration of the $\mathcal{A}$-modality $\Box$ on an element of $|\mathcal{A}|$.

Property 208 ($\Box$-context action)**.** *The function $\star : \mathbb{N} \times |\mathcal{A}| \to |\mathcal{A}|$ defined as $n \star p \stackrel{\mathsf{def}}{=} \Box^n(p)$. $\star$ is an action of $\mathbb{N}_{\mathsf{adj}}$ on $|\mathcal{A}|$.*

Proof.

1. Because $\Box$ is additive, we can prove by induction on $n \in \mathbb{N}$ that $n \star (p+q) \preceq n \star p + n \star q$.
2. Since $\Box$ is increasing, if $p \preceq q$, we have $n \star p \preceq n \star q$ by induction on n.
3. Finally, since $p \preceq \Box(p)$, we have $n \leq m$ implies $n \star p \preceq m \star q$ by induction.

□

Definition 209 (□-context extension). *We define the* $(n+1)$*-**MA*** $\mathcal{A}^{\{\Box\}} \stackrel{\text{def}}{=} \mathcal{A} \ltimes_{\Box} \mathbb{N}_{\text{adj}}$ *as the semi-direct product of* $\mathcal{A}$ *and* $\mathbb{N}_{\text{adj}}$*. We call this construction over* $\mathcal{A}$ *the* □***-context extension of*** $\mathcal{A}$*.*

We now want to check that in this new $(n+1)$-**MA**:

1. We can extend the $\mathcal{A}$ into a $\mathcal{A}^{\Box}$-modality.
2. In addition this new modality has an adjoint.

Remember that the canonical extension of □ to $\mathcal{A}^{\{\Box\}}$ is defined as follows:

$$\Box(p,n) \stackrel{\text{def}}{=} (\Box(p),n)$$

Property 210. □ *is an adjoint* $\mathcal{A}^{\{\Box\}}$*-modality.*

Proof.

1. It is immediate that □ is a $\mathcal{A}^{\{\Box\}}$-modality since □ commutes with the action $\star$ and by Theorem 197.
2. We now show that □ has an $\mathcal{A}^{\{\Box\}}$-adjoint. We pose $\overline{\Box}(p,n) = (p,n+1)$. Then we have

$$\begin{array}{rcl} \Box(p,n) \bullet (q,m) & = & (\Box(p),n) \bullet (q,m) \\ & = & (\Box^{m+1}(p) \bullet \Box^{n}(q), m) \end{array}$$

But $\mathcal{C}_{\mathcal{A}^{\Box}}(\Box^{m+1}(p) \bullet \Box^{n}(q), m) = \mathcal{C}_{\mathcal{A}^{\Box}}(\Box^{m+1}(p) \bullet \Box^{n}(q), m+1)$. Moreover, we have the following equalities:

$$\begin{array}{rcl} (\Box^{m+1}(p) \bullet \Box^{n}(q), m+1) & = & (p,n) \bullet (q,m+1) \\ & = & (p,n) \bullet \overline{\Box}(q,m) \end{array}$$

Therefore, by combining these pieces of reasoning, we obtain that

$$\mathcal{C}_{\mathcal{A}^{\Box}}(\Box(p,n) \bullet (q,m)) = \mathcal{C}_{\mathcal{A}^{\Box}}((p,n) \bullet \overline{\Box}(q,m))$$

□

We then dispose of a technique that permits to build a new adjoint modality out of *any* modality.

6.2 | Bounded-time monitoring

This section is devoted to the study of a particularly interesting class of 1-monitoring algebras, namely the **quantitative monitoring algebras**. This generalizes the linear-time algebra example shown in Subsection 4.5.1. We first define the notion of **quantitative forcing monoid**.

Definition 211 (Quantitative forcing monoid). *A **quantitative forcing monoid** is a forcing monoid $\mathcal{M}$ with a function $\|.\| : \mathcal{M} \to \overline{\mathbb{N}}$ and an element $\mathbf{1} \in \mathcal{M}$ such that:*

1. *$\|.\|$ is sup-additive:*
$$\|p\| + \|q\| \leq \|p \bullet q\|$$
2. *$\|.\|$ is increasing:*
$$p \preceq q \Rightarrow \|p\| \leq \|q\|$$
3. *$1 \leq \|\mathbf{1}\|$ and $\|\mathbf{1}\| \in \mathbb{N}$*
4. *$p \mapsto \mathbf{1} \bullet p$ is strong.*

Remark 212. *If $\mathcal{M}$ is additive, the condition 4 is unnecessary since $p \mapsto p + \mathbf{1}$ is always strong in that context by Property 80.*

Remark 213. *The notion of* resource monoid *introduced in [DLH05, DLH11] is a particular case of quantitative monoid. A resource monoid comes with an anti-distance $\mathcal{D}(p, q)$ on the monoid $\mathcal{M}$. We can then pose $\|p\| = \mathcal{D}(p, \mathbf{0})$ to retrieve a quantitative monoid.*

We introduce a new value $\underline{\infty}$ in the language, with the following reduction rule at level 0 (and hence at higher levels):

$$\langle \mathsf{case}\ \underline{\infty}\ \mathsf{of}\ x.t \parallel x.u, E\rangle_0 \to \langle u[\underline{\infty}/x], E\rangle_0$$

Hence, when evaluating $\underline{\infty}$, the case construction will always choose the second branch without decreasing the argument.

Definition 214 (Quantitative 1-**MA**). *A 1-**MA** $\mathcal{A}$ is said to be **quantitative** iff:*

- *$|\mathcal{A}|$ is a quantitative forcing monoid.*
- *$\mathcal{C}_\mathcal{A}$ is equal to:*
$$\mathcal{C}_\mathcal{A}(p) \stackrel{\mathsf{def}}{=} \{\ \underline{n} \mid n \in \overline{\mathbb{N}} \wedge n \geq \|p\|\ \}$$

Proof. This defines a 1-**MA**, because $\|.\|$ is compatible with $\preceq$. □

For instance if $\|p\| = \infty$, then $\mathcal{C}_\mathcal{A}(\infty) = \{\underline{\infty}\}$.

Example 215.

- *A simple example of quantitative forcing monoid is the set of integers $\mathbb{N}$ with the usual addition and order, together with the function $\|n\| \stackrel{\mathsf{def}}{=} n$ and $\mathbf{1} \stackrel{\mathsf{def}}{=} 1$. In this quantita-*

tive forcing monoid, no element p is such that $\|p\| = \infty$.

- *A variation of the previous example is $\overline{\mathbb{N}}$ with $\|n\| \stackrel{\text{def}}{=} n$ and $\mathbf{1} \stackrel{\text{def}}{=} 1$. Obviously, $\|\infty\| = \infty$.*
- *A more complex example is given by the additive elementary monoid $\mathcal{M}_{\mathsf{elem}}$ of Example 203. It can be endowed with the following structure:*

$$\begin{aligned} \|(n,m,f)\| &\stackrel{\text{def}}{=} n.f(n+2^n m) \\ \mathbf{1} &\stackrel{\text{def}}{=} (1,0,x \mapsto x) \end{aligned}$$

The monitor α_{time} already mentioned in Subsection 2.3.2 is a $\mathcal{A}$-monitor in every quantitative **MA**.

Property 216. *If $\mathcal{A}$ is quantitative then the following pair $(\alpha_{\mathsf{time}}, f)$ is a $\mathcal{A}$-monitor:*

- $\alpha_{\mathsf{time}} \stackrel{\text{def}}{=} \lambda x.\mathsf{case}\, x\, \mathsf{of}\, x.\Omega \parallel x.\mathsf{ret}(x)$
- $f : p \mapsto \mathbf{1} \bullet p$

Proof.

- f is strong by condition 4 of the definition of quantitative forcing monoid.
- By Property 157, it is enough to show that

$$\alpha_{\mathsf{time}} \Vdash_0 \forall x \in |\mathcal{A}|.\mathcal{C}_{\mathcal{A}}(\mathbf{1} \bullet x) \multimap \mathcal{C}_{\mathcal{A}}(x)$$

Let $E \in \mathcal{C}_{\mathcal{A}}(p)^{\perp_0}$ Take $\underline{k} \in \mathcal{C}_{\mathcal{A}}(\mathbf{1} \bullet p)$, that is $k \geq \|\mathbf{1} \bullet p\| \geq \|p\| + 1$. Since $k \geq 1$, we have

$$C_0 = \langle \alpha_{\mathsf{time}}, \mathsf{a}(\underline{k}).E\rangle_0 \to \underbrace{\langle \underline{k-1}, E\rangle_0}_{\text{noted } C_1}$$

(with $\infty - 1 = \infty$). Since $k \geq \|p\| + 1$, we have $k - 1 \geq \|p\|$ and therefore $\underline{k-1} \in \mathcal{C}_{\mathcal{A}}(p)$. Hence, the configuration $C_1 \in \perp\!\!\!\perp$, so by $\to$-saturation of $\perp\!\!\!\perp$, we have $C_0 \in \perp\!\!\!\perp$, and then our result.

□

This monitor is the same as the one defined in Subsection 2.3.2. When using it, the memory cell acts as a countdown, which makes the computation diverges if it reaches $\underline{0}$, hence allowing to observe bounded-time termination. However, putting $\underline{\infty}$ in the memory cell plays the role of *giving up* the bounded-time termination analysis, since the memory cell cannot reach $\underline{0}$.

Lemma 217. *Suppose that $\mathcal{A}$ is quantitative and ρ a $\mathcal{A}$-model that satisfies the condition of Lemma 132, i.e. for every predicate variable of arity S:*

$$\forall s \in S, \exists v \in \mathbb{V}, (v, \mathbf{0}) \in \rho(X)(s)$$

If $(t,p) \in \|N\|_{\mathcal{A},1,\rho}$ *then for any* $n \geq \|p\|$*, we have*

$$\langle t, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{nil}\rangle_1 \text{ terminates}$$

Proof. By Lemma 132, we know that there exists E such that $(E,\mathbf{0}) \in [\![N]\!]_{\mathcal{A},1,\rho}$. Hence we know that if $n \geq \|p\|$,

$$\langle t, \mathsf{a}(\underline{\mathsf{n}}).E\rangle_1 \text{ terminates}$$

But if this configuration terminates, it is easy to see that so does

$$\langle t, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{nil}\rangle_1$$

□

Corollary 218. *Suppose that* $\vdash_0 t : N$*. Then*

$$\langle t, \mathsf{nil}\rangle_0 \text{ terminates on a value in less than } |t|_\lambda \ \lambda\text{-steps}$$

Proof. Consider the quantitative **MA** based on $(\mathbb{N}, +, 0, \leq, n \mapsto n)$. Let $k = |t|_\lambda$ the number of λ constructors in t. Notice that $\|k\| = k \neq \infty$. To show that $\langle t, \mathsf{nil}\rangle_0$ terminates on a value in less than k λ-steps, it is enough to show that

$$\langle \{t\}^{\alpha_{\mathsf{time}}}, \mathsf{a}(\underline{\mathsf{k}}).\mathsf{nil}\rangle_1 \in \perp\!\!\!\perp$$

Let ρ be a $\mathcal{A}$-model that satisfies the condition of Lemma 132 (such a model always exists). By Theorem 140, we have

$$(\{t\}^{\alpha_{\mathsf{time}}}, k) \Vdash_{\mathcal{A},1} N[\rho]$$

We conclude by Lemma 217. □

If this corollary implies a linear-time termination theorem for a specific core linear language, Lemma 217 can be used to obtain bounded-time termination theorems for more expressive languages. We will illustrate this in Chapter VII .

6.3 | Stratification

Many programmig features rely on some kind of *circularity*: recursive types, higher-order references or recursive definitions of programs. In all those examples, the treatment of the circularity can be reduced to the treatment of *recursive types*. Indeed, higher-order references can be justified through a proper program transformation typed using recursive types, and fixed-point combinators can be typed using recursive types. In terms of model, it means the ability to prove the existence of fixed-points for certain maps. We consider a class of monitoring algebras called **stratified** and show that a general fixed-point theorem can be formulated and proved. This theorem is obtained as a corollary of Banach fixed-point theorem.

6.3.1 Banach Fixed-point theorem

We begin with a well-known result: the **Banach fixed-point theorem** for complete metric spaces. We remind to the reader some basic definitions and facts about metric spaces.

Definition 219 ((Pseudo-)metric and ultrametric spaces)**.** *Given a set X, we say that a function $d : X \times X \to \mathbb{R}$ is a **pseudo-metric** iff for all $x, y, z \in X$ we have:*

1. $d(x, x) = 0$
2. $d(x, y) \geq 0$
3. $d(x, y) = d(y, x)$
4. $d(x, z) \leq d(x, y) + d(y, z)$

*We say that d is a **metric** if it satisfy the additional following principle:*

$$d(x, y) = 0 \text{ implies that } x = y$$

*We moreover say that d is an **ultrametric** if it satisfies the following generalization of 4:*

$$d(x, z) \leq \max(d(x, y), d(y, z))$$

- *A **pseudo-metric space** is a pair (X, d) such that X is a set of points and d a pseudo-metric on X.*
- *An **metric space** is a pair (X, d) such that X is a set of points and d an metric on X.*
- *An **ultrametric space** is a pair (X, d) such that X is a set of points and d an ultrametric on X.*

Proposition 220. *Let (X, d) be a pseudo-metric space. If we pose the following equivalence relation:*

$$x \approx y \Leftrightarrow d(x, y) = 0$$

then the quotient space $(X/\approx, d)$ is a metric space.

Definition 221 (Product metric space)**.** *If (X, d_X) and (Y, d_Y) are two metric spaces, we define the product metric space $(X \times Y, d_X \times d_Y)$ where*

$$d_{X\times Y}((x, y), (x', y')) \stackrel{\text{def}}{=} \max(d_X(x, x'), d_Y(y, y'))$$

Definition 222 (Cauchy sequence). *Let (X, d) be a metric space. A sequence $(x_n)_{n\in\mathbb{N}} \in X^{\mathbb{N}}$ is a **Cauchy sequence** iff*

$$\forall \epsilon > 0, \exists N \in \mathbb{N}, \forall m, n \geq N, d(x_n, x_m) \leq \epsilon$$

Definition 223 (Completeness). *A metric space (X, d) is **complete** iff any Cauchy sequence $(x_n)_{n\in\mathbb{N}}$ converges to a limit x:*

$$\forall \epsilon > 0, \exists N \in \mathbb{N}, \forall n \geq N, d(x, x_n) \leq \epsilon$$

Property 224. *If (X, d_X) and (Y, d_Y) are complete, then so is $(X \times Y, d_{X\times Y})$.*

Definition 225 (Contractive and non-expansive maps). *Let (X, d_X) and (Y, d_Y) be two complete metric spaces. Let $T : X \to Y$ be a map. We say that*

- *T is **contractive** iff there is $q \in [0, 1[$ such that $d_Y(T(x), T(y)) \leq q.d_X(x, y)$.*
- *T is **non-expansive** iff for all $x, y \in X$, we have $d_Y(T(x), T(y) \leq d_X(x, y)$*

Property 226. *If $T : Y \to Z$ is a non-expansive map and $C : X \to Y$ is a contractive map then $T \circ C$ is contractive. Similarly, if $C : Z \to X$ is contractive then $C \circ T$ is contractive.*

We can now state the Banach fixed-point theorem for complete metric spaces. It says that in all complete metric spaces, contractive maps admit unique fixed-points. It applies in particular to complete ultrametric spaces as well.

Theorem 227 (Banach fixed-point theorem). *Let (X, d) be a non-empty complete metric space. Any contractive map $T : X \to X$ admits a unique fixed point x^* (i.e. , $T(x^*) = x^*$).*

6.3.2 Stratified monitoring algebras

We now define the class of the **stratified monitoring algebras**. Given such a **MA** $\mathcal{A}$, we show that $\mathcal{I}(\mathbb{V}_{\mathcal{A}})$ and $\mathcal{I}(\mathbb{E}_{\mathcal{A}})$ are completely pseudo-metrizable (we in fact even show that we can endow them with a structure of complete pseudo-*ultrametric* space). This allows us to use the Banach fixed-point theorem on these spaces, which will be used to obtain models of various kinds of recursive types.

Notation 228. *In this subsection, all the definitions and properties hold indifferently for $\mathbb{V}_{\mathcal{A}}$ and $\mathbb{E}_{\mathcal{A}}$. In what follows, $\mathbb{X} \in \{\mathbb{E}, \mathbb{V}\}$ is fixed.*

Definition 229 (Stratified forcing monoid and **MA**)**.** *A **stratified forcing monoid** is a structure $(\mathcal{M}, \phi)$ where:*

- *$\mathcal{M}$ is a forcing monoid*
- *ϕ is a function called the **stratification map** $\mathcal{M} \to \overline{\mathbb{N}}$ such that:*
  - *$p \preceq q \Rightarrow \phi(q) \leq \phi(p)$*

*A n-**MA** $\mathcal{A}$ is **stratified** iff its carrier is.*

It is in fact the set $\mathcal{I}(\mathbb{X}_{\mathcal{A}})^S$, with S being any set, that we endow with a structure of metric space. It naturally represents a subset of the predicates on S. For example the following type with y a free variable of sort S:

$$\Downarrow(\forall x \in S. X(x) \multimap \Uparrow X(y))$$

can be seen as a map that takes $X : S \to \mathcal{I}(\mathbb{V}_{\mathcal{A}})$ and returns an element of $S \to \mathcal{I}(\mathbb{V}_{\mathcal{A}})$.

Definition 230 (n-approximation)**.** *Let $n \in \overline{\mathbb{N}}$. Let $X \in \mathcal{I}(\mathbb{X}_{\mathcal{A}})$. Then we define its n-**approximation** as:*

$$\pi_n(X) \stackrel{\text{def}}{=} s \mapsto \{\ (x, p) \in X(s) \mid \phi(p) \leq n\ \}$$

In particular, $\pi_\infty(X) = X$.

We begin by endowing $\mathcal{I}(\mathbb{X}_{\mathcal{A}})^S$ with a structure of pseudo-ultrametric space based on the notion of n-approximation.

Definition 231 (Cantor ultrametric)**.** *Let $X, Y \in \mathcal{I}(\mathbb{X}_{\mathcal{A}})^S$. Then we define:*

$$\begin{array}{rcl} D(X,Y) & \stackrel{\text{def}}{=} & \sup(\{\ k \in \overline{\mathbb{N}} \mid \pi_k(X) = \pi_k(Y)\ \}) \\ d(X,Y) & \stackrel{\text{def}}{=} & 2^{-D(X,Y)} \end{array}$$

with $2^{-\infty} \stackrel{\text{def}}{=} 0$.

Lemma 232. *Let $k \in \overline{\mathbb{N}}$. If $k \leq D(X,Y)$ then $\pi_k(X) = \pi_k(Y)$.*

Proof. It is clear, because $\pi_n(X) = \pi_n(Y)$ implies that $\pi_k(X) = \pi_k(Y)$ for any $k \leq n$. □

We now prove that the distance induces an ultrametric on $\mathcal{I}(\mathbb{X}_{\mathcal{A}})^S$.

Property 233. *$(\mathcal{I}(\mathbb{X}_{\mathcal{A}}))^S, d)$ is a pseudo-metric space. It is in fact a pseudo ultrametric space.*

Proof. We show that d is a pseudo-ultrametric, which implies that it is a pseudo-metric. We only prove the ultrametric inequality.

We want to prove the following inequality:

$$d(X,Z) \leq \max(d(X,Y), d(Y,Z))$$

We have $\max(d(X,Y), d(Y,Z)) = \max(2^{-D(X,Y)}, 2^{-D(Y,Z)})$. There are two possible cases:

- If $D(X,Y) < D(Y,Z)$, then $2^{-D(Y,Z)} < 2^{-D(X,Y)}$. By Lemma 232 we have

  $$\pi_{D(X,Y)}(Y) = \pi_{D(X,Y)}(Z)$$

  But we also know that

  $$\pi_{D(X,Y)}(X) = \pi_{D(X,Y)}(Y)$$

  Hence,

  $$\pi_{D(X,Y)}(X) = \pi_{D(X,Y)}(Z)$$

  By definition of $D(X,Z)$ it means that

  $$D(X,Y) \leq D(X,Z)$$

  Therefore, by combining everything:

  $$d(X,Z) = 2^{-D(X,Z)} \leq 2^{-D(X,Y)} = \max(2^{-D(X,Y)}, 2^{-D(Y,Z)}) = \max(d(X,Y), d(Y,Z))$$

- If $D(Y,Z) \leq D(X,Y)$ then the proof is symmetric.

□

We don't have the separation property, but as stated in Proposition 220, by quotienting using the following equivalence

$$X \approx Y \Leftrightarrow \forall k \in \mathbb{N}, \pi_k(X) = \pi_k(Y)$$

we obtain a metric space. We implicitely work in this space from now on.

We now want to show that our metric space is **complete**. We use an intermediate lemma that rephrase what it means to converge or be a Cauchy sequence in our particular ultrametric space.

Lemma 234. *The following equivalences hold:*

- $(X_n)_{n\in\mathbb{N}}$ *is Cauchy* $\Longleftrightarrow \forall k \in \mathbb{N}, \exists N \in \mathbb{N}, \forall m,n \geq N, k \leq D(X_n, X_m)$.
- $(X_n)_{n\in\mathbb{N}}$ *converges to* $X \Longleftrightarrow \forall k \in \mathbb{N}, \exists N \in \mathbb{N}, \forall n \geq N, k \leq D(X, X_n)$.

Proof. We only prove the first statement, the second one being similar.

- Suppose that $(X_n)_{n\in\mathbb{N}}$ is Cauchy. Let $k \in \mathbb{N}$. There exists $\epsilon > 0$ such that $k \leq -\log_2(\epsilon)$. Because X_n is Cauchy, we have $N \in \mathbb{N}$ such that

  $$\forall n, m \geq N, d(X_n, X_m) \leq \epsilon$$

  We want to prove that if $n, m \geq N$, we have

  $$k \leq D(X_n, X_m)$$

  But $d(X_n, X_m) = 2^{-D(X_n, X_m)}$, hence

  $$-\log_2(\epsilon) \leq D(X_n, X_m)$$

  We conclude because $k \leq -\log_2(\epsilon)$.

- Suppose that $(X_n)_{n\in\mathbb{N}}$ satisfies the right condition. Let $\epsilon > 0$. There exists $k \in \mathbb{N}$ such that $2^{-k} \leq \epsilon$. Hence, we have an $N \in \mathbb{N}$ such that for all $m, n \geq N$,

  $$k \leq D(X_n, X_m)$$

  This implies that

  $$d(X_n, X_m) = 2^{-D(X_n, X_m)} \leq 2^{-k} \leq \epsilon$$

□

Property 235. *Every Cauchy sequence on $(\mathcal{I}(\mathbb{X}_{\mathcal{A}})^S, d)$ converges.*

Proof. Let $(X_n)_{n\in\mathbb{N}}$ be a Cauchy sequence. We construct a new set $X \in \mathcal{I}(\mathbb{X}_{\mathcal{A}})^S$ slice by slice, n-approximation by n-approximation, which is enough since we are working modulo $\approx$. We define $G_n = \pi_n(X)$ by induction on $n \in \mathbb{N}$ and then pose X as follows:

$$X = \bigcup_{i\in\mathbb{N}} G_i$$

To make it work, we just need to ensure that $G_i \subseteq G_{i+1}$.

Let $k \in \mathbb{N}$. Then by Lemma 234, we have $N(k) \in \mathbb{N}$ such that for all $n, m \geq N(k)$,

$$k \leq D(X_n, X_m)$$

That means by Lemma 232 that $\forall n \geq N(k)$,

$$\pi_k(X_n) = \pi_k(X_{N(k)})$$

We pose for any $k \in \mathbb{N}$

$$G_k = \pi_k(X_{N(k)})$$

We immediately have that $G_k \subseteq G_{k+1}$ for any $k \in \mathbb{N}$. By posing

$$X = \bigcup_{i\in\mathbb{N}} G_i$$

we have that $\pi_n(X) = G_n$. We prove that $X \in \mathcal{I}(\mathbb{X}_{\mathcal{A}})^S$:

- $X \in \mathcal{I}(\mathbb{X}_{\mathcal{A}})^S$. Indeed, for $s \in S$, if we have $(x,p) \in X(s)$ and $p \preceq q$, then $\phi(q) \leq \phi(p)$ by the stratification condition. But $(x,p) \in \pi_{\phi(p)}(X)(s) = G_{\phi(p)}(s) = \pi_{\phi(p)}(X_{N(\phi(p))})(s)$. On the other hand, since $\pi_{\phi(p)}(X_{N(\phi(p))})(s)$ is upward closed (because of the stratification condition), we have $(x,q) \in \pi_{\phi(p)}(X_{N(\phi(p))})(s) = G_{\phi(p)}(s) \subseteq X(s)$.

Finally, it is easy to see that X_n converges to X □

We have successfully endowed $\mathcal{I}(\mathbb{X}_{\mathcal{A}})^S$ with a structure of complete metric space. We can therefore apply the Banach fixed-point theorem.

Theorem 236 (Fixed-point Theorem)**.** *Let $\mathcal{A}$ be a stratified monitoring algebra. Suppose that $F : \mathcal{I}(\mathbb{X}_{\mathcal{A}})^S \to \mathcal{I}(\mathbb{X}_{\mathcal{A}})^S$ is contractive. Then there exists $X \in \mathcal{I}(\mathbb{X}_{\mathcal{A}})^S$ such that:*

$$F(X) \approx X$$

This fixed-point is denoted by μF.

Proof. It is a direct corollary of the Banach fixed-point theorem. □

Theorem 237 (Solution to a finite set of recursive equations)**.** *Let $\mathcal{A}$ be a stratified monitoring algebra. Suppose that we dispose of n contractive maps $F_1, \ldots, F_n : (\mathcal{I}(\mathbb{X}_{\mathcal{A}})^S)^n \to (\mathcal{I}(\mathbb{X}_{\mathcal{A}})^S)^n$. Then there exists n sets $X_1, \ldots, X_n$ that satisfy the following finite set of equations:*

$$\begin{cases} X_1 \approx F_1(X_1, \ldots, X_n) & (E_1) \\ X_2 \approx F_2(X_1, \ldots, X_n) & (E_2) \\ \quad \ldots & \\ X_n \approx F_n(X_1, \ldots, X_n) & (E_n) \end{cases}$$

Proof. It is enough to remark that we can form the following map Φ:

$$\Phi \stackrel{\text{def}}{=} (X_1, \ldots, X_n) \to (F_1(X_1, \ldots, X_n), \ldots, F_n(X_1, \ldots, X_n))$$

This map is a contractive map from $(\mathcal{I}(\mathbb{X}_{\mathcal{A}})^S)^n$ to $(\mathcal{I}(\mathbb{X}_{\mathcal{A}})^S)^n$. We can then use Banach fixed-point theorem to conclude. □

Remark 238. *It is always possible to stratify a **MA** $\mathcal{A}$ by choosing $\phi(p) = 0$ for all $p \in |\mathcal{A}|$, but then the only contractive maps are the constant maps. In general, even if we have a fixed-point theorem, its interest really depends on the kind of maps that can be proved to be contractive.*

6.4 | Step-indexing

We now explain how we can apply Theorem 236 to the step-indexing algebra $\mathcal{A}_{\mathsf{step}}$ and retrieve the results stated in section 4.5.3. We then explain how to transport non-expansiveness and contractiveness of certain maps in $\mathcal{A}_{\mathsf{step}}$ to other **MA**s.

6.4.1 Preliminaries

We first give many useful technical results. This subsection can be skipped in first reading, as it is mostly useful for the proofs of this section, but not for its understanding.

Property 239. *Suppose that $\mathcal{A}$ is stratified. Let $\odot$ be a simple $\mathcal{A}$-connective, with $\sigma(-) + \sigma(+) = n$, such that:*

- *If $\odot(p_1, \dots, p_n) \preceq r$ then there exists $q_1, \dots, q_n \in |\mathcal{A}|$ such that:*

$$\begin{cases} \forall i \in [1, n], p_i \preceq q_i \\ \phi(q_i) \preceq \phi(r) \\ \odot(q_1, \dots, q_n) \preceq r \end{cases}$$

*Then the map $\odot : \mathcal{I}(\mathbb{V}_{\mathcal{A}})^{\sigma(+)} \times \mathcal{I}(\mathbb{E}_{\mathcal{A}})^{\sigma(-)} \to \mathcal{I}(\mathbb{X}_{\mathcal{A}})$ is non-expansive. By convenience, if $\odot$ satisfies that condition we say that it is **non-expansive**.*

Proof.

Suppose that for some $k \in \mathbb{N}$, we have $\pi_k(X_i) = \pi_k(Y_i)$. We want to prove that $\pi_k(\odot(X_1, \dots, X_n)) = \pi_k(\odot(Y_1, \dots, Y_n))$. The situation is symmetric, so we only prove an inclusion. Let $(\odot(x_1, \dots, x_n), r) \in \odot(X_1, \dots, X_n)$. We know there are $p_1, \dots, p_n$ such that:

- $\odot(p_1, \dots, p_n) \preceq r$
- $(x_i, p_i) \in X_i$

Then, there exists $q_1, \dots, q_n$ such that:

1. $p_i \preceq q_i$
2. $\phi(q_i) \preceq k$
3. $\odot(q_1, \dots, q_n) \preceq r$

We have because of the points $1, 2, 3$ and 4 that:

$$\forall i \in [1, n], (x_i, q_i) \in \pi_k(X_i) = \pi_k(Y_i)$$

Hence, $(\odot(x_1, \dots, x_n), \odot(q_1, \dots, q_n)) \in \odot(Y_1, \dots, Y_n)$. By $\preceq$-saturation we also have

$$(\odot(x_1, \dots, x_n), r) \in \odot(Y_1, \dots, Y_n)$$

□

Property 240. *Let $\mathcal{A}$ be stratified. Suppose that:*

- *For all $p, q \in |\mathcal{A}|$ there exists $p', q' \in |\mathcal{A}|$ such that*
  - $p \preceq p'$
  - $\phi(p') \leq \phi(q)$
  - $p' \bullet q \preceq p \bullet q$
  - $q \bullet p' \preceq q \bullet p$

*Then both $\Uparrow$ and $\Downarrow$ are non-expansive. By convenience we will say that $\perp_{\mathcal{A}}$ is **non-expansive** if this condition is satisfied.*

Proof. We only prove the case where $X, Y \in \mathcal{I}(\mathbb{E}_{\mathcal{A}})$, the other case being symmetric. We only need to show that given $\pi_k(X) = \pi_k(Y)$, we have $\pi_k(X^{\perp_{\mathcal{A},k}}) = \pi_k(Y^{\perp_{\mathcal{A},k}})$. Let $(x,p) \in \pi_k(X^{\perp_{\mathcal{A},k}})$ and $(y,q) \in Y$. We know that $\phi(p) \leq k$. We want to prove that

$$(x,p) \perp_{\mathcal{A},k} (y,q)$$

But there is q' such that $q \preceq q'$ and $\phi(q') \leq k$. Hence $(y,q') \in \pi_k(Y) = \pi_k(X)$. Therefore we have

$$(x,p) \perp_{\mathcal{A},k} (y,q')$$

Since, $p \bullet q' \preceq p \bullet q$ we have

$$(x,p) \perp_{\mathcal{A},k} (y,q)$$

□

Remark 241. *It is worth remarking that if $\mathcal{A}$ is commutative, hence $+$ and $\bullet$ being the same operation, it is enough that $+$ is non-expansive (in the sense of Property 239) for the condition of Property 240 to be met.*

Property 242. *Let $\mathcal{A}$ be a stratified **MA**. Then the following map is non-expansive:*

$$\exists_s : \begin{cases} \mathcal{I}(\mathbb{X}_{\mathcal{A}})^{S \times s} & \to & \mathcal{I}(\mathbb{X}_{\mathcal{A}})^{S} \\ X & \to & \bigcup_{x \in s} X(x) \end{cases}$$

Proof. Suppose that $\pi_k(X) = \pi_k(Y)$. Let $(x,p) \in \pi_k(\bigcup_{a \in s} X(a))$. Then there exists $a \in s$ such that $(x,p) \in \pi_k(X(a)) = \pi_k(Y(a))$. Hence $(x,p) \in \bigcup_{a \in s} Y(a)$. □

Definition 243 (Step-indexed monitoring algebra)**.** *A **step-indexed monitoring algebra** is a stratified **MA** $\mathcal{A}$ such that:*

- *$+$ is non-expansive.*

- $\bot_{\mathcal{A}}$ *is non-expansive.*

Definition 244 (Next modality). *In a step-indexed* ***MA*** $\mathcal{A}$*, a* ***next modality*** *is a* $\mathcal{A}$*-modality* $\triangleright$ *such that if* $\triangleright(p) \preceq r$ *then there exists* q *such that*

$$\begin{cases} p \preceq q \\ \phi(q) + 1 \preceq \phi(r) \\ \triangleright(q) \preceq r \end{cases}$$

Lemma 245. *If* $\mathcal{A}$ *is step-indexed, then any next* $\mathcal{A}$*-modality is contractive.*

Proof. Let $X, Y \in \mathcal{I}(\mathbb{V}_{\mathcal{A}})$. Suppose that $\pi_k(X) = \pi_k(Y)$. We want to show that $\pi_{k+1}(\triangleright X) = \pi_{k+1}(\triangleright Y)$. It is sufficient to show one of the two inclusions, the other case being symmetric. Let $(v, p) \in \pi_{k+1}(\triangleright X)$: we want to prove that $(v, p) \in \triangleright Y$ (it is sufficient since $\phi(p) \leq k + 1$). We have $\triangleright(q) \preceq p$ for some q such that $(v, q) \in X$. We have $\phi(\triangleright(q)) = \phi(q)+1$. Hence we have $\phi(p) \leq \phi(q)+1$.

Because $\mathcal{A}$ is step-indexed, there exists q' such that $q \preceq q'$ and $\phi(q') + 1 \leq \phi(p)$, with $\triangleright(q') \preceq p$. Then by $\preceq$ -saturation of X, we have $(v, q') \in X$.

- If $\phi(p) = 0$ then $\phi(q') \leq \phi(p) = 0$. Moreover $\triangleright(q') \preceq p$. Then $(v, q') \in \pi_k(X) = \pi_k(Y)$. Hence $(v, q') \in Y \subseteq \triangleright Y$.
- If $\phi(p) = m + 1$ for some $m \in \mathbb{N}$, then $m \leq k$. We moreover know that $\phi(q') + 1 \leq m + 1 \leq k + 1$ hence $\phi(q') \leq k$. We then have $(v, q') \in \pi_k(X) = \pi_k(Y)$. Then $(v, \triangleright(q')) \in \triangleright Y$. But $\triangleright(q') \preceq p$ hence the result.

□

Property 246. *Let* $F : X \times Y \mapsto F(X, Y)$ *be a map such that* $X \mapsto F(X, Y)$ *and* $Y \mapsto F(X, Y)$ *are non-expansive. Suppose moreover that* $\triangleright$ *is a next modality. Then,*

$$Y \mapsto \mu(X \mapsto \triangleright F(X, Y)) \text{ is contractive}$$

Proof. We want to prove that if $\pi_k(Y) = \pi_k(Z)$ then

$$\pi_{k+1}(\mu(X \mapsto \triangleright F(X, Y))) = \pi_{k+1}(\mu(X \mapsto \triangleright F(X, Z)))$$

We prove this by induction on $k \in \mathbb{N}$. We have

$$\mu(X \mapsto \triangleright F(X, Y)) = \triangleright F(\mu(X \mapsto \triangleright F(X, Y)), Y)$$

Since $\triangleright$ is contractive, we only need to prove that

$$\pi_k(F(\mu(X \mapsto \triangleright F(X, Y)), Y)) = \pi_k(F(\mu(X \mapsto \triangleright F(X, Z)), Z))$$

But since F is non-expansive, we only need to show that the k-approximations of its argument are equal. For the first argument, this is the induction hypothesis. For the second it is the hypothesis. □

Let $\mathcal{A}$ be step-indexed. Consider a type P that is built using the core grammar of LCBPV types augmented with any numbers of $\mathcal{A}$-connectives that satisfy the condition of Property 239. Let ρ be a $\mathcal{A}$-valuation, $x_1, \dots, x_n$ are first-order variables of respective sorts $S_1, \dots, S_n$ and X a type variable of arity $S_1 \times \cdots \times S_n$ (denoted by S). Then P induces a map:

$$\|P\|_{\mathcal{A},k,\rho}(X) : \left\{ \begin{array}{rcl} \mathcal{I}(\mathbb{V}_{\mathcal{A}})^S & \to & \mathcal{I}(\mathbb{V}_{\mathcal{A}})^S \\ C & \mapsto & s \mapsto \|P\|_{\mathcal{A},k,\rho[\vec{x} \leftarrow s, X \leftarrow C]} \end{array} \right.$$

Lemma 247. *Let $k \in \mathbb{N}$. For any positive predicate P of arity S which is the composition of the core connectives and $\mathcal{A}$-connectives that satisfy the condition of Property 239, and for any positive type variable X of arity S', the map $\|P\|_{\mathcal{A},k,\rho}(X)$ is non-expansive.*

Proof. This map is the composition of non-expansive maps (by Property 239 and Property 240), its induced map $\|P\|_{\mathcal{A},k,\rho}(X)$ is itself non-expansive. □

6.4.2 Step-indexing Algebra

We now consider a specific step-indexed algebra, which has already been considered: the **step-indexing monitoring algebra**. We remind its definition, and prove some elementary facts about it.

Definition 248 (Step-indexing algebra)**.** *The 1-**MA** $\mathcal{A}_{\mathsf{step}}$ is defined as:*

- $|\mathcal{A}_{\mathsf{step}}| \stackrel{\mathsf{def}}{=} (\overline{\mathbb{N}}, \min, \infty, \geq)$
- $\mathcal{C}_{\mathcal{A}_{\mathsf{step}}}(n) \stackrel{\mathsf{def}}{=} \{\ \underline{k} \mid k \leq n\ \}$

$\mathcal{A}_{\mathsf{step}}$ is idempotent, so by Property 167, the contraction rule is $\mathcal{A}_{\mathsf{step}}$-sound. $\mathcal{A}_{\mathsf{step}}$ is moreover trivially stratified, by choosing $\phi(n) \stackrel{\mathsf{def}}{=} n$. It is also step-indexed, as suggested by its name.

Fact 249. *$\mathcal{A}_{\mathsf{step}}$ is step-indexed.*

Proof. By Remark 241, we only have to check that $\min$ is non-expansive. Suppose that $\min(m, n) \geq k$. We then have:

$$\left\{ \begin{array}{l} m \geq k \text{ and } n \geq k \\ k \leq k \\ \min(k, k) \geq k \end{array} \right.$$

Hence the conclusion. □

This in particular implies that $\multimap, \otimes, \Uparrow.$ and $\Downarrow$ all induce non-expansive maps.

Property 250 (Approximation). *If $t \in \mathbb{P}$ and $Y \in \mathcal{I}(\mathbb{E}_{\mathcal{A}_{\mathsf{step}}})$ then for any $k \in \mathbb{N}$, we have the following equivalence:*

$$(t, \infty) \in Y^{\perp_{\mathcal{A},k}} \Leftrightarrow \forall n \in \mathbb{N}, (t, n) \in Y^{\perp_{\mathcal{A},k}}$$

Proof. We only prove the $\Leftarrow$ implication, the other one being immediate by $\geq$-saturation. Suppose that for every $n \in \mathbb{N}$, $(t, n) \in Y^{\perp_{\mathcal{A},k}}$ and take $(E, m) \in Y$. We want to show that $(t, E, \underbrace{\min(\infty, m)}_{=m}) \in \perp\!\!\!\perp_{\mathcal{A},k}$.

Two cases are possible:

- If $m \in \mathbb{N}$, then we know that $(t, m) \in Y^{\perp_{\mathcal{A},k}}$ and since $\min(m, m) = m$, we have the result.
- If $m = \infty$, then we need to show that for any $k \in \mathbb{N}$, we have

  $$(\star) \quad \langle t, \mathsf{a}(\underline{\mathsf{k}}).E\rangle_k \in \perp\!\!\!\perp$$

  But we know that $(t, k) \in Y^{\perp_{\mathcal{A},k}}$ by hypothesis. Moreover, by $\geq$-saturation of Y, we have $(E, k) \in Y$. Hence, $(t, E, k) \in \perp\!\!\!\perp_{\mathcal{A},k}$ and therefore we deduce $(\star)$.

□

Remark 251. *A consequence of this property is that for every $X, Y \in \mathcal{I}(\mathbb{E}_{\mathcal{A}_{\mathsf{step}}})$ (resp. $X, Y \in \mathcal{I}(\mathbb{V}_{\mathcal{A}_{\mathsf{step}}})$) we have the following implication:*

$$X \approx Y \Longrightarrow X^{\perp_{\mathcal{A},k}} = Y^{\perp_{\mathcal{A},k}}$$

6.4.3 A contractive modality

In order to use the fixed-point theorem we proved for stratified **MA**s, we define a new $\mathcal{A}_{\mathsf{step}}$-modality which will turn out to be a *next modality*, hence contractive. Therefore, by composing this modality with any non-expansive map induced by a type yields a contractive map (by Property 226). This will be used to define different kinds of **recursive types**.

Definition 252 (Next modality). *We define $\triangleright$ as the following next $\mathcal{A}_{\mathsf{step}}$-modality:*

$$\triangleright(n) \stackrel{\mathsf{def}}{=} n + 1$$

Proof. It is easy to see that it is indeed order-preserving and sub-additive. It is also trivial that it is a next modality. □

Property 253. *This modality satisfies multiple remarkable properties:*

1. $X \subseteq \triangleright X$
2. $\triangleright$ *distributes over* $\otimes$

Proof.

1. If $(v, n) \in X$, then $(v, n+1) \in \triangleright X$, but since $n \leq n+1$, we also have $(v, n) \in \triangleright X$.
2. This is an immediate consequence of the following equality:

$$\min(n, m) + 1 = \min(n+1, m+1)$$

□

6.4.4 Guarded recursive types

We now have all the ingredients to interpret **recursive types**. We show that we can interpret them in two flavours: **guarded recursive types** and **non-guarded recursive types**. We begin by extending the original grammar of λ_{LCBPV} positive types as follows:

$$P, Q \quad ::= \quad \ldots \mid \mu X.P$$

such that in the constructor $\mu X.P$:

- P is a positive predicate of arity S
- X is a positive predicate variable of the same arity S.

Since $\triangleright$ is contractive, Property 226 immediately tells us that all maps induced by types of the form $\triangleright P$ are contractive. Hence, they have fixed-points (unique modulo $\approx$).

Theorem 254 (Fixed-points). *Let P be a positive predicate of arity $S = S_1 \times \ldots S_n$ and X be a positive type variable. Then for any level $k \in \mathbb{N}$ and any $\mathcal{A}_{\mathsf{step}}$-model ρ the map $\| \triangleright P \|_{\mathcal{A}_{\mathsf{step}},k,\rho}$ has a fixed-point $\mu X. \| \triangleright P \|_{\mathcal{A}_{\mathsf{step}},k,\rho}$, which is unique modulo $\approx$.*

Proof. It is a simple application of the Banach fixed-point theorem, by remarking that since $\triangleright$ is contractive and $\|P\|_{\mathcal{A}_{\mathsf{step}},k,\rho}$ is non-expansive by Lemma 247 and Property 246, $\| \triangleright P \|_{\mathcal{A}_{\mathsf{step}},k,\rho}$ is contractive by Property 226. Therefore this map as a unique fixed-point modulo $\approx$. □

We use Theorem 254 to extend the interpretation of positive types by

$$\|\mu X.P\|_{\mathcal{A}_{\mathsf{step}},k,\rho} \stackrel{\mathsf{def}}{=} \mu X. \| \triangleright P \|_{\mathcal{A}_{\mathsf{step}},k,\rho}$$

It is time to wonder what kind of typing rules we can prove to be sound in $\mathcal{A}_{\mathsf{step}}$. Consider the following **guarded recursive types** fold and unfold rules, respectively for the positive recursive types, described in Figure 2.

Positive recursive types

$$\frac{\mathcal{E};\Gamma \vdash_1 v : (\mu X.P, n) \qquad n \in \mathbb{N}}{\mathcal{E};\Gamma \vdash_1 v : (\triangleright P[\mu X.P/X], n)} \qquad \frac{\mathcal{E};\Gamma \vdash_1 v : (\triangleright P[\mu X.P/X], n) \qquad n \in \mathbb{N}}{\mathcal{E};\Gamma \vdash_1 v : (\mu X.P, n)}$$

Figure 2: Guarded recursive types

Proposition 255. *The typing rules for guarded recursive types described in Figure 2 are $\mathcal{A}_{\mathsf{step}}$-sound.*

Proof. This is a direct corollary of Theorem 254. □

Remark 256. *As a corollary of Proposition 255 and of the parametric soundness theorem, every program typable in $\lambda_{\mathsf{LCBPV}}^{\otimes\mathsf{Nat}\forall}$ augmented with the guarded fold and unfold rules are terminating. The fact that every occurence of the variable we consider is guarded by the next modality $\triangleright$ prevents divergence.*

6.4.5 Non-guarded recursive types

What if we want to obtain unrestricted recursive types, *i.e.* get rid of the $\triangleright$ modality? First, concerning the fold rule, we can simplify it since $X \subseteq \triangleright X$ (Property 253) as follows:

$$\frac{\mathcal{E};\Gamma \vdash_1 v : (P[\mu X.P/X], n)}{\mathcal{E};\Gamma \vdash_1 v : (\triangleright P[\mu X.P/X], n)}{\mathcal{E};\Gamma \vdash_1 v : (\mu X.P, n)}$$

This is the usual fold rule for recursive types. Now if we turn our attention to the unfold rule and use the definition of $\triangleright$, we obtain:

$$\frac{\mathcal{E};\Gamma \vdash_1 v : (\mu X.P, n+1)}{\mathcal{E};\Gamma \vdash_1 v : (P[\mu X.P/X], n)}$$

In order to obtain an unrestricted unfold rule, the idea is to find a way to pass from n to $n+1$. Remind the following lemma proved in Section 4.5.3.

Lemma 257. *If we take* $\alpha_{\mathsf{step}} = \mathsf{case}\, x \,\mathsf{of}\, x.✠ \parallel x.\mathsf{ret}(x)$*, the following rule is* $\mathcal{A}_{\mathsf{step}}$*-sound:*

$$\frac{\mathcal{E};\Gamma \vdash_1 t : (N, n)}{\mathcal{E};\Gamma \vdash_1 (\!|t|\!)_1^{\alpha_{\mathsf{step}}} : (N, n+1)}$$

We would like to precompose the unfold typing rule with the observation rule. The problem with the unfold rule is that we can't precompose it with the observation typing rule since it can only be applied to computations. However, we can consider the following rule:

$$\frac{\Gamma \vdash v : \mu X.P \qquad \Gamma, x : P[\mu X.P/X] \vdash t : N}{\Gamma \vdash \mathsf{let\,fold}\ x\ =\ v\ \mathsf{in}\ t : N}$$

Remark 258. *This corresponds to the fold rule given by Levy in his PhD thesis and in [Lev99].*

In our framework, we consider a corresponding realizability typing rule.

Property 259. *The following rule is* $\mathcal{A}_{\mathsf{step}}$*-sound.*

$$\frac{\mathcal{E};\Gamma \vdash_1 v : (\mu X.P, n) \qquad \mathcal{E};\Delta, x : P[\mu X.P/X] \vdash_1 t : (N, m)}{\mathcal{E};\Gamma,\Delta \vdash_1 \mathsf{ret}(v)\ \mathsf{to}\ x.(\!|t|\!)_1^{\alpha_{\mathsf{step}}} : (N, \min(m, n))}$$

Proof. First, consider the following part of the derivation:

$$\frac{\dfrac{\mathcal{E};\Delta, x : P[\mu X.P/X] \vdash_1 t : (N, m)}{\mathcal{E};\triangleright\Delta, x : \triangleright P[\mu X.P/X] \vdash_1 (\!|t|\!)_1^{\alpha_{\mathsf{step}}} : (N, m)}}{\mathcal{E};\Delta, x : \triangleright P[\mu X.P/X] \vdash_1 (\!|t|\!)_1^{\alpha_{\mathsf{step}}} : (N, m)}$$

On the other hand, we have:

$$\frac{\dfrac{\mathcal{E};\Gamma \vdash_1 v : (\mu X.P, n)}{\mathcal{E};\Gamma \vdash_1 v : (\triangleright P[\mu X.P/X], n)}}{\mathcal{E};\Gamma \vdash_1 \mathsf{ret}(v) : (\Uparrow(\triangleright P[\mu X.P/X]), n)}$$

Notice that if $n = \infty$, this still works since we can prove the conclusion for every $n \in \mathbb{N}$, and then since the conclusion is negative, we can use Property 250 to re-obtain ∞. Finally, we deduce the following conclusion

$$\mathcal{E};\Gamma,\Delta \vdash_1 \mathsf{ret}(v)\ \mathsf{to}\ x.(\!|t|\!)_1^{\alpha_{\mathsf{step}}} : (N, \min(m, n))$$

□

Remark 260. *An important consequence when considering these non-guarded recursive types is that we loose termination. Indeed, if we use the observation* $(\!|.|\!)_1^{\alpha_{\mathsf{step}}}$ *in a program, and if we don't know in advance how many times this observation will arrive in head posi-*

tion, we can't tell if the term terminates or not: even if the program diverges, for any value $\underline{\mathsf{n}}$ *we can put in the memory cell, the observation will stop after a finite number of times it arrives in head position, even though the execution could have continued.*

Guarded recursive types

$$\frac{\mathcal{E};\Gamma \vdash_1 v : (\triangleright(P[\mu X.P/X]), n)}{\mathcal{E};\Gamma \vdash_1 v : (\mu X.P, n)} \qquad \frac{\mathcal{E};\Gamma \vdash_1 v : (\mu X.P, n)}{\mathcal{E};\Gamma \vdash_1 v : (\triangleright(P[\mu X.P/X]), n)}$$

Non-guarded recursive types

$$\frac{\mathcal{E};\Gamma \vdash_1 v : (P[\mu X.P/X], n)}{\mathcal{E};\Gamma \vdash_1 v : (\mu X.P, n)}$$

$$\frac{\mathcal{E};\Gamma \vdash_1 v : (\mu X.P, n) \qquad \mathcal{E};\Delta, x : P[\mu X.P/X] \vdash_1 t : (N, m)}{\mathcal{E};\Gamma,\Delta \vdash_1 \mathsf{ret}(v)\ \mathsf{to}\ x.(\!|t|\!)_1^{\alpha_{\mathsf{step}}} : (N, \min(m, n))}$$

Figure 3: Guarded and non-guarded recursive types

6.4.6 Call-by-name and call-by-value translation

To illustrate how positive recursive types with the rules presented in Figure 3 are useful, we extend the translation of the call-by-name and call-by-value calculi already defined in Section 2.2 to recursive types.

6.4.7 Preservation

We give a final result concerning the preservation of the step-indexed structure by the different iterations. If $\mathcal{A}$ is stratified (by $\phi : |\mathcal{A}| \to \overline{\mathbb{N}}$) and $\mathcal{B}$ is a $\mathcal{A}$-**MA**, we can canonically extend the stratification to any simple iteration $\mathcal{A} \lhd \mathcal{B}$ or centered semi-direct iteration $\mathcal{A} \bowtie^{\delta} \mathcal{B}$ by choosing:

$$\phi(a, b) \stackrel{\mathsf{def}}{=} \phi(a)$$

Theorem 261. *If* $\mathcal{A}$ *is step-indexed, then any simple iteration* $\mathcal{D} = \mathcal{A} \lhd \mathcal{B}$ *is also step-indexed.*

Proof. We want to prove that the $+$ on $|\mathcal{D}|$ is non-expansive. Suppose that $(a, n) + (b, m) \preceq_{|\mathcal{D}|} (c, k)$. Then it means that:

$$\begin{cases} a + b \preceq_{|\mathcal{A}|} c \\ n + m \preceq_{|\mathcal{B}|} k \end{cases}$$

Extension of types

$$A, B \quad ::= \quad \ldots \mid X \mid \mu X.A$$

$$\frac{\Gamma \vdash_{\mathsf{Aff}} t : \mu X.A}{\Gamma \vdash_{\mathsf{Aff}} \mathsf{unfold}\, t : A[\mu X.A/X]} \qquad \frac{\Gamma \vdash_{\mathsf{Aff}} t : A[\mu X.A/X]}{\Gamma \vdash_{\mathsf{Aff}} \mathsf{fold}\, t : \mu X.A}$$

Syntax

$$\begin{array}{rcl} t, u & ::= & \ldots \mid \mathsf{fold}\, t \mid \mathsf{unfold}\, t \\ v, w & ::= & \ldots \mid \mathsf{fold}\, v \end{array}$$

Call-by-value reduction

$$\begin{array}{rcl} \langle \mathsf{fold}\,(t), E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle t, \mathsf{fold}\,.E\rangle_{\mathsf{V}} \\ \langle v, \mathsf{fold}\,.E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle \mathsf{fold}\, v, E\rangle_{\mathsf{V}} \\ \langle \mathsf{unfold}\,(t), E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle t, \mathsf{unfold}\,.E\rangle_{\mathsf{V}} \\ \langle \mathsf{fold}\,(v), \mathsf{unfold}\,.E\rangle_{\mathsf{V}} & \to_{\mathsf{V}} & \langle v, E\rangle_{\mathsf{V}} \end{array}$$

Call-by-name reduction

$$\begin{array}{rcl} \langle \mathsf{unfold}\,(t), E\rangle_{\mathsf{N}} & \to_{\mathsf{N}} & \langle t, \mathsf{unfold}\,.E\rangle_{\mathsf{N}} \\ \langle \mathsf{fold}\,(t), \mathsf{unfold}\,.E\rangle_{\mathsf{N}} & \to_{\mathsf{N}} & \langle t, E\rangle_{\mathsf{N}} \end{array}$$

Call-by-value translation

$$\begin{array}{rcl} (X)^{\mathsf{V}} & \stackrel{\mathsf{def}}{=} & X \\ (\mu X.A)^{\mathsf{V}} & \stackrel{\mathsf{def}}{=} & \mu X.(A)^{\mathsf{V}} \\ (\mathsf{fold}\, t)^{\mathsf{V}} & \stackrel{\mathsf{def}}{=} & (t)^{\mathsf{V}} \\ (\mathsf{unfold}\, t)^{\mathsf{V}} & \stackrel{\mathsf{def}}{=} & (t)^{\mathsf{V}}\ \mathsf{to}\ x.(\!|\mathsf{ret}(x)|\!)_1^{\alpha_{\mathsf{step}}} \end{array}$$

Call-by-name translation

$$\begin{array}{rcl} (X)^{\mathsf{N}} & \stackrel{\mathsf{def}}{=} & \Uparrow X \\ (\mu X.A)^{\mathsf{N}} & \stackrel{\mathsf{def}}{=} & \Uparrow \mu X.\Downarrow (A)^{\mathsf{N}} \\ (\mathsf{fold}\, t)^{\mathsf{N}} & \stackrel{\mathsf{def}}{=} & \mathsf{ret}(\mathsf{thunk}((t)^{\mathsf{N}})) \\ (\mathsf{unfold}\, t)^{\mathsf{N}} & \stackrel{\mathsf{def}}{=} & t\ \mathsf{to}\ x.(\!|\mathsf{force}(x)|\!)_1^{\alpha_{\mathsf{step}}} \end{array}$$

Figure 4: Extension of the linear λ-calculus with recursive types

Since $\mathcal{A}$ is step-indexed we obtain two elements $a', b' \in |\mathcal{A}|$ such that:

$$\begin{cases} a \preceq_{|\mathcal{A}|} a' \\ b \preceq_{|\mathcal{A}|} b' \\ \phi(a' + b') \leq \phi(c) \\ a' + b' \preceq_{|\mathcal{A}|} c \end{cases}$$

Therefore we have $(a,n) \preceq_{|\mathcal{D}|} (a',n)$, $(b,m) \preceq_{|\mathcal{D}|} (b',m)$. On the other hand,

$$\begin{array}{rcll} \phi((a',n)+(b',m)) & = & \phi(a'+b') & \\ & \leq & \phi(a'+b') & \text{by monotonicity of } \delta \\ & \leq & \phi(c) & \end{array}$$

Finally, we have $(a',n)+(b',m) = (a'+b', n+m) \preceq (c, n+m) \preceq (c,k)$. □

Property 262. *Let $\mathcal{A}$ be step-indexed, and consider a simple iteration $\mathcal{D} = \mathcal{A} \lhd \mathcal{B}$. If $\odot$ is a $\mathcal{B}$-connective, then its canonical extension to $\mathcal{D}$ is non-expansive.*

Proof. Straightforward. □

Property 263. *Let $\mathcal{A}$ be step-indexed, and consider a simple iteration $\mathcal{D} = \mathcal{A} \lhd \mathcal{B}$. If $\triangleright$ is a next modality in $\mathcal{B}$, then its canonical extension to $\mathcal{D}$ is also a next modality.*

Proof. Straightforward. □

In the case of the semi-direct iteration, the situation is a little bit more complicated. If $\mathcal{A}'$ is a step-indexed algebra that is a factor of $\mathcal{A}$. Then in a centered semi-direct iteration $\mathcal{D} = \mathcal{A} \ltimes^{\delta} \mathcal{B}$, if δ leaves $\mathcal{A}'$ untouched, then its step-indexed structure is transported to $\mathcal{D}$.

Theorem 264. *Suppose that $\mathcal{A}'$ is step-indexed. Let $\mathcal{A}$ be a n-**MA**, whose $\mathcal{A}'$ is a factor of. We note ι the canonical injection of $|\mathcal{A}'|$ in $|\mathcal{A}|$. Let $\mathcal{B}$ be a $\mathcal{A}$-**MA**, and δ an action of $|\mathcal{B}|$ over $|\mathcal{A}|$. Suppose that*

$$\forall p \in |\mathcal{A}'|, \delta(\iota(p)) = \iota(p)$$

Then $\mathcal{D}$ is step-indexed.

Finally, we give a general theorem that ensures the existence of unrestricted fixed-points.

Theorem 265. *Suppose that:*

- *$\mathcal{A}$ is a step-indexed k-**MA** .*
- *$\triangleright$ is a next $\mathcal{A}$-modality.*
- *$(\alpha, x \mapsto \triangleright x)$ is a $\mathcal{A}$-monitor.*

Then using the same interpretation of recursive types as for $\mathcal{A}_{\mathsf{step}}$, the following rules are $\mathcal{A}$-sound as soon as P is built using the core connectives and non-expansive connectives:

$$\frac{\mathcal{E};\Gamma \vdash_k v : (P[\mu X.P/X], p) \qquad \phi(p) \in \mathbb{N}}{\mathcal{E};\Gamma \vdash_k v : (\mu X.P, p)}$$

$$\frac{\mathcal{E};\Gamma \vdash_k v : (\mu X.P, p) \qquad \mathcal{E};\Delta, x : P[\mu X.P/X] \vdash_k t : (N, q)}{\mathcal{E};\Gamma,\Delta \vdash_k \mathsf{ret}(v)\ \mathsf{to}\ x.(\!|t|\!)^{\alpha}_{k} : (N, p+q)}$$

6.5 | Higher-order references

If first-order references can be added by considering a simple **MA**, as seen in Section 4.5, it is not the case of higher-order references. Indeed, higher-order references pose a circularity problem: the values which are put in the state are themselves able to manipulate the state. Usually, realizability models for languages featuring such higher-order references are based on the technique of step-indexing, or are based on some encoding in recursive types. In our framework, we can define a general method to build realizability models of references, including higher-order references, by relying on the results proved in Section 6.3. The method goes schematically as follows:

1. Choosing a step-indexed monitoring algebra $\mathcal{A}$, in the sense of Section 6.3.
2. Identify a type P that induces a contractive map.
3. Define a new $\mathcal{A}$-**MA** $\mathcal{B}$, using the fixed-point induced by P to define the content of the memory cell.
4. Form a simple iteration of $\mathcal{A}$ and $\mathcal{B}$.
5. We can then prove a soundness theorem for the swap instruction, as done in Section 4.5.

6.5.1 Preliminaries

In the following, we fix a step-indexed k-**MA** $\mathcal{A}$. We also suppose that we have chosen in it a contractive positive type P, *i.e.* a type built using the basic connectives, any non-expansive simple $\mathcal{A}$-connective, and such that the outer-most $\mathcal{A}$-connective is contractive in the sense of Section 6.4.

Example 266. *For instance, if $\mathcal{A}$ is equipped with a next modality $\triangleright$, we can choose any non-expansive type P and form $\triangleright P$ which is contractive.*

Notice that since P is made out of the core connectives and simple $\mathcal{A}$-connectives, it admits a forcing interpretation $P^*(\iota)$ as pointed out in Section 4.3.

Property 267. *If $\mathcal{M}$ is any forcing monoid, then the following map is contractive for any level $m \in \mathbb{N}$:*

$$\begin{cases} \mathcal{I}(\mathbb{V}_{\mathcal{A}})^{\mathcal{M}} & \to & \mathcal{I}(\mathbb{V}_{\mathcal{A}})^{\mathcal{M}} \\ C & \mapsto & (p \mapsto \|(P)^*(\iota)\|_{\mathcal{A},m,[\iota \leftarrow p, \mathsf{C} \leftarrow C]}) \end{cases}$$

where $(P)^*(\iota)$ is the forcing type transformation induced by the forcing structure $\mathcal{F} = (\mathcal{M}, \mathsf{C})$ with C being a predicate variable of arity $\mathcal{M}$.

Proof. It is enough to remark that $(P)^*(\iota)$ only makes use of the connectives of λ_{LCBPV}, the simple $\mathcal{A}$-connectives used by P (which are all non-expansive). Moreover its outer most connective $\odot$ is contractive. Indeed, remark that $(P)^*(p)$ (suppose for more clarity that $\odot$ is binary) is defined as:

$$(P)^*(p) = \exists q_1 \in \mathcal{M}, \exists q_2 \in \mathcal{M}, \{\odot(q_1, q_2) \preceq_{\mathcal{M}} p\} \wedge \odot(Q^*(q_1), N^*(q_2))$$

Hence the map induced by the forcing interpretation of P is contractive. □

6.5.2 General case

We now define a $\mathcal{A}$-**MA**, which takes advantage of the structure of step-indexed algebra. This new $\mathcal{A}$-**MA** is basically *not indexed*, similarly to $\mathcal{A}_{\mathsf{ref}[1]}$ defined in Subsection 4.5.2.

Definition 268. *We define the following $\mathcal{A}$-**MA** denoted $\mathcal{B}$:*

- *The forcing monoid $|\mathcal{B}|$ is the trivial one-element forcing monoid $\{\star\}$.*
- *$\mathcal{C}_{\mathcal{B}}$ is defined as the fixed-point of a function F defined as:*

$$F \stackrel{\mathsf{def}}{=} \begin{array}{rcl} (\mathbb{V}_{\mathcal{A}})^{\{\star\}} & \to & (\mathbb{V}_{\mathcal{A}})^{\{\star\}} \\ X & \mapsto & y \mapsto \|(P)^*(x)\|_{\mathcal{A}_0,k+1,[x \leftarrow y, \mathsf{C} \leftarrow X]} \end{array}$$

We can then define $\mathcal{C}_{\mathcal{B}}$ as a fixpoint of F:

$$\mathcal{C}_{\mathcal{B}} \stackrel{\mathsf{def}}{=} \mu F$$

- Notice that to define the test function we use the forcing interpretation of P inside $\mathcal{A}_0$ but *at level* $k+1$, although the level of $\mathcal{A}$ is k. This is because thanks to the fixed-point, it will represent the content of the memory cell in the simple iteration $\mathcal{A} \lhd \mathcal{B}$, whose level is $k+1$.

Definition 269 (Higher-order reference algebra)**.** *The resulting* $(k+1)$*-**MA** is defined as the simple iteration of* $\mathcal{A}$ *and* $\mathcal{B}$*:*

$$\mathcal{A}_{\mathsf{ref}(P)} \stackrel{\mathsf{def}}{=} \mathcal{A} \lhd \mathcal{B}$$

In this new **MA**, the content of the test function is indeed the set of realizers of P, as witnessed by the following property.

Property 270. *The following equivalence holds:*

$$(v, \overrightarrow{w}) \in \mathcal{C}_{\mathcal{A}_{\mathsf{ref}(P)}}(\star, p) \iff \exists r \in |\mathcal{A}|, \phi(r) \in \mathbb{N} \wedge \begin{cases} (v, (\star, r)) \in \|P\|_{\mathcal{A}_{\mathsf{ref}(P)}, k+1, []} \\ \overrightarrow{w} \in \mathcal{C}_{\mathcal{A}}(r \bullet p) \end{cases}$$

Proof. By definition,

$$\mathcal{C}_{\mathcal{A}_{\mathsf{ref}(P)}}(\star, p) = \{\ (v, \overrightarrow{w}) \mid \exists r \in |\mathcal{A}|, (v, r) \in \mathcal{C}_{\mathcal{B}}(\star) \wedge \overrightarrow{w} \in \mathcal{C}_{\mathcal{A}}(r \bullet p)\ \}$$

Therefore it is enough to prove that $(v, r) \in \mathcal{C}_{\mathcal{B}}(\star)$ is equivalent to $(v, (\star, r)) \in \|P\|_{\mathcal{A}_{\mathsf{ref}(P)}, k+1,}$ with $\phi(r) \in \mathbb{N}$. But by definition, $(v, r) \in \mathcal{C}_{\mathcal{B}}(\star)$ means that $(v, r) \in \mu(X \mapsto \|(P)^*(\star)\|_{\mathcal{A}, k+1, [\mathsf{C} \leftarrow X]})$. Hence $\phi(r) \in \mathbb{N}$ and by unfolding the fixed-point this is equivalent to $(v, r) \in \|(P)^*(\star)\|_{\mathcal{A}, k+1, [\mathsf{C} \leftarrow \mathcal{C}_{\mathcal{B}}(\star)]}$. By the connection theorem, this is in turn equivalent to $(v, (\star, r)) \in \|(P)\|_{\mathcal{A}_{\mathsf{ref}(P)}, k+1, []}$. Hence the conclusion.

□

We now show that we can add a new swap instruction to our language and a new $\mathcal{A}_{\mathsf{ref}(P)}$-sound typing rule. In Subsection 4.5.2 we have introduced the primitive swap. We here introduce a generalization of swap to any level: swap_{n+1}, whose reduction rule is as follows:

$$\langle \mathsf{swap}_{n+1}(v), \overrightarrow{\mathsf{a}(w')}.\mathsf{a}(w).E \rangle_{n+1} \quad \xrightarrow{n+1} \quad \langle \mathsf{ret}(w), \overrightarrow{\mathsf{a}(w')}.\mathsf{a}(v).E \rangle_{n+1}$$

Theorem 271. *The following rule is* $\mathcal{A}_{\mathsf{ref}(P)}$*-sound.*

$$\frac{\mathcal{E}; \Gamma \vdash_{k+1} v : (P, (\star, p)) \qquad \phi(p) \in \mathbb{N}}{\mathcal{E}; \Gamma \vdash_{k+1} \mathsf{swap}_{k+1}(v) : (\Uparrow P, (\star, p))}$$

Proof. Suppose that $\mathcal{E}; \Gamma \vdash_{k+1} v : (P, (\star, p))$ is $\mathcal{A}_{\mathsf{ref}(P)}$-sound. For the sake of simplicity, suppose Γ is empty. Then, we know by hypothesis that $(v, (\star, p)) \in \|P\|_{\mathcal{A}_{\mathsf{ref}(P)}, k+1, \rho}$. Let $(E, (\star, q)) \in \|P\|_{\mathcal{A}_{\mathsf{ref}(P)}, k+1, \rho}{}^{\perp_{\mathcal{A}_{\mathsf{ref}(P)}, k+1}}$ and $(w, \overrightarrow{w}) \in \mathcal{C}_{\mathcal{A}_{\mathsf{ref}(P)}}(\star, p \bullet q)$. We know by Property 270 that there exists some $r \in |\mathcal{A}|$ such that:

- $(w, (\star, r)) \in \|P\|_{\mathcal{A}_{\mathsf{ref}(P)}, k+1, []}$

- $\overrightarrow{w'} \in \mathcal{C}_{\mathcal{A}}(r \bullet p \bullet q)$

We want to prove that $\langle \mathsf{swap}(v), \mathsf{a}(w).\overrightarrow{\mathsf{a}(w')}.E\rangle_{k+1} \in \perp\!\!\!\perp$. But this configuration reduces to

$$\langle \mathsf{ret}(w), \mathsf{a}(v).\overrightarrow{\mathsf{a}(w')}.E\rangle_{k+1}$$

so it is enough to show that this last configuration is in $\perp\!\!\!\perp$. It is enough to show that $(v, \overrightarrow{w'}) \in \mathcal{C}_{\mathcal{A}_{\mathsf{ref}(P)}}(\star, r \bullet q)$. We have:

- $(v, (\star, p)) \in \|P\|_{\mathcal{A}_{\mathsf{ref}(P)}, k+1, \rho}$ on one hand
- $\overrightarrow{w'} \in \mathcal{C}_{\mathcal{A}}(r \bullet p \bullet q) = \mathcal{C}_{\mathcal{A}}(p \bullet r \bullet q)$ on the other hand

So by the other implication of Property 270, we obtain that $(v, \overrightarrow{w'}) \in \mathcal{C}_{\mathcal{A}_{\mathsf{ref}(P)}}(\star, r \bullet q)$, hence the conclusion.

□

6.5.3 Particular instances

We give various concrete instances of this construction, by choosing different starting algebras, different stratifications and different contractive maps.

6.5.3.1 First-order references

We can retrieve the first-order references already mentionned in Section 4.5 as a degenerate case. We show that our method instanciates into a method to add first-order references to any **MA**. Let $\mathcal{A}$ be a n-**MA**. Then we endow $\mathcal{A}$ with a structure of step-indexed **MA** by chosing the trivial stratification:

$$\phi \stackrel{\mathsf{def}}{=} \begin{cases} |\mathcal{A}| & \to & \overline{\mathbb{N}} \\ p & \mapsto & 0 \end{cases}$$

In the induced metric space, the only contractive maps are the constant maps. Hence Nat induces a contractive map. By Theorem 271, we obtain that the following rule is $\mathcal{A}_{\mathsf{ref}(\mathsf{Nat})}$-sound:

$$\frac{\mathcal{E}; \Gamma \vdash_{k+1} v : (\mathsf{Nat}, (\star, p))}{\mathcal{E}; \Gamma \vdash_{k+1} \mathsf{swap}_{k+1}(v) : (\Uparrow\mathsf{Nat}, (\star, p))}$$

6.5.3.2 Guarded higher-order references

We have seen in Section 6.4 that using the step-indexing algebra $\mathcal{A}_{\mathsf{step}}$, we can interpret both guarded and non-guarded recursive types. Similarly, we can use $\mathcal{A}_{\mathsf{step}}$ to obtain guarded higher-order references as well as non-guarded higher-order references. Guarded references are obtained in the following way:

- Consider the step-indexing algebra $\mathcal{A}_{\mathsf{step}}$.

- By choosing a positive type P, we know that the map induced by the interpretation at any level of $\triangleright P$ is contractive in $\mathcal{A}_{\mathsf{step}}$.
- We build $\mathcal{A}_{\mathsf{step}}[\mathsf{ref}(P)]$.

We conclude by Theorem 271 obtaining what we call **guarded higher-order references**:

Property 272. *The following rule is $\mathcal{A}_{\mathsf{step}}[\mathsf{ref}(P)]$-sound:*

$$\frac{\mathcal{E};\Gamma \vdash_2 v : (\triangleright P, (\star, p))}{\mathcal{E};\Gamma \vdash_2 \mathsf{swap}_2(v) : (\Uparrow \triangleright P, (\star, p))}$$

6.5.3.3 Non-guarded higher-order references

To obtain **non-guarded higher-order references**, similarly to non-guarded recursive types, we reuse the guarded case. Indeed, it is enough to consider the rule previously obtained:

$$\frac{\mathcal{E};\Gamma \vdash_2 v : (\triangleright P, (\star, p))}{\mathcal{E};\Gamma \vdash_2 \mathsf{swap}_2(v) : (\Uparrow \triangleright P, (\star, p))}$$

We precompose it with the $\triangleright$ promotion, thus obtaining:

$$\frac{\mathcal{E};\Gamma \vdash_2 v : (P, (\star, n))}{\mathcal{E};\Gamma \vdash_2 \mathsf{swap}_2(v) : (\Uparrow \triangleright P, (\star, n))}$$

By Property 177 and Proposition 170, we know that if we pose:

$$\alpha'_{\mathsf{step}} \stackrel{\mathsf{def}}{=} \lambda x.(\!|\mathsf{ret}(x)|\!)_1^{\alpha_{\mathsf{step}}}$$

then $(\alpha'_{\mathsf{step}}, \triangleright)$ is a $\mathcal{A}_{\mathsf{step}}[\mathsf{ref}(P)]$-monitor. Hence, we can finally post-compose the previous typing rule with the monitor typing rule, following Lemma 191 we have

Property 273. *The following rule is $\mathcal{A}_{\mathsf{step}}[\mathsf{ref}(P)]$-sound:*

$$\frac{\mathcal{E};\Gamma \vdash_2 \mathsf{swap}_2(v) : (\Uparrow \triangleright P, (\star, n))}{\mathcal{E};\Gamma \vdash_2 \mathsf{swap}_2(v)\ \mathsf{to}\ x.(\!|\mathsf{ret}(x)|\!)_1^{\alpha_{\mathsf{step}}} : (\Uparrow P, (\star, n))}$$

Chapter VII

Some applications of the monitoring algebra theory

Contents

This chapter is devoted to the study of three more complex applications of the monitoring algebras. Three main examples are used to illustrate the use of **MA**s:

- A first application concerns substructural naive set theories, *i.e.* set theories with an unrestricted comprehension scheme, but based on substructural logic (for instance light logics [Gir98]). Such theories, where contraction is weakened, are known to be consistent. But until now, no semantical proof of the consistency of these theories was known. This situation is a generalization of what happens in the case of linear recursive types:

in general one can add linear recursive types in any type system without harming termination. This fact is known since Girard [Gir92], but again, no semantical proof of this fact was known. Our development gives a solution to this problem as well. We begin by this example, because the technique used here will surprisingly be reused in the two other examples.

- The second application is the correctness of a polynomial-time programming language. We introduce a call-by-value programming language based on soft linear logic [Laf04], that features unrestricted recursive types. Despite the presence of unrestricted recursive types, all typable programs enjoy a termination property. Moreover, the bound on the reduction of those programs is a polynomial in their size. This language gives in fact a characterization of the complexity classes **P** and **FP**. We show that using monitoring algebras we can prove the bounded-time termination property of typable programs.
- Finally, we consider a core programming language with higher-order references, very much inspired by [AFM07], that has the particularity of supporting *strong updates*. Strong updates allow to change the type of the reference *in the course of the execution*. When used in an unrestricted way this usually result in the loss of program safety. This language however relies on the use of linear capabilities to access the reference, which ensures the termination of programs. The elaboration of a good monitoring algebra and the proof of termination reuses the technique used for the linear naive set theory example.

7.1 | Linear naive set theory

We have seen in the Section 6.4 that by considering the step-indexing algebra $\mathcal{A}_{\mathsf{step}}$, it is possible to obtain two kinds recursive types:

- Guarded recursive types: in that case the realizability model still implies termination.
- Non-guarded recursive types: in which case we loose termination and only observe safety.

There is another kind of recursive type which, like guarded recursive types, do not harm termination: the **linear fixpoints**. It is well-known [Gir92] that in the context of linear logic, adding recursive types $\mu X.L$ for every *linear* type L (*i.e.* that does not contain any occurence of the exponential modality !) does not break strong normalization, even though they are not guarded.

Here, we turn our interest to an even more general and older question: the consistency of **linear naive set theory**. By naive set theory, we mean a set theory with *the principle of unrestricted comprehension*: for every predicate $P(x)$ there exists a set $\{\ x \mid P\ \}$ such that:

$$\forall t,\ (t \in \{\ x \mid P\ \} \text{ is logically equivalent to } P(t))$$

It is well-known that in intuitionistic or classical logic, this principle yields an inconsistency (for example considering the set $\{\ x \mid x \notin x\ \}$ implies Russel' Paradox). Grishin [Gri82] first

introduced in 1982 a naive set theory based on a contraction-free logic and showed the consistency of such a theory. More expressive (and consistent) naive set theories have then been proposed, based on light logics [Gir98, Ter04]: these theories are based on logics that have a form of contraction that is not as powerful as the linear logic one but ensure consistency. The consistency of these theories is given by the cut-elimination theorem, usually proved by a syntactic argument [Ter04, Shi94]. The question of finding a semantical proof of the consistency of these theories has been regularly raised [Kom89, Shi94, Ter02] but to our knowledge, no such semantical proof has been given. We propose such a proof in the case of linear naive set theory (*i.e.* , without any contraction), based on the combination of two 1-**MA**s:

- The quantitative 1-**MA** based on $\mathbb{N}$.
- The step-indexing **MA** $\mathcal{A}_{\mathsf{step}}$.

The termination argument is then obtained by a surprising interaction between the two monitors α_{time} and α_{step}, which compete to make their respective counter reach 0 first. We finally show evidence that this proof can be extended to more expressive naive set theories such as elementary set theory or light affine set theory [Ter02].

7.1.1 Linear set theory

The syntax of linear naive set theory is an extension of the core λ_{LCBPV}.

$$\begin{array}{rccl} & \alpha & \in & Var \\ \textbf{Naive sets} & r,s & ::= & \alpha \mid \{\ \alpha \mid P\ \} \\ \textbf{Pos. types} & P,Q & ::= & P \otimes Q \mid r \in s \mid \exists \alpha.P \mid \Downarrow N \\ \textbf{Neg. types} & N,M & ::= & P \multimap N \mid \Uparrow P \mid \forall \alpha.P \end{array}$$

- A first-order term is either a first-order variable α or a naive set of the form $\{\ \alpha \mid P\ \}$, which morally represents the set of all sets s such that $P[s/\alpha]$. Here P is supposed to be a predicate such that $\mathsf{FV}(P) \subseteq \{\alpha\}$.
- The domain of quantification is the set of all closed first-order terms, *i.e.* sets of the form $\{\ \alpha \mid P\ \}$ where $\mathsf{FV}(P) \subseteq \{\alpha\}$ called **naive sets**. Formally, since this domain is defined by mutual recursion with the grammar of types, we should say that the domain of quantification is $\mathbb{N}$, and then give a bijection between $\mathbb{N}$ and the set of closed naive sets afterwards. Hence the quantification is simply the usual first-order quantification with $\mathbb{N}$ as the domain. We leave that implicit since it has no technical interest.
- The syntax of types is augmented with a membership relation $\in$ between two naive sets.

Here are the additional typing rules for this system, which constitute the naive comprehension rule and are somewhat similar to the rule for folding and unfolding of unrestricted recursive

types as in Section 6.4 :

$$\frac{\mathcal{E};\Gamma \vdash v : P[r/\alpha]}{\mathcal{E};\Gamma \vdash v : r \in \{\ \alpha \mid P\ \}} \qquad \frac{\mathcal{E};\Gamma \vdash v : r \in \{\ \alpha \mid P\ \} \qquad \mathcal{E};\Delta, x : P[r/\alpha] \vdash t : N}{\mathcal{E};\Gamma,\Delta \vdash \mathsf{ret}(v)\ \mathsf{to}\ x.t : N}$$

Similarly to second-order encoding, other connectives of Affine Logic like the **absurdity 0** or the **additive disjunction** $\oplus$ can be encoded using naive sets following [Ter04]:

$$\begin{array}{rcl} t_0 & \stackrel{\mathsf{def}}{=} & \{\ \alpha \mid \alpha \in \alpha\ \} \\ \mathbf{0} & \stackrel{\mathsf{def}}{=} & \forall \alpha. t_0 \in \alpha \\ P \oplus Q & \stackrel{\mathsf{def}}{=} & \forall \alpha. ((A \multimap t_0 \in \alpha) \multimap (B \multimap t_0 \in \alpha) \multimap t_0 \in \alpha) \end{array}$$

Here t_0 can be replaced by any naive set. We can consider the set $\{\ \alpha \mid (\alpha \in \alpha) \multimap \mathbf{0}\ \}$, whose existence usually implies inconstency due to Russel's paradox. Here it becomes harmless. Indeed the theory is consistent despite the presence of the unrestricted comprehension rule.

Theorem 274 (Linear naive set theory consistency)**.** *Linear naive set theory is consistent. More precisely, there is no term of type* $\bot$*.*

In this system, we can find a certain type P that is logically equivalent to $P \multimap \bot$, for example:

$$P \stackrel{\mathsf{def}}{=} \{\ \alpha \mid (\alpha \in \alpha) \multimap \bot\ \} \in \{\ \alpha \mid (\alpha \in \alpha) \multimap \bot\ \}$$

However, the idea behind the consistency result of linear naive set theory (or linear recursive types) is that it is impossible to duplicate P to obtain P and $P \multimap \bot$ at the same time, and apply one to the other to deduce $\bot$.

7.1.2 A realizability model

We now define a step-indexed monitoring algebra $\mathcal{B}$ in which we can interpret linear naive set theory. The difficulty lies in the interpretation of the membership relation, which is defined by solving a set of recursive equations in $\mathcal{B}$.

7.1.2.1 Monitoring algebra

The monitoring algebra chosen is a very simple one, resulting of the product of two already known **MA**s.

- We start with the simplest quantitative 1-**MA** $\mathcal{A}$, based on $\mathbb{N}$ and already defined in Example 215:

$$(\mathbb{N}, +, 0, \leq, x \mapsto x)$$

- We then take the product of $\mathcal{A}$ and the step-indexing monitoring algebra $\mathcal{A}_{\mathsf{step}}$ of Section 6.4:

$$\mathcal{B} \overset{\mathsf{def}}{=} \mathcal{A} \times \mathcal{A}_{\mathsf{step}}$$

Here are a few properties satisfied by $\mathcal{B}$:

- $\mathcal{B}$ is step-indexed by Theorem 261.
- $\triangleright(n,k) \overset{\mathsf{def}}{=} (n,k+1)$ defines a next $\mathcal{B}$-modality, by Property 263.
- $(\lambda x.(\!|\mathsf{ret}(x)|\!)_1^{\alpha_{\mathsf{time}}}, (n,k) \mapsto (n+1,k))$ is a $\mathcal{B}$-monitor by Property 181.
- $(\alpha_{\mathsf{step}}, (n,k) \mapsto (n,k+1))$ is a $\mathcal{B}$-monitor by Property 181.

7.1.2.2 The interpretation

To interpret the problematic unrestricted comprehension scheme, we use the following argument. Consider a derivation π in linear naive set theory. In π, only a finite number $s_1, \dots, s_k$ of naive sets appear. Those sets can be written:

$$\begin{cases} s_1 &= \{\ \alpha \mid P_1\ \} \\ & \dots \\ s_k &= \{\ \alpha \mid P_k\ \} \end{cases}$$

To interpret the general formula $r \in s$, we know that it is enough that r and s take their value amongst the sets $s_1, \dots, s_k$. That is we can restrict the quantification domain to those sets. Hence $r \in s$ will always be of the form $s_i \in s_j$ with $i,j \in [1,k]$, *i.e.* $s_i \in \{\ \alpha \mid P_j\ \}$. What we want is the denotation of this formula to be logically equivalent to the denotation of $P_j(s_i)$. This poses a problem of circularity, as witnessed by the following example, where $s_i \overset{\mathsf{def}}{=} \{\ \alpha \mid \alpha \in \alpha \multimap \bot\ \}$:

$$s_i \in s_i \ \multimap\!\!\circ\ (s_i \in s_i) \multimap \bot$$

Let's note the interpretation, still to be defined, of $s_i \in s_j$ by the variable $X_{s_i s_j}$. If ρ is a $\mathcal{B}$-model, then it is easy to see that the interpretation $\|P_i\|_{\mathcal{B},2,\rho}$ of each type P_i is uniquely determined by the $X_{s_i s_j}$. For example the type $P = \alpha \in \beta \otimes \beta \in \gamma$ induces the following interpretation:

$$X_{\rho(\alpha)\rho(\beta)} \otimes X_{\rho(\beta)\rho(\gamma)}$$

The solution is to consider the following finite set of recursive equations:

$$\begin{cases} X_{s_1 s_1} &\approx \triangleright P_1[s_1/\alpha] & (E_{11}) \\ X_{s_1 s_2} &\approx \triangleright P_2[s_1/\alpha] & (E_{12}) \\ & \dots & \\ X_{s_i s_j} &\approx \triangleright P_j[s_i/\alpha] & (E_{ij}) \\ & \dots & \\ X_{s_k s_k} &\approx \triangleright P_k[s_k/\alpha] & (E_{kk}) \end{cases}$$

This system of recursive equation can be solved since all maps are contractive and there is a finite number of equations, by Theorem 237. Hence we define the interpretation of $s_i \in s_j$ as the solution of (E_{ij}):

$$\|s \in r\|_{\mathcal{B},2,\rho} \stackrel{\text{def}}{=} X_{[\![s]\!]_\rho [\![r]\!]_\rho}$$

Lemma 275. *The following rules are $\mathcal{B}$-sound:*

$$\frac{\mathcal{E};\Gamma \vdash_2 v : (P[r/\alpha], p)}{\mathcal{E};\Gamma \vdash_2 v : (r \in \{\ \alpha \mid P\ \}, p)}$$

$$\frac{\mathcal{E};\Gamma \vdash_2 v : (r \in \{\ \alpha \mid P\ \}, p) \qquad \mathcal{E};\Delta, x : P[r/\alpha] \vdash_2 t : (N, q)}{\mathcal{E};\Gamma,\Delta \vdash_2 \mathsf{ret}(v)\ \mathsf{to}\ x.(\!|t|\!)_2^{\alpha_{\mathsf{step}}} : (N, p+q)}$$

Proof. By Theorem 265. □

7.1.3 Consistency result

We have a sound *partial* interpretation of linear naive set theory in $\mathcal{A}_3$: it is parametrized over a finite set of the naive sets that can appear in the course of a typing derivation. We now want to prove that it implies the consistency of this theory. The main argument will be the termination of all well-typed programs. However, since we use a similar constructions to the unrestricted recursive types and we introduce the α_{step} monitor, we need a special argument to justify that we don't loose termination.

We begin by annotating the typable programs using the two monitors α_{step} and α_{time}. In fact, we annotate the typing derivation of a programs. We annotate only the typing rule that concerns the unfolding of the naive comprehension scheme, using the two observations $(\!|.|\!)_2^{\alpha_{\mathsf{step}}}$ and $(\!|.|\!)_1^{\alpha_{\mathsf{time}}}$. This corresponds to a proof translation:

$$\pi \rightsquigarrow \pi^\bullet$$

that is the identity everywhere except for the following rule (we abusively identify $\pi^\bullet$ with the underlying program $t^\bullet$ or $v^\bullet$):

$$\frac{\mathcal{E};\Gamma \vdash v : r \in \{\ x \mid P\ \} \qquad \mathcal{E};\Delta, x : P[r/x] \vdash t : N}{\mathcal{E};\Gamma,\Delta \vdash \mathsf{ret}(v)\ \mathsf{to}\ x.t}$$

$$\rightsquigarrow$$

$$\frac{\mathcal{E};\Gamma \vdash_2 v^\bullet : (r \in \{\ x \mid P\ \}, p) \qquad \mathcal{E};\Delta, x : P[r/x] \vdash_2 t^\bullet : (N, q)}{\mathcal{E};\Gamma,\Delta \vdash_2 \mathsf{ret}(v^\bullet)\ \mathsf{to}\ x.(\!|(\!|t^\bullet|\!)_2^{\alpha_{\mathsf{step}}}|\!)_1^{\alpha_{\mathsf{time}}} : (N, p+q+(1,0))}$$

Basically, the argument is now that each time the monitor α_{step} is triggered, the monitor α_{time} has to be triggered first. Suppose that the two memory cells are initialized with a same value $\underline{\mathsf{n}}$. It means we launch a configuration of the form

$$\langle t^{\bullet}, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{a}(\underline{\mathsf{n}}).\mathsf{nil}\rangle_2$$

In that case, the daimon ✠ cannot be triggered. Indeed, in the case ✠ is triggered, it means that α_{step} has been called while the value of the (second) counter is 0: but that would mean α_{time} is called with the value of the corresponding (first) counter being 0, which would trigger the execution of Ω and that is absurd (because it would prevent ✠ from being triggered). Since it cannot be triggered, it means that when observing the termination of the translated program, we are sure that the original program also terminates.

We now can give a proof of Theorem 274. That is, there is no term typable by $\bot$.

Proof. Reasoning by absurd, suppose that we have $\vdash_0 t : \bot$. Then we know that $(t, (n, \infty)) \in \|\bot\|_{\mathcal{B},2,[]}$. Hence $(t, (n, n)) \in \|\bot\|_{\mathcal{B},2,[]}$. Let $E = \mathsf{f}(x.\Omega).\mathsf{nil}$. We know that $(E, (0, 0)) \in [\![\bot]\!]_{\mathcal{B},2,[]}$ and $(\mathsf{nil}, (0, 0)) \in [\![\bot]\!]_{\mathcal{B},2,[]}$. Therefore we have

$$C_0 \stackrel{\mathsf{def}}{=} \langle t, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{a}(\underline{\mathsf{n}}).\mathsf{nil}\rangle_2 \in \perp\!\!\!\perp$$

and

$$C_1 \stackrel{\mathsf{def}}{=} \langle t, \mathsf{a}(\underline{\mathsf{n}}).\mathsf{a}(\underline{\mathsf{n}}).E\rangle_2 \in \perp\!\!\!\perp$$

There are two possibilities:

- If the configuration C_0 terminates on a configuration of the form

  $$\langle \mathsf{ret}(v), \mathsf{a}(\underline{\mathsf{k}}).\mathsf{a}(\underline{\mathsf{k}'}).\mathsf{nil}\rangle_2$$

  Then necessarily,

  $$C_1 \to^* \langle \Omega, \mathsf{a}(\underline{\mathsf{k}}).\mathsf{a}(\underline{\mathsf{k}'}).\mathsf{nil}\rangle_2 \text{ that diverges}$$

  But we know that C_1 terminates, which is absurd.

- If C_0 terminates on ✠, then it means that it has been triggered by the monitor α_{step}. But to trigger ✠, it means that the monitor is called after the second counter has reached $\underline{0}$. This is absurd because before each call to α_{step} in t^* there is a call to α_{time}. Since the counters are initialized with the same value $\underline{\mathsf{n}}$, if one counter is equal to $\underline{0}$ then the next call to a monitor will make the reduction diverge, because of α_{time}. Hence ✠ is never called.

□

7.1.4 Considerations

- We have given a semantical proof that the linear naive set theory is consistent. This theory is extremely weak since it does not allow any kind of sharing. However, notice that the construction and the proof of Theorem 274 do *not* depend of the choice of the quantitative monitoring algebra $\mathcal{A}$, as soon as the element $\mathbf{1}$ is such that $\|\mathbf{1}\| \in \mathbb{N}$. It means that we can choose another more expressive quantitative 1-**MA**. By considering:

  - The elementary quantitative monoid $\mathcal{M}_{\mathsf{elem}}$ described in Example 203 and Example 215, we then can interpret elementary naive set theory, based on elementary linear logic [Gir98].
  - The light quantitative monoid described in [BM12] allows to interpret Light Affine Set Theory [Ter04].
  - We can in fact revisit all quantitative monoids described in the various papers [Bru13, BM12, DLH11].

  This gives an interesting point of view: we can explore the possibilities offered by the semantics (here the notion of quantitative monoid) to look for new systems (here consistent naive set theories).

- It is known that any naive set theory based on linear logic without any restriction is unsound. In fact, it is the same as using unrestricted recursive types with linear logic: this yields divergence. This phenomenon can be reread in our framework in the form of the following algebraic result:

  Fact 276. *There exists no quantitative monoid $\mathcal{M}$ with an exponential modality in the sense of Definition 202 such that for all $p \in \mathcal{M}$, $\|p\| < \infty$.*

  Indeed, otherwise we could prove the consistency of naive set theory. Of course this result can be proved directly.

7.2 | Polynomial-time programming language with recursive type

In this section, we consider a more concrete example: a programming language with unrestricted recursive types based on a substructural logic called soft affine logic (**SAL**) [BM04, Laf04]. It means that the usual exponential of linear logic ! is weakened: the contraction rule is restricted. As a result this type system ensures that all typable programs enjoy a polynomial-time termination property. This example is the occasion to showcase the methodology of monitoring algebras: we detail each step of the correctness proof of a concrete language. It is basically as follows:

- We define the syntax of a call-by-value language with an explicit modality and its type system based on **SAL**.
- We build step by step a monitoring algebra that integrates the features required to show the correctness of the language: a quantitative 1-**MA**, step-indexing to deal with recursive types, the addition of strong modalities to reduce under !. We then show a soundness theorem for this monitoring algebra, reusing many of the theorems and properties proved in this thesis.

- We finally define a translation of the language into the call-by-push-value, and show that the soundness of the monitoring algebra defined imply the polynomial-time termination property.

This language is inspired by [BM12].

7.2.1 The language

Syntax

We define the syntax of the language alongside its reduction. It constitutes an extension of the core call-by-value calculus of Subsection 2.2.3. In addition to the base constructors, we have an explicit modality !, and the constructors unfold and fold for the recursive types.

$$\begin{array}{rrcl} \textbf{Values} & v, w & ::= & x \mid \lambda x.t \mid * \mid \underline{\mathsf{n}} \mid !v \mid \mathsf{fold}\,(v) \\ \textbf{Terms} & t, u & ::= & v \mid \underline{\mathsf{s}}(t) \mid (t)u \mid !t \mid \mathsf{let}\,!x = t\,\mathsf{in}\,u \mid \mathsf{fold}\,(t) \mid \mathsf{unfold}\,(t) \end{array}$$

The language is executed with a call-by-value strategy, in an abstract machine based on the machine presented in Section 2.2. Hence we use the same notations when possible. We define the syntax of **environments**, which is the same as for the core language of Subsection 2.2.3 but augmented with a special context $!.E$ which represents a !-box.

$$\begin{array}{rrcl} \textbf{Environments} & E & ::= & \mathsf{nil} \mid \mathsf{a}(v).E \mid \mathsf{f}(t).E \mid !.E \mid \mathsf{fold}\,.E \mid \mathsf{unfold}\,.E \\ \textbf{Configurations} & C & ::= & \langle t, E\rangle \end{array}$$

Finally, the reduction relation is extended as follows:

$$\begin{array}{rcl} \langle !(t), E\rangle & \to_{\mathsf{V}} & \langle t, !.E\rangle \\ \langle v, !.E\rangle & \to_{\mathsf{V}} & \langle !v, E\rangle \\ \langle \mathsf{unfold}\,(t), E\rangle & \to_{\mathsf{V}} & \langle t, \mathsf{unfold}\,.E\rangle \\ \langle \mathsf{fold}\,(v), \mathsf{unfold}\,.E\rangle & \to_{\mathsf{V}} & \langle v, E\rangle \end{array}$$

Notice that ! is a call-by-value modality in the sense of Section 6.1. This is very important as it is crucial for its expressivity [BM04, Laf04], unlike linear logic that is fully expressive without reducing under !.

Soft type system

We now define the accompanying type system. It is, like the syntax, an extension of the type system presented in Section 2.2. In addition, we add type variables and (unrestricted) recursive types.

$$\text{Types} \quad A, B \quad ::= \quad \mathsf{Nat} \mid X \mid A \multimap B \mid A \otimes B \mid !A \mid \mu X.A$$

The typing relation is denoted $\vdash_{\mathsf{SAL}}$ and the complete set of rules is the following:

$$\frac{}{\Gamma, x : A \vdash_{\mathsf{SAL}} x : A} \qquad \frac{}{\Gamma \vdash_{\mathsf{SAL}} \underline{0} : \mathsf{Nat}} \qquad \frac{\Gamma \vdash_{\mathsf{SAL}} t : \mathsf{Nat}}{\Gamma \vdash_{\mathsf{SAL}} \underline{\mathsf{s}}(t) : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash_{\mathsf{SAL}} t : \mathsf{Nat} \qquad \Delta, x : \mathsf{Nat} \vdash_{\mathsf{SAL}} u_1 : A \qquad \Delta, x : \mathsf{Nat} \vdash_{\mathsf{SAL}} u_2 : A}{\Gamma, \Delta \vdash_{\mathsf{SAL}} \mathsf{case}\, t \,\mathsf{of}\, x.u_1 \parallel x.u_2 : A}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{SAL}} t : B}{\Gamma \vdash_{\mathsf{SAL}} \lambda x.t : A \multimap B} \qquad \frac{\Gamma \vdash_{\mathsf{SAL}} t : A \multimap B \qquad \Delta \vdash_{\mathsf{SAL}} u : A}{\Gamma, \Delta \vdash_{\mathsf{SAL}} (t)u : B}$$

$$\frac{\Gamma \vdash_{\mathsf{SAL}} t : A \qquad \Delta \vdash_{\mathsf{SAL}} u : B}{\Gamma, \Delta \vdash_{\mathsf{SAL}} (t, u) : A \otimes B} \qquad \frac{\Gamma \vdash_{\mathsf{SAL}} t : A \otimes B \qquad \Delta, x : A, y : B \vdash_{\mathsf{SAL}} u : C}{\Gamma, \Delta \vdash_{\mathsf{SAL}} \mathsf{let}\,(x, y) \;=\; t \,\mathsf{in}\, u : C}$$

$$\frac{\Gamma, x_1 : A, \ldots, x : A \vdash_{\mathsf{SAL}} t : B}{\Gamma, x : !A \vdash_{\mathsf{SAL}} t[x/x_1, \ldots, x/x_n] : B}\ \mathsf{Mplex}_n \qquad \frac{\Gamma \vdash_{\mathsf{SAL}} t : B}{!\Gamma \vdash_{\mathsf{SAL}} !t : !B}\ \mathsf{Prom}$$

$$\frac{\Gamma \vdash_{\mathsf{SAL}} t : !A \qquad \Delta, x : !A \vdash_{\mathsf{SAL}} u : B}{\Gamma, \Delta \vdash_{\mathsf{SAL}} \mathsf{let}\, !x \;=\; t \,\mathsf{in}\, u : B} \qquad \frac{\Gamma \vdash_{\mathsf{SAL}} t : \mu X.A}{\Gamma \vdash_{\mathsf{SAL}} \mathsf{unfold}\,(t) : A[\mu X.A/X]}$$

$$\frac{\Gamma \vdash_{\mathsf{SAL}} t : A[\mu X.A/X]}{\Gamma \vdash_{\mathsf{SAL}} \mathsf{fold}\,(t) : \mu X.A}$$

Here are some remarks on the typing rules.

- The recursive types are unrestricted, in the sense that they are not guarded.
- The promotion rule of ! is the usual functorial promotion.
- The contraction rule is not quite the same as in linear logic. It takes the form of a scheme of rules, called the **multiplex** rules:

  $$!A \multimap \underbrace{A \otimes \cdots \otimes A}_{n \text{ times}}$$

  Whereas the linear logic contraction is:

  $$!A \multimap !A \otimes !A$$

Definition 277 (!-depth). *If π is a typing derivation, its* ***!-depth*** *is the maximum number of nested promotion rule* (Prom) *in π.*

We intend to prove the following polynomial-time termination theorem. It says that there exists a family of polynomials of each degree, such that for each typable program, a bound on its execution time is given by the application of the polynomial whose degree is the !-depth of its derivation to the size of this program.

Theorem 278 (Polynomial-time termination). *There exists a family of polynomials $(P_d)_{d\in\mathbb{N}}$ such that P_d is of degree d, such that if $\vdash_{\mathsf{SAL}} t : A$ is proved with a !-depth equals to n, then $\langle t, \mathsf{nil}\rangle$ terminates on a value using at most $P_n(|t|_\lambda + |t|_{\mathsf{unfold}})$ λ and* unfold *reduction steps.*

7.2.2 Monitoring algebra

We now build step-by-step a n-monitoring algebra that induces a realizability model of this language (or more precisely of the translation into the call-by-push-value of this language). As a corollary of the soundness of the realizability model, we obtain a proof of the polynomial time reduction theorem. The different steps are the following ones:

1. Define a quantitative 1-**MA** $\mathcal{A}_1$ that allows to observe bounded-time termination (as shown in Section 6.2) and in which the typing rule of a restricted functional fragment, where promotion is restricted to values.

2. Based on $\mathcal{A}_1$, we define a 2-**MA** $\mathcal{A}_2$ in which we apply the modality completion procedure described in Section 6.1, allowing us to obtain the soundness of the unrestricted promotion for !.

3. We then use a similar construction to the one described in Section 7.1 for the linear naive set theory: we use the product with the step-indexing **MA** to obtain a step-indexed 3-**MA** $\mathcal{A}_3$, and interpret unrestricted recursive types.

Step 0: soft forcing monoid

The first step is to define a class of forcing monoids such that every k-**MA** whose carrier belongs to this class induces a model of the functional fragment of the previously defined typing rules. We will then use this to check that at each step of our construction the k-**MA** we have obtained still induces a model of the core of our language.

Definition 279 (Soft forcing monoid).
A ***soft forcing monoid*** *is a structure $(\mathcal{M}, +, \mathbf{0}, \bullet, \leq, !, (r_n)_{n\in\mathbb{N}})$ such that:*

- *$(\mathcal{M}, +, \mathbf{0}, \bullet, \leq)$ is a forcing monoid.*

- $(r_n)_{n\in\mathbb{N}}$ *is a family of elements of* $\mathcal{M}$ *indexed by* $\mathbb{N}$.
- $! : \mathcal{M} \to \mathcal{M}$ *is a sub-additive function that satisfies the following* ***multiplex condition****:*

$$\forall p \in \mathcal{M}, n.p \preceq !p + r_n$$

We say that a k*-**MA** is **soft** if its carrier is.*

Example 280. *Any idempotent **MA** is also soft, by choosing* $!p \overset{\text{def}}{=} p$ *and* $r_n \overset{\text{def}}{=} \mathbf{0}$.

Property 281. *If* $\mathcal{M}$ *and* $\mathcal{N}$ *are two soft forcing structures, then so is* $\mathcal{M}\times\mathcal{N}$ *(by considering the product of the two modalities* $!$*, and the product of the two families* r_n*).*

Proof. Immediate. □

This class of monitoring algebras enjoys an extended soundness result. In addition to the linear core of the type system, new rules concerning ! are sound. These new rules allows to have some restricted form of duplication.

Theorem 282 (Soft soundness)**.** *If* $\mathcal{A}$ *is a soft* k*-**MA**, then the following modality typing rules are sound:*

$$\frac{\mathcal{E};\Gamma \vdash_k v : (P,p)}{\mathcal{E};!\Gamma \vdash_k !v : (!P,!p)} \qquad \frac{\mathcal{E};\Gamma, x_1 : P, \ldots, x_n : P \vdash_k t : (N,p)}{\mathcal{E};\Gamma, z : !P \vdash_k t[z/x_1,\ldots,z/x_n] : (N, p + r_n)}$$

Proof.

1. The first rule is simply the usual promotion rule, which is sound because of Property 190.
2. Let ρ be a $\mathcal{A}$-valuation, $\sigma \in \|\Gamma\|_{\mathcal{A},k,\rho}$ and $(v,q) \in \|!P\|_{\mathcal{A},k,\rho}$. We know there exists q' such that $!q' \preceq q$ and $(v,q') \in \|P\|_{\mathcal{A},k,\rho}$. We know by hypothesis that

$$(t[v/x_1,\ldots,v/x_n], p + \underbrace{q' + \cdots + q'}_{n \text{ times}})[\sigma] \in \|N\|_{\mathcal{A},k,\rho}$$

   We have $n.q' \preceq !q' + r_n \preceq q + r_n$ hence

$$(t[v/x, v/y], p + q + r_n)[\sigma] \in \|N\|_{\mathcal{A},k,\rho}$$

□

Step 1: Polynomial-time termination

The starting point of the construction is the **soft bounded-time MA**. It is an example of quantitative **MA**, in the sense of Section 6.2, whose carrier is a soft forcing monoid. We begin by defining the carrier of this algebra.

Definition 283. *$\mathcal{M}_{\mathsf{poly}}$ is the commutative forcing monoid such that:*

- *The set of pairs $(n, f) \in \mathbb{N} \times \mathbb{N}^{\mathbb{N}}$ where f is an $\leq$-preserving function.*
- $\mathbf{0} \stackrel{\mathsf{def}}{=} (0, x \mapsto 0)$
- $(n, f) + (m, g) \stackrel{\mathsf{def}}{=} (\max(n, m), f + g)$
- $(n, f) \preceq (m, g)$ *iff:*
  - $n \leq m$
  - $\forall x \geq m, f(x) \leq g(x)$

Proof.

- $+$ is clearly an associative and commutative operation with $\mathbf{0}$ being its neutral.
- We prove that $\preceq$ is compatible with $+$:
  - It is clear that $\mathbf{0} \preceq (n, f)$ for any (n, f).
  - Suppose that $(n, f) \preceq (m, g)$. So $(n, f) + (k, h) = (\max(n, k), f + h)$ and $(m, g) + (k, h) = (\max(m, k), g + h)$. Clearly since $n \leq m$ we have $\max(n, k) \leq \max(m, k)$. Let $x \geq \max(m, k) \geq m$. Then we have $(f + h)(x) = f(x) + h(x) \leq g(x) + h(x) = (g + h)(x)$ since $(n, f) \preceq (m, g)$.

□

Property 284. *We endow $\mathcal{M}_{\mathsf{poly}}$ with:*

1. *A structure of soft forcing monoid, by choosing:*
   - $!(n, f) \stackrel{\mathsf{def}}{=} (n, x \mapsto (x + 1) f(x))$
   - $r_n \stackrel{\mathsf{def}}{=} (n, x \mapsto 0)$
2. *A structure of quantitative forcing monoid, by defining:*
   - $\|(n, f)\| \stackrel{\mathsf{def}}{=} f(n)$
   - $\mathbf{1} \stackrel{\mathsf{def}}{=} (0, x \mapsto 1)$

Proof.

1. We need to prove that ! is a modality and that it satisfies the multiplex condition.

- First, we want to prove that $!((n,f)+(m,g)) \preceq !(n,f)+!(m,g)$. It is immediate since:

$$\begin{array}{rcl} !((n,f)+(m,g)) & = & !(\max(n,m), f+g) \\ & = & (\max(n,m), x \mapsto (x+1)(f+g)(x)) \\ & = & (\max(n,m), x \mapsto (x+1)f(x) + (x+1)g(x)) \\ & = & !(n,f)+!(m,g) \end{array}$$

- We now want to prove that $n.(k,f) \preceq !(k,f) + r_n$. We have $n.(k,f) = (\max(k,n), n.f)$ on the one hand. On the other hand, $!(k,f) + r_n = (\max(k,n), x \mapsto (x+1).f(x))$. Let $x \geq \max(k,n)$. Then $n.f(x) \leq (\max(k,n)+1)f(x) \leq (x+1)f(x)$. Hence the result.

2. First, since $\mathcal{M}_{\mathsf{poly}}$ is additive, $x \mapsto \mathbf{1} \bullet x$ is immediately strong by Property 80.

- We prove that $\|.\|$ is sup-additive.

$$\begin{array}{rcl} \|(n,f)\| + \|(k,g)\| & = & n.f(n) + k.g(k) \\ & \leq & \max(n,k).f(n) + \max(n,k).g(k) \\ & \leq & \max(n,k).f(\max(n,k)) + \max(n,k).g(\max(n,k)) \\ & = & \max(n,k).(f+g)(\max(n,k)) \\ & = & \|(\max(n,k), f+g)\| \\ & = & \|(n,f)+(k,g)\| \end{array}$$

- We prove that $\|.\|$ is increasing. Suppose that $(n,f) \preceq (m,g)$. Then since $n \leq m$ and f is increasing, $n.f(n) \leq m.f(n) \leq m.f(m)$. But $f(m) \leq g(m)$ so $m.f(m) \leq m.g(m)$.
- We have $\|(0, x \mapsto 1)\| = 1$.

□

Definition 285 (Soft bounded-time **MA**). *We define the soft bounded-time **MA** $\mathcal{A}_1$ as the quantitative* 1*-**MA** induced by the quantitative forcing monoid $\mathcal{M}_{\mathsf{poly}}$.*

Lemma 286. ! *satisfies additional properties:*

- *For all $p \in |\mathcal{A}_1|$, $p \preceq_{|\mathcal{A}_1|} !p$.*
- *For all $p, q \in |\mathcal{A}_1|$, $!(p+q) = !p + !q$.*

Proof. Immediate. □

We remind the reader that by Property 216 the pair $(\alpha_{\mathsf{time}}, x \mapsto x + \mathbf{1})$ is a $\mathcal{A}_1$-monitor, where:

$$\alpha_{\mathsf{time}} = \lambda x.\mathsf{case}\, x\, \mathsf{of}\, x.\Omega \parallel x.\mathsf{ret}(x)$$

Step 2: Reducing under the modality

The connective ! is a $\mathcal{A}_1$*-modality*. It means that it is *restricted to value*, or in other words it is not allowed for the reduction to happen under !. We remind that in order to do that, we need

! to have a right adjoint: this is impossible in $\mathcal{A}_1$. To obtain an adjoint modality, we apply the construction described in Section 6.1, which automatically adds an adjoint to !. We define the following 2-**MA**:

$$\mathcal{A}_2 \stackrel{\text{def}}{=} \mathcal{A}_1^{\{!\}} = \mathcal{A}_1 \ltimes_! \mathbb{N}_{\text{adj}}$$

Proposition 287.

1. *$\mathcal{A}_2$ is soft.*
2. *! is an adjoint $\mathcal{A}_2$-modality.*
3. *The $\mathcal{A}_1$-monitor $(\alpha_{\text{time}}, x \mapsto x + \mathbf{1})$ is preserved.*

Proof.

1. $\mathbb{N}_{\text{adj}}$ is soft when considering the modality $n \mapsto n$ because it is idempotent, as remarked in Example 280. Hence $\mathcal{A}_2$ is also soft as a semi-direct product of two soft **MA**s and by Property 281.
2. The fact that the new $\mathcal{A}_2$-modality ! is an adjoint modality is a direct consequence of Property 210.
3. To show that the $\mathcal{A}_1$-monitor is preserved, we use Property 188. It requires the strong function $x \mapsto x + \mathbf{1}$ to weakly commute with the modality !. Let $(n, f) \in |\mathcal{A}_1|$, we have $(!(n, f)) + \mathbf{1} = (n, x \mapsto (x+1)f(x) + 1) \preceq (n, x \mapsto (x+1)(f(x) + 1)) = !((n, f) + \mathbf{1})$.

□

Step 3: Linear type fixed-points

We now have enough to interpret the functional core of our type system, including the unrestricted modality !. The third step consists in adding recursive types. We define the 3-**MA**

$$\mathcal{A}_3 \stackrel{\text{def}}{=} \mathcal{A}_{\text{step}} \times \mathcal{A}_2$$

Proposition 288.

1. *$\mathcal{A}_3$ is soft.*
2. *! is an adjoint $\mathcal{A}_3$-modality.*
3. *$\mathcal{A}_3$ is step-indexed and has a next modality $\triangleright$.*
4. *The canonical extension of ! to $\mathcal{A}_3$ is non-expansive.*
5. *The two monitors based on α_{time} and α_{step} are preserved.*

Proof.

1. Since $\mathcal{A}_{\mathsf{step}}$ is idempotent, it is also soft. But the product of two soft forcing monoids is itself soft by Property 281, hence the result.
2. Adjoint modalities are preserved by direct product of **MA**s by Theorem 198.
3. The product of any algebra with a step-indexed algebra is itself step-indexed by Theorem 261. It moreover has a next modality because $\mathcal{A}_{\mathsf{step}}$ has and by Property 263.
4. Since ! comes from a simple $\mathcal{A}_2$-connective, its canonical extension is non-expansive by Property 262.
5. Immediate by Property 181.

□

Result: Soundness

Consider the following grammar of types:

$$\begin{array}{rcl} P, Q & ::= & \mathsf{Nat} \mid X \mid \mathbb{1} \mid P \otimes Q \mid \Downarrow N \mid !P \mid \triangleright P \mid \mu X.P \\ N, M & ::= & \Uparrow P \mid P \multimap N \end{array}$$

Then as a corollary of Proposition 288 and using the step-indexing theorem 265, the interpretation of all those types is such that in addition to the typing rules of λ_{LCBPV}, the following typing rules are $\mathcal{A}_3$-sound.

$$\frac{\mathcal{E}; \Gamma \vdash_3 t : (\Uparrow P, p)}{\mathcal{E}; !\Gamma \vdash_3 t : (\Uparrow !P, !p)} \qquad \frac{\mathcal{E}; \Gamma, x_1 : P, \ldots, x_n : P \vdash_3 t : (N, p)}{\mathcal{E}; \Gamma, z : !P \vdash_3 t[z/x_1, \ldots, z/x_n] : (N, p + r_n)}$$

$$\frac{\mathcal{E}; \Gamma \vdash_3 t : (N, p)}{\mathcal{E}; \Gamma \vdash_3 (\!|t|\!)_1^{\alpha_{\mathsf{time}}} : (N, p + \mathbf{1})} \qquad \frac{\mathcal{E}; \Gamma \vdash_3 v : (P[\mu X.P/X], p)}{\mathcal{E}; \Gamma \vdash_3 v : (\mu X.P, p)}$$

$$\frac{\mathcal{E}; \Gamma \vdash_1 v : (\mu X.P, p) \qquad \mathcal{E}; \Delta, x : P[\mu X.P/X] \vdash_1 t : (N, q)}{\mathcal{E}; \Gamma, \Delta \vdash_1 \mathsf{ret}(v) \text{ to } x.(\!|t|\!)_1^{\alpha_{\mathsf{step}}} : (N, p + q)}$$

Hence, so is

$$\frac{\mathcal{E}; \Gamma \vdash_1 v : (\mu X.P, p) \qquad \mathcal{E}; \Delta, x : P[\mu X.P/X] \vdash_1 t : (N, q)}{\mathcal{E}; \Gamma, \Delta \vdash_1 (\!|\mathsf{ret}(v) \text{ to } x.(\!|t|\!)_3^{\alpha_{\mathsf{step}}}|\!)_1^{\alpha_{\mathsf{time}}} : (N, p + q + \mathbf{1})}$$

7.2.3 Translation

To obtain a soundness result for the original language, we end this section by providing a translation of our source language to the monitoring calculus. It is a simple extension of the

call-by-value translation $(.)^{\mathsf{V}}$ developped in Section 2.2. The translation is unchanged on the purely linear and functional calculus and extended as follows:

$$\begin{aligned}
(!t)^{\mathsf{V}} &\stackrel{\text{def}}{=} (t)^{\mathsf{V}} \text{ to } x.\mathsf{ret}(x) \\
(\text{let } !x = t \text{ in } u)^{\mathsf{V}} &\stackrel{\text{def}}{=} (t)^{\mathsf{V}} \text{ to } x.(u)^{\mathsf{V}} \\
(\mathsf{fold}\, t)^{\mathsf{V}} &\stackrel{\text{def}}{=} (t)^{\mathsf{V}} \\
(\mathsf{unfold}\, t)^{\mathsf{V}} &\stackrel{\text{def}}{=} (\!|(t)^{\mathsf{V}} \text{ to } x.(\!|\mathsf{ret}(x)|\!)_3^{\alpha_{\mathsf{step}}}|\!)_1^{\alpha_{\mathsf{time}}}
\end{aligned}$$

Theorem 289. *The extended translation preserves reduction and typing.*

Proof. This is an extension of Theorems 55 and 56 and has already been proved in Section 6.1 for the modality and in Section 6.4 for the recursive types. □

Theorem 290 (Polynomial-time termination)**.** *There exists a family of polynomials $(P_d)_{d\in\mathbb{N}}$ such that P_d is of degree d, such that if $\vdash_{\mathsf{SAL}} t : A$ is proved with a !-depth equals to n, then $\langle t, \mathsf{nil}\rangle$ terminates on a value using at most $P_n(|t|_\lambda + |t|_{\mathsf{unfold}})$ λ and* unfold *reduction steps.*

Proof. Suppose that $\vdash_{\mathsf{SAL}} t : A$. Then we have

$$;\vdash_3 \{(t)^{\mathsf{V}}\}^{\alpha_{\mathsf{time}}} : (\Uparrow(A)^{\mathsf{V}}, (p, 0, \infty))$$

Hence, $(\{(t)^{\mathsf{V}}\}^{\alpha_{\mathsf{time}}}, (p, 0, \|p\|)) \in \|(A)^{\mathsf{V}}\|_{\mathcal{A}_3,3,[]}$ by $\preceq$-saturation. Since $(A)^{\mathsf{V}}$ is positive, we immediately have that:

$$(\mathsf{nil}, \mathbf{0}) \in [\![\Uparrow(A)^{\mathsf{V}}]\!]_{\mathcal{A}_3,3,[]}$$

Hence if we pose $k = \|p\|$,

$$C = \langle\{(t)^{\mathsf{V}}\}^{\alpha_{\mathsf{time}}}, \mathsf{a}(\underline{\mathsf{k}}).\mathsf{a}(*).\mathsf{a}(\underline{\mathsf{k}}).\mathsf{nil}\rangle_3 \in \perp\!\!\!\perp$$

- First, as in the proof of Theoreom 274, each time α_{step} has been triggered, it means that the monitor α_{time} has been triggered *before*. Since we put the same value in the two counters, it means that α_{step} is never called in front of a counter that equals $\underline{0}$. Otherwise, α_{time} would have been called before and triggered the execution of Ω, making the whole configuration diverge. Since we know that $C \in \perp\!\!\!\perp$, it means that this situation does not happen. Hence α_{time} and α_{step} are both triggered less than k times, and do not *alter* the convergence of the execution.
- Second, by examining the typing rules, it is easy to see that p is built by adding $\mathbf{1}$ each time there is an observation α_{time}, and by applying ! for each promotion rule encountered. Therefore $\|p\|$ is a polynomial in the size of t, whose degree is less than the maximum number of nested ! promotion rules used.

Finally, using the simulation theorem, these two points ensure that the original configuration $\langle t, \mathsf{nil}\rangle$ terminates on a value, using a number of λ-steps and unfold -steps which is bounded by $\|p\|$, which is itself bounded by a polynomial in the size of t, whose degree is the !-depth of the typing derivation of t. □

7.3 | A linear calculus for strong updates

We study a small language that features a higher-order reference, but has the ability to perform **strong updates**, very much inspired by the work of Ahmed et al. [AFM07]: in fact, modulo the addition of dynamic references, we could translate their core language into this one. Strong updates means that the type of the memory cell can *change* during the execution. This kind of feature is usually not sound, but here the access to the memory cell is restricted thanks to a linear discipline. We show that a simple monitoring algebra can be given by reusing a quantitative 1-**MA**, the step-indexing algebra and a variant of the higher-order references construction of Section 6.5, and a soundness theorem be proved such that it implies the termination of that language.

7.3.1 The language

The language and its type system are given as an extension of $\lambda_{\mathsf{LCBPV}}^{\otimes\mathsf{Nat}\forall}$ called $\lambda_{\mathsf{LCBPV}}^{\mathsf{cap}}$. We first give the syntax of this core language, as an extension of λ_{LCBPV} values and computations.

$$\begin{array}{rrcl} \textbf{Values} & v, w & ::= & \ldots \mid \mathsf{cap} \\ \textbf{Computations} & t, u & ::= & \ldots \mid \mathsf{swap}(v, w) \end{array}$$

- The syntax of values is extended with a **capability** cap, that is used to grant access to the memory cell.
- We extend the computations with a variant of the swap instruction, that requires the use of a capability, hence the additional argument.

We also define a new reduction relation $\to_{\mathsf{Str}}$. It is a relation between configurations of the form

$$\langle t, E \star v \rangle_{\mathsf{Str}}$$

where t and v are terms of the extended grammar we just defined, and E is a usual environment resulting of this extension. The value v represents the content of the memory cell. The reduction relation $\to_{\mathsf{Str}}$ is a simple variant of λ_{LCBPV} reduction, with a rule for the swap constructor.

$$
\begin{array}{rcl}
\langle (t)v, E \star w\rangle_{\mathsf{Str}} & \to_{\mathsf{Str}} & \langle t, \mathsf{a}(v).E \star w\rangle_{\mathsf{Str}} \\
\langle \lambda x.t, v.E \star w\rangle_{\mathsf{Str}} & \to_{\mathsf{Str}} & \langle t[v/x], E \star w\rangle_{\mathsf{Str}} \\
\langle t \text{ to } x.u, E \star w\rangle_{\mathsf{Str}} & \to_{\mathsf{Str}} & \langle t, \mathsf{f}(x.u).E \star w\rangle_{\mathsf{Str}} \\
\langle \mathsf{ret}(v), \mathsf{f}(x.u).E \star w\rangle_{\mathsf{Str}} & \to_{\mathsf{Str}} & \langle u[v/x], E \star w\rangle_{\mathsf{Str}} \\
\langle \mathsf{force}(\mathsf{thunk}(t)), E \star w\rangle_{\mathsf{Str}} & \to_{\mathsf{Str}} & \langle t, E \star w\rangle_{\mathsf{Str}} \\
\langle \mathsf{let}\,(x,y) \;=\; (v,w)\,\mathsf{in}\,t, E \star u\rangle_{\mathsf{Str}} & \to_{\mathsf{Str}} & \langle t[v/x, w/y], E \star u\rangle_{\mathsf{Str}} \\
\langle \mathsf{swap}(\mathsf{cap}, v), E \star w\rangle_{\mathsf{Str}} & \to_{\mathsf{Str}} & \langle \mathsf{ret}((\mathsf{cap}, w)), E \star v\rangle_{\mathsf{Str}}
\end{array}
$$

We also extend the syntax of $\lambda^{\otimes\mathsf{Nat}\forall}_{\mathsf{LCBPV}}$ types as follows:

$$
\begin{array}{rrcl}
\textbf{Types} & P, Q & ::= & \ldots \mid !P \mid \mathsf{Cap}\ P \\
\textbf{Modes} & \kappa, \iota & ::= & \mathsf{lin} \mid \mathsf{exp}
\end{array}
$$

- The types are just an extension of $\lambda^{\otimes\mathsf{Nat}\forall}_{\mathsf{LCBPV}}$ types with the exponential ! of linear logic and **capability types** $\mathsf{Cap}\ P$.

- Finally, the **modes** are used to restrict the application of the ! promotion rule: it is restricted to the exp mode.

Definition 291. *We define the operation $\vee$ on modes as follows:*

$$
\begin{cases}
\mathsf{lin} \vee \kappa &= \mathsf{lin} \\
\kappa \vee \mathsf{lin} &= \mathsf{lin} \\
\mathsf{exp} \vee \mathsf{exp} &= \mathsf{exp}
\end{cases}
$$

The typing judgments are of the form

$$\kappa \mid \Gamma \vdash_{\mathsf{Str}} a : A$$

where κ is a mode and Γ is a usual positive context, while a is a term and A a formula of the same polarity as a. The typing rules given below can be seen as an extension of the core $\lambda^{\otimes\mathsf{Nat}\forall}_{\mathsf{LCBPV}}$ rules, without inequational theory, but with a linear logic exponential and a typing rule for swap. The first component κ of a judgment represents a possible restriction on the application of ! promotion rule.

$$\frac{}{\mathsf{exp} \mid \Gamma, x : P \vdash_{\mathsf{Str}} x : P}$$

$$\frac{\mathsf{exp} \mid !\Gamma \vdash_{\mathsf{Str}} v : P}{\mathsf{exp} \mid !\Gamma \vdash_{\mathsf{Str}} v : !P} \qquad \frac{\kappa \mid \Gamma, x : !P, y : !P \vdash_{\mathsf{Str}} a : A}{\kappa \mid \Gamma, x : !P \vdash_{\mathsf{Str}} a[x/y] : A} \qquad \frac{\kappa \mid \Gamma, x : P \vdash_{\mathsf{Str}} a : A}{\kappa \mid \Gamma, x : !P \vdash_{\mathsf{Str}} a : A}$$

$$\frac{\kappa \mid \Gamma \vdash_{\mathsf{Str}} v : P}{\kappa \mid \Gamma \vdash_{\mathsf{Str}} \mathsf{ret}(v) : \Uparrow P} \qquad \frac{\kappa \mid \Gamma \vdash_{\mathsf{Str}} t : \Uparrow P \qquad \iota \mid \Delta, x : P \vdash_{\mathsf{Str}} u : M}{\kappa \vee \iota \mid \Gamma, \Delta \vdash_{\mathsf{Str}} t \ \mathsf{to}\ x.u : M}$$

$$\frac{\kappa \mid \Gamma \vdash_{\mathsf{Str}} t : N}{\kappa \mid \Gamma \vdash_{\mathsf{Str}} \mathsf{thunk}(t) : \Downarrow N} \qquad \frac{\kappa \mid \Gamma \vdash_{\mathsf{Str}} \Gamma v : \Downarrow N}{\kappa \mid \Gamma \vdash_{\mathsf{Str}} \mathsf{force}(v) : N}$$

$$\frac{}{\mathsf{exp} \mid \Gamma \vdash_{\mathsf{Str}} * : \mathbb{1}} \qquad \frac{\kappa \mid \Gamma \vdash_{\mathsf{Str}} v : \mathbb{1} \qquad \iota \mid \Delta \vdash_{\mathsf{Str}} t : N}{\kappa \vee \iota \mid \Gamma, \Delta \vdash_{\mathsf{Str}} \mathsf{let} * = v \,\mathsf{in}\, t : N} \qquad \frac{\kappa \mid \Gamma \vdash_{\mathsf{Str}} t : N}{\kappa \mid \Gamma \vdash_{\mathsf{Str}} \lambda x.t : P \multimap N}$$

$$\frac{\kappa \mid \Gamma \vdash_{\mathsf{Str}} t : P \multimap N \qquad \iota \mid \Delta \vdash_{\mathsf{Str}} v : P}{\kappa \vee \iota \mid \Gamma, \Delta \vdash_{\mathsf{Str}} (t)v : N}$$

$$\frac{\kappa \mid \Gamma \vdash_{\mathsf{Str}} v : P \qquad \iota \mid \Delta \vdash_{\mathsf{Str}} w : Q}{\kappa \vee \iota \mid \Gamma, \Delta \vdash_{\mathsf{Str}} (v, w) : P \otimes Q} \qquad \frac{\kappa \mid \Gamma \vdash_{\mathsf{Str}} v : P \otimes Q \qquad \iota \mid \Delta, x : P, y : Q \vdash_{\mathsf{Str}} t : N}{\kappa \vee \iota \mid \Gamma, \Delta \vdash_{\mathsf{Str}} \mathsf{let}\,(x, y) = v \,\mathsf{in}\, t : N}$$

$$\frac{\kappa \mid \Gamma \vdash_{\mathsf{Str}} v : \mathsf{Cap}\ P \qquad \iota \mid \Delta \vdash_{\mathsf{Str}} w : Q}{\mathsf{lin} \mid \Gamma, \Delta \vdash_{\mathsf{Str}} \mathsf{swap}(v, w) : \Uparrow(\mathsf{Cap}\ Q \otimes P)}$$

7.3.2 Monitoring algebra

We define a 3-**MA** $\mathcal{A}_3$ that interprets this core language. More precisely we will show that this language can be translated into a fragment of the language interpreted by $\mathcal{A}_3$. The different factors of $\mathcal{A}_3$ are the following ones:

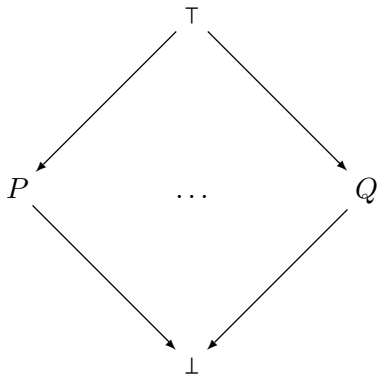
1. A quantitative 1-**MA** $\mathcal{A}_1$ that allows to observe bounded-time termination (as shown in Section 6.2). where promotion is restricted to values.

2. The step-indexing algebra $\mathcal{A}_{\mathsf{step}}$ of Section 6.4.

3. We then define a $(\mathcal{A}_1 \times \mathcal{A}_{\mathsf{step}})$-**MA** $\mathcal{B}$ that is similar to the higher-order reference algebra used in Section 6.5, except that the forcing monoid is no more a singleton. Instead its elements represent the type of the content of the memory cell, reflecting the presence of strong updates that can change its type. We then use a simple iteration similar to 6.5 to build $\mathcal{A}_3 \stackrel{\mathsf{def}}{=} (\mathcal{A}_1 \times \mathcal{A}_{\mathsf{step}}) \lhd \mathcal{B}$.

The monitoring algebra

1. $\mathcal{A}_1$ is the quantitative 1-monitoring algebra based on $\overline{\mathbb{N}}$ and already described in Example 215.
   - $(\overline{\mathbb{N}}, +, 0, \leq)$
   - $\mathcal{C}_{\mathcal{A}_0}(n) \stackrel{\text{def}}{=} \{\ \underline{k} \mid k \geq n\ \}$
2. We then form the product $\mathcal{A}_2 = \mathcal{A}_1 \times \mathcal{A}_{\mathsf{step}}$. $\mathcal{A}_2$ is step-indexed by Theorem 261 and has a next modality $\triangleright$ defined as
$$\triangleright(n, l) \stackrel{\text{def}}{=} (n, l+1)$$
3. The final monitoring algebra is a variation on the higher-order reference construction proposed in Section 6.5.
   - The carrier of $\mathcal{B}$ is the additive forcing monoid whose elements consist in all positive types P generated by the grammar of $\lambda^{\mathsf{cap}}_{\mathsf{LCBPV}}$, augmented with two symbols $\bot$ and $\top$. This set can be endowed with the following structure of complete lattice as follows:



   For the corresponding preorder $\sqsubseteq$, $\bot$ is the least element and $\top$ is the greatest element. The forcing monoid is then the lattice, the $+$ operation being the join $\vee$, the neutral element being $\bot$ and the preorder being $\sqsubseteq$.
   - If P is a positive type, the value of the relativized test function $\mathcal{C}_{\mathcal{B}}(P)$ is defined as the fixed-point of the following function F:
$$F_P : \begin{array}{rcl} \mathbb{V}_{\mathcal{A}_2}^{|\mathcal{B}|} & \to & \mathbb{V}_{\mathcal{A}_2}^{|\mathcal{B}|} \\ X & \mapsto & y \mapsto \|(P)^*(x)\|_{\mathcal{A}_0, n+1, [x \leftarrow y, \mathsf{C} \leftarrow X]} \end{array}$$
   We can then define $\mathcal{C}_{\mathcal{B}}$ as follows:
$$\mathcal{C}_{\mathcal{B}} \stackrel{\text{def}}{=} X \mapsto \begin{cases} (\mu F_P)(\bot) & \text{if } X = P \\ \varnothing & \text{if } X = \top \\ \mathbb{V}_{\mathcal{A}_1} & \text{if } X = \bot \end{cases}$$

There is one subtelty that has to be addressed in this definition. We define the test function using the forcing interpretation of every positive type P generated by the grammar of $\lambda^{\mathsf{cap}}_{\mathsf{LCBPV}}$. But we have not defined the interpretation of the type $\mathsf{Cap}\ P$ nor its forcing interpretation. We define its interpretation as the upward closure of the following singleton:

$$\mathsf{Cap}\ P \stackrel{\mathsf{def}}{=} \{(\mathsf{cap}, (0, \infty, P))\}$$

As already said in Section 4.3, every singleton type can be seen as the composition of a connective and the type $\mathbf{1}$, so we can abusively speak of $\mathsf{Cap}\ P$ as a connective. This is in particular a separable connective on $\mathcal{A}_2$ and $\mathcal{B}$, as defined in Section 5.2. Notice that this definition does not need $\mathcal{B}$ to be defined, but only its carrier, hence the definition of the test function $\mathcal{C}_{\mathcal{B}}$ still makes sense.

Property 292. *$\mathcal{B}$ is a $\mathcal{A}_2$-**MA**.*

We can then form the simple iteration :

$$\mathcal{A}_3 \stackrel{\mathsf{def}}{=} \mathcal{A}_2 \lhd \mathcal{B}$$

Properties of $\mathcal{A}_3$

We now prove important properties of $\mathcal{A}_3$. The first one shows that we can interpret the linear logic exponential !. Indeed we can define an exponential $\mathcal{A}_3$-modality in the sense of Definition 202. It is defined as the product of exponentials on the different factors of $\mathcal{A}_3$. The exponential on $\overline{\mathbb{N}}$ is the same as in Example 203:

$$! \stackrel{\mathsf{def}}{=} n \mapsto \begin{cases} 0 & \text{if } n = 0 \\ \infty & \text{otherwise} \end{cases}$$

On $\mathcal{A}_{\mathsf{step}}$, it is simply the identity. On $\mathcal{B}$, it is defined similarly as on $\overline{\mathbb{N}}$:

$$! \stackrel{\mathsf{def}}{=} X \mapsto \begin{cases} \bot & \text{if } X = \bot \\ \top & \text{otherwise} \end{cases}$$

Finally, the modality resulting of the product is:

$$!(n, l, X) \stackrel{\mathsf{def}}{=} (!n, l, !X)$$

Property 293. *! is an exponential $\mathcal{A}_3$-modality.*

Proof. It is enough to show that ! comes from a product of exponential modalities, thanks to Property 204. Since $\mathcal{A}_{\mathsf{step}}$ is idempotent the identity is exponential thanks to Lemma 205. We only prove that ! is exponential on $\mathcal{A}_1$, the $\mathcal{B}$ case being very similar.

- (Additivity) Let $n, m \in \mathbb{N}$. If $n + m = 0$ then $n = m = 0$. Hence, $!(n + m) = 0 = !n + !m$. Otherwise it means that n or m is not equal to 0. Let's suppose that $n > 0$. We then have $!(n + m) = \infty = \infty + !m = !n + !m$.
- (Contraction) Let $n \in \mathbb{N}$. Clearly, $!n + !n = !n$ (because $0 + 0 = 0$ and $\infty + \infty = \infty$).
- (Dereliction) Let $n \in \mathbb{N}$. We need to prove that $n \leq !n$. If $n = 0$ then $n \leq !n$. Otherwise, $!n = \infty$ and hence $n \leq \infty$.
- (Digging) Let $n \in \mathbb{N}$, then we want to prove that $!!n \leq !n$.
  - If $!n = 0$ then $!!n = 0$ and it is immediate.
  - If $!n = \infty$ it implies that $!!n = \infty$ and hence $!!n = !n$.

□

Corollary 294. *The following rules are $\mathcal{A}_3$-sound.*

$$\frac{\mathcal{E}; !\Gamma \vdash_3 v : (P, p)}{\mathcal{E}; !\Gamma \vdash_3 v : (!P, !p)} \qquad \frac{\mathcal{E}; \Gamma, x : !P, y : !P \vdash_3 a : (A, p)}{\mathcal{E}; \Gamma, x : !P \vdash_3 a[x/y] : (A, p)} \qquad \frac{\mathcal{E}; \Gamma, x : P \vdash_3 a : (A, p)}{\mathcal{E}; \Gamma, x : !P \vdash_3 a : (A, p)}$$

Proof. It is a corollary of Property 201. □

Lemma 295. *The following rule is $\mathcal{A}_3$-sound.*

$$\frac{\mathcal{E}; \Gamma \vdash_3 v : (\mathsf{Cap}\ P, p) \qquad \mathcal{E}; \Delta \vdash_3 w : (Q, q)}{\mathcal{E}; \Gamma, \Delta \vdash_3 \mathsf{swap}(v, w)\ \mathsf{to}\ x.(\!|\!|\mathsf{ret}(x)|\!|_2^{\alpha_{\mathsf{step}}}|\!|_1^{\alpha_{\mathsf{time}}} : (\Uparrow(\mathsf{Cap}\ Q \otimes P), p + q + \mathbf{1})}$$

Proof. By Lemma 191, it is enough to show that the following rule is $\mathcal{A}_3$-sound.

$$\frac{\mathcal{E}; \Gamma \vdash_3 v : (\mathsf{Cap}\ P, p) \qquad \mathcal{E}; \Delta \vdash_3 w : (Q, q)}{\mathcal{E}; \Gamma, \Delta \vdash_3 \mathsf{swap}(v, w) : (\Uparrow(\triangleright(\mathsf{Cap}\ Q \otimes P)), p + q)}$$

But since $\triangleright \mathsf{Cap}\ Q = \mathsf{Cap}\ Q$ and that $\triangleright$ commutes with $\otimes$, it is enough to show that the following rule is $\mathcal{A}_3$-sound.

$$\frac{\mathcal{E}; \Gamma \vdash_3 v : (\mathsf{Cap}\ P, p) \qquad \mathcal{E}; \Delta \vdash_3 w : (Q, q)}{\mathcal{E}; \Gamma, \Delta \vdash_3 \mathsf{swap}(v, w) : (\Uparrow(\mathsf{Cap}\ Q \otimes \triangleright P), p + q)}$$

For simplicity, we suppose that $\Gamma = \Delta = \varnothing$. We then suppose that $(v, p) \in \|\mathsf{Cap}\ P\|_{\mathcal{A}_3,3,\rho}$ and $(w, q) \in \|Q\|_{\mathcal{A}_3,3,\rho}$. Let $(E, r) \in [\![\Uparrow(\mathsf{Cap}\ Q \otimes \triangleright P)]\!]_{\mathcal{A}_3,3,\rho}$ and $(\underline{\mathsf{m}}, \underline{\mathsf{k}}, u) \in \mathcal{C}_{\mathcal{A}_3}(p + q + r)$. We know by definition that $p \geq (0, \infty, P)$, $v = \mathsf{cap}$. Let's pose $q = (n, l, R)$ and $r = (n', l', R')$. We then have $p + q + r \geq (n + n', \min(l, l'), P \vee R \vee R'$.

- Suppose that $P \vee R \vee R' = \top$. This is absurd since in that case $\mathcal{C}_{\mathcal{A}_3} p + q + r = \varnothing$.
- Suppose that $P \vee R \vee R' = P$. Then there exists $(n'', l'') \in \overline{\mathbb{N}} \times \overline{\mathbb{N}}$ such that

$$\begin{cases} \underline{\mathsf{m}} & \in & \mathcal{C}_{\mathcal{A}_1}(n + n' + n'') \\ \underline{\mathsf{k}} & \in & \mathcal{C}_{\mathcal{A}_{\mathsf{step}}}(\min(l, l', l'')) \\ (u, (n'', l'')) & \in & \mathcal{C}_{\mathcal{B}}(P) \end{cases}$$

That means that by the connection theorem, we have:

$$(u, (n'', l'', \bot)) \in \| \triangleright P \|_{\mathcal{A}_3, 3, \rho}$$

We want to prove that

$$\langle \mathsf{swap}(\mathsf{cap}, w), \mathsf{a}(\underline{\mathsf{m}}).\mathsf{a}(\underline{\mathsf{k}}).\mathsf{a}(u).E \rangle_3 \in \perp\!\!\!\perp$$

But it reduces to

$$\langle \mathsf{ret}(\mathsf{cap}, u), \mathsf{a}(\underline{\mathsf{m}}).\mathsf{a}(\underline{\mathsf{k}}).\mathsf{a}(w).E \rangle_3 \in \perp\!\!\!\perp$$

- First, we have $((\mathsf{cap}, u), (n'', l'', Q)) \in \| \mathsf{Cap}\ Q \otimes \triangleright P \|_{\mathcal{A}_3, 3, \rho}$. It is then enough to show that

$$(\underline{\mathsf{m}}, \underline{\mathsf{k}}, w) \in \mathcal{C}_{\mathcal{A}_3}(n'' + n', \min(l'', l'), Q)$$

- But we have clearly that

$$\begin{cases} (\underline{\mathsf{m}}, \underline{\mathsf{k}}) & \in & \mathcal{C}_{\mathcal{A}_2}(n + (n' + n''), \min(l, \min(l', l''))) \\ (w, (n, l, Q)) & \in & \| Q \|_{\mathcal{A}_3, 3, \rho} \subseteq \| \triangleright Q \|_{\mathcal{A}_3, 3, \rho} \end{cases}$$

  By a fixed-point argument we obtain $(w, (n, l)) \in \mathcal{C}_{\mathcal{B}}(Q)$. Hence the conclusion $(\underline{\mathsf{m}}, \underline{\mathsf{k}}, w) \in \mathcal{C}_{\mathcal{A}_3}(n'' + n', \min(l'', l'), Q)$.

□

7.3.3 Correctness

Definition 296. *The annotation map $(.)^\bullet$ is defined on the* swap *constructor:*

$$(\mathsf{swap}(v, w))^\bullet = \mathsf{swap}(v^\bullet, w^\bullet)\ \mathsf{to}\ x.(\!|\mathsf{ret}(x)|\!)_2^{\alpha_{\mathsf{step}}}|\!)_1^{\alpha_{\mathsf{time}}}$$

And $(.)^\bullet$ commutes with every other constructor.

Lemma 297.

- *If $\kappa \mid \vdash_{\mathsf{Str}} t : N$ then there exists $n \in \mathbb{N}$ such that $(t^\bullet, (n, \infty, \bot)) \in \| N \|_{\mathcal{A}_3, 3, []}$.*
- *If $\kappa \mid \vdash_{\mathsf{Str}} v : P$ then there exists $n \in \mathbb{N}$ such that for all $k \in \mathbb{N}$, $(v^\bullet, (n, k, \bot)) \in \| P \|_{\mathcal{A}_3, 3, []}$.*

Proof. It is a consequence of Theorem 140, Lemma 295 and Corollary 294. We can always choose $\bot$ as the third component of the annotation because the only type that can make it change is $\mathsf{Cap}\ P$, but we did not include any rule to introduce a term of type $\mathsf{Cap}\ P$. □

Theorem 298. *If* $\kappa \mid \vdash_{\mathsf{Str}} t : \mathsf{Cap}\ P \multimap N$ *and* $\iota \mid \vdash_{\mathsf{Str}} v : P$ *then* $\langle (t)\mathsf{cap}, \mathsf{nil} \star v \rangle_{\mathsf{Str}}$ *terminates.*

Proof. On the one hand, since $\kappa \mid \vdash_{\mathsf{Str}} t : \mathsf{Cap}\ P \multimap N$ and by Lemma 297 we have $k \in \mathbb{N}$ such that we can derive:

$$(t^\bullet, (k, \infty, \bot)) \in \|\mathsf{Cap}\ P \multimap N\|_{\mathcal{A}_3,3,[]}$$

On the other hand, because $\iota \mid \vdash_{\mathsf{Str}} v : P$ we have $n \in \mathbb{N}$ such that:

$$(v^\bullet, (n, n+k, \bot)) \in \|P\|_{\mathcal{A}_3,3,[]}$$

Since $\infty \geq n + k$ we obtain:

$$(t^\bullet, (k, n+k, \bot)) \in \|\mathsf{Cap}\ P \multimap N\|_{\mathcal{A}_3,3,[]}$$

We also have $(\mathsf{cap}, (0, \infty, P)) \in \|\mathsf{Cap}\ P\|_{\mathcal{A}_3,3,[]}$, hence:

$$((t)^\bullet\mathsf{cap}, (k, n+k, P)) \in \|N\|_{\mathcal{A}_3,3,[]} \qquad (\star)$$

Using Lemma 132, we then have:

$$(((t^\bullet)\mathsf{cap}, \mathsf{nil}), (k, n+k, P)) \in \perp\!\!\!\perp_{\mathcal{A}_3,3}$$

Moreover, we have $(v^\bullet, (n, \infty, \bot)) \in \|P\|_{\mathcal{A}_3,3,[]} \subseteq \|\triangleright P\|_{\mathcal{A}_3,3,[]}$, so by a fixed-point argument we also have

$$(v^\bullet, (n, \infty)) \in \mathcal{C}_{\mathcal{B}}(P) \qquad (1)$$

We also know that

$$(\underline{\mathsf{n}+\mathsf{k}}, \underline{\mathsf{n}+\mathsf{k}}) \in \mathcal{C}_{\mathcal{A}_2}(n+k, n+k) \qquad (2)$$

So by combining (1) and (2), we obtain:

$$(\underline{\mathsf{n}+\mathsf{k}}, \underline{\mathsf{n}+\mathsf{k}}, v^\bullet) \in \mathcal{C}_{\mathcal{A}_3}(k, n+k, P)$$

Finally, because of $(\star)$ we obtain that

$$\langle (t^\bullet)\mathsf{cap}, \mathsf{a}(\underline{\mathsf{n}+\mathsf{k}}).\mathsf{a}(\underline{\mathsf{n}+\mathsf{k}}).\mathsf{a}(v^\bullet).\mathsf{nil} \rangle_3 \text{ terminates on a value}$$

But, by the same reasoning as in Section 7.1, we obtain that the second counter of level 2 never reaches 0 and hence the daimon is never triggered. That means that

$$\langle (t)\mathsf{cap}, \mathsf{nil} \star v \rangle_{\mathsf{Str}} \text{ terminates}$$

□

7.3.4 Discussion

This semantical proof of termination suggests variations of this core language. Indeed, following [AFM07], we used a linear discipline to tame the access to the memory cell, hence allowing for strong updates. In our proof, the presence of higher-order references is allowed by the step-indexing algebra, but is counter-balanced by the quantitative 1-**MA** $\mathcal{A}_1$, which restores

termination. The use of $\mathcal{A}_1$ is rather restrictive, since it does not allow *any* duplication of a capability. The same discussion we had in the case of linear naive set theory applies here. We could replace $\mathcal{A}_1$ by any quantitative 1-**MA** which contains greatest element, since the definition of the modality ! would remain unchanged. In such **MA**s, it is possible to define, in addition to this exponential modality, *substructural* modalities, as the one mentioned in Section 7.2 or in Example 203. These modalities allow for a certain amount of duplication, without reaching the power of the usual linear logic exponential. Thus we believe that the cohabitation of these two types of exponentials in an extension of [AFM07] could greatly improve the expressivity of this core calculus by allowing a certain amount of sharing of the capabilities.

Chapter VIII

Conclusion

In this thesis, we have a defined a new realizability framework that allows to generalize several known techniques and reobtain various language correctness proofs. This framework moreover allows to reuse parts of those proofs as they are, hence simplifying the new semantics and proofs. We plan to continue simplifying the theory of monitoring algebras, as well as extending it with new techniques and semantical blocks. We intend to investigate how wide is the range of applications of our theory and find concrete examples where we can simplify existing proofs, as well as to apply it on new results. We already know several interesting results that can be drastically simplified by considering some slight extensions of the present work: in the context of functional reactive programming [Kri13] and in the context of indexed linear logic [BGMZ, GS14]. In addition, we have identified several interesting research directions.

8.1 | Realizability algebras

A first line of work concerns the definition a notion of **linear realizability algebra**, similar to Krivine's realizability algebra, which would generalize our unary realizability, monitoring algebras and our linear version of forcing. This would allow us to:

- have a unique soundness result for all those semantics.
- carry a general study of iteration, that would generalize the simple iteration and our two connection theorems.

In general, it is interesting to understand which results are specific to the monitoring algebras, and which ones hold in a more general context. Another possible outcome would be to find formal links with Krivine's realizability algebras.

8.2 | Control operators

We have not talked about control operators in this thesis. However, Krivine's realizability is somehow hardwired to accept control operators like call-cc. To add such a control operator to a **MA**, it seems that we need two different things:

- The first one is the commutativity of the forcing monoid. Indeed, we need to be able to take an environment and transform it into a value: this reflects at the forcing level by the commutativity of $\bullet$. This is enough to obtain an *affine form* of control operators that do not duplicate the environment (unlike call-cc).
- The second ingredient is then the possibility of duplicating the environment, which can be done in several ways. For instance this is possible if the forcing monoid admits an exponential modality or is idempotent.

The first point is not always met, for example in the case of a semi-direct iteration. If we consider the construction of Section 6.1 used to add an adjoint modality (which makes use of the semi-direct iteration), we conjecture that this is a manifestation of a well-known phenomenon: the *value restriction*. Indeed in a call-by-value language featuring control operators, one needs to restrict the exponential modality and the quantifiers to values. More about this can be found in Munch's PhD thesis [MM]. Similarly, the construction used to add adjoint modalities is not compatible with the commutativity requirement. We plan to pursue a study of control operators and the associated phenomenon in our framework.

8.3 | Generalization of the forcing monoid

We have used a particular notion to represent annotations: the forcing monoid. It has been defined to abstract over all the examples we had in mind at the beginning of this work, but not too general to simplify the presentation. However, we already are aware of examples of annotated realizability semantics that do not fit in the framework presented in this thesis. Here are some possibilities of generalizing the forcing monoid:

- When we look at the orthogonality relation induced by a **MA** $\mathcal{A}$, it looks as follows (we forget about the level):

$$(t,p)\perp_{\mathcal{A}}(E,q) \Leftrightarrow (t,E,p\bullet q)\in \perp\!\!\!\perp_{\mathcal{A}}$$

  Here, the elements p and q that annotate the environment and the computation are both elements of a same set $\mathcal{M}$, and we make them interact using the operation $\bullet$, returning also an element of $\mathcal{M}$. However there are examples of quantitative realizability semantics that distinguish between three kinds of annotations: p is an element of $\mathcal{M}_\Lambda$, q an element of $\mathcal{M}_\Pi$ and $p\bullet q$ is of a third kind $\mathcal{M}_{\Lambda\bullet\Pi}$. This is the case of [BM12] where $\mathcal{M}_\Lambda$ is a quantitative monoid, $\mathcal{M}_\Pi$ a set of functions $f:\mathcal{M}_\Lambda\to\mathcal{M}_\Lambda$ and $p\bullet f$ is the application $f(p)\in\mathcal{M}_\Lambda$. We already have started investigating a generalization of forcing monoids

that allow to deal with three kinds of elements. This generalization does not seem to break any of the results presented here (in particular the soundness theorem and the connection theorem).

- In [BG], we considered a quantitative semantics for a linear dependent type system [DLG11] that allows to give exact bounds on the time complexity of **PCF** programs. The types are not dependent of terms, as in dependent type theories, but of an external notion of first-order terms which represent the complexity bounds. That kind of dependency has been used in several other works [GHH+13, DLP13]. The difficulty lies in the fact that annotations can contain annotation variables and hence are dependent themselves. We believe our framework can also be extended to handle that very general situation, but it remains to see how difficult that would be.

8.4 | Combination with the **KFAM**

Our forcing program transformation is in some aspects orthogonal to the forcing transformation unveiled by Miquel [Miq11]. Indeed, in the **KFAM**, the combinators pushed on the protected memory cell are the terms behind the proofs of the basic facts about the set of forcing conditions: associativity, commutativity, idempotency, etc. In our forcing transformation these basic properties are computationally trivial since the notion of forcing monoid is *external* to the type system. In our case, the combinators used on the protected memory cell of the **MAM**, the **monitors**, are proofs of certain saturation properties of the forcing predicate C. It seems to us that if the monitoring algebra theory is reformulated in an adequate framework (higher-order logic or a powerful enough type theory), it would be possible to combine these two different aspects of the forcing program transformations, thus obtaining an abstract machine that combines both features.

The forcing transformation used to obtain the **KFAM** implement a mechanism that specifically monitors the management of explicit environments. Hence, we believe it could bring to our framework a greater analysis power, since some properties of the execution are tightly coupled to the substitution of variables and the duplication/erasing of environments: this could for example allow us to analyse precise space consumption or control flow analysis.

8.5 | Extension to type theory

Our framework is based on a first-order logic extending polarized linear logic. We intend in the future to reformulate this framework is the context of dependent type theories, like the calculus of constructions. It is already possible to extend classical realizability to the calculus of constructions as shown by Miquel [Miq07]. We believe such a reformulation would add more flexibility to the definition of new types and proofs of correctness. This would permit the following directions to be explored.

Binary logical relations

In this thesis, we have only studied *unary logical relations*, in the sense that in our framework, it is not possible to relate two *terms*:

$$(t, t') \Vdash A$$

This is useful to study contextual equivalences of programs. Our framework could be extended to such binary relations. But more interestingly, we conjecture that if reformulated in a sufficiently expressive theory, *i.e.* a dependent type theory or a set theory, we could retrieve binary logical relations as a particular case of *simple iteration*, *i.e.* we would have the following composition:

$$(t, t') \Vdash A \Leftrightarrow t \Vdash (t' \Vdash A)$$

This would amount to take programs, environments and configurations as forcing conditions.

Coq formalization

Formalizing the theory of monitoring algebras in Coq is one of our major future goals. However, as it is, it seems to us that it would be really diffuclut. We think that reformulating our work using a dependent type theory would make this task much more conceivable.

8.6 | A forcing study of the semi-direct iteration

We have considered the particular construction of the semi-direct iteration in Chapter V . This construction has proved to be useful. However, unlike the simple iteration, it fails to satisfy the connection theorem of Section 5.2. In a certain respect, it is not even clear why we call this operation *iteration* if it does not satisfy the connection theorem. We can in fact underline the part of the proof of Lemma 184 that fails: the positive and negative shifts. Suppose we consider the following semi-direct iteration:

$$\mathcal{D} \overset{\text{def}}{=} \mathcal{A} \ltimes_{\delta} \mathcal{B}$$

If we carefully look at the proof of the negative shift (for example), one wants to prove that:

$$\forall (E, (q, m)) \in [\![N]\!]_{\mathcal{D}, k+1, \rho}, (t, E, (p, n) \bullet (q, m)) \in \perp\!\!\!\perp \qquad (\star)$$

is equivalent to the following:

$$(t, p) \in \| \forall m \in |\mathcal{B}|. \mathcal{C}_{\mathcal{B}}(n \bullet m) \multimap N^{\circ}(m) \|_{\mathcal{A}, k, \overline{\rho}}$$

But because we use the the semi-direct product instead of the product, we can only prove that $(\star)$ is equivalent to the following:

$$\forall m \in |\mathcal{B}|, (t, \delta_m(p)) \in \| \mathcal{C}_{\mathcal{B}}(n \bullet m) \multimap N^{\circ}(m) \|_{\mathcal{A}, k, \rho}$$

This suggests that we could define a new connective $\forall^{\delta}$ whose interpretation would be basically such that:

$$(t, p) \in \| \forall^{\delta} x \in S. N \| \Leftrightarrow \forall m \in S, (t, \delta_m(p)) \in \| N[m/x] \|$$

This new connective, which a form of dependent product, allows to define a new forcing interpretation of the negative shift as follows:

$$N^*(m) \stackrel{\text{def}}{=} \forall^{\delta} m \in |\mathcal{B}|.\mathcal{C}_{\mathcal{B}}(n \bullet m) \multimap N^{\circ}(m)$$

This defines a whole new class of forcing models that we plan to investigate.

- First, we would like to consider this new forcing type translation and formally generalize the connection theorem to semi-direct iteration[1].
- We then want to see if the new forcing models coming from this interpretation have some interest on their own (in the context of set theory for instance). This new interpretation of the shift connectives seems to correspond to a *dependent* version of orthogonality that we intend to study.

Here again, formalizing our work in a dependent type theory could help.

8.7 | Store-passing translation

In [Pot11] Pottier gives a typed store-passing translation of a language with general references to a variant of system $\mathbf{F}_\omega$ with guarded recursive kinds, *i.e.* he introduces Nakano's recursion modality at the level of kinds. Pottier claims that it is the first type-preserving store-passing translation for general references. However, to achieve this result, Pottier needs to extend $\mathbf{F}_\omega$ so that in some sense it already integrates a form of step-indexing. After having extended our framework to at least system $\mathbf{F}_\omega$, by using our forcing program transformation induced by the step-indexing algebra $\mathcal{A}_{\mathsf{step}}$, we could translate his system into the simpler system $\mathbf{F}_\omega$. By composing it with Pottier's translation, one would obtain the first translation for general references into system $\mathbf{F}_\omega$. In general, it is interesting to see that if we have obtained a realizability model of a programming language by only using successive simple iterations of 1-**MA**s, we obtain at the same time a program transformation by composing the forcing transformations.

8.8 | Categorical formulation

We have not touched upon any category-theoretical consideration in the course of this thesis. We think it would be worth investigating two different possible research directions.

- It is possible to define a category of monitoring algebras, whose objects are the monitoring algebras. By quotienting the set of computations by the contextual equivalence:

  $$t \equiv_\perp u \Leftrightarrow \{t\}^\perp = \{u\}^\perp$$

  We can define a meaningful notion of morphisms between two **MA**s $\mathcal{A}$ and $\mathcal{B}$ by considering a pair:

[1] This would by the way justify the name *iteration*.

  - A sub-additive function $\phi : |\mathcal{A}| \to |\mathcal{B}|$ between the carriers of the **MA**s.
  - The equivalence classe of a computation γ that realizes the following:

$$\gamma \Vdash_0 \forall p \in |\mathcal{A}|.\mathcal{C}_{\mathcal{A}}(p) \multimap \Uparrow\mathcal{C}_{\mathcal{B}}(\phi(p))$$

  We believe we could reformulate many of the concepts presented in this thesis in a more categorical flavor, maybe leading to simplifications and new concepts. In particular, in this particular category, is it possible to give a categorical status to the simple iteration construction?

- As already mentioned, Cohen's forcing can be reformulated in topos theory. Topos theory has already proved to give a framework in which the use of logical relations can be drastically simplified [BMSS11, JTS12] and it presents the advantage of working in the context of dependent type theories. A first account of Krivine's realizability algebras in topos theory has been given by Streicher [Str13] and the product realizability [Kri11] (that combines Krivine's classical realizability with forcing) has been shown to be an application of Pitts iteration [Pit81]. We believe investigating how our work relates with topos theory would constitute a very interesting line of work.

Bibliography

[AFM07] Amal Ahmed, Matthew Fluet, and Greg Morrisett, *l^3: A linear language with locations*, Fundamenta Informaticae **77** (2007), no. 4, 397–449.

[AM01] Andrew W Appel and David McAllester, *An indexed model of recursive types for foundational proof-carrying code*, ACM Transactions on Programming Languages and Systems (TOPLAS) **23** (2001), no. 5, 657–683.

[AMRV07] Andrew W Appel, Paul-André Mellies, Christopher D Richards, and Jérôme Vouillon, *A very modal model of a modern, major, general type system*, ACM SIGPLAN Notices **42** (2007), no. 1, 109–122.

[BBTS07] Bodil Biering, Lars Birkedal, and Noah Torp-Smith, *Bi-hyperdoctrines, higher-order separation logic, and abstraction*, ACM Transactions on Programming Languages and Systems (TOPLAS) **29** (2007), no. 5, 24.

[BG] Aloïs Brunel and Marco Gaboardi, *Realizability models for a linear dependent pcf*.

[BGMZ] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic, *A core quantitative coeffect calculus*.

[BH09] Nick Benton and Chung-Kil Hur, *Biorthogonality, step-indexing and compiler correctness*, ACM Sigplan Notices **44** (2009), no. 9, 97–108.

[BM04] Patrick Baillot and Virgile Mogbil, *Soft lambda-calculus: a language for polynomial time computation*, Foundations of software science and computation structures, Springer, 2004, pp. 27–41.

[BM12] Aloïs Brunel and Antoine Madet, *Indexed realizability for bounded-time programming with references and type fixpoints*, Programming Languages and Systems, Springer, 2012, pp. 264–279.

[BMSS11] Lars Birkedal, Rasmus Ejlers Mogelberg, Jan Schwinghammer, and Kristian Stovring, *First steps in synthetic guarded domain theory: step-indexing in the topos of trees*, Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on, IEEE, 2011, pp. 55–64.

[BRS+11] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang, *Step-indexed kripke models over recursive worlds*, ACM SIGPLAN Notices **46** (2011), no. 1, 119–132.

[Bru13] Aloïs Brunel, *Quantitative classical realizability*, Information and Computation (2013).

[BSS10] Lars Birkedal, Jan Schwinghammer, and Kristian Støvring, *A metric model of lambda calculus with guarded recursion*, Fixed Points in Computer Science 2010 (2010), 19.

[BST09] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg, *Realizability semantics of parametric polymorphism, general references, and recursive types*, Foundations of Software Science and Computational Structures, Springer, 2009, pp. 456–470.

[CH00] Pierre-Louis Curien and Hugo Herbelin, *The duality of computation*, ACM sigplan notices, vol. 35, ACM, 2000, pp. 233–243.

[Coh63] Paul J Cohen, *The independence of the continuum hypothesis*, Proceedings of the National Academy of Sciences of the United States of America **50** (1963), no. 6, 1143.

[Coh64] _____, *The independence of the continuum hypothesis, ii*, Proceedings of the National Academy of Sciences of the United States of America **51** (1964), no. 1, 105.

[DAB09] Derek Dreyer, Amal Ahmed, and Lars Birkedal, *Logical step-indexed logical relations*, Logic In Computer Science, 2009. LICS'09. 24th Annual IEEE Symposium on, IEEE, 2009, pp. 71–80.

[DJ03] Vincent Danos and Jean-Baptiste Joinet, *Linear logic and elementary time*, Information and Computation **183** (2003), no. 1, 123–137.

[DLG11] Ugo Dal Lago and Marco Gaboardi, *Linear dependent types and relative completeness*, Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on, IEEE, 2011, pp. 133–142.

[DLH05] Ugo Dal Lago and Martin Hofmann, *Quantitative models and implicit complexity*, FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science (2005), 189–200.

[DLH10a] _____, *Bounded linear logic, revisited*, Logical Methods in Computer Science **6** (2010), no. 4.

[DLH10b] _____, *A semantic proof of polytime soundness of light affine logic*, Theory of Computing Systems **46** (2010), 673–689.

[DLH11] _____, *Realizability models and implicit complexity*, Theoretical Computer Science **412** (2011), no. 20, 2029 – 2047, Girard's Festschrift.

[DLM04] Ugo Dal Lago and Simone Martini, *Phase semantics and decidability of elementary affine logic*, Theoretical Computer Science **318** (2004), no. 3, 409–433.

[DLP13] Ugo Dal Lago and Barbara Petit, *The geometry of types*, The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13, Proceedings, 2013, pp. 167–178.

[DYBG+13] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang, *Views: compositional reasoning for concurrent programs*, ACM SIGPLAN Notices **48** (2013), no. 1, 287–300.

[Fil89] Andrzej Filinski, *Declarative continuations: An investigation of duality in programming language semantics*, Category Theory and Computer Science, Springer, 1989, pp. 224–249.

[GB40] Kurt Gödel and George William Brown, *The consistency of the axiom of choice and of the generalized continuum-hypothesis with the axioms of set theory*, no. 3, Princeton University Press, 1940.

[GDR09] Marco Gaboardi and Simona Ronchi Della Rocca, *From light logics to type assignments: a case study*, Logic Journal of IGPL **17** (2009), no. 5, 499–530.

[GHH+13] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce, *Linear dependent types for differential privacy*, ACM SIGPLAN Notices, vol. 48, ACM, 2013, pp. 357–370.

[Gir87] Jean-Yves Girard, *Linear logic*, Theoretical computer science **50** (1987), no. 1, 1–101.

[Gir91] _____, *A new constructive logic: classical logic*, Mathematical Structures in Computer Science **1** (1991), no. 3, 255–296.

[Gir92] _____, *A fixpoint theorem in linear logic. an email posting to the mailing list linear@ cs*, 1992.

[Gir98] _____, *Light linear logic*, Information and Computation **143** (1998), no. 2, 175–204.

[Gir01] _____, *Locus solum: From the rules of logic to the logic of rules*, Mathematical Structures in Computer Science **11** (2001), no. 3, 301–506.

[GMRDR12] Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca, *An implicit characterization of pspace*, ACM Transactions on Computational Logic (TOCL) **13** (2012), no. 2, 18.

[Gri82] VN Grišin, *Predicate and set-theoretic calculi based on logic without contractions*, Mathematics of the USSR-Izvestiya **18** (1982), no. 1, 41.

[Gri89] Timothy G Griffin, *A formulae-as-type notion of control*, Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1989, pp. 47–58.

[GS14] Dan R Ghica and Alex Smith, *Bounded linear types in a resource semiring*, European Symposium on Programming (ESOP), Grenoble, France. Springer, 2014.

[Hof03] Martin Hofmann, *Linear types and non-size-increasing polynomial time computation*, Information and Computation **183** (2003), no. 1, 57–85.

[Hos12] Naohiko Hoshino, *Step indexed realizability semantics for a call-by-value language based on basic combinatorial objects*, Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science, IEEE Computer Society, 2012, pp. 385–394.

[How69] William A Howard, *The formulae-as-types notion of construction.*

[JT11] Guilhem Jaber and Nicolas Tabareau, *Decomposing logical relations with forcing.*

[JTS12] Guilhem Jaber, Nicolas Tabareau, and Matthieu Sozeau, *Extending type theory with forcing*, Logic in Computer Science (LICS), 2012 27th Annual IEEE Symposium on, IEEE, 2012, pp. 395–404.

[KBH12] Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann, *Higher-order functional reactive programming in bounded space*, POPL, 2012, pp. 45–58.

[Kle45] Stephen Cole Kleene, *On the interpretation of intuitionistic number theory.*

[Kom89] Yuichi Komori, *Illative combinatory logic based on bck-logic."*, Math. Japonica **34** (1989), 585–596.

[Kri03] Jean-Louis Krivine, *Dependent choice,âĂŸquoteâĂŹand the clock*, Theoretical Computer Science **308** (2003), no. 1, 259–276.

[Kri07] _____, *A call-by-name lambda-calculus machine*, Higher-Order and Symbolic Computation **20** (2007), no. 3, 199–207.

[Kri09] _____, *Realizability in classical logic*, Panoramas et synthèses **27** (2009), 197–229 (Anglais).

[Kri10a] _____, *Realizability algebras: a program to well order R*, manuscript (2010).

[Kri10b] _____, *Realizability algebras II: new models of ZF+ DC*, Arxiv preprint arXiv:1007.0825 (2010).

[Kri11] _____, *Realizability algebras: a program to well order r*, Logical Methods in Computer Science (LMCS) **7** (2011), no. 3, 1–47.

[Kri13] Neelakantan R. Krishnaswami, *Higher-order functional reactive programming without spacetime leaks*, Proceedings of the 18th ACM SIGPLAN international conference on Functional programming, ACM, 2013, pp. 221–232.

[Laf96] Yves Lafont, *The undecidability of second order linear logic without exponentials*, Journal of Symbolic Logic (1996), 541–548.

[Laf97] _____, *The finite model property for various fragments of linear logic*, Journal of Symbolic Logic **62** (1997), no. 4, 1202–1208.

[Laf04] _____, *Soft linear logic and polynomial time*, Theoretical Computer Science **318** (2004), no. 1-2, 163–180.

[Lev99] Paul Blain Levy, *Call-by-push-value: a subsuming paradigm*, Typed Lambda Calculi and Applications, Springer, 1999, pp. 228–243.

[Lev03] _____, *Call-by-push-value: A functional/imperative synthesis*, vol. 2, Springer, 2003.

[Miq07] Alexandre Miquel, *Classical program extraction in the calculus of constructions*, Computer Science Logic, Springer, 2007, pp. 313–327.

[Miq11] _____, *Forcing as a program transformation*, Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on, IEEE, 2011, pp. 197–206.

[MM] Guillaume Munch-Maccagnoni, *Models of a non-associative composition.*

[MM09] _____, *Focalisation and classical realisability*, Computer Science Logic, Springer, 2009, pp. 409–423.

[Mog91] Eugenio Moggi, *Notions of computation and monads*, Information and computation **93** (1991), no. 1, 55–92.

[MT10] Paul-André Mellies and Nicolas Tabareau, *Resource modalities in tensor logic*, Annals of Pure and Applied Logic **161** (2010), no. 5, 632–653.

[Nak00] Hiroshi Nakano, *A modality for recursion*, LICS, 2000, pp. 255–266.

[Oka99] Mitsuhiro Okada, *Phase semantic cut-elimination and normalization proofs of first- and higher-order linear logic*, Theoretical Computer Science **227** (1999), no. 1-2, 333–396.

[Oka02] _____, *A uniform semantic proof for cut-elimination and completeness of various first and higher order logics*, Theoretical Computer Science **281** (2002), no. 1, 471–498.

[Par92] Michel Parigot, *$\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction*, Logic programming and automated reasoning, Springer, 1992, pp. 190–201.

[Pit81] Andrew M Pitts, *The theory of triposes*, 1981.

[Pot11] François Pottier, *A typed store-passing translation for general references*, ACM SIGPLAN Notices, vol. 46, ACM, 2011, pp. 147–158.

[PS98] Andrew M Pitts and Ian DB Stark, *Operational reasoning for functions with local state*, Higher order operational techniques in semantics (1998), 227–273.

[RP10] J. Reed and B.C. Pierce, *Distance makes the types grow stronger: A calculus for differential privacy*, ACM SIGPLAN Notices, vol. 45, ACM, 2010, pp. 157–168.

[SBP13] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson, *Impredicative concurrent abstract predicates*, Under submission (2013).

[Shi94] M. Shirahata, *Linear set theory*, PHD Thesis (1994).

[ST71] Robert M Solovay and Stanley Tennenbaum, *Iterated cohen extensions and souslin's problem*, Annals of Mathematics (1971), 201–245.

[Str13] Thomas Streicher, *Krivine's classical realisability from a categorical perspective*, Mathematical Structures in Computer Science **23** (2013), no. 06, 1234–1256.

[Ter02] Kazushige Terui, *Light logic and polynomial time computation*, Ph.D. thesis, Keio University, 2002.

[Ter04] ———, *Light affine set theory: a naive set theory of polynomial time*, Studia Logica **77** (2004), no. 1, 9–40.