Translating types and effects with state monads and linear logic

Paolo Tranquilli

To cite this version:

Paolo Tranquilli. Translating types and effects with state monads and linear logic. 2010. hal-00465793

HAL Id: hal-00465793

https://hal.archives-ouvertes.fr/hal-00465793

Preprint submitted on 21 Mar 2010

Translating Types and Effects with State Monads and Linear Logic

Paolo Tranquilli
LIP, ENS Lyon, Université de Lyon
(UMR 5668 CNRS ENS Lyon UCBL INRIA)
Email: paolo.tranquilli@ens-lyon.fr

***Abstract*—We study a lambda-calculus with references and a types and effects system. In the first part of the paper, we translate it into the ordinary lambda-calculus with products, implementing an interacting family of state monads localized at sets of regions. In general the target language must be endowed with recursive types. However we prove that the stratification condition on regions, already used in type and effect systems to assure termination, is equivalent to completely avoid the use of recursion in the types used in the translation. We thus obtain a logical characterization of stratification, and by simulation we also provide a new proof that it yields termination. In the second part of the paper we extend the call-by-value translation of ordinary lambda-terms in linear logic proof nets to the calculus with references. This allows for a parallel evaluation of the calculus that preserves its sequential semantics.**

I. Introduction

Mainstream programming paradigms are pervaded with side effects. The great majority of programs do not simply calculate a function, but carry out a whole lot of other actions that may influence the result: interacting with the user or with other processes, jumping to particular parts of its code, accessing memory,. . . There is a lot of research in computer science that goes towards controlling such side effects. Indeed programs that make large, uncontrolled use of side effects are harder to understand, verify or optimize.

Among the abstract tools that have been developed to this end and that are of interest to this work are *types and effects* systems [1] and *monads* [2]. The objective of the former is to analyze statically side effects by annotating in some way the ordinary types of programs. A typical way to analyze memory access is abstract memory into different entities called regions; then one decorates types with the set of regions which the typed program can access, possibly specifying what kind of access it needs. The annotated types become then informative on what and where can something happen when calling the function. A suitable level of abstraction from the actual workings of memory management allow to carry out a static analysis. For example such an approach has been successfully used to analyze the problem of heap memory deallocation ([3], leading to the so-called region based memory management).

Monads are a tool directly coming from category theory which envisages to encapsulate and abstract away the details of side effects while remaining in a "clean" typed world. The idea is that a monad T can be seen as a type constructor modeling a computational paradigm where (effect-less) *values* of type A are separated from *computations* of type $T(A)$. All the details are left to the monad's *unit* $A \to T(A)$, embedding values into computations, and its *multiplication* $T^2(A) \to T(A)$, determining how computations should compose, possibly interacting with one another. Since their inception they made it to be a highlight of Haskell's type system and way of programming.

Both approaches rely on a common ground: types as a tool to study and/or discipline programs. To this end when it comes to memory access, the typical result is either allowing effective parallelization (like in the original type and effect proposal [1]), or ensure type safety (e.g. no "wrong" data occurs during execution), or allow timely memory deallocation as already mentioned. However there is another property that in general type systems have been studied to deliver: *termination*, i.e. a certificate that the program will eventually yield a result.

Until recently this particular aspect has not been much studied in the presence of side effects involving memory access, especially when higher order types are possibly referenced. Indeed it was long known [4] that apart from the classical way of obtaining a (diverging) fix-point operator through self application, which is easily forbidden by types, such a term can be encoded through well-typed self reference. Using the syntax we will show in Figure 1, a diverging term can be easily written following this idea:

$$\nu r{\Leftarrow}\lambda x.\,\mathtt{get}(r)x.\,\mathtt{get}(r)\,\langle\rangle\,.$$

Such a term can be read as "store in the location r the higher order function that reads from r what it should do, then apply it". Indeed, marking as $F = \lambda x.\mathtt{get}(r)x$ the execution yields

$$\nu r{\Leftarrow}F.\,\mathtt{get}(r)\,\langle\rangle \to \epsilon r.\,\mathtt{get}(r)\,\langle\rangle\,, r{\Leftarrow}F \\ \to \epsilon r.F\,\langle\rangle\,, r{\Leftarrow}F \to \epsilon r.\,\mathtt{get}(r)\,\langle\rangle\,, r{\Leftarrow}F \to \cdots$$

$r{\Leftarrow}F$ is detached in a store accessible from the term. The ϵr is a feature of our syntax which acts as a place-holder to garbage-collect values of the store once computation has ended (more details in section II). Returning to the self-referencing, divergent term, we can set r to hold functions of type $1 \to 1$, and the resulting term will be typed as 1. We may get a hint as to why the program loops (but a priori no solution) by annotating types and seeing that in fact r stores functions $1 \xrightarrow{\{r\}} 1$: the set added to the arrow indicates that functions stored in r may access r, so circularity may ensue.

Recent works explored the idea of *stratification* of regions to avoid such circularities [5], [6] and yield termination not only for sequential but also for cooperative multithreading programs. The idea is that one must follow a precise order when assigning types to regions, which induces an ordering on regions so that, intuitively, a region may affect or read only regions that are strictly smaller. This has a distinct logical scent to it: even more so when one see their proof technique, which they carry via reducibility candidates.

In this work we set out to study types and effects from the logical point of view, and stratification together with them. We first chose the viewpoint of Girard's linear logic (LL [7]=, which has already been employed in the study of λ-calculus. One of the theoretical notions in LL's toolbox are two translations of intuitionistic logic into LL (also first described in [7]). These two mappings are based on two different encodings of the intuitionistic arrow $A \to B$ (corresponding via the Curry-Howard proofs-as-program paradigm to the type of functions from A to B): $!(A \multimap B)$ and $!A \multimap B$. The $\multimap$ is the *linear arrow*, typing proofs/programs which use their hypotheses exactly once. The power of duplication is regained (and controlled) via the *exponential modality* ! ("off course"). The two translation then mark two different perspectives: in $!(A \multimap B)$ we are saying that functions are the duplicable objects, and thus the values that can be passed around, while in the other we are saying that being an argument suffices to be duplicated. Indeed it was shown in [8] that the two translation are intimately linked with two paradigms of evaluation of functional programs, respectively call-by-value and call-by-name. We concentrate on the first, which is used more often in calculi with references.

The syntax of choice for LL are *proof nets*, graph-theoretical representation of proofs that have the advantage of exposing parallel features of deterministic and sequential computation. Its recent developments in the direction of non-deterministic *differential* extensions [9] reaching concurrency [10] make this kind of investigation the ideal launching pad towards extending logical interpretations of concurrent computation, in the sense of Curry-Howard. For now we restrain to the sequential case and leave multithreading for future work: here proof nets provide for a direct representation of the dependencies among different parts of the terms by means of wires (we follow the interaction net paradigm [11]). In particular effect annotations translate into several wires carrying around values: if a particular part of the program does not use a particular region, its wire will run to the next term in the evaluation order, which will be able to get and process the value before the preceding term has actually finished. Sequential semantics is preserved, as any evaluation of proof nets corresponds to the one of the term.

As it turns out, translating this kind of system is not a feature exclusive to LL. Though it exposes the parallel features of effect annotating, the constructions it uses can still be carried out in the environment of ordinary λ-calculus. Unsurprisingly, effects correspond exactly to state monads: annotation simply allows to restrict the state monads to the memory actually used. We thus implement a family T_e of monads indexed by sets of regions, where combining a computation $T_{e_1}(A \to T_{e_3}(B))$ with one in $T_{e_2}(A)$ yields a computation in $T_{e_1 \cup e_2 \cup e_3}(B)$, i.e. affected regions are clearly summed up during evaluation. As the state monad can be encoded internally in types, it turns out that the types assigned to regions in e are in fact directly referred to in $T_e(A)$. If we go back to the self-referencing type, we obtain the translation $(1 \xrightarrow{\{r\}} 1)^\circ = 1 \to T_r(1) = 1 \to X_r \to (X_r \times 1)$, where X_r is the (translation of) the type assigned to r... that is $(1 \xrightarrow{\{r\}} 1)^\circ$ itself! We therefore get for r the *recursive* type $X_r = 1 \to X_r \to (X_r \times 1)$. These are perfectly fine for type safety, except they cannot ensure termination. As it turns out, this unstratified instance is not a case: stratification is equivalent to providing a type to all regions without reverting to recursive ones. Stratification *is* logic.

Outline: In the upcoming section II we introduce Λ_{reg}, the calculus on which we base our work. It is a deterministic and single threaded variant of the one of [6]. The only different feature it has is *local values*: the store works in fact like a stack, and when a value is introduced it covers what was assigned to that region beforehand, until the computation in which it is used (and possibly updated) returns a value. Then the value gets garbage collected and the previous value is exposed again. What are localized however are instances of the store, not the regions. We prove that the type and effect system (stratified or not) guarantees that a program never locks (Lemma 3): for example if it requests a value there will be one for him in the store. We make some comparisons with other calculi.

In section III we implement localized monads in $\Lambda_\times$, the simply typed λ-calculus with products. We then use them to completely translate Λ_{reg}, proving two results: stratification is equivalent to using simple types without fixpoints of formulae (Proposition 12), and a Λ_{reg} terms evaluates to a value V iff its translation evaluates too to the translation of V (Theorem 14). In particular we get a new proof that stratification yields termination (Corollary 15).

In section IV we pass to LL proof nets. We redefine the translation, that is essentially what done with $\Lambda_\times$ passed through the call-by-value translation; we then show simulation (Theorem 23) and that we can follow any reduction strategy (limited to depth 0) in the proof net corresponding to a term M: we are guaranteed that we will find the value of M if there is one (Theorem 24).

In section V we make some final remarks and present our future objectives.

Notations: We will use multisets (i.e. functions from a set to natural numbers, the multiplicities) with additive notation, so that $\mu_1 + \mu_2$ is disjoint union, $\mu_1 \leq \mu_2$ means that μ_1 is a submultiset of μ_2, and $\mu_1 - \mu_2$ is multiset subtraction. Given a relation $\to$, the notation $\xrightarrow{*}$ is for the transitive reflexive closure of $\to$.

II. The λ-Calculus with Regions

In this section we present the λ-calculus with regions we will use for our results.

A. Syntax and Reduction

Figure 1 presents the syntax of **terms**, **stores** and the reduction of their interaction. A ν-step is one reducing a $\nu r \Leftarrow V.N$ subterm. As usual, we associate both abstractions and applications, i.e. $\lambda x, y.M = \lambda x.\lambda y.M$ and $MN_1N_2 = (MN_1)N_2$. We also use the imperative notation $M; N$ to denote $(\lambda d.N)M$ with $d \notin \text{FV}(N)$.

Syntax

$$\begin{array}{lll}
x, y & \text{(variables)} & \\
r, s & \text{(regions)} & \\
U, V & ::= x \mid \langle\rangle \mid \lambda x.M & \text{(values)} \\
M, N & ::= V \mid MN \mid \nu r{\Leftarrow}M.N \mid \epsilon r.M & \\
 & \qquad \mid \mathtt{set}(r, M) \mid \mathtt{get}(r) & \text{(terms)} \\
S, T & ::= \varepsilon \mid r{\Leftarrow}V \mid S, T & \text{(stores)} \\
E, F & ::= [\,] \mid EM \mid VE \mid \mathtt{set}(r, E) & \\
 & \qquad \mid \nu r{\Leftarrow}E.M \mid \epsilon r.E & \text{(eval. contexts)}
\end{array}$$

Structural congruence for stores

$$\varepsilon, S \equiv S \equiv S, \varepsilon, \quad (S_1, S_2), S_3 \equiv S_1, (S_2, S_3)$$

$$r{\Leftarrow}U, s{\Leftarrow}V \equiv s{\Leftarrow}V, r{\Leftarrow}U \quad \text{if } r \neq s.$$

Reduction

$$\begin{array}{ll}
E[(\lambda x.M)V], S & \to E[M\{V/x\}], S, \\
E[\nu r{\Leftarrow}U.M], S & \to E[\epsilon r.M], r{\Leftarrow}U, S \\
E[\mathtt{set}(r, V)], r{\Leftarrow}U, S & \to E[\langle\rangle], r{\Leftarrow}V, S \\
E[\mathtt{get}(r)], r{\Leftarrow}U, S & \to E[U], r{\Leftarrow}U, S \\
E[\epsilon r.V], r{\Leftarrow}U, S & \to E[V], S.
\end{array}$$

Figure 1. Syntax and reduction of Λ_{reg}.

A first abstraction with respect to other calculi with references is that we do not directly employ true references that can be passed around. In fact we identify regions with *locations*: every region in a given moment provides only one value. Notice indeed that though there may be multiple values for r in the store S, as stores are *not* commutative in general, there will always be a single value "on the surface" for a given region: reduction is completely deterministic.

Stores are therefore functions assigning *stacks* to regions. This feature is introduced by the interaction between $\nu r{\Leftarrow}V$, that pushes a value in the store, and the ϵr it leaves behind. The latter is a helper constructor that waits until the term it is attached to becomes a value and then removes the entry from the store possibly revealing what was assigned to r before. $\nu r/\epsilon r$ thus implement a kind of local entry in the store. However it must not be confused with a private region (like the PRIVATE constructor from [1]), as it does not bind r in any way. In particular any reference to r inside the body of a function is unaffected by the $\nu r/\epsilon r$ reductions and my be evaluated later to refer to other values of r. Also the value at the top of the stack can "leak" below after that it has been destroyed if a function passes it around.

We take the opportunity to present an example using all the reduction rules in Figure 1 and exposing what we just wrote. Let $\mathtt{tt}$ and $\mathtt{ff}$ be $\lambda x, y.x$ and $\lambda x, y.y$ respectively. Then

$$\begin{array}{l}
\nu r{\Leftarrow}\,\mathtt{tt}\,.\,\mathtt{get}(r)(\nu{\Leftarrow}\,\mathtt{ff}\,.(\lambda x, y.\,\mathtt{set}(r, x); x)\,\mathtt{get}(r))\,\langle\rangle\,\langle\rangle \\
\xrightarrow{*} \epsilon r.\,\mathtt{tt}(\nu{\Leftarrow}\,\mathtt{ff}\,.(\lambda x, y.\,\mathtt{set}(r, x); x)\,\mathtt{get}(r))\,\langle\rangle\,\langle\rangle\,, r{\Leftarrow}\,\mathtt{tt} \\
\xrightarrow{*} \epsilon r.(\nu{\Leftarrow}\,\mathtt{ff}\,.(\lambda x, y.\,\mathtt{set}(r, x); x)\,\mathtt{get}(r))\,\langle\rangle\,, r{\Leftarrow}\,\mathtt{tt} \\
\to \epsilon r.(\epsilon r.(\lambda x, y.\,\mathtt{set}(r, x); x)\,\mathtt{get}(r))\,\langle\rangle\,, r{\Leftarrow}\,\mathtt{ff}, r{\Leftarrow}\,\mathtt{tt} \\
\xrightarrow{*} \epsilon r.(\epsilon r.\lambda y.\,\mathtt{set}(r, \mathtt{ff}); \mathtt{ff})\,\langle\rangle\,, r{\Leftarrow}\,\mathtt{ff}, r{\Leftarrow}\,\mathtt{tt} \\
\to \epsilon r.(\lambda y.\,\mathtt{set}(r, \mathtt{ff}); \mathtt{ff})\,\langle\rangle\,, r{\Leftarrow}\,\mathtt{tt} \\
\xrightarrow{*} \epsilon r.\,\mathtt{ff}, r{\Leftarrow}\,\mathtt{ff} \to \mathtt{ff}\,.
\end{array}$$

Notice how the "internal" $\mathtt{ff}$ got to be the final result.

$$\begin{array}{lr}
e, f & \text{(finite sets of regions)} \\
A, B ::= \mathbf{1} \mid A \xrightarrow{e} B & \text{(types)} \\
\Gamma, \Delta ::= x_1 : A_1, \ldots, x_n : A_n & \text{(variable context)} \\
R, S ::= r_1 : A_1, \ldots, r_n : A_n & \text{(region context)}
\end{array}$$

Typing

$$\frac{}{R; \Gamma \vdash x : A, \emptyset} \qquad \frac{}{R; \Gamma \vdash \langle\rangle : 1, \emptyset}$$

$$\frac{R; \Gamma, x : A \vdash M : B, e}{R; \Gamma \vdash \lambda x.M : A \xrightarrow{e} B, \emptyset}$$

$$\frac{R; \Gamma \vdash M : A \xrightarrow{e_3} B, e_1 \quad R; \Gamma \vdash N : A, e_2}{R; \Gamma \vdash MN : B, e_1 \cup e_2 \cup e_3}$$

$$\frac{R, r : A; \Gamma \vdash M : A, e_1 \quad R, r : A; \Gamma \vdash N : B, e_2 \cup \{r\}}{R, r : A; \Gamma \vdash \nu r{\Leftarrow}M.N : B, e_1 \cup (e_2 \setminus \{r\})}$$

$$\frac{R, r : A; \Gamma \vdash M : A, e \cup \{r\}}{R, r : A; \Gamma \vdash \epsilon r.M : A, e \setminus \{r\}}$$

$$\frac{R, r : A; \Gamma \vdash M : A, e}{R, r : A; \Gamma \vdash \mathtt{set}(r, M) : 1, e \cup \{r\}}$$

$$\frac{}{R, r : A; \Gamma \vdash \mathtt{get}(r) : A, \{r\}}$$

$$\frac{R; \Gamma \vdash M : A, e \quad e \subsetneq e'}{R; \Gamma \vdash M : A, e'}$$

$$\frac{R; \vdash M : A, e \qquad \forall r{\Leftarrow}V \in S : R; \vdash V : R(r), \emptyset}{R; \vdash M, S : A, e}$$

Stratification

$$\frac{}{\emptyset \vdash} \qquad \frac{R \vdash A}{R, r : A \vdash} \qquad \frac{R \vdash}{R \vdash 1}$$

$$\frac{R \vdash A \quad R \vdash B \quad e \subseteq \mathrm{dom}(R)}{R \vdash A \xrightarrow{e} B}$$

Figure 2. Type and effect system for Λ_{reg}.

B. *Types, Effects and Stratification*

Figure 2 presents all the actors involved in types and effects inference of Λ_{reg}. Notice how effects are cumulative save for ν/ϵ which effectively erases the effect from, so to say, the active ones. Notice also how dummy effects can be freely added. In the same figure we also present the stratification condition on the region context R, denoted by $R \vdash$. The following are basic results on the type system.

Lemma 1.

- *if* $R; \Gamma \vdash M : A, e$ *and* $x \notin \mathrm{dom}(\Gamma)$, *then* $R; \Gamma, x : A \vdash M, e$;
- *if* $R; \Gamma \vdash V : A, \emptyset$ *and* $R; x : A, \Gamma \vdash M : B, e$, *then* $R; \Gamma \vdash M\{V/x\} : B, e$;
- *if* $R; \vdash M, S : A$ *and* $M, S \to M', S'$, *then* $R; \vdash M', S' : A$.

States, programs, results: Types allow to forbid unwanted configurations, like trying to apply a base value as a function. However memory access provides for other ways of misbehaving: for example asking for the value in an empty memory cell, $(\mathtt{get}(r), \varepsilon)$ or trying to write to an unallocated slot $(\mathtt{set}(r, V), \varepsilon)$. In fact the types and effect system, together with the νr construct we employ, allow to avoid this kind of situations. We just need to start from a closed term with no pending effects (i.e. $R; \vdash M : A, \emptyset$), provided M does not use any ϵr. Nevertheless in order to reason inductively on programs and prove their properties we also need to describe the intermediate states between the starting program and (hopefully) the value it reaches. In the following we will do exactly so introducing the notion of *external state*, checking on what regions and how many of them a term must have access to.

The domain $\mathrm{dom}(S)$ of a store S is defined as the *multiset* of r's for which there is $r \Leftarrow V \in S$. Formally, $\mathrm{dom}(\varepsilon) = [\,]$, $\mathrm{dom}(r \Leftarrow V) = [r]$ and $\mathrm{dom}(S, T) = \mathrm{dom}(S) + \mathrm{dom}(T)$. Let $\mathrm{ar}(M)$ (the **active regions**) be the partial function taking terms to *multisets* of regions defined inductively as follows ($\perp$ stands for undefined).

$$\begin{aligned}
&\mathrm{ar}(x) = \mathrm{ar}(\langle\rangle) = \mathrm{ar}(\mathtt{get}(r)) := [\,]\,,\\
&\mathrm{ar}(\mathtt{set}(r, M)) = \mathrm{ar}(M),\\
&\mathrm{ar}(MN) := \begin{cases} \mathrm{ar}(N) & \text{if } M \text{ is a value,}\\ \mathrm{ar}(M) & \text{if } M \text{ not a value and } \mathrm{ar}(N) = [\,],\\ \perp & \text{otherwise,}\end{cases}\\
&\mathrm{ar}(\lambda x.M) := \begin{cases} [\,] & \text{if } \mathrm{ar}(M) = [\,],\\ \perp & \text{otherwise,}\end{cases}\\
&\mathrm{ar}(\nu r \Leftarrow M.N) := \begin{cases} \mathrm{ar}(M) & \text{if } \mathrm{ar}(N) = [\,],\\ \perp & \text{otherwise,}\end{cases}\\
&\mathrm{ar}(\epsilon r.M) = \mathrm{ar}(M) + [r]\,.
\end{aligned}$$

What this function does is check how many νr's were activated and turned into ϵr's waiting for an evaluation to end. Non-definedness marks that something is wrong: there is an ϵr that could not possibly be generated during evaluation (for example under a λ).

Definition 2. A **state** is a pair M, S of a term M and a store S such that

- M, S is typable with some type and effects A, e,
- $\mathrm{ar}(M)$ is defined and $\mathrm{ar}(M) \leq \mathrm{dom}(S)$, and
- $e \subseteq |\,\mathrm{dom}(S) - \mathrm{ar}(M)|$.

A state S, M is **external** if $\mathrm{dom}(S) = \mathrm{ar}(M)$ (in particular it has $e = \emptyset$). A **program** is an external state M, ε: in particular M, ε is a program iff M is closed, typable with some $A, \emptyset$, and not containing any subterm $\epsilon r.N$, so that $\mathrm{ar}(M) = [\,]$. A **result** is a state V, S whose term is a value. We call external results V, ε directly values.

While asking that a program do not contain ϵr constructs may seem sensible, marking it as a helper constructor used for evaluation but not available for programming (similar to the *PRIVATE* constructor of [1]), the condition on states by means of the "ar" function might seem a bit awkward. However the stability of external states (Lemma 3) and the one we will present later (Lemma 8) indeed characterize external states as exactly the residuals of programs. Non-external states are needed to carry out some proofs by induction: intuitively, memory access operations are always evaluated in an internal state, otherwise they would fail in retrieving or setting a value.

The following result states that the conditions imposed on programs/states are stable under reduction and that they guarantee that either we have divergence or get a value/result. In particular no deadlock related to memory occurs: memory access ($\mathtt{get}(r)$ and $\mathtt{set}(r, V)$) and deallocation ($\varepsilon r.V$) do not produce "segmentation faults", i.e. are always evaluated when there is an $r \Leftarrow V$ in the store, so their reduction is defined. Moreover if M, ε is a program and evaluates to a result V, S, then it will be necessarily a value V, ε, i.e. garbage collection will have been done.

Lemma 3. *If M, S is a state, then either it is a result V, S or $M, S \to M', S'$ with M', S' a state too. Moreover $\mathrm{dom}(S') - \mathrm{ar}(M') = \mathrm{dom}(S) - \mathrm{ar}(M)$; in particular if M, S is external so is M', S'.*

C. Alternatives

In this section we discuss some alternatives for the syntax, either coming from the literature, or necessary later in the paper.

1) Subtyping: The reader could notice the lack of subtyping in our system, as in [1], [6]: we retained just the possibility to add dummy effects. We will here explain how this does not really affect the expressiveness (though it may affect the conciseness of terms when subtyping *is* needed).

Subtyping is given by the following inductive definition:

$$\frac{}{A \leq A} \qquad \frac{A' \leq A \quad B \leq B' \quad e \subseteq f}{A \xrightarrow{e} B \leq A' \xrightarrow{f} B'}$$

Transitivity follows from a proof by induction. Let $\vdash_{\mathrm{s}}$ be the type system with the same rules done in Figure 2 and the following one for subtyping:

$$\frac{R; \Gamma \vdash_{\mathrm{s}} M : A, e \quad A \leq B}{R; \Gamma \vdash_{\mathrm{s}} M : B, e}$$

Lemma 4. *For every derivation of $R; \Gamma \vdash_{\mathrm{s}} M : A, e$ there is one of the same assertion where the subtyping rule appears only under axioms (namely the rules for variable and for $\mathtt{get}$).*

The subtyping rule cannot be completely removed preserving the same assertion $M : A$. Nevertheless one may notice that η-expansion allows some form of supertyping: e.g. we have $x : 1 \xrightarrow{e} 1 \vdash \lambda y.xy : 1 \xrightarrow{e'} 1$ with $e \subseteq e'$, without recurring to the explicit subtyping rule but just using the dummy effects one. However η-expansion behaves very badly in general with call-by-value, as it may turn a non-value into a value[1]. Though η-expansion of values is acceptable, it does not suffice here. For example, if we want to cast $x : 1 \to 1 \to 1$ to its supertype $1 \to 1 \xrightarrow{e} 1$, we would need to η-expand to $\lambda y, z.xyz$, in particular expanding xy which is not a value. We thus provide a particular combination of η and β expansions which do the trick.

Let $\rightsquigarrow$ be the reflexive and compatible closure of

$$\begin{aligned}
V \;&\rightsquigarrow \lambda x.I(Vx), \quad x \notin \mathrm{FV}(V),\\
\mathtt{get}(r) &\rightsquigarrow I\,\mathtt{get}(r),
\end{aligned}$$

where $I = \lambda x.x$, the identity. Notice that under non-weak reduction, the right hand side of $\rightsquigarrow$ reduces to the η-expansion of the right hand side. Notice also that $M \rightsquigarrow M'$

[1] For example, if M is a diverging term, its η-expansion is a value and is thus terminating.

implies that M is a value iff M' is one too. We extend $\rightsquigarrow$ also to stores. Finally, by compatibility one easily has that if $M \rightsquigarrow M'$ and $V \rightsquigarrow V'$ then $M\{V/x\} \rightsquigarrow M'\{V'/x\}$.

Lemma 5. *If $R;\Gamma \vdash_{\mathrm{s}} M : A, e$ is derivable, then there is M' with $M \overset{*}{\rightsquigarrow} M'$ with $R;\Gamma \vdash M' : A, e$ without subtyping.*

Lemma 6. *If $M \rightsquigarrow M'$ and $S \rightsquigarrow S'$, then $M, S \Downarrow V, S_0$ iff $M', S' \Downarrow V', S_0'$ with $V \rightsquigarrow V'$ and $S' \rightsquigarrow S_0'$.*

The above two lemmas combined together say that the system without subtyping we presented is expressive as much as the one with subtyping, modulo doing an expansion on terms which allows mimicking subtyping by just adding dummy effects.

2) Reference types: Another feature that might seem strange to have missing are explicit types for references, and the possibility to treat references/regions as data, possibly to be passed between programs, like in [1]. The language presented in this paper must mainly be seen as a tool to abstract away some specific features of references, for example with the objective of proving termination.

In any case given enough expressiveness one can encode reference types. In fact if we suppose to have an implementation in λ-calculus of lists and natural numbers (for example using polymorphic λ-calculus, see the discussion we make in section V), it is not hard to implement general references. We here give an informal description of it.

For references of type A, let be given a region $r : \mathtt{List}_A$ (of list type). Then let

$$\begin{aligned} \mathtt{new}_r &:= \lambda v.(\lambda \ell.\,\mathtt{set}(r,\ell);\mathtt{length}\,\ell)\,\mathtt{get}(r)\\ \mathtt{write}_r &:= \lambda n,v.\,\mathtt{set}(r,\mathtt{update}\,\mathtt{get}(r)nv),\\ \mathtt{read}_r &:= \lambda n.\,\mathtt{pick}\,\mathtt{get}(r)n,\end{aligned}$$

where the new terms employed (`length`, `update` and `pick`) are rather self-explanatory. Then `new` can be interpreted to assign a value to a new slot in the store and pass a reference to it (which quite bluntly is the size of the list and thus the index of the last element inserted in it. Given such an integer `write` and `read` can then update and obtain the values from the list. The term must only be wrapped into $\nu r \Leftarrow \mathtt{empty}.M$, initializing the list to be empty.

3) Storeless reduction: We will here present an alternative definition of reduction that internalizes stores and does not use the helper ϵ construct. We will use such a syntax to define the translation of states into linear logic proof nets in section IV.

Let **ν-evaluation** contexts be generated by the same rules as evaluation contexts (Figure 1), adding however $\nu \Leftarrow V.E$. Given a ν-evaluation context E, let $\mathrm{PR}(E)$ (the private regions of E) be the set of r's so that E's hole is in the scope of a νr. Now given a program M (in particular without ϵr's), we define the **storeless reduction** $M \rightarrowtail M'$ as:

$$E[(\lambda x.M)V] \rightarrowtail E[N\{V/x\}], \quad E[\nu r \Leftarrow V.U] \rightarrowtail E[U]$$
$$E[\nu r \Leftarrow U.F[\mathtt{set}(r,V)]] \rightarrowtail E[\nu r \Leftarrow V.F[\langle\rangle]]$$
$$E[\nu r \Leftarrow U.F[\mathtt{get}(r)]] \rightarrowtail E[\nu r \Leftarrow V.F[V]],$$

where E and F are ν-evaluation contexts such that $r \notin \mathrm{PR}(F)$. Intuitively, the νr in $\nu r \Leftarrow U.F$ is the first value for r whose scope captures the location we are evaluating. We recall that a ν-step is one that in regular reduction reduces a $\nu r \Leftarrow V.N$ subterm adding V to the store.

Syntax

$$\begin{array}{lll} U, V & ::= x \mid \langle\rangle \mid \lambda x.M \mid \langle U, V\rangle & \text{(values)}\\ M, N & ::= V \mid MN \mid \pi_1 M \mid \pi_2 M \mid \langle M, N\rangle & \text{(terms)}\\ E, F & ::= [\,] \mid EM \mid VE \mid \pi_1 E \mid \pi_2 E & \\ & \qquad \mid \langle E, N\rangle \mid \langle V, E\rangle & \text{(eval. contexts)} \end{array}$$

Reduction

$$(\lambda x.M)V \to M\{V/x\}, \qquad \pi_i\langle V_1, V_2\rangle \to V_i.$$

Figure 3. Syntax and reduction of $\Lambda_\times$.

$$\begin{array}{lll} X, Y & & \text{(type variables)}\\ A, B & ::= X \mid \mathbf{1} \mid A \to B \mid A \times B & \text{(types)}\\ \Gamma, \Delta & ::= x_1 : A_1, \dots, x_n : A_n & \text{(variable context)}\\ E, F & ::= X_1 \doteq A_1, \dots, X_k \doteq A_k, \dots & \text{(systems of eq.)} \end{array}$$

Typing

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash \langle\rangle : 1}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A \times B} \qquad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \pi_i M : A_i}$$

Figure 4. Type system for $\Lambda_\times$.

Now we will show how to lift a state to a program that reduces to it via ν-steps (which, we recall, are those reducing νr's), and which will be able to simulate the reduction of the state via $\rightarrowtail$ (Lemma 9). It will be then the case that the lifted program perfectly simulates the evolution of the state it lifted from by means of storeless reduction.

Definition 7. The **lifting** $(M, S)^\nu$ of a state be defined as M if $M = V$ or $M = \mathtt{get}(r)$, and otherwise:

$$\begin{aligned} (MN, S)^\nu &:= \begin{cases} M(N, S)^\nu & \text{if } M \text{ is a value,}\\ (M, S)^\nu & \text{otherwise.}\end{cases}\\ (\nu r \Leftarrow M.N, S)^\nu &:= \nu r \Leftarrow (M, S)^\nu.N,\\ (\mathtt{set}(r, M), S)^\nu &:= \mathtt{set}(r, (M, S)^\nu)\\ (\epsilon r.M, S, r \Leftarrow V)^\nu &:= \nu r \Leftarrow V.(M, S)^\nu. \end{aligned}$$

Lemma 8. *For every external state M, S, $(M, S)^\nu$ is defined and is the unique program such that $(M, S)^\nu, \epsilon \xrightarrow{*} M, S$ using only ν-steps.*

Lemma 9. *Let (M, S) be an external state. Then $(M, S) \to (M', S')$ with a non-ν-step iff $(M, S)^\nu \rightarrowtail (M', S')^\nu$. If $(M, S) \to (M', S')$ with a ν-step, then $(M, S)^\nu = (M', S')^\nu$.*

III. Types and Effects into Monads

In this section we define the translation of Λ_{reg} into the ordinary λ-calculus with products. As it turns out, on the type level this corresponds to introducing state *monads* [2]. Indeed, the annotated arrow $A \xrightarrow{e} B$ will correspond to the arrow $A \to T_e(B)$ where T_e is a state monad indexed by the set of regions.

In Figure 3 we show the syntax of such calculus, which we denote by $\Lambda_\times$. Types and typing rules for $\Lambda_\times$ are shown in Figure 3. As usual, $\mathrm{FV}(A)$ denotes the set of variables appearing in A. For x and y distinct variables we will write $\lambda\langle x, y\rangle.M$ to mean $\lambda p.(\lambda x, y.M)(\pi_1 p)(\pi_2 p)$. We will denote the generalized product (parenthesized on the left) by $\prod_{i\in I} A_i$, if I has an order associated with it. We define the empty product as 1, and generalized tuples by $\langle M_1, \ldots, M_{k+1}\rangle := \langle\langle M_1, \ldots, M_k\rangle, M_{k+1}\rangle$ for $k \geq 2$, and $\langle M\rangle := M$. The corresponding projections π_i^k are obtained by combining the two projections π_i.

As we will want to account also for the unstratified case, we have also introduced type variables and systems of equations, which are considered here as functions (possibly with infinite domain) from some type variables to *non-atomic* types, presented as sets of pairs $X \doteq E(X)$. Given such a system of equations, it can be used to define recursive types by considering the structural equivalence $\equiv_E$ generated by $X \equiv_E E(X)$ for $X \in \mathrm{dom}(E)$ (i.e. $\equiv_E$ is the equivalence relation generated by the context closure of the equations $\doteq$ appearing in E).

Definition 10. We say that E is **solvable** if there is an assignment σ_E from $\mathrm{dom}(E)$ to *closed* (i.e. variable-free) types so that for all $X \in \mathrm{dom}(E)$ we have $X \equiv_E \sigma_E(X)$.

The above is equivalent to asking $\sigma_E(X) = \sigma_E(E(X))$, where $\sigma_E(\ )$ is extended on all types as usual, letting it be the identity on variables outside $\mathrm{dom}(E)$ and proceeding by induction. Notice that if E is solvable, then $\mathrm{FV}(E(X)) \subseteq \mathrm{dom}(E)$ for all $X \in \mathrm{dom}(E)$, the solution σ_E is unique, and the quotient by $\equiv_E$ gives just the types A with $\mathrm{FV}(A) \cap \mathrm{dom}(E) = \emptyset$, in particular no truly recursive types (such as $X = 1 \times X$) are induced.

A. State Monads

In this section we will implement side effects in $\Lambda_\times$ by using state monads. Let there be a distinguished type variable X_r for each region r, and let E be a system of equations on the X_r's. From now on, types will be considered modulo E. Let there be also a fixed order on regions, so that sets of regions are considered presented according to this order.

Given a finite set e of regions, the **type of** e**-stores** is $P_e := \prod_{r\in e} X_r$. The **state monad** localized at e is defined by the type constructor $T_e(A) = P_e \to (P_e \times A)$. This is the classic state monad, modeling the fact that a computation will start with a certain state and return possibly another one together with the result. We just parametrize it by a finite product of types, indicating to what values it will have access.

Let us first introduce some $\Lambda_\times$ terms working with stores. Given $r \in e$, we define $\pi_r^e M$ as $\pi_i^k M$ where $k = \# e$ and i is r's position in e. We generalize to $e = \{r_1, \ldots, r_k\} \subseteq f$ by setting $\pi_e^f M = \langle \pi_{r_1}^f M, \ldots, \pi_{r_k}^f\rangle$.

We will now prove some of the main properties of the terms presented in Figure 5. We call e-stores the values of type P_e. Given an e-store S and an f-store T then if $e \cap f = \emptyset$, $S + T$ is defined as the $e \cup f$-store given by joining the two; if $e \subseteq f$ then $T|_e$ is the store T restricted to regions in e. In fact,$S + T$ and $T|_e$ are the values of $\mathtt{upd}_{e,f}\, ST$ and $\pi_e^f T$ respectively.

Monadic structure

$$\mathtt{let}\,x\,\mathtt{be}\,M\,\mathtt{in}\,N := \lambda s.(\lambda\langle s_1, x\rangle.N s_1)(M s),$$
$$[M] := \lambda s.\langle s, M\rangle.$$

Derived typing rules

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash [M] : T_e(A)}$$
$$\frac{\Gamma \vdash M : T_e(A) \quad x : A, \Gamma \vdash N : T_e(B)}{\Gamma \vdash \mathtt{let}\,x\,\mathtt{be}\,M\,\mathtt{in}\,N : T_e(B)}$$

Additional projections

$$\pi_i^k M := \begin{cases} M & \text{if } i = k = 1, \\ \pi_i M & \text{if } k = 2, \\ \pi_2 M & \text{if } i = k > 2, \\ \pi_i^{k-1}\pi_1 M & \text{if } i < k \text{ and } k > 2, \end{cases}$$

$$\pi_r^e M := \pi_i^{\# e}, \quad \text{where } i \text{ is } r\text{'s place in } e,$$
$$\pi_f^e M := \langle \pi_r^e M\rangle_{r\in f}, \quad \text{if } f \subseteq e.$$

Additional terms

$$\mathtt{upd}_{e,f} := \lambda s, t.\langle c_r\rangle_{r\in e\cup f} \text{ with } c_r = \begin{cases} \pi_r^e s & \text{if } r \in e, \\ \pi_r^f t & \text{otherwise,} \end{cases}$$
$$\mathtt{upd}_{r,e} := \mathtt{upd}_{\{r\},e},$$
$$\mathtt{g} := \lambda x.\langle x, x\rangle,$$
$$\mathtt{s} := \lambda x, d.\langle x, \langle\rangle\rangle,$$
$$\mathtt{n}_r^e := \lambda x, p, s.(\lambda\langle s_1, v\rangle.\langle \pi_{e\setminus\{r\}}^{e\cup\{r\}} s_1, v\rangle)(p(\mathtt{upd}_{r,e\cup\{r\}}\, xs)),$$
$$\mathtt{cast}_{e,f} := \begin{cases} I & \text{if } f \subseteq e\text{; otherwise:} \\ \lambda p, s.(\lambda\langle s_1, v\rangle.\langle \mathtt{upd}_{e,f}\, s_1 s, v\rangle)(p(\pi_e^{e\cup f} s)) \end{cases}$$

Types of additional terms

$$\vdash \mathtt{upd}_{e,f} : P_e \to P_f \to P_{e\cup f},$$
$$\vdash \mathtt{g} : T_{\{r\}}(X_r),$$
$$\vdash \mathtt{s} : X_r \to T_{\{r\}}(1),$$
$$\vdash \mathtt{n}_r^e : X_r \to T_{e\cup\{r\}}(A) \to T_{e\setminus\{r\}}(A),$$
$$\vdash \mathtt{cast}_{e,f} : T_e(A) \to T_{e\cup f}(A),$$

Mixing monads

$$\mathtt{let}_{e,f}\,x\,\mathtt{be}\,M\,\mathtt{in}\,N := \mathtt{let}\,x\,\mathtt{be}\,\mathtt{cast}_{e,f}\,M\,\mathtt{in}\,\mathtt{cast}_{f,e}\,N,$$

$$\frac{\Gamma \vdash M : T_e(A) \quad x : A, \Gamma \vdash N : T_f(B)}{\Gamma \vdash \mathtt{let}_{e,f}\,x\,\mathtt{be}\,M\,\mathtt{in}\,N : T_{e\cup f}(B)}$$

Figure 5. Implementation of localized state monads.

Lemma 11. *We have the following properties on the terms introduced in Figure 5.*

- *If $S : P_{e\setminus\{r\}}$, $V : X_r$ is a value (so $\langle V\rangle$ is an $\{r\}$-store) and $M : T_{e\cup\{r\}}(A)$ then $\mathtt{n}_r^e VMS \xrightarrow{*} \langle S', U\rangle$ iff $M(S + \langle V\rangle) \xrightarrow{*} \langle S' + \langle V'\rangle, U\rangle$ for some $V' : X_r$.*
- *If $S : P_{e\cup f}$, $M : T_e(A)$ then $\mathtt{cast}_{e,f}\, MS \xrightarrow{*} \langle S', U\rangle$ iff $MS|_e \xrightarrow{*} \langle S'|_e, U\rangle$ and $S'|_{f\setminus e} = S|_{f\setminus e}$.*
- *If $S : P_{e\cup f}$, $M : T_e(A)$, $x : A \vdash N : T_f(B)$, then $(\mathtt{let}_{e,f}\,x\,\mathtt{be}\,M\,\mathtt{in}\,N)S \xrightarrow{*} \langle S', U\rangle$ iff $MS|_e \xrightarrow{*} \langle T, V\rangle$ and $N\{V/x\}(T|_{f\cap e} + S|_{f\setminus e}) \xrightarrow{*} \langle S'|_f, U\rangle$.*

B. Translating the Types

We are now ready to translate Λ_{reg} types into $\Lambda_\times$. In the following we define such a translation, and one taking region contexts into systems of equations on $\Lambda_\times$ types.

$$1^\circ := 1, \quad (A \xrightarrow{e} B)^\circ := A^\circ \to T_e(B^\circ),$$
$$\emptyset^\circ := \emptyset, \quad (R, r : A)^\circ := R^\circ, X_r \doteq A^\circ$$

$$\frac{}{x : A \vdash x : A, \emptyset \mapsto [x]} \qquad \frac{}{\vdash \langle\rangle : 1, \emptyset \mapsto [\,\langle\rangle\,]}$$

$$\frac{x : A \vdash M : B, e \mapsto M'}{\vdash \lambda x.M : A \xrightarrow{e} B, \emptyset \mapsto [\lambda x.M']}$$

$$\frac{\vdash M : A \xrightarrow{e_3} B, e_1 \mapsto M' \quad \vdash N : A, e_2 \mapsto N'}{\begin{array}{r}\vdash MN : B, e_1 \cup e_2 \cup e_3 \mapsto \mathtt{let}_{e_1, e_2 \cup e_3}\, f\, \mathtt{be}\, M'\, \mathtt{in} \\ \mathtt{let}_{e_2, e_3}\, a\, \mathtt{be}\, N'\, \mathtt{in}\, fa\end{array}}$$

$$\frac{\vdash M : A, e_1 \mapsto M' \quad \vdash N : B, e_2 \cup \{r\} \mapsto N'}{\begin{array}{r}\vdash \nu r{\Leftarrow}M.N : B, e_1 \cup (e_2 \setminus \{r\}) \qquad\qquad \\ \mapsto \mathtt{let}_{e_1, e_2 \setminus \{r\}}\, v\, \mathtt{be}\, M'\, \mathtt{in}\, \mathtt{n}_r^{e_2}\, vN'\end{array}}$$

$$\frac{\vdash M : R(r), e \mapsto M'}{\vdash \mathtt{set}(r, M) : 1, e \cup \{r\} \mapsto \mathtt{let}_{e, \{r\}}\, v\, \mathtt{be}\, M\, \mathtt{in}\, \mathtt{s}\, v}$$

$$\frac{}{\vdash \mathtt{get}(r) : R(r), \{r\} \mapsto \mathtt{g}}$$

$$\frac{\vdash M : A, e \mapsto M' \quad e \subseteq e'}{\vdash M : A, e' \mapsto \mathtt{cast}_{e,e'}\, M'}$$

$$\frac{R; \vdash M : A, \emptyset \mapsto M'}{R; \vdash M, \varepsilon : A, \emptyset \mapsto \pi_2(M\,\langle\rangle)}$$

Figure 6. The rules defining the translation $\mapsto$ from Λ_{reg} programs to $\Lambda_\times$, passing through the localized monadic structure.

As it can be seen, the annotated function type $A \xrightarrow{e} B$ gets translated, in category theoretic terms, to arrows of the Kleisli category for the monad T_e. The region context just sets a system of equations that equates a variable X_r with the translation of the type associated with r.

Stratification: We will account for stratification by showing that via our translation it is equivalent to avoiding the use of recursive types, as the associated system of equations is solvable (Definition 10). We will thus give a strong logical justification as to why stratification ensures termination, even if the result is not new: it allows to internalize all in an ordinary λ-calculus.

Proposition 12. *R° is solvable iff the stratification condition $R \vdash$ holds.*

In fact, the proof of the above proposition shows how stratification gives an order in which each indeterminate of R° find its value.

C. Translating the Terms

We will now turn our attention to terms. In the following we fix a region context R and consider all $\Lambda_\times$ types under the structural equivalence $\equiv_E$ (possibly as non-recursive types if, as seen, R is stratified). In Figure 6 we define the translation of Λ_{reg} *programs* (to be more precise type derivations of programs) to $\Lambda_\times$ terms. Notice in particular that no translation is given for $\epsilon r.M$: as we will prove the simulation of an entire evaluation rather than a step by step one, this does not pose particular problems. Notice that the translation of a value V typed with $A, \emptyset$ is necessarily $[V']$ for a value V'.

$$\begin{array}{lr} X, Y & \text{(type variables)} \\ A, B ::= X \mid X^\perp \mid \mathbf{1} \mid \perp \mid A \otimes B \mid A \parr B & \text{(types)} \\ E, F ::= X_1 \doteq A_1, \dots, X_k \doteq A_k & \text{(systems of eq.)} \end{array}$$

Translation

$$1^\bullet := !1, \qquad \{r_1, \dots, r_k\}^\bullet := \bigotimes_{i=1}^{k} !X_r,$$

$$(A \xrightarrow{e} B)^\bullet := !\big((A^\bullet \otimes e^\bullet) \multimap (e^\bullet \otimes B^\bullet)\big),$$

$$(\emptyset)^\bullet = \emptyset \qquad (R, r : A)^\bullet = R^\bullet, X_r \doteq A^\bullet$$

Figure 7. Recursive linear logic types and type and effect translation.

Proposition 13. *Let $\Gamma^\circ(x) := (\Gamma(x))^\circ$. Then:*

- *if $R; \Gamma \vdash M : A, e \mapsto M'$ then $\Gamma^\circ \vdash M' : T_e(A^\circ)$ is derivable in $\Lambda_\times$ modulo $\equiv_{R^\circ}$; in particular programs $M, \varepsilon : A, \emptyset$ are mapped to closed terms of type A°;*
- *if $R; x : A\Gamma \vdash M : A, e \mapsto M'$ and $R; \Gamma \vdash V : A, \emptyset \mapsto [V']$ then $R; \Gamma \vdash M\{V/x\} : A, e \mapsto M'\{V'/x\}$.*

Here follows the main result of this section, stating that the $\Lambda_\times$ term associated with each Λ_{reg} term evaluates to the same result (up to translation).

Theorem 14. *If $R; \vdash M, \varepsilon : A, \emptyset \mapsto M'$ is a Λ_{reg} program with its associated M' term, then M evaluates to the value V, ε iff $M' \xrightarrow{*} V'$ with $V \mapsto [V']$.*

Corollary 15. *If $R \vdash$ is stratified and $R; \vdash M, \varepsilon$ then M terminates to a value.*

IV. Types and Effects into Linear Logic

In this section we draw from the intuitions gained from the translation of Λ_{reg} into $\Lambda_\times$ to translate $\Lambda_\times$ into LL proof nets. We have two starting points: the translation of call-by-value λ-calculus [7], [8], and the exponential isomorphism $!(A \times B) \cong !A \otimes !B$ that can be used for pairs.

A. The nets

First of all, in Figure 7 we define LL's formulae (and provide already the translation of Λ_{reg} types). As usual, the dual is involutive ($A^{\perp\perp} = A$) and defined via De Morgan laws $1^\perp = \perp$ and $(A \otimes B)^\perp = A^\perp \parr B^\perp$. The linear arrow $A \multimap B$ is defined as $A^\perp \parr B$. For example chasing dualities on the translation of the arrow with effects we have

$$(A \xrightarrow{\{r_i\}_i} B)^\bullet = \Big(\big((A^\bullet)^\perp \parr \textstyle\bigparr_i ?X_{r_i}^\perp\big) \parr \big(\bigotimes_i !X_{r_i} \otimes B^\bullet\big)\Big).$$

Again, in order to be able to translate also in absence of stratification, we consider systems of equations, and the induced structural equivalence. Once again, X_r are special variables marked with regions. The definition of solvability of such a system is identical to what done in section III: the only difference is one takes into account substitution of dual variables. We will present LL proof nets in the interaction net style [11], and range over them with letters such as π, σ.

Definition 16. LL nets[2] are, intuitively, cells linked with wires. A net π is thus given by a set of **cells**, to each of

[2] We keep the definition informal, for more details the reader is referred to [12].

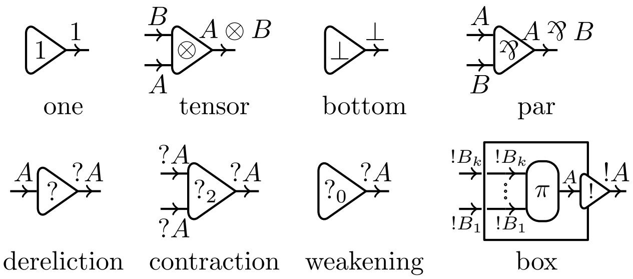


Figure 8. The cells of LL, together with their typing rules.

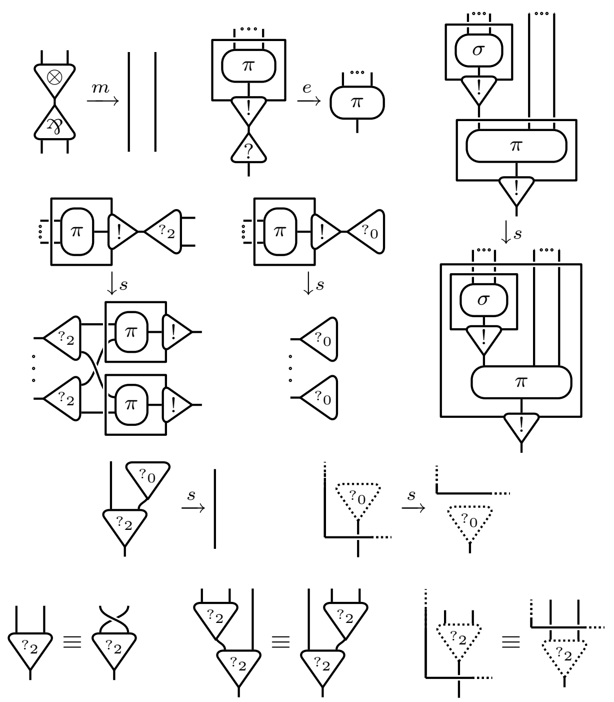


Figure 9. Reduction and equivalence rules of LL.

which a number of disjoint **ports** and a symbol is assigned; each port, belonging to a cell or **free** (i.e. a conclusion) belongs exactly to one **wire**, which in fact is a set of two ports. A net is **typed** if there is an assignment to **directed wires** such that reversing the direction of the wire we get the dual type, and such that other properties are satisfied.

The graphical representation of cells, together with their names, their symbols, their number of ports and how they must be typed are depicted in Figure 8. In particular exponential **boxes** can be formally considered as cells having whole nets as symbols, their contents.

Notice that we chose a *planar* presentation, i.e. the ⅋ has their premises flipped with respect to ⊗. The **depth** of an element in a net is the number of nesting boxes containing it. In particular depth 0 is outside any box.

Reduction can be defined as usual as a context closure, once contexts in this graphical setting are defined. Again, we will skip the details. Roughly, it amounts to finding a pattern in the net, and replacing it with another net gluing wires back.

Definition 17. Figure 9 shows the reduction and equivalence rules we employ on LL nets. The m and e (multiplicative and exponential) reductions are considered *only at depth* 0. The s (structural) reduction is considered at any depth. We denote simply by $\rightarrow$ the union of $\xrightarrow{m}$, $\xrightarrow{e}$ and $\xrightarrow{s}$. We say that π is in **0-depth normal form** if it is normal for m and e.

In the definition of normal form we ignore s reductions, but they are strongly normalizing anyway (Lemma 19). Equivalences account for inessential differences of the nets which have no counterpart in models and from the computational point of view: commutativity and associativity of contraction, and its commutation with box borders. Such equivalences where already studied in literature about explicit substitutions [13], or for differential nets [14][3]. We do not use the syntax automatically quotienting such equivalences (as in [15]), as we want to keep the dereliction on box step separate from structural ones. We need also to use the other reductions we list in the structural ones: neutrality of weakening over contraction and pulling weakenings out of boxes. As usual, a **correctness criterion** is enforced on nets to guarantee their good computational behaviour. One of the most used is the switching acyclicity one [16].

Definition 18. A **switching path** is a path passing through adjacent wires at depth 0, which never passes by two premises of a par or a contraction. A net is **switching acyclic** (or a **proof net**) if it has no switching cycles and inductively all the box contents are switching acyclic too.

From now on all nets we will consider are implicitly switching acyclic. We will use the following properties of proof nets and the reductions we listed.

Lemma 19. *In* untyped LL *(and so also in presence of recursive types) one has the following properties.*

- $\xrightarrow{s}$ *is strongly normalizing;*
- $\rightarrow$ *is confluent;*
- π *is strongly normalizing for* $\rightarrow$ *iff it is weakly so.*

B. Translating the Types

While introducing LL types in Figure 7 we also defined the translation of Λ_{reg} types. Now we explain a bit informally how the translation we presented is related to the one we gave for $\Lambda_\times$ in section III. Let A^* be the translation of $\Lambda_\times$ types into LL's ones, defined by

$$1^* := !1, \quad X^* := !X, \quad (A \to B)^* := !(A^* \multimap B^*),$$
$$(A \times B)^* = A^* \otimes B^*.$$

A^* is in fact the classical call-by-value translation with pairs added in: seen that all A^* start with an !, $A^* \otimes B^*$ is indeed isomorphic to an LL product. Now if we compose the translation given in section III with this one (*before* any system of equations is applied) , we obtain

$$(A \xrightarrow{e} B)^{\circ *} = (A^\circ \to P_e \to (P_e \times B^\circ))^*$$
$$= !\big(A^{\circ *} \multimap !(e^\bullet \multimap (e^\bullet \otimes B^{\circ *}))\big).$$

[3] Here we will skip the subtleties linked to reduction modulo an equivalence. However the results of Lemma 19 are valid in this framework.

However we know from the intended behaviour of the translation that the function taking stores P_e is duplicated only when part of a function taking actual values: in terms of monads, it is a computation, not a value. In other words, the inner ! above is useless to our objective. So we pass to

$$!(A^{\circ *} \multimap e^\bullet \multimap (e^\bullet \otimes B^{\circ *})) \cong !\big((A^{\circ *} \otimes e^\bullet) \multimap (e^\bullet \otimes B^{\circ *})\big),$$

where the uncurrying isomorphism $A \multimap B \multimap C \cong (A \otimes B) \multimap C$ (monoidal closedness) shows that our translation is essentially the one we presented into $\Lambda_\times$ passed through the call-by-value translation.

The following result has the same statement and proof of Proposition 12.

Proposition 20. *$R \vdash$ is stratified iff $R^\bullet$ is solvable.*

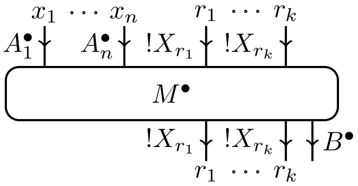
C. Translating the Terms

We will define the translation as a mapping $M \mapsto M^\circ$ between typed terms and nets. Even more explicitly than we have done before, we will identify a term with its type derivation. This poses problems of representation: a type derivation may have dummy variables in the context; similarly it may also have dummy effects. We sidestep the problem by considering $y : D, \Gamma(x) \vdash M : A, e \equiv \Gamma \vdash M : A, e \equiv \Gamma \vdash, e \cup \{r\}$, if the center one is derivable. Clearly similar equivalences will have to be considered on nets also.

Given a term

$$R; x_1 : A_1, \ldots, x_k : A_k \vdash M : B, \{\, r_1, \ldots, r_k \,\}$$

its translation $M^\bullet$ will have the following general form.



(1)

In particular, we have on top a wire for every variable (labelled with it) and one for every affected region, "entering" the net. Below there are corresponding wires for regions, and one for the output on the right. Informally, a translated term may be seen graphically as a unit processing a stream (the region wires passing through it) based on inputs (the variables) and giving out an output. With this graphical convention in fact the parts of the net that are evaluated beforehand are always on top. At times for the sake of space we may also draw proof nets from left to right rather than from top to bottom.

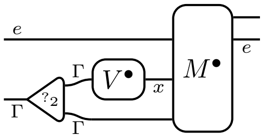
Given a context Γ (resp. a set of regions e) a wire labelled by Γ (resp. by e) will stand for multiple wires labelled by variables in $\mathrm{dom}(\Gamma)$ (resp. regions in e), typed accordingly. In Figure 10 we show the rules defining the translation $M^\bullet$. Again, notice that no rule is given for $\epsilon r.M$, like for the $\Lambda_\times$ translation. However we use lifting to extend the translation to all external states by setting $(M, S)^\bullet := ((M, S)^\nu)^\bullet$. We refer to Definition 7 and the subsequent results for the definition and the properties of the lifting $(M, S)^\nu$.

First, the upcoming lemma shows that the translation is "statically" correct: it indeed yields proof nets, that is switching acyclic nets.

Lemma 21. *For every typed term $R; \Gamma \vdash M : A, e$, the net $M^\bullet$ is typed (modulo $R^\bullet$) and switching acyclic.*

The following is a standard result when translating calculi into nets.

Lemma 22 (substitution)**.**



$\xrightarrow{s*} (M\{V/x\})^\bullet$.

Now we show the main results of this part of the paper.

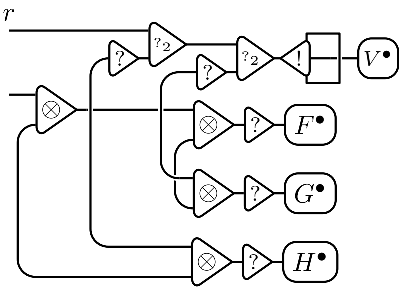
Theorem 23 (Simulation)**.** *Let M, S be an external state. Then:*

- *if $M, S = V, \varepsilon$ is a value, then $(V, \varepsilon)^\bullet$ is in 0-depth normal form.*
- *if $M, S \to M', S'$ with a non-ν-step then $(M, S)^\bullet \xrightarrow{+} (M', S')^\bullet$ with exactly one dereliction on box step;*
- *if $M, S \to M', S'$ with a ν-step, then $(M, S)^\bullet = (M', S')^\bullet$.*

The following theorem finally tells that we can in fact calculate directly with proof nets: we can reduce in parallel the net while preserving the sequential semantics enforced by effects.

Theorem 24. *If $(M, S)^\bullet \xrightarrow{*} \pi$, with π a 0-depth normal form, then there is a value V such that $M, S \to V, \varepsilon$ and $\pi = V^\bullet$.*

An informal example: Let us show the point we made in the introduction with an informal example. Suppose we have a state $(\mathtt{set}(r, V); F)(G\,\mathtt{get}(r))(H\,\mathtt{get}(r)), r \Leftarrow U$ and F is typed with effects not containing r. Then its translation will be (omitting U that is erased any way):



Even supposing that the term F is undergoing some heavy calculations, the translation exposes the fact that the store containing V can not only interact with the $\mathtt{get}(r)$ in argument position for G (possibly starting computaion in G, it can be delivered even to $H^\bullet$, even before G's $\mathtt{get}$ "accepts" to take the value. Wires expose directly what dependencies are present in the term and where values can arrive even before the evaluation strategy permits it, all this while guaranteeing the same final result.

V. Concluding remarks

In this paper we gave a logical account of a λ-calculus with references, interpreting it as monads in ordinary λ-calculus and as proof nets in LL. There are some final points and future perspectives that can be discussed.

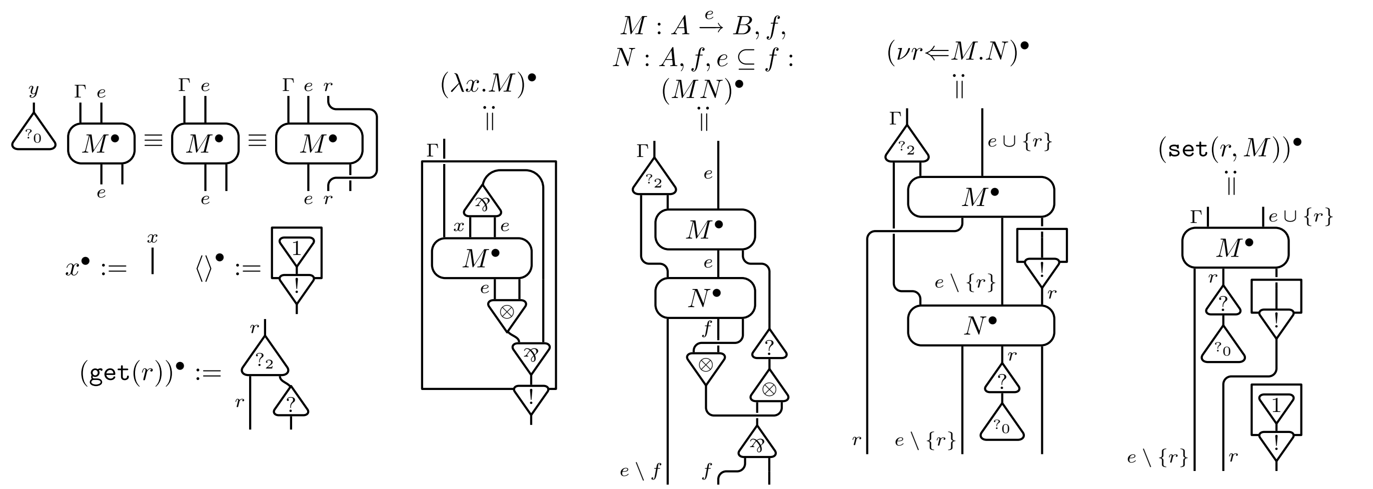


Figure 10. The translation of Λ_{reg} programs into proof nets. In the rule for equivalence, $y \notin \mathrm{dom}(\Gamma)$ and $r \notin e$ are required. Also, we suppose that the wire for r is moved to its position in $e \cup \{r\}$.

Polymorphism: Indeed we did not include any kind of polymorphism in our treatment. However *type* polymorphism (generalization to $\forall X.A$ and instantiation to $A[B/X]$) can straightforwardly be added to the type system and does not entail any difficulty in the translation, though the types of regions need to be closed. Polymorphism of regions is probably also possible to handle, but needs some more investigation.

Multithreading: what is lacking the most with respect to other proposals of calculi (or type systems) is multithreading and concurrency. Indeed the starting objective of this work was to combine call-by-value translation of λ-calculus together with the communication zones which were employed in [10] for a bisimulation between (a fragment of) π-calculus and differential nets. Indeed by slightly generalizing to differential nets and non-determinism the translation presented in this work and combining it with elements of [10], one gets a translation of a multithreaded version of the calculus. However the target nets are very easily cyclic. For example $\mathtt{set}(r, \mathtt{get}(r)) | \mathtt{set}(r, \mathtt{get}(r))$, which may in general be any two threads cooperatively updating a shared variable, is (it seems) necessarily cyclic. No particular computational property can be therefore entailed, save for simulation. The problem seems to linked with how logic in general and proof nets in particular handle dependency. In proof nets dependency (which may be tracked with switching paths) can never be created. In particular in $\mathtt{set}(r, \mathtt{get}(r)) | \mathtt{set}(r, \mathtt{get}(r))$ there is a potential dependency of each of the $\mathtt{get}$'s from the other $\mathtt{set}$, so that from the logical point of view there is a circular dependency which is somewhat hidden by prefixing. Indeed also in π-calculus' translation a simple process like $c(x).\bar{c}\langle x\rangle\, |c(x).\bar{c}\langle x\rangle$ is mapped to a cyclic net.

It seems then the only direction for truly using linear logic with concurrency is either to restrict programs in order to fall within $\mathtt{LL}$'s scope (such as forbidding processes like the one pointed above), or rather find a new meaning to correctness to account for such concurrent behaviours.

References

[1] J. M. Lucassen and D. K. Gifford, "Polymorphic effect systems," in *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* New York, NY, USA: ACM, 1988, pp. 47–57.

[2] E. Moggi, "Notions of computation and monads," *Information and Computation*, vol. 93, no. 1, pp. 55–92, Jul. 1991.

[3] M. Tofte and J.-P. Talpin, "Region-based memory management," *Inf. Comput.*, vol. 132, no. 2, pp. 109–176, 1997.

[4] P. J. Landin, "The mechanical evaluation of expressions," *The Computer Journal*, vol. 6, no. 4, pp. 308–320, January 1964. [Online]. Available: http://dx.doi.org/10.1093/comjnl/6.4.308

[5] G. Boudol, "Fair cooperative multithreading," in *CONCUR*, ser. Lecture Notes in Computer Science, vol. 4703. Springer, 2007, pp. 272–286.

[6] R. M. Amadio, "On stratified regions," in *APLAS*, ser. Lecture Notes in Computer Science, Z. Hu, Ed., vol. 5904. Springer, 2009, pp. 210–225.

[7] J.-Y. Girard, "Linear Logic," *Th. Comp. Sc.*, vol. 50, pp. 1–102, 1987.

[8] J. Maraist, M. Odersky, D. N. Turner, and P. Wadler, "Call-by-name, call-by-value, call-by-need and the linear lambda calculus," *Theor. Comput. Sci.*, vol. 228, no. 1-2, pp. 175–210, 1999.

[9] T. Ehrhard and L. Regnier, "Differential interaction nets," *Theor. Comput. Sci.*, vol. 364, no. 2, pp. 166–195, 2006.

[10] T. Ehrhard and O. Laurent, "Interpreting a finitary pi-calculus in differential interaction nets," in *CONCUR*, ser. Lecture Notes in Computer Science, L. Caires and V. T. Vasconcelos, Eds., vol. 4703. Springer, 2007, pp. 333–348.

[11] Y. Lafont, "From proof nets to interaction nets," in *Advances in Linear Logic*, ser. London Mathematical Society Lecture Note Series, J.-Y. Girard, Y. Lafont, and L. Regnier, Eds., vol. 222. Cambridge University Press, 1995, pp. 225–247.

[12] L. Vaux, "λ-calcul différentiel et logique classique : interactions calculatoires," Thèse de Doctorat, Université de la Méditerranée, 2007.

[13] R. Di Cosmo, D. Kesner, and E. Polonovski, "Proof nets and explicit substitutions," *Mathematical Structures in Comp. Sci.*, vol. 13, no. 3, pp. 409–450, jun 2003.

[14] P. Tranquilli, "Intuitionistic differential nets and lambda calculus," 2008, theoretical Computer Science, to appear.

[15] V. Danos, "La logique linéaire appliquée à l'étude de divers processus de normalisation (principalement du λ-calcul)," Thèse de Doctorat, Université Paris VII, 1990.

[16] V. Danos and L. Regnier, "The structure of multiplicatives," *Archive for Mathematical Logic*, vol. 28, pp. 181–203, 1989.

Proofs

A. The λ-Calculus with Regions

Lemma 1.

- if $R;\Gamma \vdash M : A, e$ and $x \notin \mathrm{dom}(\Gamma)$, then $R;\Gamma, x : A \vdash M, e$;
- if $R;\Gamma \vdash V : A, \emptyset$ and $R; x : A, \Gamma \vdash M : B, e$, then $R;\Gamma \vdash M\{V/x\} : B, e$;
- if $R; \vdash M, S : A$ and $M, S \to M', S'$, then $R; \vdash M', S' : A$.

Proof: Standard inductions on the height of the type derivation. For the second point for each axiom $R;\Gamma', x : A \vdash x : A, \emptyset$ one has that $\Gamma' \supseteq \Gamma$, so an application of the first point can yield $R;\Gamma' \vdash V : A$ which thus can replace the axiom. The third point is also proved by induction, where the case when reducing $(\lambda x.M)V$ follows from the second point. For memory access operations subject reduction is straightforward. ■

Lemma 3. If M, S is a state, then either it is a result V, S or $M, S \to M', S'$ with M', S' a state too. Moreover $\mathrm{dom}(S') - \mathrm{ar}(M') = \mathrm{dom}(S) - \mathrm{ar}(M)$; in particular if M, S is external so is M', S'.

Proof: Suppose M is not a result. Let $\mu(M, S) := \mathrm{dom}(S) - \mathrm{ar}(M)$: M, S is a state (resp. an external state) iff μ is defined (i.e. positive) and contains the effects of M (resp. if $\mu(M, S) = []$ with no effects). We will show by induction on M that $M, S \to M', S'$ with $\mu(M', S') = \mu(M, S)$. Checking that the reduct is typed with the same effects of M will be omitted as it is guaranteed by Lemma 1. The cases where there is a subterm of M which is not a value and should be evaluated are handled simlarly to the proof of Lemma 4. For example, if $M = VN$ then N, S is an (external) state and not a result, so $N, S \to N', S'$ and thus $M, S \to VN', S'$, an (external) state.

Particular care must only be reserved for $M = \epsilon r.N$. In this case , if N is typed with effects e then $\epsilon r.N$ will be with $f \supseteq e \setminus \{r\}$. Now $\mu(N, S) = \mu(\epsilon r.N, S) + [r]$, so $e \subseteq f \cup \{r\} \subseteq |\mu(N, S)|$ and N, S is a state (necessarily non-external). By inductive hypothesis $N, S \to N', S'$, so $\epsilon r.N, S \to \epsilon r.N', S'$ with μ invariant.

Suppose then that M has no direct subterm to be evaluated. If $M = V_1 V_2$, then $V_1 = \lambda x.N$ as it must be closed and typed with a function space. Then $M, \varepsilon \to N\{V_2/x\}, \varepsilon$, which and μ cannot have changed as value substitution leaves $\mathrm{ar}(N) = []$ unchanged, and the store is the same.

If $M = \pi r{\Leftarrow}V.N$ then $M, S \to \epsilon r.N, S, r{\Leftarrow}V$. μ remains constant as the ϵr balances the new value in the store. If on the other hand $M = \epsilon r.V$, then $r \in \mathrm{ar}(\epsilon r.V) \leq \mathrm{dom}(S)$ and thus $S = T, r{\Leftarrow}U$ and $M, S \to V, T$ which is a result (with $\mu(V, T) = \mathrm{dom}(T) = \mathrm{dom}(S) - [r] = \mu(M, S)$).

Finally, if $M = \mathtt{get}(r)$ or $\mathtt{set}(r, V)$, then M must be typed with an effect containing r, so $r \in \mu(M, S) = \mathrm{dom}(S)$ and the reduction of the two can take place giving in fact a result. The domain of the store and the inexistent active regions remain unchanged. ■

Lemma 4. For every derivation of $R;\Gamma \vdash_{\mathrm{s}} M : A, e$ there is one of the same assertion where the subtyping rule appears only under axioms (namely the rules for variable and for `get`).

Proof: Standard induction on the size of the starting inference. By size we take the number of non-subtyping rules, and we assume all adiacent subtyping rules are merged together into one. Supposing the inference ends with a subtyping rule, the proof is split by cases on the rule immediately preceding it. If the subtyping rule can be pushed up on the premises, the inductive hypothesis yields the result. The only interesting cases are application and abstraction. Omitting R and Γ, one transforms the inferences in the following way.

$$\dfrac{\dfrac{\vdash_{\mathrm{s}} M : A \xrightarrow{e_3} B, e_1 \quad \vdash_{\mathrm{s}} N : A, e_2}{\vdash_{\mathrm{s}} MN : B, e_1 \cup e_2 \cup e_3}}{\vdash_{\mathrm{s}} MN : C, e_1 \cup e_2 \cup e_3}$$

$$\mapsdown$$

$$\dfrac{\dfrac{\vdash_{\mathrm{s}} M : A \xrightarrow{e_3} B, e_1}{\vdash_{\mathrm{s}} M : A \xrightarrow{e_3} C, e_1} \quad \vdash_{\mathrm{s}} N : A, e_2}{\vdash_{\mathrm{s}} MN : C, e_1 \cup e_2 \cup e_3}$$

$$\dfrac{\dfrac{\begin{array}{c}\delta\\ \vdots\\ x : A \vdash_{\mathrm{s}} M : B, e\end{array}}{\vdash_{\mathrm{s}} \lambda x.M : A \xrightarrow{e} B, \emptyset}}{\vdash_{\mathrm{s}} \lambda x.M : A' \xrightarrow{e'} B', \emptyset} \quad \mapsto \quad \dfrac{\dfrac{\begin{array}{c}\delta'\\ \vdots\\ x : A' \vdash_{\mathrm{s}} M : B', e\end{array}}{x : A' \vdash_{\mathrm{s}} M : B', e'}}{\vdash_{\mathrm{s}} \lambda x.M : A' \xrightarrow{e'} B', \emptyset}$$

where δ' is δ where all axioms introducing $x : A$ are turned into ones introducing A' followed by a subtyping rule giving $x : A$ (as $A' \xrightarrow{e'} B' \geq A \xrightarrow{e} B$ implies $A' \leq A$). Then δ' has the same size of δ and inductive hypothesis applies. ■

Lemma 5. If $R;\Gamma \vdash_{\mathrm{s}} M : A, e$ is derivable, then there is M' with $M \overset{*}{\leadsto} M'$ with $R;\Gamma \vdash M' : A, e$ without subtyping.

Proof: One starts by applying Lemma 4 to get an inference where subtypings are just below axioms. Then for every $B \leq C$ let $F_{B,C}[\,]$ be the one-hole context (not an evaluation one) defined inductively by $F_{B,B}[\,] := [\,]$ and

$$F_{B \xrightarrow{e} C, D \xrightarrow{f} E}[\,] := \lambda y.(\lambda z.F_{C,E}[z])([\,]F_{D,B}[y]),$$

with y and z fresh. Then by induction on the subtyping one sees that $x : B \vdash F_{B,C}[x] : C, \emptyset$ and $x \overset{*}{\leadsto} F_{B,C}[x]$. Now substituting every variable occurrence x in M (resp. every $\mathtt{get}(r)$ occurrence) with $F_{B,C}[x]$ (resp. with $(\lambda x.F_{B,C}[x])\,\mathtt{get}(r)$) where B is its type in the context at the moment of introduction, (resp. the type of r) and C the one after any subtyping following its introduction, we get an M' with $M \leadsto M'$ and the same type under the $\vdash$ inference. ■

Lemma 6. If $M \leadsto M'$ and $S \leadsto S'$, then $M, S \Downarrow V, S_0$ iff $M', S' \Downarrow V', S'_0$ with $V \leadsto V'$ and $S' \leadsto S'_0$.

Proof: First suppose $M, S \Downarrow V, S_0$ and let us reason by induction on the length of the normalization. If M is itself a value we are done. Let us split by cases otherwise. All cases where M is not the redex fired immediately are handled by the compatibility of $\leadsto$.

If $M = \mathtt{set}(r, U)$ and $\epsilon r.U$ we are easily done, as both sides reduce in just one step.

If $S = r{\Leftarrow}U, T$ and $M = \mathtt{get}(r)$, then $S' = r{\Leftarrow}U', T'$ with $U \leadsto U'$, and M' is either $\mathtt{get}(r)$, or $I\,\mathtt{get}(r)$. In both cases after one or two step we get to U' and we are done.

If $M = (\lambda x.M_1)V_1 \to M_1\{V_1/x\}$ then $M' = V_2'V_1'$ where $V_1 \rightsquigarrow V_1'$, and either $V_2' = \lambda x.M_1'$ with $M_1 \rightsquigarrow M_1'$, or $V_2' = \lambda y.I((\lambda x.M_1')y)$ with $M_1' = M_1$ (and in particular $M_1 \rightsquigarrow M_1'$). In any case by inductive hypothesis, as $M_1'\{V_1'/x\} \leftsquigarrow M_1\{V_1/x\} \Downarrow V$ (turning the store S into S'), we have $M_1'\{V_1'/x\} \Downarrow V'$ as by the thesis. In both cases we conclude: in particular in the latter one we have $S', V_2'V_1' \to S', I((\lambda x.M_1')V_1) \xrightarrow{*} S_0'IV' \to S_0'V$.

For the if part, the reasoning follows in reverse the steps taken for the only if part. ■

Lemma 8. For every external state M, S, $(M,S)^\nu$ is defined and is the unique program such that $(M,S)^\nu, \epsilon \xrightarrow{*} M, S$ using only ν-steps.

Proof: By induction on M. If $(M,S)^\nu = M$ we notice that $S = \varepsilon$ (as $\mathrm{ar}(M) = [\,]$) and we are done. If $M = V_1N_2$ with V_1 a value, then N_2, S is an external state, and by inductive hypothesis $N_2, S \leftarrow (N_2, S)^\nu, \varepsilon$ by expanding νr's. Then as $N_1(N_2,S)^\nu, \varepsilon \to N_1N_2, S$ we are done. The other case for application, for $\nu r{\Leftarrow}N_1.N_2$ and for $\mathtt{set}(r, M)$ are similar. For $M = \epsilon s.N$, it must be the case that $S = T, s{\Leftarrow}V$ (as $s \in \mathrm{ar}(M) = \mathrm{dom}(S)$). As N, T is an external state inductive hypothesis gives $(N,T)^\nu, \varepsilon \to N, T$ by ν-steps. Then $(M,S)^\nu = \nu s{\Leftarrow}V.(N,T)^\nu$, apart from being defined, has the desired property: $\nu s{\Leftarrow}V.(N,T)^\nu, \varepsilon \to \epsilon s.(N,T)^\nu, s{\Leftarrow}V \xrightarrow{*} \epsilon s.N, T, s{\Leftarrow}V$.

Uniqueness follows from the fact if two programs reduce to the same external state M, S just by ν-steps, then such reductions must be exaclty the same: their number is the cardinality of S, the position of reduced νr's are marked by ϵr's in M, and the values assigned by the reduced νr's are determined by their order in S. ■

Lemma 9. Let (M,S) be an external state. Then $(M,S) \to (M',S')$ with a non-ν-step iff $(M,S)^\nu \rightarrowtail (M',S')^\nu$. If $(M,S) \to (M',S')$ with a ν-step, then $(M,S)^\nu = (M',S')^\nu$.

Proof: Let $E[R] = M$ with E a regular evaluation context and R the redex fired in the reduction. Then there is a ν-context E^ν such that $E^\nu[R] = (M,S)^\nu$, and such that E^ν differs from E by having νr's in place of ϵr's. Moreover if S' is the sequence of $r{\Leftarrow}V$ obtained from each context $\nu r{\Leftarrow}V.F$ building E^ν, starting from the hole up, we see that $S' = S$ (all this is easily shown by induction on E). In particular the first $r{\Leftarrow}V$ in S for any given r is characterized by having $E^\nu = E'[\nu r{\Leftarrow}V.E'']$ with $r \notin \mathrm{PR}(E'')$. In fact such construction can also be reversed: every ν-context F such that $(M,S)^\nu = F[R]$ with R a redex turns to a regular context F^ϵ by stripping all ν's, so both sides of the equivalence are valid. ■

B. Types and effects into Monads

Lemma 11. We have the following properties on the terms introduced in Figure 5.

- If $S : P_{e\setminus\{r\}}$, $V : X_r$ is a value (so $\langle V\rangle$ is an $\{r\}$-store) and $M : T_{e\cup\{r\}}(A)$ then $\mathtt{n}_r^e\, V M S \xrightarrow{*} \langle S', U\rangle$ iff $M(S + \langle V\rangle) \xrightarrow{*} \langle S' + \langle V'\rangle, U\rangle$ for some $V' : X_r$.
- If $S : P_{e\cup f}$, $M : T_e(A)$ then $\mathtt{cast}_{e,f}\, M S \xrightarrow{*} \langle S', U\rangle$ iff $MS|_e \xrightarrow{*} \langle S'|_e, U\rangle$ and $S'|_{f\setminus e} = S|_{f\setminus e}$.
- If $S : P_{e\cup f}$, $M : T_e(A)$, $x : A \vdash N : T_f(B)$, then $(\mathtt{let}_{e,f}\, x\ \mathtt{be}\ M\ \mathtt{in}\ N)S \xrightarrow{*} \langle S', U\rangle$ iff $MS|_e \xrightarrow{*} \langle T, V\rangle$ and $N\{V/x\}(T|_{f\cap e} + S|_{f\setminus e}) \xrightarrow{*} \langle S'|_f, U\rangle$.

Proof: For $\mathtt{n}_r^e\, V M S$, it reduces to $F(M(S + \langle V\rangle))$ where $F = \lambda\langle s_1, v\rangle.\langle \pi^{e\cup\{r\}}_{e\setminus\{r\}} s_1, v\rangle$. This reduces to some $F\langle S' + \langle V'\rangle, U\rangle$ iff $M(S + \langle V\rangle)$ evaluates to that value, and then $F\langle S' + \langle V'\rangle, U\rangle \xrightarrow{*} \langle S', U\rangle$.

$\mathtt{cast}_{e,f}\, M S$ follows similarily: it reduces to $(\lambda\langle s_1, v\rangle.\mathtt{upd}_{e,f}\, s_1 S, v)(MS|_e)$, which goes to $\langle S'|_e + S|_{f\setminus e}, U\rangle$ iff $MS|_e$ evaluates to $\langle S'|_e, U\rangle$.

The third point comines the expected behaviour of the implemented $\mathtt{let}$ construct with the one for $\mathtt{cast}$. Indeed $(\mathtt{let}_{e,f}\, x\ \mathtt{be}\ M\ \mathtt{in}\ N) \xrightarrow{*} (\lambda\langle s_1, x\rangle.\,\mathtt{cast}_{f,e}\, N s_1)(\mathtt{cast}_{e,f}\, M S)$, which by the above reduces to $\mathtt{cast}_{f,e}\, N\{V/x\}(T + S|_{f\setminus e})$ iff $MS|_e \xrightarrow{*} \langle T, V\rangle$, and it will then reduce to $\langle S', U\rangle$ iff $N\{V/x\}(T + S|_{f\setminus e})|_f = N\{V/x\}(T|_{e\cap f} + S|_{f\setminus e})$ will also. ■

Proposition 12. R° is solvable iff the stratification condition $R \vdash$ holds.

Proof: Let Ξ denote either $R \vdash A$ or $R \vdash$, and let $|\Xi|$ be defined by $|R \vdash\,| := \sum_{r\in\mathrm{dom}(R)}(1 + |R(r)|)$ and $|R \vdash A| := |R \vdash\,| + |A|$, with the size $|\,.\,|$ defined on types as usual. Resoning by induction on $|\Xi|$, we show that Ξ is derivable iff R° is solvable and $\mathrm{FV}(A) \subseteq \mathrm{dom}(R^\circ)$, if A is present. Let us reason by cases on Ξ.

$\boldsymbol{\Xi = R \vdash 1}$: inductive hypothesis yields that $R \vdash$ iff R° solvable. As $\mathrm{FV}(1) = \emptyset$ and $R \vdash$ iff $R \vdash 1$ we are done.

$\boldsymbol{\Xi = R \vdash A \xrightarrow{e} B}$: Ξ is derivable iff $R \vdash A$, $R \vdash B$ and $e \subseteq \mathrm{dom}(R)$, with the latter equivalent to $\forall r \in e : X_r \in \mathrm{dom}(R^\circ)$. Then inductive hypothesis gives that Ξ is derivable iff R° is solvable and $\mathrm{FV}((A \xrightarrow{e} B)^\circ) = \mathrm{FV}(A) \cup \mathrm{FV}(B) \cup \{\, X_r \mid r \in e \,\} \subseteq \mathrm{dom}(R^\circ)$.

$\boldsymbol{\Xi = R \vdash}$: if $R = \emptyset$ there is nothing to prove, as both ends of the equivalence are always true. Otherwise suppose first that $R \vdash$ is derivable, so that $R = R_0, r : A$ and $R_0 \vdash A$. By inductive hypothesis (as $|R_0, r : A \vdash\,| = |R_0 \vdash A|{+}1$) R_0° is solvable and $\mathrm{FV}(A^\circ) \subseteq \mathrm{dom}(R_0^\circ)$, which entails that $\sigma_{R_0^\circ}(A^\circ)$ is closed. Assigning such a value to X_r gives then a solution for R°.

Let us start on the other hand with R° solvable. Take then a region $r \in \mathrm{dom}(R)$ so that $\sigma_{R^\circ}(R(r)^\circ)$ is maximal in size, and consider R_0 to be R restricted to $\mathrm{dom}(R)\setminus\{r\}$. Now we see that $X_r \notin \mathrm{FV}(R(s)^\circ)$ for all $s \in \mathrm{dom}(R)$: if it was the case, then $\sigma_{R^\circ}(R(s)^\circ)$ would contain $\sigma_{R^\circ}(R(r)^\circ)$ as a proper subformula, which would violate maximality. This entails on one side that σ_{R° provides a solution also for R_0°, once restricted to its domain; on the other that $\mathrm{FV}(R(r)^\circ) \subseteq \mathrm{dom}(R^\circ) \setminus \{X_r\} = \mathrm{dom}(R_0^\circ)$. By inductive hypothesis we conclude that $R_0 \vdash R(r)$, from which $R \vdash$ can be inferred. ■

Proposition 13. Let $\Gamma^\circ(x) := (\Gamma(x))^\circ$. Then:

- if $R; \Gamma \vdash M : A, e \mapsto M'$ then $\Gamma^\circ \vdash M' : T_e(A^\circ)$ is derivable in $\Lambda_\times$ modulo $\equiv_{R^\circ}$; in particular programs $M, \varepsilon : A, \emptyset$ are mapped to closed terms of type A°;
- if $R; x : A\Gamma \vdash M : A, e \mapsto M'$ and $R; \Gamma \vdash V : A, \emptyset \mapsto [V']$ then $R; \Gamma \vdash M\{V/x\} : A, e \mapsto M'\{V'/x\}$.

Proof: Straightforward induction on the derivation, using the derived typing rules shown in Figure 5. The equivalence $\equiv_{R^\circ}$ is used with `set` and `get`, to equate A° to X_r if $r : A \in R$. For the point about substitution, it suffices to see that replacing all axioms $x : A \vdash x : A \mapsto [x]$ with $\vdash V : A \mapsto [V']$ gives exactly what needed in the end. ■

Theorem 14. If $R; \vdash M, \varepsilon : A, \emptyset \mapsto M'$ is a Λ_{reg} program with its associated M' term, then M evaluates to the value V, ε iff $M' \xrightarrow{*} V'$ with $V \mapsto [V']$.

Proof: Given a Λ_{reg} store S (typed according to R) and a set of regions $e \subseteq |\operatorname{dom}(S)|$, there is a unique partition of $S = S_1, S_2$ so that $|\operatorname{dom}(S_1)| = e$ and S_1 is made by singletons (in a way, $\operatorname{dom}(S_1)$ *is* e). Let us define $\mathrm{t}_e(S)$ as the translation of such an S_1 into a $\Lambda_\times$ e-store: for each $r{\Leftarrow}V_r \in S_1$ (V_r is uniquely determined by r) we have that $R; \vdash V_r : R(r), \emptyset \mapsto [V'_r]$ for some $\Lambda_\times$ value $V'_r : R(r)^\circ = X_r$, so we define $\mathrm{t}_e(S)$ as $\langle V'_r \rangle_{r \in e}$. Clearly $\mathrm{t}_e(\varepsilon) = \langle\rangle$; moreover if $e \subseteq f$ then $\mathrm{t}_f(S) = \mathrm{t}_e(S)|_f$.

Now we will prove that every Λ_{reg} state M, S with $R; \vdash M : A, e \mapsto M'$ evaluates to a result V, S' iff $M' \, \mathrm{t}_e(S)$ evaluates to $\langle T, V' \rangle$ with $T = \mathrm{t}_e(S')$ and $R; \vdash V : A, \emptyset \mapsto [V']$ (recall that by the condition imposed on states $e \subseteq |\operatorname{dom}(S)|$). In particular for a program M, ε this amounts to saying that M evaluates to V iff $M' \langle\rangle \xrightarrow{*} \langle \langle\rangle, V' \rangle$ with $V \mapsto [V']$, which is what requested.

Proof of only if: We will proceed by induction on the length of the reduction of M, S: for the base case in which M is already a value then the only difference with the expected result is that the type derivation might have added effects in the end, i.e. $e \neq \emptyset$. Supposing no consecutive rules adding effects are used, this amounts to having $M' = \mathtt{cast}_{\emptyset,e} [V']$ with $M : A, \emptyset \mapsto [V']$. Then $M' \, \mathrm{t}_e(S) \xrightarrow{*} \langle \mathrm{t}_e(S), V' \rangle$ which is what required.

For the inductive step we again settle first the case where the last rule of the derivation of M just adds effects, $M' = \mathtt{cast}_{e',e} M''$ with $R; \vdash M : A, e' \mapsto M''$. Then Lemma 11 reduces the problem to $M \mapsto M''$, with the same starting term as before. Now we go on depending on the shape of M.

$\boldsymbol{M = N_1 N_2}$: $M' = \mathtt{let}_{e_1, e_2 \cup e_3} f \, \mathtt{be} \, N'_1 \, \mathtt{in} \, \mathtt{let}_{e_2, e_3} a \, \mathtt{be} \, N'_2 \, \mathtt{in} \, f a$. As M, S evaluates, then $N_1, S \xrightarrow{*} \lambda x.N_3, S''$, $N_2, S'' \xrightarrow{*} U, S'''$ and $N_3\{U/x\}, S''' \xrightarrow{*} V, S'$, all with reductions strictly shorter than the global one (as the sum must include also the β-reduction). So inductive hypothesis gives $N'_1 \, \mathrm{t}_e(S)|_{e_1} \xrightarrow{*} \langle \mathrm{t}_e(S'')|_{e_1}, [\lambda x.N'_3] \rangle$ with $N_3 \mapsto N'_3$ and $N_2 \, \mathrm{t}_e(S'')|_{e_2} \rightarrow * \langle \mathrm{t}_e(S''')|_{e_2}, [U'] \rangle$ with $U \mapsto [U']$. By Proposition 13 $N_3\{U/x\} \mapsto N'_3\{U'/x\}$, so a third application of inductive hypothesis yields that $N'_3\{U'/x\} \, \mathrm{t}_e(S''')|_{e_3} \xrightarrow{*} \langle \mathrm{t}_e(S')|_{e_3}, V' \rangle$ with $V \mapsto [V']$. Applying two times Lemma 11 gives what needed.

$\boldsymbol{M = \nu r{\Leftarrow}N_1.N_2}$. We will have a reduction $N_1, S \xrightarrow{*} V_1, S''$, then a reduction of $N_2, r{\Leftarrow}V_1, S'' \xrightarrow{*} V, r{\Leftarrow}V_2, S'$. A value for r must remain in the store because of Lemma 3, so that we have the final step $\epsilon r.V, r{\Leftarrow}V_2, S' \rightarrow V, S'$. By inductive hypothesis we get $N'_1 \, \mathrm{t}_e(S)|_{e_1} \xrightarrow{*} \langle \mathrm{t}_e(S'')|_{e_1}, V'_1 \rangle$ and $N'_2 \, \mathrm{t}_{e_2 \cup \{r\}}(r{\Leftarrow}V_1, S'') = N'_2(\mathrm{t}_{e_2 \setminus \{r\}}(S'') + \langle V_1 \rangle) \xrightarrow{*} \langle \mathrm{t}_{e_2 \setminus \{r\}}(S') + \langle V'_2 \rangle, V' \rangle$. Then applying two times Lemma 11 yields first that $\mathtt{n}_r^{e_2} \, V'_1 N'_2 \, \mathrm{t}_e(S'')|_{e_2 \setminus \{r\}} \xrightarrow{*} \langle \mathrm{t}_e(S')|_{e_2 \setminus \{r\}}, V' \rangle$ and then that the `let` construction of M' gives that $M' \, \mathrm{t}_e(S) \xrightarrow{*} \langle \mathrm{t}_e(S'), V' \rangle$, all with $V \mapsto [V']$.

$\boldsymbol{M = \mathtt{set}(r, N)}$ (with $N \mapsto N'$): then $N, S \xrightarrow{*} U_1, S''$ and with a further step $M, S'' \xrightarrow{*} \langle\rangle, r{\Leftarrow}U_1, T''$ where in fact $S'' = r{\Leftarrow}U_2, T''$ (Lemma 3). By inductive hypothesis we obtain $N' \, \mathrm{t}_e(S) \xrightarrow{*} \langle \mathrm{t}_{e \setminus \{r\}}(T'') + \langle U'_2 \rangle, U'_1 \rangle$ with $U_i \mapsto U'_i$. Now as $\mathtt{s} \, U'_1 U'_2 \xrightarrow{*} \langle U'_1, \langle\rangle \rangle$ Lemma 11 for the `let` construction yields $M' \, \mathrm{t}_e(S) = (\mathtt{let}_{e,\{r\}} v \, \mathtt{be} \, N' \, \mathtt{in} \, \mathtt{s} \, v) \, \mathrm{t}_e(S) \xrightarrow{*} \langle \mathrm{t}_{e \setminus \{r\}}(T'') + \langle U'_1 \rangle, \langle\rangle \rangle = \langle \mathrm{t}_e(S'), \langle\rangle \rangle$, and $\langle\rangle \mapsto [\langle\rangle]$.

$\boldsymbol{M = \mathtt{get}(r)}$: then $S = r{\Leftarrow}V, T$ (with $V \mapsto [V']$) and $M, S \rightarrow V, S$ and $M' \, \mathrm{t}_{\{r\}}(S) = \mathtt{g} \, V' \rightarrow \langle V', V' \rangle$ and we are done.

Proof of if: In fact the opposite direction retraces exactly the same steps, only doing an induction on the length of the reduction of $M' \, \mathrm{t}_e(S)$ and using the opposite directions of the equivalences of Lemma 11. ■

C. Types and effects into linear logic

Lemma 19. In *untyped* LL (and so also in presence of recursive types) one has the following properties.

- $\xrightarrow{s}$ is strongly normalizing;
- $\rightarrow$ is confluent;
- π is strongly normalizing for $\rightarrow$ iff it is weakly so.

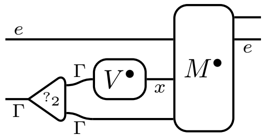
Proof: The reduction $\xrightarrow{s}$ is strongly normalizing by the finite developments theorem, which with equivalences has been shown in [14]. It follows with a quick check that $\xrightarrow{s}$ is confluent. Now m and e are trivially strongly confluent, and the three strongly commute with one another (e.g. $\xleftarrow{e}\xrightarrow{s} \subseteq \xrightarrow{s=} e$), so the three together are confluent.

For strong normalization, there is a proof with a slightly different syntax in

> D. de Carvalho, M. Pagani, and L. Tortora de Falco, "A semantic measure of the execution time in linear logic," 2008, to appear on Theoretical Computer Science.

One can alternatively show it by introducing a counter increasing at every m or e reduction, which cannot decrease at s ones. Here one proves that reduction is still confluent, and conclude: the counter of the normal form bounds the number of total e or m steps that can be done, and there cannot be an infinite number of s steps either. ■

Lemma 22 (substitution).



$\xrightarrow{s*} (M\{V/x\})^\bullet$.

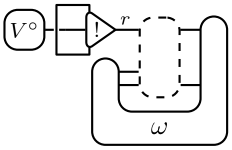
Proof (sketch): After noticing that values are always boxes, the proof follows by induction. Equivalence on contractions commuting with box borders are needed to make the trailing contractions enter boxes in the $\lambda x.M$ case. ■

Theorem 23 (Simulation). Let M, S be an external state. Then:

- if $M, S = V, \varepsilon$ is a value, then $(V, \varepsilon)^\bullet$ is in 0-depth normal form.
- if $M, S \rightarrow M', S'$ with a non-ν-step then $(M, S)^\bullet \xrightarrow{+} (M', S')^\bullet$ with exactly one dereliction on box step;
- if $M, S \rightarrow M', S'$ with a ν-step, then $(M, S)^\bullet = (M', S')^\bullet$.

Proof: The first and the last points are straightforward from the definition. For the second we show that if $M \rightarrowtail N$ then $M^\bullet \xrightarrow{+} N^\bullet$ with exactly one dereliction on box reduction, and conclude by Lemma 9.

First notice that ν-evaluation contexts translate to proof net contexts with the hole at depth 0. Moreover we can prove that if $r \notin \mathrm{PR}(F[\ ])$ then the context $(\nu r{\Leftarrow}V.F[\,])^\bullet$ is of the form



i.e. the translation of V is directly available to the hole. This follows from the definition of ν-evaluation contexts: all the parts of the term the during the translation would be on top (or on the left) of the hole are necessarily values. In particular the effect r is present in them only as a wire (via the $\equiv$ relation). The only exception would be another $\nu r{\Leftarrow}U$, which would interrupt it, but it is explicitly forbidden by $r \notin \mathrm{PR}(F)$. We can thus suppose such part of the net when carrying out the cases.

Indeed it suffices now to reduce each case separately, employing Lemma 22 for the $(\lambda x.M)V$ reduction. ■

Theorem 24. If $(M,S)^\bullet \xrightarrow{*} \pi$, with π a 0-depth normal form, then there is a value V such that $M,S \to V,\varepsilon$ and $\pi = V^\bullet$.

Proof: By Lemma 19 $(M,S)^\bullet$ is strongly normalizing for the 0-depth reduction we employ. So in particular applying Theorem 23 we obtain that $(M,S)^\bullet$ normalizes to $V^\bullet$ (as M,S cannot make infinite ν-steps). Unicity of 0-depth normal form entails $\pi = V^\bullet$. ■