A DISSECTION OF L

ARNAUD SPIWACK

Inria – Ens – Paris, France
e-mail address: arnaud@spiwack.net

Abstract. This article describes a one-sided variant of system L whose typing corresponds to linear sequent calculus and its application. A polarised version of the system is introduced to control the reduction strategy. The polarised type system is then extended to dependent linear types. The system with dependent type supports dependent elimination of positive connectives.

Introduction

It is my strong belief that, if natural deduction has enlightened our path through programming language design in the past half-century, sequent calculus will have a preponderant role in the next one. Ever since the turn of the millennium, new results have been pointing towards this conclusion.

Sequent calculus has been linked to strategy of evaluation [CH00, DL06, Zei08]. Lengrand, Dyckhoff and McKinna have shown [LDM10] that the edge which sequent calculus has in proof search extends to the dependently typed case. The quite popular bidirectional type-checking discipline is well-modelled by sequent calculus [DK13]. Sequent calculus is also know to be connected to program optimisation [Mar95].

Another tool which is becoming important for programming languages is linear logic. It offers an alternative to commutative monads to represent effects [BW96], there is a fragment, known as effect calculus [EMgS09], which can be use to encode arbitrary monadic effects. An outstanding recent manifestation of linear types is [KB11] where linear typing is leveraged for graphical interface programming.

The main technology for sequent-calculus based programming languages originates in [CH00] where it was called $\bar{\lambda}\mu\tilde{\mu}$-calculus. It is more commonly known as system L (for Gentzen's name for sequent calculi: LK and LJ). The *tour de force* of L, in my opinion, is to provide a syntax for classical sequent calculus proofs in which, like λ-calculus for natural deduction, contraction and weakening are done through variables: a bound variable which isn't used is weakened, if it is used twice or more it is contracted. This is, I would argue, why it makes a good foundation for programming languages. There has also been linear incarnations of system L [MM09].

1998 ACM Subject Classification: F.3.3.
Key words and phrases: Sequent calculus, Dependent types, Linear logic, Polarised logic, System L.

LOGICAL METHODS
IN COMPUTER SCIENCE
DOI:10.2168/LMCS-???

Despite the growing importance of sequent calculus in programming language design, the literature on system L is scattered, and it still all too often feels impenetrable to outsiders. There is, however, an excellent introductory write-up by Philip Wadler on the classical version in [Wad03]. The most thorough studies of system L to date are Hugo Herbelin's habilitation [Her05] (see in particular the first chapter for historical notes) and Guillaume Munch-Maccagnoni's doctoral thesis [MM13].

My goal with this article is threefold. First, I aim at giving an overview of the system L literature. Second, I want to outline how sequent calculus is relevant for programming language design. Third and last, I give a proposal for a dependently typed linear sequent calculus based on system L.

Readers familiar with system L who wish to jump quickly to dependent types should skim through Figures 2 and 4 then go to Section 5.

Acknowledgement. This article may not have been written if it had not been for the long discussions I had with Pierre-Louis Curien, Hugo Herbelin, Guillaume Munch-Maccagnoni, and Pierre-Marie Pédrot.

1. Core L

Stripped down to its bare minimum, L appears as a very simple calculus whose syntax is made of two kinds of objects, terms (t, u, v, ...) and commands (c)

$$\begin{array}{lcl} t, u & ::= & x \mid \mu x.\, c \\ c & ::= & \langle\, t \mid u \,\rangle \end{array}$$

Together with reduction rules

$$\begin{array}{lcl} \langle\, t \mid \mu x.\, c \,\rangle & \rightsquigarrow & c[\, x \backslash t \,] \\ \langle\, \mu x.\, c \mid t \,\rangle & \rightsquigarrow & c[\, x \backslash t \,] \end{array}$$

The intent being that the vertical bar be read as commutative. We shall use it as such from now on.

Given a command c, the term $\mu x.\, c$ can be thought as "let x be the current context, do c". Conversely, for t and u, two terms $\langle\, t \mid u \,\rangle$ runs t with context u (or symmetrically, u with context t) it is read "t against u".

The reduction rules look quite similar to β-reduction, however this core calculus does not nearly have the power of λ-calculus. Indeed the fact that there are two kinds of object is crucial here: from a functional programming perspective, it is like if the only construct was let... in. Contrary to λ-calculus we have practically no computation power without additional constructs.

Nonetheless, we can already observe undesirable behaviours. For instance it is easy to cook up a non-terminating command $\langle\, \mu x.\, \langle\, x \mid x \,\rangle \mid \mu x.\, \langle\, x \mid x \,\rangle \,\rangle$. Much worse: any two commands c_1 and c_2 have a common antecedent $\langle\, \mu\alpha.\, c_1 \mid \mu\alpha.\, c_2 \,\rangle$ where α is fresh.

1.1. Typing as classical sequent calculus.

The original typing rules [CH00] for L corresponded closely to classical sequent calculus. We shall present, in this section, a one-sided variant of the classical core L. Therefore we shall require that every formula A has a dual $A^\perp$ such that $A^{\perp\perp} = A$.

The dualisation should not be understood as a connective – core L has none – but as an operation on types. Duality tracks positional information: a variable of type A on the right is

the same as a variable of type $A^\perp$ on the left. Therefore, there is no difference between either side and the variables can be arranged on a single side (or any convenient arrangement). In classical logic, negation, which *is* a connective is a reflection of dualisation, and it may be tempting – and is indeed often done – to forgo the negation altogether and keep only dualisation. In that case, the de Morgan laws are not just tautologies, they are *definitions* for the negation. An option which is more appealing from a programming language standpoint is to keep negation as a connective, give it a dual, and equip them both with introduction rules [Har12, Chapter 31][MM13, Chapter 4]. Duality, on the other hand, does not have introduction rules – unless we count the identity and cut rules as introduction rules.

To follow the tradition of programming languages, let us choose to leave all the variables on the left-hand side of the sequents. The tradition in proof theory, on the other hand, is rather to keep the variables on the right. The latter is better suited for interpretation in terms of proof nets [Gir96] or monoidal category [See89]. On the other hand, the left-handed variant works very well with L. Duality ensures that the difference is only in the eye of the reader: mathematically, these are all the same objects.

The typing of command is a simple assignment of types to its free variables: commands are self-contained, they don't have a "return type".

$$\Gamma \vdash c$$

Terms, on the other hand, have an intrinsic type in addition to the type assignment of their variables. From a logical standpoint, we shall need a distinguished formula which, since it does not correspond to a variable, we shall display it on the right-hand side of the sequent:

$$\Gamma \vdash t : A$$

Keep in mind, though, that a term typing judgement, despite the similarity with natural deduction sequents, is still a one-sided sequent. Indeed, a one-sided sequent is, by definition, a sequent where formulæ can flow freely between left and right.

The typing rules for variables and interaction correspond, on the logical side, to identity and cut respectively:

$$\frac{}{\Gamma, x{:}A \vdash x : A}\ \mathrm{id}$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : A^\perp}{\Gamma \vdash \langle\, t \,|\, u \,\rangle}\ \mathrm{cut}$$

The cut rule emphasises the rôle of of the dual type $A^\perp$ in the programming point of view: $A^\perp$ is the type of the contexts of A. Also, as $A^{\perp\perp} = A$, A is, conversely, the type of context of $A^\perp$: the idempotence of the duality operator goes hand to hand with the commutativity of the interaction.

The typing rule for μ abstraction does not correspond to a logical rule: from the point of view of sequent calculus, it corresponds to choosing a formula, and placing it to the right-hand side of the sequent to make it *active*.

$$\frac{\Gamma, x{:}A \vdash c}{\Gamma \vdash \mu x.\, c : A^\perp}\ \mu$$

From a programming point of view, $\mu x.\, c$ expects a value for x and continues with c. In other words, $\mu x.\, c$ interacts with values of type A: it has type $A^\perp$.

What makes is so that these typing rules correspond to classical logic is that *weakening* and *contraction* are admissible. In fact contraction is even derivable:

$$\frac{\dfrac{\Gamma, x{:}A, y{:}A \vdash c}{\Gamma, x{:}A \vdash \mu y.\, c : A^\perp} \quad \overline{\Gamma, x{:}A \vdash x : A}}{\Gamma, x{:}A \vdash \langle\, \mu y.\, c \,|\, x \,\rangle}$$

Weakening cannot be defined as such a macro, as the context only grows upwards. However, it is not difficult to check that any unused variable will be absorbed by the identity rules. Just like in natural deduction, this implicit weakening is what allows to give type to terms of the form $\mu\alpha.\, c$ for a fresh α.

As long as there is no type A such that $A^\perp = A$, the reduction of typed terms is terminating. In particular the aforementioned $\langle\, \mu x.\, \langle\, x \,|\, x \,\rangle \,|\, \mu x.\, \langle\, x \,|\, x \,\rangle \,\rangle$ cannot be typed. On the other hand, non-confluence is still as acute as in the untyped calculus: the untyped example translates to a cut between to weakenings. Let $\Gamma \vdash c_1$ and $\Gamma \vdash c_2$ be two commands typed in the same context, and α and β two fresh variable then we have the following derivation:

$$\frac{\dfrac{\Gamma, \alpha{:}A^\perp \vdash c_1}{\Gamma \vdash \mu\alpha.\, c_1 : A} \quad \dfrac{\Gamma, \beta{:}A \vdash c_2}{\Gamma \vdash \mu\beta.\, c_2 : A^\perp}}{\Gamma \vdash \langle\, \mu\alpha.\, c_1 \,|\, \mu\beta.\, c_2 \,\rangle}$$

Which we can conclude by weakening. So again, any two typed commands have a common antecedent. This is not specifically a property of L: in classical sequent calculus, a cut between two weakening exhibits the same non-confluent behaviour.

1.2. **Typing as linear sequent calculus.** In order to address the issue of non-confluence, we move away from classical logic and favour linear logic. The effect on the core calculus is minimal. The identity and cut rules are modified to prevent implicit weakening and conversion:

$$\frac{}{x{:}A \vdash x : A}\ \text{id}$$

$$\frac{\Gamma \vdash t : A \quad \Delta \vdash u : A^\perp}{\Gamma, \Delta \vdash \langle\, t \,|\, u \,\rangle}\ \text{cut}$$

The μ rule, on the other hand is left unchanged:

$$\frac{\Gamma, x{:}A \vdash c}{\Gamma \vdash \mu x.\, c : A^\perp}\ \mu$$

Would we want to limit the *exchange* rules, like in non-commutative logic [AR99], we would have to tweak the μ rule, but we will be content with treating the comma, in contexts, as commutative.

2. Linear L

As mentioned in the previous section, core L does not have all that much computing abilities. This is remedied by the introduction of logical connectives in types, and corresponding constructions in terms. In this section we shall extend L to reflect the whole range of linear logic connectives.

2.1. **Multiplicative fragment.** Throughout this article, the connectives' introduction rules come in two varieties: some are *value* constructors, while the introduction rules of the dual type are *computation* constructors, the syntax of which is inspired by pattern-matching.

For instance, the introduction rules of multiplicative connectives $A \otimes B$ and $A \parr B$ correspond, respectively, to a pair of two terms, and matching on a pair. They are written as follows:

$$t,\, u \quad ::= \quad \dots \mid (t,u) \mid \mu\,(x,u)\,.\,c$$

A benefit of this syntax is that the reduction rules are pretty easy to figure. Here is the case of pairs:

$$\left\langle\, (t,u) \,\middle|\, \mu\,(x,y)\,.\,c \,\right\rangle \quad \rightsquigarrow \quad c[\,x\backslash t, y\backslash u\,]$$

The typing rules follow naturally, keeping in mind that these are linear logic connectives in particular the pair (x,x) is ill-typed:

$$\frac{\Gamma \vdash t : A \quad \Delta \vdash u : B}{\Gamma, \Delta \vdash (t,u) : A \otimes B}\ \otimes$$

$$\frac{\Gamma, x{:}A, y{:}B \vdash c}{\Gamma \vdash \mu\,(x,y)\,.\,c : A^\perp \parr B^\perp}\ \parr$$

The multiplicative connectives alone bring a lot of expressiveness: linear λ-calculus is macro-expressible. The intuition comes from abstract machines: in abstract machines for λ-calculus, λ-terms interact with a stack. The idea is to represent the stack as iterated pairs; then the application must be a *push* operation, adding its second operand on the stack, while abstraction must *pop* the first stack element and substitute it in the function body.

This is an important insight, so it bears repeating: in L the stack is *first-class*. Programming in L is quite like programming directly an abstract machine. Giving a definition to application and abstraction hence amounts to solving the equations:

$$\langle\, t\ u \,|\, v\,\rangle \rightsquigarrow \left\langle\, t \,\middle|\, (u,v)\,\right\rangle$$
$$\left\langle\, \lambda x.\, t \,\middle|\, (u,v)\,\right\rangle \rightsquigarrow \left\langle\, t[\,x\backslash u\,] \,\middle|\, v\,\right\rangle$$

Solving these equations ensures that the reduction rules of L simulate β-reduction:

$$\left\langle\, (\lambda x.\, t)\ \ u \,\middle|\, v\,\right\rangle \rightsquigarrow \left\langle\, \lambda x.\, t \,\middle|\, (u,v)\,\right\rangle \rightsquigarrow \left\langle\, t[\,x\backslash u\,] \,\middle|\, v\,\right\rangle$$

The equation defining the application is of particular interest because it is a special case of an adjoint equation [MM13, Chapter 1], that is an equation of the form

$$\left\langle\, \varphi[t] \,\middle|\, u\,\right\rangle \rightsquigarrow \left\langle\, t \,\middle|\, \psi[u]\,\right\rangle$$

For two functions on terms φ and ψ compatible with substitution. One rôle played by the μ-binder is to solve these adjoint equation generically. Specifically, when ψ is completely specified, $\varphi = \mu\alpha.\ \langle\, t \,|\, \psi[\alpha]\,\rangle\,[t]$, for a fresh variable α:

$$\left\langle\, \varphi[t] \,\middle|\, u\,\right\rangle = \left\langle\, \mu\alpha.\ \left\langle\, t \,\middle|\, \psi[\alpha]\,\right\rangle \,\middle|\, u\,\right\rangle \rightsquigarrow \left\langle\, t \,\middle|\, \psi[u]\,\right\rangle$$

Accordingly, application $t\ u$ is defined as $\mu\alpha.\ \langle\, t \,|\, (u,\alpha)\,\rangle$: explicitely, the current context (*i.e.* stack) is named α and t is then run against (u,α), the current stack on top of which u has been pushed.

Abstraction does not enjoy such a generic description. However, the equation suggests that $\lambda x.\, t$ ought to be defined as $\mu\,(x,\alpha)\,.\,\langle\, t \,|\, \alpha\,\rangle$: the first element of the stack is called x and the rest α, for a fresh α, the body t of the abstraction is then run against α, the popped stack. Remember that x is binds a variable of t, so there is, implicitly a substitution in t. In

fact, since we are in linear logic, typing will impose that x is, in some sense, used exactly once in t. This definition is indeed a valid solution for the λ-abstraction equation.

Writing $A \multimap B$ for the linear arrow connective $A^\perp \parr B$, the typing rules of application and abstraction are familiar:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash t : A \multimap B \quad \dfrac{\Delta \vdash u : A \quad \overline{\alpha{:}B^\perp \vdash \alpha : B^\perp}\ \text{id}}{\Delta, \alpha{:}B^\perp \vdash (u,\alpha) : A \otimes B^\perp}\ \otimes}{\Gamma, \Delta, \alpha{:}B^\perp \vdash \left\langle t \,\middle|\, (u,\alpha) \right\rangle}\ \text{cut}}{\Gamma, \Delta \vdash \mu\alpha.\, \left\langle t \,\middle|\, (u,\alpha) \right\rangle : B}\ \mu}{\Gamma, \Delta \vdash t\ u : B}\ \text{definition}$$

$$\dfrac{\dfrac{\dfrac{\Gamma, x{:}A \vdash t : B \quad \overline{\alpha{:}B^\perp \vdash \alpha : B^\perp}\ \text{id}}{\Gamma, x{:}A, \alpha{:}B^\perp \vdash \langle t \,|\, \alpha \rangle}\ \text{cut}}{\Gamma \vdash \mu\,(x,\alpha)\,.\,\langle t \,|\, \alpha\rangle : A \multimap B}\ \parr}{\Gamma \vdash \lambda x.\, t : A \multimap B}\ \text{definition}$$

To the binary multiplicative connectives $A \otimes B$ and $A \parr B$ correspond the nullary 1 and $\bot$ respectively. The constructor $()$ of type 1 can be seen as an empty stacks, while $\mu().\, c$ is essentially the command c reified as a term: it discards the (anyway empty) stack and runs c. The reader may refer to Figure 1 – which also contains the upcoming additive connectives – for the typing rules of these nullary connectives.

2.2. **Additive fragment.** The additive connectives $A \oplus B$ and $A \& B$ bring something radically new to simply typed lambda calculus: case analysis. Case analysis can be encoded in pure λ-calculus, or in system F, but simply typed lambda calculus has no way of representing it. Their are two value constructors for $A \oplus B$, $1.t$ and $2.t$, respectively injective A and B into $A \oplus B$. Terms of type $A \& B$ are made with a computation constructor which branches on whether it is run against a $1.t$ or a $2.t$:

$$t,\, u \quad ::= \quad \ldots \mid 1.t \mid 2.t \mid \left\{\mu\,(1.x)\,.\,c_1\,,\ \mu\,(2.y)\,.\,c_2\right\}$$

The computation constructor $\{\mu\,(1.x)\,.\,c_1\,,\ \mu\,(2.y)\,.\,c_2\}$, is written as a set of two pattern-matching clauses, the appropriate clause is selected by the reduction rules:

$$\begin{array}{rcl} \left\langle 1.t \,\middle|\, \left\{\mu\,(1.x)\,.\,c_1\,,\ \mu\,(2.y)\,.\,c_2\right\} \right\rangle & \rightsquigarrow & c_1[\,x\backslash t\,] \\ \left\langle 2.t \,\middle|\, \left\{\mu\,(1.x)\,.\,c_1\,,\ \mu\,(2.y)\,.\,c_2\right\} \right\rangle & \rightsquigarrow & c_2[\,y\backslash t\,] \end{array}$$

In the typing rule for $\{\mu\,(1.x)\,.\,c_1\,,\ \mu\,(2.y)\,.\,c_2\}$, as required by linear logic, both branches share the same typing context:

$$\dfrac{\Gamma, x{:}A \vdash c_1 \quad \Gamma, y{:}B \vdash c_2}{\Gamma \vdash \left\{\mu\,(1.x)\,.\,c_1\,,\ \mu\,(2.y)\,.\,c_2\right\} : A^\perp \& B^\perp}\ \&$$

This typing rule can be framed in terms of the computational interpretation of L: linearity imposes that each variable must be used exactly once. Since one of the branches (say c_2) is dropped by the reduction rule, none of the variables of c_2 are used, therefore all of the variables in the context must be used exactly once in c_1. And symmetrically, they must be used exactly once in c_2.

In the dual typing rules, the missing type is materialised from thin air, since the corresponding branch is dropped by reduction:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash 1.t : A \oplus B} \oplus\mathrm{l}$$

$$\frac{\Gamma \vdash u : B}{\Gamma \vdash 2.u : A \oplus B} \oplus\mathrm{r}$$

There is a dual way to think about additive connectives, in which $A\&B$ is the type of *records* (with two fields labelled 1 and 2), and the injections $1.t$ and $2.t$ play the role of *projections*. Similarly to the case of λ-calculus in Section 2.1, we read $1.k$ as a stack which begins with the first projection, and $2.k$ with the second projection.

Hence the projections $t.1$ and $t.2$ push the appropriate instruction on the top of the stack and then run t. They are, like application, solutions to adjoint equations:

$$\begin{array}{rcl} t.1 & = & \mu\alpha.\, \langle\, t \,|\, 1.\alpha \,\rangle \\ t.2 & = & \mu\alpha.\, \langle\, t \,|\, 2.\alpha \,\rangle \end{array}$$

and the record $\{1 = t\,,\, 2 = u\}$ (whose syntax is inspired by ML) pattern-matches on the top of the stack:

$$\{1 = t\,,\, 2 = u\} \quad = \quad \big\{ \mu\,(1.\alpha)\,.\,\langle\, t \,|\, \alpha \,\rangle\ ,\ \mu\,(2.\alpha)\,.\,\langle\, u \,|\, \alpha \,\rangle \big\}$$

The main difference between multiplicative pairs and additive records – apart from the existence of projections in the latter – is that the former are values whereas the latter are computations. The best way to think about the distinction between values and computation may be to pretend that computations can have side effects. Under that view, a record stays unevaluated, and only the effects of the selected field will happen. On the other hand, a multiplicative pair (t, u) can be evaluated further, and the effects of both t and u will occur.

The binary $A \oplus B$ and $A\&B$ are accompanied by the nullary 0 – the empty type – and $\top$, whose sole constructor is $\{\}$ the empty case analysis (a.k.a *ex falso quodlibet*). The complete rules for the multiplicative and additive fragment of linear L can be found in Figure 1.

Notice how all the syntactic sugar so far defines computations. This is no coïncidence: computations, as their name suggests, is where the fun happens. In other words, we program functions, not pairs. The raw syntax of L is dry, and wholly unfamiliar; however, with a few macros, familiar programming construct emerge, and L looks like a normal programming language.

2.3. **Exponentials.** The typing rules presented so far are purely linear, in the sense that there is no contraction – or weakening – happening. For instance, the term (x, x) can be given a type in no context. Linear logic has the connective $!A$ to represent the formulæ which can be contracted and weakened on the left-hand side of sequents, and its dual $?A$ for the formulæ which can be contracted and weakened on the right-hand side of sequents. In short: a function of type $!A \multimap ?B$ can *use* any number of copies of its argument of type A, and may choose to return a B or not.

One way to incorporate the exponential connectives within linear L, proposed in [MM09], is the following rule:

$$\frac{\Gamma, x{:}!A, y{:}!A \vdash c}{\Gamma, x{:}!A \vdash c[\, y \backslash x \,]}$$

Syntax

$$\begin{array}{lcl} t,\, u & ::= & x \mid \mu x.\, c \mid (t,u) \mid \mu\,(x,u)\,.\,c \mid () \mid \mu().\, c \\ & & 1.t \mid 2.t \mid \{\mu\,(1.x)\,.\,c_1\,,\,\mu\,(2.y)\,.\,c_2\} \mid \{\} \\ c & ::= & \langle\, t \,|\, u \,\rangle \end{array}$$

Reduction

$$\begin{array}{lcl} \langle\, t \,|\, \mu x.\, c \,\rangle & \rightsquigarrow & c[\, x \backslash t\,] \\ \langle\, (t,u) \,|\, \mu\,(x,y)\,.\,c \,\rangle & \rightsquigarrow & c[\, x \backslash t, y \backslash u\,] \\ \langle\, () \,|\, \mu().\, c \,\rangle & \rightsquigarrow & c \\ \langle\, 1.t \,|\, \{\mu\,(1.x)\,.\,c_1\,,\,\mu\,(2.y)\,.\,c_2\}\,\rangle & \rightsquigarrow & c_1[\, x \backslash t\,] \\ \langle\, 2.t \,|\, \{\mu\,(1.x)\,.\,c_1\,,\,\mu\,(2.y)\,.\,c_2\}\,\rangle & \rightsquigarrow & c_2[\, y \backslash t\,] \end{array}$$

Derived syntax

$$\begin{array}{lcl} \lambda x.\, t & = & \mu\,(x,\alpha)\,.\,\langle\, t \,|\, \alpha\,\rangle \\ t\ u & = & \mu\alpha.\,\langle\, t \,|\, (u,\alpha)\,\rangle \\ \{1 = t\,,\, 2 = u\} & = & \{\mu\,(1.\alpha)\,.\,\langle\, t \,|\, \alpha\,\rangle\ ,\ \mu\,(2.\alpha)\,.\,\langle\, u \,|\, \alpha\,\rangle\} \\ t.1 & = & \mu\alpha.\,\langle\, t \,|\, 1.\alpha\,\rangle \\ t.2 & = & \mu\alpha.\,\langle\, t \,|\, 2.\alpha\,\rangle \end{array}$$

Typing

$$\frac{}{x{:}A \vdash x : A}\ \mathrm{id} \qquad \frac{\Gamma \vdash t : A \quad \Delta \vdash u : A^{\perp}}{\Gamma, \Delta \vdash \langle\, t \,|\, u\,\rangle}\ \mathrm{cut}$$

$$\frac{\Gamma, x{:}A \vdash c}{\Gamma \vdash \mu x.\, c : A^{\perp}}\ \mu$$

$$\frac{\Gamma \vdash t : A \quad \Delta \vdash u : B}{\Gamma, \Delta \vdash (t,u) : A \otimes B}\ \otimes \qquad \frac{\Gamma, x{:}A, y{:}B \vdash c}{\Gamma \vdash \mu\,(x,y)\,.\,c : A^{\perp} \parr B^{\perp}}\ \parr$$

$$\frac{}{\vdash () : 1}\ 1 \qquad \frac{\Gamma \vdash c}{\Gamma \vdash \mu().\, c : \perp}\ \perp$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash 1.t : A \oplus B}\ \oplus\mathrm{l} \qquad \frac{\Gamma, x{:}A \vdash c_1 \quad \Gamma, y{:}B \vdash c_2}{\Gamma \vdash \{\mu\,(1.x)\,.\,c_1\,,\,\mu\,(2.y)\,.\,c_2\} : A^{\perp} \& B^{\perp}}\ \&$$

$$\frac{\Gamma \vdash u : B}{\Gamma \vdash 2.u : A \oplus B}\ \oplus\mathrm{r}$$

No rule for 0 $\qquad \dfrac{}{\Gamma \vdash \{\} : \top}\ \top$

Derived typing rules

$$\frac{\Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \multimap B}\ \lambda \qquad \frac{\Gamma \vdash t : A \multimap B \quad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}\ \mathrm{app}$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash \{1 = t\,,\, 2 = u\} : A \& B}\ \mathrm{record} \qquad \frac{\Gamma \vdash t : A \& B}{\Gamma \vdash t.1 : A}\ \pi_1$$

$$\frac{\Gamma \vdash t : A \& B}{\Gamma \vdash t.2 : B}\ \pi_2$$

Figure 1: Multiplicative and additive fragment

This rule mimics accurately the traditional linear sequent calculus, however it does not follow the discipline of classical L, where contraction and weakening is only a matter of using a variable several times or not at all.

In order to retain this property, we choose to use the dyadic presentation of linear logic. This presentation, due to Andreoli [And92], classifies hypotheses according to whether they are *duplicable* or not, and renders the duplicable hypotheses in a separate context which behaves additively. The exponential connective $!A$ reflects duplicable hypotheses in the linear context.

The typing judgement in (dyadic) linear L of the form $\Xi \ ; \ \Gamma \vdash t : A$ and $\Xi \ ; \ \Gamma \vdash c$, where Γ is the linear context, and Ξ is the new duplicable context. Contraction and weakening are, like in classical L in Section 1.1, are implicit as the duplicable context is copied by cut and ignored by identity (more generally, the duplicable context is distributed on each premise of inference rules):

$$\frac{\Xi \ ; \ \Gamma \vdash t : A \quad \Xi \ ; \ \Delta \vdash u : A^\perp}{\Xi \ ; \ \Gamma, \Delta \vdash \langle \, t \, | \, u \, \rangle} \text{ cut}$$

$$\frac{}{\Xi \ ; \ x{:}A \vdash x : A} \text{ id}$$

The addition of a new context in our sequents requires the addition of a new *structural* rule to relate the new context to the old. A common form of structural rule for the duplicable context is the *copy*, or *absorbtion*, rule:

$$\frac{\Xi, A \ ; \ \Gamma \vdash A^\perp}{\Xi, A \ ; \ \Gamma \vdash}$$

By the copy rule, copies of a duplicable hypothesis A in the linear context, can be contracted into A. This rule does not follow the style of L as it would create a new kind of command. The equivalent rule originally proposed by Andreoli [And92], on the other hand, fits the design of L:

$$\frac{}{\Xi, x{:}A \ ; \ \cdot \vdash x : A} \text{ id'}$$

The duplicable identity rule makes it so that a duplicable hypothesis of type A can be used as a term of type A. This rule can simply be read as stating that variables in the duplicable context are duplicable variables. For example:

$$\frac{\dfrac{}{x{:}A \ ; \ \cdot \vdash x : A} \text{ id'} \quad \dfrac{}{x{:}A \ ; \ \cdot \vdash x : A} \text{ id'}}{x{:}A \ ; \ \cdot \vdash (x, x) : A \otimes A} \otimes$$

Variables in the duplicable context can, hence, be used freely, whereas variables of the linear context must be used in a linear fashion. Precisely what we expected to achieve.

As a sanity check, let us consider the derivation of the copy rule from the duplicable identity:

$$\frac{\Xi, x{:}A \ ; \ \Gamma \vdash t : A^\perp \quad \dfrac{}{\Xi, x{:}A \ ; \ \cdot \vdash x : A} \text{ id'}}{\Xi, x{:}A \ ; \ \Gamma \vdash \langle \, t \, | \, x \, \rangle} \text{ cut}$$

For the exponential connectives themselves, $!A$ has a value constructor, written $\lfloor t \rfloor$, which marks the term t as being duplicable (*i.e.* t does not use any linear variables), and $?A$ has a dual computation constructor:

$$t, u \quad ::= \quad \ldots \mid \lfloor t \rfloor \mid \mu \lfloor x \rfloor \, . \, c$$

SYNTAX

$$\begin{array}{lll} t,\, u & ::= & \ldots \mid \lfloor t \rfloor \mid \mu \lfloor x \rfloor . \, c \\ c & ::= & \langle \, t \mid u \, \rangle \end{array}$$

REDUCTION

$$\begin{array}{lll} \langle \, t \mid \mu x. \, c \, \rangle & \rightsquigarrow & c[\, x \backslash t \,] \\ \langle \, (t, u) \mid \mu \, (x, y) \, . \, c \, \rangle & \rightsquigarrow & c[\, x \backslash t, y \backslash u \,] \\ \langle \, () \mid \mu (). \, c \, \rangle & \rightsquigarrow & c \\ \langle \, 1.t \mid \{ \mu \, (1.x) \, . \, c_1 \, , \, \mu \, (2.y) \, . \, c_2 \} \, \rangle & \rightsquigarrow & c_1[\, x \backslash t \,] \\ \langle \, 2.t \mid \{ \mu \, (1.x) \, . \, c_1 \, , \, \mu \, (2.y) \, . \, c_2 \} \, \rangle & \rightsquigarrow & c_2[\, y \backslash t \,] \\ \langle \, \lfloor t \rfloor \mid \mu \lfloor x \rfloor \, . \, c \, \rangle & \rightsquigarrow & c[\, x \backslash t \,] \end{array}$$

TYPING

$$\frac{}{\Xi \, ; \; x{:}A \vdash x : A} \; \mathrm{id} \qquad \frac{\Xi \, ; \; \Gamma \vdash t : A \quad \Xi \, ; \; \Delta \vdash u : A^{\perp}}{\Xi \, ; \; \Gamma, \Delta \vdash \langle \, t \mid u \, \rangle} \; \mathrm{cut}$$

$$\frac{}{\Xi, x{:}A \, ; \; \cdot \vdash x : A} \; \mathrm{id'} \qquad \frac{\Xi \, ; \; \Gamma, x{:}A \vdash c}{\Xi \, ; \; \Gamma \vdash \mu x. \, c : A^{\perp}} \; \mu$$

$$\frac{\Xi \, ; \; \Gamma \vdash t : A \quad \Xi \, ; \; \Delta \vdash u : B}{\Xi \, ; \; \Gamma, \Delta \vdash (t, u) : A \otimes B} \; \otimes \qquad \frac{\Xi \, ; \; \Gamma, x{:}A, y{:}B \vdash c}{\Xi \, ; \; \Gamma \vdash \mu \, (x, y) \, . \, c : A^{\perp} \parr B^{\perp}} \; \parr$$

$$\frac{}{\Xi \, ; \; \cdot \vdash () : 1} \; 1 \qquad \frac{\Xi \, ; \; \Gamma \vdash c}{\Xi \, ; \; \Gamma \vdash \mu (). \, c : \bot} \; \bot$$

$$\frac{\Xi \, ; \; \Gamma \vdash t : A}{\Xi \, ; \; \Gamma \vdash 1.t : A \oplus B} \; \oplus \mathrm{l} \qquad \frac{\Xi \, ; \; \Gamma, x{:}A \vdash c_1 \quad \Xi \, ; \; \Gamma, y{:}B \vdash c_2}{\Xi \, ; \; \Gamma \vdash \{ \mu \, (1.x) \, . \, c_1 \, , \, \mu \, (2.y) \, . \, c_2 \} : A^{\perp} \& B^{\perp}} \; \&$$

$$\frac{\Xi \, ; \; \Gamma \vdash u : B}{\Xi \, ; \; \Gamma \vdash 2.u : A \oplus B} \; \oplus \mathrm{r}$$

$$\text{No rule for } 0 \qquad \frac{}{\Xi \, ; \; \Gamma \vdash \{\} : \top} \; \top$$

$$\frac{\Xi \, ; \; \cdot \vdash t : A}{\Xi \, ; \; \cdot \vdash \lfloor t \rfloor : !A} \; ! \qquad \frac{\Xi, x{:}A \, ; \; \Gamma \vdash c}{\Xi \, ; \; \Gamma \vdash \mu \lfloor x \rfloor \, . \, c : ?A^{\perp}} \; ?$$

Figure 2: Linear L

The typing rules correspond, respectively, to the *promotion* and *dereliction* rules of linear logic:

$$\frac{\Xi \, ; \; \cdot \vdash t : A}{\Xi \, ; \; \cdot \vdash \lfloor t \rfloor : !A} \; !$$

$$\frac{\Xi, x{:}A \, ; \; \Gamma \vdash c}{\Xi \, ; \; \Gamma \vdash \mu \lfloor x \rfloor \, . \, c : ?A^{\perp}} \; ?$$

The rest of the rules for linear L can be found in Figure 2. They are almost identical to the rules of the multiplicative and additive fragment.

The choice of using substitution to embody contractions like in [MM09] or the dyadic system has non-trivial implications: if they are logically equivalent, they do not have the

same computational behaviour. In the substitution system, for instance, the sequent

$$x{:}!A \vdash \mu\alpha.\, \langle\, (x,x) \,|\, \alpha \,\rangle : !A \otimes !A$$

is derivable. In the dyadic system, it is replaced, depending on the context, by either

$$x{:}A \;;\; \cdot \vdash \left(\lfloor x \rfloor, \lfloor x \rfloor\right) : !A \otimes !A$$

or the more complex

$$\cdot \;;\; x{:}!A \vdash \mu\beta.\, \left\langle\, x \,\middle|\, \mu\, \lfloor \alpha \rfloor \,.\, \left\langle\, \left(\lfloor \alpha \rfloor, \lfloor \alpha \rfloor\right) \,\middle|\, \beta \,\right\rangle \,\right\rangle : !A \otimes !A$$

More acutely, in [MM09], it is required for type safety that the promotion rule is a computation constructor. Because of this, using the substitution system would be incompatible with the treatment of Section 4.

3. Practical L

Linear L can be intimidating. It may feels verbose and impractical to write in directly. We will use this section to review syntactic short-cuts and concrete examples in linear L. The author hopes to convince the reader that the idea of programming in linear L is not too far fetched. In fact, linear L makes a decent programming language, and an useful intermediate language. The reason why such a claim can be confidently made, is that standard programming constructs are *macro-expressible* in linear μ.

In Sections 4 and 5, we will refine linear L further. Each refinement has this same property that we can recover a usual programming language by simple macro expansion.

3.1. **Patterns.**

A most useful concept in linear μ is *nested patterns*, where we extend the atomic patterns into a full-blown pattern-matching rule. Using the whole range of patterns, however, is a bit involved, and unnecessary for this article, so we shall restrict ourselves to the irrefutable patterns

$$p,\, q \quad ::= \quad x \mid \lfloor x \rfloor \mid () \mid (p,q)$$

Note how the $\lfloor x \rfloor$ pattern is not recursive. This is not entirely necessary, but it would be a significant complication: in the following typing rules, the rule for $\lfloor x \rfloor$ would not generalise easily to nested patterns because in the pattern $\lfloor (x,y) \rfloor$, the variables x and y should not be duplicable (in other words, $!\,(A \otimes B)$ does not have projections).

$$\frac{}{\cdot \;;\; x{:}A \vdash_p x : A}\ \mathrm{id} \qquad \frac{}{x{:}A \;;\; \cdot \vdash_p \lfloor x \rfloor : !A}\ \mathrm{id'}$$

$$\frac{}{\vdash_p () : 1}\ 1 \qquad \frac{\Xi \;;\; \Gamma \vdash_p p : A \quad \Theta \;;\; \Delta \vdash_p q : B}{\Xi, \Theta \;;\; \Gamma, \Delta \vdash_p (p,q) : A \otimes B}\ \otimes$$

The treatment of duplicable variables is significantly different with respect to the usual typing rules: they are treated linearly, and they are solely introduced by the $\lfloor x \rfloor$ construction.

The first application of patterns is a generalisation of the idiom $\langle\, t \,|\, x \,\rangle$ that the reader may have noticed appeared a number of times earlier in this article. It is typed as follows:

$$\frac{\Xi \;;\; \Gamma \vdash t : A \quad \overline{\Xi \;;\; x{:}A^\perp \vdash x : A^\perp}}{\Xi \;;\; \Gamma, x{:}A^\perp \vdash \langle\, t \,|\, x \,\rangle}$$

In other words, using the terminology that the type of t is the active formula, $\langle\, t \,|\, x \,\rangle$ deactivates the type of t and give it the name x.

It is useful to generalise this idiom to a $\langle\, t \,|\, p \,\rangle$ where p is a pattern. In this case, we may see t as a box with a number of free wires – its inputs and outputs – and we give a name to individual wires. This is a fairly common idiom throughout this article, sufficiently so that we give it an derived typing rule. Let $\Theta\ ;\ \Delta \vdash_p p : A$ be a pattern, whose typing derivation will be kept implicit, then we have the following derived rule:

$$\frac{\Xi, \Theta\ ;\ \Gamma \vdash t : A^\perp}{\Xi, \Theta\ ;\ \Gamma, \Delta \vdash \langle\, t \,|\, p \,\rangle}\ \mathrm{cut}\, p$$

The case of the variable-pattern is the simple version of the idiom as seen above. It is clear, in general, that if $\Theta\ ;\ \Delta \vdash_p p : A$, then $\Xi, \Theta\ ;\ \Delta \vdash p : A$, which, with the help of the cut rule proves the derived rule. Note that this derived rule stays correct for non-linear patterns where duplicable variables are used multiple times.

The $\mathrm{cut}\, p$ rule allows for terser proofs even in the very common variable, as we shall see immediately. Indeed, a more significant use of deep patterns is, of course, pattern matching: we define the $\mu p.\, c$ by induction on patterns. In the base cases, $\mu p.\, c$ already exists, so only $\mu\,(p, q)\,.\,c$ is left to be defined:

$$\mu\,(p, q)\,.\,c = \mu\,(\alpha, \beta)\,.\,\left\langle\, \alpha \,\middle|\, \mu p.\, \langle\, \beta \,|\, \mu q.\, c \,\rangle \,\right\rangle$$

There is a derived typing rule for $\mu p.\, c$, for $\Theta\ ;\ \Delta \vdash p : A$:

$$\frac{\Xi, \Theta\ ;\ \Gamma, \Delta \vdash c}{\Xi\ ;\ \Gamma \vdash \mu p.\, c : A^\perp}\ \mu p$$

The base cases are already provided by the typing rules of linear μ, here is the proof (by induction) of the pair pattern. Let $\Theta\ ;\ \Delta \vdash_p p : A$ and $\Psi\ ;\ \Omega \vdash_p q : B$ be two patterns such that μp and μq are known to hold, we have $\Theta, \Psi\ ;\ \Delta, \Omega \vdash_p (p, q) : A \otimes B$ and the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Xi, \Theta, \Psi\ ;\ \Gamma, \Delta, \Omega \vdash c}{\Xi, \Theta\ ;\ \Gamma, \Delta \vdash \mu q.\, c : Q^\perp}\ \mu q}{\Xi, \Theta\ ;\ \Gamma, \Delta, \beta{:}Q \vdash \langle\, \beta \,|\, \mu q.\, c \,\rangle}\ \mathrm{cut}\,\beta}{\Xi\ ;\ \Gamma, \beta{:}Q \vdash \mu p.\, \langle\, \beta \,|\, \mu q.\, c \,\rangle : P^\perp}\ \mu p}{\Xi\ ;\ \Gamma, \alpha{:}P, \beta{:}Q \vdash \left\langle\, \alpha \,\middle|\, \mu p.\, \langle\, \beta \,|\, \mu q.\, c \,\rangle \,\right\rangle}\ \mathrm{cut}\,\alpha}{\Xi\ ;\ \Gamma \vdash \mu\,(p, q)\,.\,c : A^\perp \parr B^\perp}\ \parr$$

We can use this pattern-matching syntax to give meaning to the very useful $\lambda p.\, t$: we define it as $\mu\,(p, \alpha)\,.\,\langle\, t \,|\, \alpha \,\rangle$. This is an extension of the definition in Section 2.1: in addition to popping the stack, $\lambda p.\, t$ pattern-matches against the top element. Of course, $\lambda p.\, t$ has a similarly concise typing rule. Let $\Theta\ ;\ \Delta \vdash_p p : A$:

$$\frac{\Xi, \Theta\ ;\ \Gamma, \Delta \vdash t : B}{\Xi\ ;\ \Gamma \vdash \lambda p.\, t : A \multimap B}\ \lambda p$$

The justification is a straightforward extension of the variable case in Section 2.1:

$$\dfrac{\dfrac{\dfrac{\Xi, \Theta\ ;\ \Gamma, \Delta \vdash t : B}{\Xi, \Theta\ ;\ \Gamma, \Delta, \alpha{:}B^\perp \vdash \langle\, t \,|\, \alpha \,\rangle}\ \mathrm{cut}\,\alpha}{\Xi\ ;\ \Gamma \vdash \mu\,(p, \alpha)\,.\,\langle\, t \,|\, \alpha \,\rangle : A^\perp \parr B}\ \mu\,(p, \alpha)}{\Xi\ ;\ \Gamma \vdash \lambda p.\, t : A \multimap B}\ \text{definition}$$

TYPING

$$\frac{}{\Xi, x{:}A \;;\; \cdot \vdash x : A}$$

$$\frac{\Xi, x{:}A \;;\; \cdot \vdash t : B}{\Xi \;;\; \cdot \vdash \lambda \lfloor x \rfloor . t : !A \multimap B} \qquad \frac{\Xi \;;\; \cdot \vdash t : !A \multimap B \quad \Xi \;;\; \cdot \vdash u : A}{\Xi \;;\; \cdot \vdash t \;\; \lfloor u \rfloor : B}$$

Figure 3: Embedding λ-calculus

With this syntax, we can revisit the duplication of $!A$ which we encountered in Section 2.3. It is, now, quite easy to write a duplication function:

$$\vdash \lambda \lfloor x \rfloor . \left(\lfloor x \rfloor , \lfloor x \rfloor \right) : !A \multimap !A \otimes !A$$

It is possible, if quite a bit of work, to extend patterns to all of the value constructors. In [CMM10], nested patterns are even primitive and are used to define everything else. Primitive patterns can be used to as a syntax for the synthetic connectives of focusing [And92].

3.2. **Natural deduction.** Going back to Figure 2, we can observe that no rule modifies the duplicable context except the dereliction rule. In the dereliction rule, the duplicable context, Ξ, of the conclusion is adjoined an extra variable x in the premise.

This means like that the duplicable context is more similar to a natural deduction context than a standard sequent calculus context. In fact, if we restrict our attention to the sequents of the form $\Xi \;;\; \cdot \vdash t : A$, we essentially get *intuitionistic natural deduction*. Types make brief appearances in the linear context, but this can be hidden by macros.

In Figure 3, we give the translation of simply typed λ-calculus inside linear μ. It is probably comforting that using the duplicable context as a natural deduction context, the intuitionistic arrow is naturally interpreted, as it is most common, as $!A \multimap B$. Conjunction can be interpreted by either conjunction connectives, though the additive conjunction is simpler (because in the case of multiplicative conjunction the right encoding is $!A \otimes !B$, instead of the more straightforward $A\&B$). Disjunction is encoded as $!A \oplus !B$ (this time we cannot use the multiplicative connective).

Intuitionistic natural deduction (a.k.a. typed λ-calculus) is indeed the logic of duplicable formulæ in dyadic linear L. However, with extra type constructor, unusual manipulations can be made. The reader who enjoys this sort of things can have fun proving that classical logic can be encoded replacing the usual double-negation modality by the "why-not" modality: classical formulæ are those such that $?A \multimap A$ holds. In that case, the disjunction becomes $?\left(!A \oplus !B\right)$ and the falsity $?0$, or, isomorphically, $?!A \parr ?!B$ and $\bot$ [Lau02, Chapter 9].

3.3. **Linear logic proofs.** Let us, now, consider a few logical principles of linear logic, starting with the isomorphism between $!\left(A\&B\right)$ and $!A \otimes !B$. Using the syntactic facilities introduced so far, the isomorphism is quite concise. We define

$$\begin{array}{lcl} \varphi & = & \lambda \lfloor x \rfloor . \left(\lfloor x.1 \rfloor , \lfloor x.2 \rfloor \right) \\ \varphi^{-1} & = & \lambda \left(\lfloor a \rfloor , \lfloor b \rfloor \right) . \left\lfloor \{1 = a \,,\, 2 = b\} \right\rfloor \end{array}$$

Which have the following types

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}A\&B\ ;\ \cdot \vdash x : A\&B}\ \text{id'}}{x{:}A\&B\ ;\ \cdot \vdash x.1 : A}\ \pi_1}{x{:}A\&B\ ;\ \cdot \vdash \lfloor x.1 \rfloor : !A}\ ! \qquad \dfrac{\dfrac{\dfrac{}{x{:}A\&B\ ;\ \cdot \vdash x : A\&B}\ \text{id'}}{x{:}A\&B\ ;\ \cdot \vdash x.2 : B}\ \pi_2}{x{:}A\&B\ ;\ \cdot \vdash \lfloor x.2 \rfloor : !B}\ !}{x{:}A\&B\ ;\ \cdot \vdash \left(\lfloor x.1 \rfloor , \lfloor x.2 \rfloor\right) : !A \otimes !B}\ \otimes}{\vdash \lambda \lfloor x \rfloor . \left(\lfloor x.1 \rfloor , \lfloor x.2 \rfloor\right) : !\left(A\&B\right) \multimap !A \otimes !B}\ \lambda \lfloor x \rfloor}{\vdash \varphi : !\left(A\&B\right) \multimap !A \otimes !B}\ \text{definition}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{a{:}A, b{:}B\ ;\ \cdot \vdash a : A}\ \text{id'} \qquad \dfrac{}{a{:}A, b{:}B\ ;\ \cdot \vdash b : B}\ \text{id'}}{a{:}A, b{:}B\ ;\ \cdot \vdash \{1 = a\,,\, 2 = b\} : A\&B}\ \text{record}}{a{:}A, b{:}B\ ;\ \cdot \vdash \left\lfloor \{1 = a\,,\, 2 = b\} \right\rfloor : !\left(A\&B\right)}\ !}{\cdot\ ;\ \cdot \vdash \lambda \left(\lfloor a \rfloor , \lfloor b \rfloor\right) . \left\lfloor \{1 = a\,,\, 2 = b\} \right\rfloor : !A \otimes !B \multimap !\left(A\&B\right)}\ \lambda \left(\lfloor a \rfloor , \lfloor b \rfloor\right)}{\cdot\ ;\ \cdot \vdash \varphi^{-1} : !A \otimes !B \multimap !\left(A\&B\right)}\ \text{definition}$$

We have

$$\left\langle\, \varphi \ \left(\varphi^{-1} \ \left(\lfloor a \rfloor , \lfloor b \rfloor\right)\right) \,\middle|\, \alpha \,\right\rangle \rightsquigarrow \left\langle\, \left(\lfloor a \rfloor , \lfloor b \rfloor\right) \,\middle|\, \alpha \,\right\rangle$$

as well as

$$\left\langle\, \varphi^{-1} \ \left(\varphi \ \lfloor x \rfloor\right) \,\middle|\, \alpha \,\right\rangle \rightsquigarrow \left\lfloor \{1 = x.1\,,\, 2 = x.2\} \right\rfloor$$

Accepting the extensionality principles that every elements of $!A$ is of the form $\lfloor x \rfloor$, every elements of $A \otimes B$ is of the form (x, y) and for every x in $A\&B$, $\{1 = x.1\,,\, 2 = x.2\} = x$, we conclude that φ and φ^{-1} form, indeed, an isomorphism.

The dual isomorphism between $?\left(A \oplus B\right)$ and $?A \parr ?B$, which we touched upon briefly in Section 3.2, has slightly more advanced proof terms, but is all the more interesting.

$$\begin{array}{rcl} \psi & = & \lambda x.\, \mu \left(\lfloor a \rfloor , \lfloor b \rfloor\right) . \left\langle\, x \,\middle|\, \left\lfloor \{1 = a\,,\, 2 = b\} \right\rfloor \,\right\rangle \\ \psi^{-1} & = & \lambda y.\, \mu \lfloor x \rfloor . \left\langle\, y \,\middle|\, \left(\lfloor x.1 \rfloor , \lfloor x.2 \rfloor\right) \,\right\rangle \end{array}$$

Notice the pattern here: ψ is quite similar to φ^{-1} – the λ of the latter becomes a μ in the former – and so is ψ^{-1} to φ. Instead of giving a direct type derivation for ψ and ψ^{-1}, which the user can work out himself as an exercise, let us define a combinator to encode this pattern, that is a proof of $(A \multimap B) \multimap (B^{\perp} \multimap A^{\perp})$:

$$\gamma \quad = \quad \lambda f.\, \lambda x.\, \mu y.\, \left\langle\, x \,\middle|\, f \ y \,\right\rangle$$

With the typing derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\ ;\ f{:}A \multimap B \vdash f : A \multimap B}\ \text{id} \qquad \dfrac{}{\cdot\ ;\ y{:}A \vdash y : A}\ \text{id}}{\cdot\ ;\ f{:}A \multimap B, y{:}A \vdash f\ y : B}\ \text{app}}{\cdot\ ;\ f{:}A \multimap B, x{:}B^{\perp}, y{:}A \vdash \left\langle\, x \,\middle|\, f \ y \,\right\rangle}\ \text{cut}\, x}{\cdot\ ;\ f{:}A \multimap B, x{:}B^{\perp} \vdash \mu y.\, \left\langle\, x \,\middle|\, f \ y \,\right\rangle : A^{\perp}}\ \mu}{\cdot\ ;\ f{:}A \multimap B \vdash \lambda x.\, \mu y.\, \left\langle\, x \,\middle|\, f \ y \,\right\rangle : B^{\perp} \multimap A^{\perp}}\ \lambda}{\vdash \lambda f.\, \lambda x.\, \mu y.\, \left\langle\, x \,\middle|\, f \ y \,\right\rangle : (A \multimap B) \multimap (B^{\perp} \multimap A^{\perp})}\ \lambda}{\vdash \gamma : (A \multimap B) \multimap (B^{\perp} \multimap A^{\perp})}\ \text{definition}$$

We now have the equivalent definitions of ψ and ψ^{-1}:

$$\begin{array}{rcl} \psi & = & \gamma\ \varphi^{-1} \\ \psi^{-1} & = & \gamma\ \varphi \end{array}$$

Both of them reduce to the corresponding original definition, and their type is clear.

The γ combinator is quite interesting. Up to the extensionality rules $x = \mu\alpha.\ \langle\, \alpha \,|\, x \,\rangle$ and $f = \lambda\alpha.\ f\ \alpha$, a function f is the same as $\lambda x.\, \mu y.\ \langle\, y \,|\, f\ x \,\rangle$. So really, γ simply exchanges x and y in the binders. This remark makes it clear that γ is involutive, hence that $A \multimap B$ and $B^\perp \multimap A^\perp$ are isomorphic. As they should be: $A \multimap B = A^\perp \parr B$ and $B^\perp \multimap A^\perp = B \parr A^\perp$, so γ witnesses the commutativity of the $\parr$ connective.

The unsugared type of γ – $(A \otimes B^\perp) \parr (B \parr A^\perp)$ – suggests another definition

$$\gamma \quad = \quad \mu\left(f, (x, y)\right).\ \left\langle\, (y, x) \,\middle|\, f \,\right\rangle$$

Which, fortunately, is a reduced form of the original definition. This new form has the advantage of a very succinct type derivation:

$$\dfrac{\dfrac{\dfrac{}{\cdot\ ;\ f{:}A^\perp \parr B \vdash f : A^\perp \parr B}\ \mathrm{id}}{\cdot\ ;\ f{:}A^\perp \parr B, x{:}B^\perp, y{:}A \vdash \left\langle\, (y, x) \,\middle|\, f \,\right\rangle}\ \mathrm{cut}\ (y, x)}{\cdot\ ;\ \cdot \vdash \mu\left(f, (x, y)\right).\ \left\langle\, (y, x) \,\middle|\, f \,\right\rangle : (A \otimes B^\perp) \parr (B \parr A^\perp)}\ \mu\left(f, (x, y)\right)$$

To conclude this section, let us consider principles corresponding to contraction and weakening. We already mentioned in Sections 2.3 and 3.1 the duplication combinator of type $!A \multimap !A \otimes !A$, corresponding to contraction of duplicating formulæ.

$$\delta \quad = \quad \lambda\lfloor x \rfloor\,.\ \left(\lfloor x \rfloor\,, \lfloor x \rfloor\right)$$

There is also an erasure combinator, of type $!A \multimap 1$ corresponding to weakening. To highlight unused variables, we may simply omit them in the binders, writing $\lfloor\ \rfloor$ instead of $\lfloor \alpha \rfloor$:

$$\varepsilon \quad = \quad \lambda\lfloor\ \rfloor\,.\,()$$

With γ we obtain corresponding principles on the type $?A$:

$$\begin{array}{c} \cdot\ ;\ \cdot \vdash \gamma\ \delta : ?A \parr ?A \multimap ?A \\ \cdot\ ;\ \cdot \vdash \gamma\ \varepsilon : \bot \multimap ?A \end{array}$$

3.4. **Programming constructs.** We have seen many construction, so far, which allow to program in the style of pure programming languages. Linear L, however, goes beyond pure languages. To illustrate this, let us consider exceptions.

Following the tradition in pure languages, we can decide to represent computations of type A which may raise an exception E by the type $A \oplus E$. The well known limit of this representation is that exception-raising expressions must be threaded explicitly. Consider three exception-raising functions $\vdash f : A \multimap B \oplus E$, $\vdash g : B \multimap C \oplus E$, and $\vdash h : C \multimap D \oplus E$, their composite, in the worst order, can be defined as:

$$\begin{array}{rcl} g \circ f & = & \lambda x.\, \mu r. \left\langle\, f\ x \,\middle|\, \left\{ \begin{array}{l} \mu\,(1.y)\,.\ \langle\, g\ y \,|\, r \,\rangle \\ \mu\,(2.e)\,.\ \langle\, e \,|\, r.2 \,\rangle \end{array} \right\} \right\rangle \\ h \circ (g \circ f) & = & \lambda x.\, \mu r. \left\langle\, (g \circ f)\ x \,\middle|\, \left\{ \begin{array}{l} \mu\,(1.z)\,.\ \langle\, h\ z \,|\, r \,\rangle \\ \mu\,(2.e)\,.\ \langle\, e \,|\, r.2 \,\rangle \end{array} \right\} \right\rangle \end{array}$$

The relative verbosity is not an issue, as it can be hidden behind (monadic) combinators. What can be an issue, on the other hand, is that each step of the program has to inspect whether the previous expression returns a value or an exception. In the worst case, as above, even when the innermost function – f – fails, there is a linear number of inspections before the total function finally fails (on the other hand $(h \circ g) \circ f$ would fail immediately if f fails).

The inspections themselves can be costly, but more importantly, it is not always possible to avoid the slow composition order. An extreme, yet not uncommon, example would be combinators like Ocaml's `List.fold_left`: even if `List.fold_left f s l` fails quickly on `f`, the whole list needs to be traversed before returning an error. With actual exceptions, on the other hand, the execution of `List.fold_left f s l` is interrupted as soon as an exception is raised.

The behaviour of exceptions can be modelled in linear L. Instead of the type $A \oplus E$, we may use the weaker type $?E \parr ?A$ to represent computation which may raise exceptions. For simplicity, we use the fact that it also reads $!E^\perp \multimap ?A$. Let $\vdash f : A \multimap ?E \parr ?B$, $\vdash g : B \multimap ?E \parr ?C$, and $\vdash h : C \multimap ?E \parr D$, their composite can be written as:

$$\begin{aligned} g \circ f &= \lambda x.\, \mu\left(\lfloor\theta\rfloor, \lfloor\rho\rfloor\right).\, \left\langle f\ x\ \lfloor\theta\rfloor \,\middle|\, \left\lfloor \mu y.\, \left\langle g\ y\ \lfloor\theta\rfloor \,\middle|\, \lfloor\rho\rfloor \right\rangle \right\rfloor \right\rangle \\ h \circ (g \circ f) &= \lambda x.\, \mu\left(\lfloor\theta\rfloor, \lfloor\rho\rfloor\right).\, \left\langle (g \circ f)\ x\ \lfloor\theta\rfloor \,\middle|\, \left\lfloor \mu z.\, \left\langle h\ z\ \lfloor\theta\rfloor \,\middle|\, \lfloor\rho\rfloor \right\rangle \right\rfloor \right\rangle \end{aligned}$$

Where $\langle\, e \,|\, \theta \,\rangle$ should be understood as *throwing* the exception e and $\langle\, a \,|\, \rho \,\rangle$ as *returning* value a. The composition still threads functions in a monadic style to use values of type $?B$ and $?C$, however, when an exception is raised, the continuation is simply not executed as there is no value of type B or C to go on with. Indeed, $f\ x\ \lfloor\theta\rfloor$ has type $?B$: if an exception is raised no value is returned. With the type $A \oplus E$, on the other hand, a value of type E is returned.

This representation of exceptions is often called the two-continuation encoding of exception. Where ρ is called the success continuation, and θ the failure continuation. Because continuations are lexically bound, the θ operation behaves differently from Ocaml's `raise`: instead of *dynamically* looking for the enclosing `try`, θ jumps to a *statically* determined context. Much like with callcc, it means that θ can be captured and escape its scope, in which case executing $\langle\, e \,|\, \theta \,\rangle$ would still resume at θ.

It should be no surprise that, actually, callcc itself can be implemented in linear L. In fact, we have already claimed in Section 3.2 that classical logic can be embedded in linear L; but we can give a more direct implementation with a more precise type than what would be achieved through such an embedding.

The type of callcc is (a linear version of) Peirce's law. The first ingredient is to take a context of type $A^\perp$ and reify it as a function $A \multimap X$ for some X. As, by definition, no value of type X will be produced, X must be of the form $?B$. In particular, X can be $\perp$, which is isomorphic to $?0$. We define trow, of type $A^\perp \multimap A \multimap ?B$:

$$\text{throw} = \lambda k.\, \lambda x.\, \mu\, \lfloor\ \rfloor.\, \langle\, x \,|\, k \,\rangle$$

It takes the continuation k to the function which, given an element x, drops the current continuation, and runs x against k. The typing derivation is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha{:}B^\perp \;;\; x{:}A \vdash x : A}\,\mathrm{id}}{\alpha{:}B^\perp \;;\; k{:}A^\perp, x{:}A \vdash \langle\, x \,|\, k \,\rangle}\,\mathrm{cut}\,k}{\cdot \;;\; k{:}A^\perp, x{:}A \vdash \mu\lfloor\alpha\rfloor\,.\,\langle\, x \,|\, k \,\rangle : ?B}\,?}{\cdot \;;\; k{:}A^\perp \vdash \lambda x.\,\mu\lfloor\alpha\rfloor\,.\,\langle\, x \,|\, k \,\rangle : A \multimap ?B}\,\lambda}{\cdot \;;\; \cdot \vdash \lambda k.\,\lambda x.\,\mu\lfloor\alpha\rfloor\,.\,\langle\, x \,|\, k \,\rangle : A^\perp \multimap A \multimap ?B}\,\lambda}{\cdot \;;\; \cdot \vdash \mathrm{throw} : A^\perp \multimap A \multimap ?B}\,\text{definition}$$

To complete callcc, let us consider Peirce's law: $((A \to B) \to A) \to A$. From a computational point of view, there are two ways an α is produced: either the body returns an α, and the current continuation is restored, or the body explicitely calls the continuation via the function $\alpha \to \beta$. Returning to linear L, it means that the current continuation must be duplicable, so callcc has a type of the form $X \multimap ?A$. Since the current continuation is duplicable, we are free to make its functional reification duplicable as well. Which gives us the final type $(!\,(A \multimap ?B) \multimap ?A) \multimap ?A$.

$$\mathrm{callcc} = \lambda f.\,\mu\lfloor k\rfloor\,.\,\big\langle\, f \;\; \lfloor \mathrm{throw}\; k\rfloor \;\big|\; \lfloor k\rfloor \,\big\rangle$$

Notice how k is indeed duplicated, hence forces the return type to be $?A$. The full type derivation is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{k{:}A^\perp \;;\; f{:}!\,(A \multimap ?B) \multimap ?A \vdash f : !\,(A \multimap ?B) \multimap ?A}\,\mathrm{id} \qquad \begin{array}{c}\vdots\\ k{:}A^\perp \;;\; \cdot \vdash \lfloor \mathrm{throw}\; k\rfloor : !\,(A \multimap ?B)\end{array}}{k{:}A^\perp \;;\; f{:}!\,(A \multimap ?B) \multimap ?A \vdash f \;\; \lfloor \mathrm{throw}\; k\rfloor : ?A}\,\mathrm{app}}{k{:}A^\perp \;;\; f{:}!\,(A \multimap ?B) \multimap ?A \vdash \big\langle\, f \;\; \lfloor \mathrm{throw}\; k\rfloor \;\big|\; \lfloor k\rfloor \,\big\rangle}\,\mathrm{cut}\;\lfloor k\rfloor}{\cdot \;;\; f{:}!\,(A \multimap ?B) \multimap ?A \vdash \mu\lfloor k\rfloor\,.\,\big\langle\, f \;\; \lfloor \mathrm{throw}\; k\rfloor \;\big|\; \lfloor k\rfloor \,\big\rangle : ?A}\,?}{\vdash \mathrm{callcc} : \big(!\,(A \multimap ?B) \multimap ?A\big) \multimap ?A}\,\lambda$$

The type of callcc illustrates how embedding classical principles in linear μ requires annotations with both exponential connectives. Depending on the precise annotations used, the ambiguous critical pairs of classical logic may be resolved by favouring one way or another [Lau02, Chapter 9]. However, it is probably unnecessary to use a callcc combinator to program in linear μ, as the μ binder already fills its function.

3.5. **Commutative cuts.** For those developing a programming language which is not based on linear L, it can still be quite serviceable as an intermediate language. In the spirit of the Glasgow Haskell Compiler (GHC) [PS98], the syntax of a programming language can be *desugared* into linear L which is quite amenable to program transformations. Desugaring is a simple transformation, which should consist of interpreting syntactic construction of the language as macros, which is the case of every definition we demonstrate in this article. The reader wishing to make the meaning of macros more formal may refer to [Fel90].

One of the roles of intermediate language optimisation is to eliminate unnecessary computations introduces by modularly written programs: a common slogan is *writing the programs we want to write, and producing the code we want to produce.* The strategy, in [PS98] is to express optimisations as *program transformations* on the intermediate language. For instance – in Ocaml syntax – `let(x,y)=(u,v) in t x y`, provided u and v are values, can be rewritten

to t u v: it does not affect the complexity of the program, but avoids unnecessary allocations which can be inefficient.

In languages based on λ-calculus, however, there are missed opportunity for reduction. Consider the term let(x,y) = let (z,i) = u in (i,z) in t. In that expression, (i,z) is built and immediately destructed. However, this unnecessary allocation cannot be eliminated by β-reduction. To handle this missed opportunity for reduction [PS98, Section 5] introduces an extra transformation (we give the transformation for pair patterns, but it can be defined, in general, for any atomic pattern matching):

$$\text{let (x,y) = let (z,i) = u in v in t} \rightsquigarrow \text{let(z,i) = u in let(x,y) = v in t}$$

This reduction is an example of what is called, in proof theory, a *commutative cut*, or commuting conversion. That is, proofs which are distinguished by the cut elimination of natural deduction, but are identified by the cut elimination of sequent calculus. They can, hence, be seen as hidden cuts. Commutative cuts always involve a pattern matching another elimination rule. Here is a commutative cut with application:

$$\text{(let (x,y) = u in v) t} \rightsquigarrow \text{let(x,y) = u in v t}$$

Transforming programs according to such commutative cut is crucial to the optimisation strategy of GHC. Going even further, the MLJ compiler [BKR99] transformed programs according to every commutative cut.

In linear L, commutative cuts are, as is expected of cut elimination in a sequent calculus, mere reductions. Indeed, the pattern matching of linear L do not return terms, but commands. Hence the term-returning flavour of pattern matching from natural deduction must be encoded, by capturing the current continuation: let(x,y) = u in v is encoded as $\mu\alpha.\ \langle\, u \,|\, \mu\,(x,y)\,.\ \langle\, v \,|\, \alpha \,\rangle\,\rangle$. We then have that (let (x,y) = u in v) t is as follows:

$$\mu\alpha.\ \left\langle\, \mu\beta.\ \langle\, u \,\middle|\, \mu\,(z,i)\,.\ \langle\, v \,|\, \beta \,\rangle\,\rangle \,\middle|\, \mu\,(x,y)\,.\ \langle\, t \,|\, \alpha \,\rangle \,\right\rangle$$

Reducing the redex for $\mu\beta$, we obtain:

$$\mu\alpha.\ \left\langle\, u \,\middle|\, \mu\,(z,i)\,.\ \langle\, v \,\middle|\, \mu\,(x,y)\,.\ \langle\, t \,|\, \alpha \,\rangle\,\rangle \,\right\rangle$$

which is almost the encoding of let(z,i) = u in let(x,y) = v in t. In fact, it is one innocuous reduction better than the actual one:

$$\mu\alpha.\ \left\langle\, u \,\middle|\, \mu\,(z,i)\,.\ \left\langle\, \mu\beta.\ \langle\, v \,\middle|\, \mu\,(x,y)\,.\ \langle\, t \,|\, \beta \,\rangle\,\rangle \,\middle|\, \alpha \,\right\rangle \,\right\rangle$$

The case of (let (x,y) = u in v) t works just as well:

$$\mu\alpha.\ \left\langle\, \mu\beta.\ \langle\, u \,\middle|\, \mu\,(x,y)\,.\ \langle\, v \,|\, \beta \,\rangle\,\rangle \,\middle|\, (t,\alpha) \,\right\rangle \rightsquigarrow \mu\alpha.\ \left\langle\, u \,\middle|\, \mu\,(x,y)\,.\ \langle\, v \,\middle|\, (t,\alpha) \,\rangle \,\right\rangle$$

Going further, Marlow [Mar95, Chapter 3] studies precisely how cut elimination for sequent calculus can model deforestation, *i.e.* elimination of unnecessary intermediate data in presence of recursive types.

The commutative cuts, in linear L, are not actual reductions, because extra μ binder are introduced by the macro-encoding of the language constructs. However, they are indeed conversions. From an intermediate language perspective, anyway, these extra binders should be eliminated as much as possible, so these reduction produce the appropriate terms.

While we are on the subject of program optimisation by reduction, linear L contributes to the subject in another way. Indeed, reductions should not modify the complexity of the

program, so they should not duplicate or drop computations. In linear L, there is only one reduction rule which may duplicate or drop computation:

$$\left\langle \lfloor t \rfloor \,\middle|\, \mu \lfloor x \rfloor . c \right\rangle \quad \rightsquigarrow \quad c[x \backslash t]$$

All the other reduction may be applied safely. Though, they should probably not be applied blindly, as *code* duplication can still occur, if several branches refer to the same variable. Note that, for non-linear languages, this requires a significantly finer translation than just marking every variable as duplicable to be actually useful.

Using linearity to guide reductions in the GHC compiler was actually proposed in [PS98, Section 4.2]. Their intermediate language does not have syntactic markers for duplication, so they rely purely on types to guide the reduction. The main difference, though, is that Haskell being a lazy language, GHC can safely drop computations, so their typing system is actually *affine*.

An affine flavour L can be defined by changing the linear context of linear L into an affine context which can be dropped at leaves. Effectively, affine L differs from linear L by three rules: the two identity rules and the introduction rule for 1, which now can have an arbitrary affine context. Affine L can be translated into linear L by translating every hypothesis x of type A in the affine context into an hypothesis x of type $A\&1$ in the linear context. This translation, however, *is not* a simple macro translation, at least not on untyped terms, as unused variables must be explicitly dropped. So the translation of the three modified rules depend on the affine variables in the context.

4. Polarised L

Linear L solves the weakening-against-weakening non-confluence example of Section 1: to erase a variable, one must introduce a binder $\mu \lfloor \ \rfloor . c$ which is not involved in a critical pair. However, there are still critical pairs of the form $\langle \mu x. c \,|\, \mu y. c' \rangle$ which can be typed in linear L. It is conceivable that the reduction of linearly typed L term is still non-confluent. And indeed, here is a counter-example.

$$\left\langle \mu x. \left\langle (x, z) \,\middle|\, v \right\rangle \,\middle|\, \mu y. \left\langle (t, y) \,\middle|\, w \right\rangle \right\rangle$$

which reduces both to

$$\left\langle \left(t, \mu x. \left\langle (x, z) \,\middle|\, v \right\rangle \right) \,\middle|\, w \right\rangle \qquad \text{and} \qquad \left\langle \left(\mu y. \left\langle (t, y) \,\middle|\, w \right\rangle, z \right) \,\middle|\, v \right\rangle$$

two distinct normal forms, yet has the following type:

$$\cdot \ ; \ t{:}A, z{:}A^{\perp}, v{:}A^{\perp} \parr A, w{:}A^{\perp} \parr A \vdash \left\langle \mu x. \left\langle (x, z) \,\middle|\, v \right\rangle \,\middle|\, \mu y. \left\langle (t, y) \,\middle|\, w \right\rangle \right\rangle$$

There are several ways to think about this example. On the first hand, it could be said that the syntax is inadequate and we should move to a syntax which identifies both terms, like proof nets. On the other hand, we can also point out that $\mu x. c$ does not really make sense by itself: it is an active term which expects a counterpart. In that view, it does not really make sense to capture such a term in a pair $(\mu x. c, u)$ where the μ cannot be resolved.

4.1. **Restricting substitution.** The solution suggested by the latter view is to take more seriously the distinction between *values* and *computations*. That is $\mu x.\, c$ is a computation, yet we take the point of view that variables should only be substituted with values. This is a form of *call by value*, even though, as we will see below, this does not preclude call by name functions. Such a restriction can be achieved by a syntactic criterion: identifying a syntactic class of values, and restricting reduction rules to only substitute values. This is the strategy used in the setting of λ-calculus [Plo75] or in the original L paper [CH00].

To offer a counter-point we will account for this restriction purely by typing. This is merely a difference in presentation, though, as the syntactic restriction can be read off directly from the typing rules. This idea leads to a *polarised* logic, where types are classified on whether their introduction rules are value constructors (*positive* types) or computation constructors (*negative* types). The restriction that variables can only be substituted with values then translates to the restriction that *variables all are of positive type*.

This rule has strong consequences: in $\mu\,(x,y)\,.\,c$, x and y must be of positive type, hence the two component of a pair must be values. In particular terms of the form $(\mu x.\, c, u)$, like above, are no longer permissible. It is now well understood [DL06, Zei08, MM09, CMM10], that this call-by-value restriction of sequent calculus is akin to focusing [And92], though, on the details, it need not correspond too closely.

The classification of types is, hence, a strong restriction which we sum up with the following grammar, where A and B denote positive types and N and M denote negative types:

$$\begin{array}{lcl} A,\, B & ::= & A \otimes B \mid 1 \mid A \oplus B \mid 0 \mid !A \mid \downarrow N \\ N,\, M & ::= & N \parr M \mid \bot \mid A \& B \mid \top \mid ?A \mid \uparrow P \end{array}$$

This grammar introduces two new dual connectives $\downarrow N$ and $\uparrow P$ – both read "shift" – to mediate between the two polarities. The shift connectives have reversed introduction rules, as $\uparrow P$ is introduced by the construction $\Uparrow v$ (read "return" v), and $\downarrow N$ by $\mu \Uparrow x\,.\,c$, despite the former being negative and the latter positive. Indeed $\Uparrow v$ is a computation and $\mu \Uparrow x\,.\,c$ a value. Sequents for values are written $\Xi\ ;\ \Gamma \vdash_v t : A$, this is purely cosmetic in this section, but the typing judgement of values and computations will be distinct in Section 5.

$$\frac{\Xi\ ;\ \Gamma, x{:}A \vdash c}{\Xi\ ;\ \Gamma \vdash_v \mu \Uparrow x\,.\,c : \downarrow A^\bot}\downarrow \qquad \frac{\Xi\ ;\ \Gamma \vdash_v t : A}{\Xi\ ;\ \Gamma \vdash \Uparrow t : \uparrow A}\uparrow$$

With the obvious reduction rule:

$$\langle\, \mu \Uparrow x\,.\,c \,|\, \Uparrow t \,\rangle \quad \rightsquigarrow \quad c[\,x \backslash t\,]$$

As often, it is usually possible to define an alternative syntax to replace the μ pattern: $\downarrow N$ can alternatively be introduced by $\Downarrow t$ – read "thunk t" – defined as:

$$\Downarrow t = \mu \Uparrow \alpha\,.\,\langle\, t \,|\, \alpha \,\rangle$$

The typing of which is given by:

$$\dfrac{\dfrac{\dfrac{\Xi\ ;\ \Gamma \vdash t : N}{\Xi\ ;\ \Gamma, \alpha{:}N^\bot \vdash \langle\, t \,|\, \alpha \,\rangle}\,\mathrm{cut}\,\alpha}{\Xi\ ;\ \Gamma \vdash_v \mu \Uparrow \alpha\,.\,\langle\, t \,|\, \alpha \,\rangle : \downarrow N}\downarrow}{\Xi\ ;\ \Gamma \vdash_v \Downarrow t : \downarrow N}\,\mathrm{definition}$$

And has as a reduction rule:

$$\langle\, \Downarrow t \,|\, \Uparrow u \,\rangle \rightsquigarrow \langle\, u \,|\, t \,\rangle$$

SYNTAX

$$\begin{array}{rcl} t,\, u & ::= & \ldots \mid \Uparrow t \mid \mu \Uparrow x \,.\, c \\ c & ::= & \langle\, t \,|\, u \,\rangle \end{array}$$

TYPES

$$\begin{array}{rcl} A,\, B & ::= & A \otimes B \mid 1 \mid A \oplus B \mid 0 \mid\, !A \mid \downarrow N \\ N,\, M & ::= & N \parr M \mid \bot \mid A \& B \mid \top \mid\, ?A \mid \uparrow P \end{array}$$

REDUCTION

$$\begin{array}{lcl} \langle\, t \,|\, \mu x.\, c \,\rangle & \rightsquigarrow & c[\, x \backslash t \,] \\ \langle\, \mu \Uparrow x \,.\, c \,|\, \Uparrow t \,\rangle & \rightsquigarrow & c[\, x \backslash t \,] \\ \langle\, (t, u) \,|\, \mu\,(x, y)\,.\, c \,\rangle & \rightsquigarrow & c[\, x \backslash t, y \backslash u \,] \\ \langle\, () \,|\, \mu().\, c \,\rangle & \rightsquigarrow & c \\ \langle\, 1.t \,|\, \{\mu\,(1.x)\,.\, c_1 \,,\, \mu\,(2.y)\,.\, c_2\} \,\rangle & \rightsquigarrow & c_1[\, x \backslash t \,] \\ \langle\, 2.t \,|\, \{\mu\,(1.x)\,.\, c_1 \,,\, \mu\,(2.y)\,.\, c_2\} \,\rangle & \rightsquigarrow & c_2[\, y \backslash t \,] \\ \langle\, \lfloor t \rfloor \,|\, \mu\, \lfloor x \rfloor \,.\, c \,\rangle & \rightsquigarrow & c[\, x \backslash t \,] \end{array}$$

TYPING

$$\frac{}{\Xi \,;\, x{:}A \vdash_v x : A}\ \mathrm{id} \qquad \frac{\Xi \,;\, \Gamma \vdash_v t : A \quad \Xi \,;\, \Delta \vdash u : A^{\perp}}{\Xi \,;\, \Gamma, \Delta \vdash \langle\, t \,|\, u \,\rangle}\ \mathrm{cut}$$

$$\frac{}{\Xi, x{:}A \,;\, \cdot \vdash_v x : A}\ \mathrm{id'} \qquad \frac{\Xi \,;\, \Gamma, x{:}A \vdash c}{\Xi \,;\, \Gamma \vdash \mu x.\, c : A^{\perp}}\ \mu$$

$$\frac{\Xi \,;\, \Gamma, x{:}A \vdash c}{\Xi \,;\, \Gamma \vdash_v \mu \Uparrow x \,.\, c : \downarrow A^{\perp}}\ \downarrow \qquad \frac{\Xi \,;\, \Gamma \vdash_v t : A}{\Xi \,;\, \Gamma \vdash \Uparrow t : \uparrow A}\ \uparrow$$

$$\frac{\Xi \,;\, \Gamma \vdash_v t : A \quad \Xi \,;\, \Delta \vdash_v u : B}{\Xi \,;\, \Gamma, \Delta \vdash_v (t, u) : A \otimes B}\ \otimes \qquad \frac{\Xi \,;\, \Gamma, x{:}A, y{:}B \vdash c}{\Xi \,;\, \Gamma \vdash \mu\,(x, y)\,.\, c : A^{\perp} \parr B^{\perp}}\ \parr$$

$$\frac{}{\Xi \,;\, \cdot \vdash_v () : 1}\ 1 \qquad \frac{\Xi \,;\, \Gamma \vdash c}{\Xi \,;\, \Gamma \vdash \mu().\, c : \bot}\ \bot$$

$$\frac{\Xi \,;\, \Gamma \vdash_v t : A}{\Xi \,;\, \Gamma \vdash_v 1.t : A \oplus B}\ \oplus\mathrm{l} \qquad \frac{\Xi \,;\, \Gamma, x{:}A \vdash c_1 \quad \Xi \,;\, \Gamma, y{:}B \vdash c_2}{\Xi \,;\, \Gamma \vdash \{\mu\,(1.x)\,.\, c_1 \,,\, \mu\,(2.y)\,.\, c_2\} : A^{\perp} \& B^{\perp}}\ \&$$

$$\frac{\Xi \,;\, \Gamma \vdash_v u : B}{\Xi \,;\, \Gamma \vdash_v 2.u : A \oplus B}\ \oplus\mathrm{r}$$

$$\text{No rule for } 0 \qquad \frac{}{\Xi \,;\, \Gamma \vdash \{\} : \top}\ \top$$

$$\frac{\Xi \,;\, \cdot \vdash_v t : A}{\Xi \,;\, \cdot \vdash_v \lfloor t \rfloor : !A}\ ! \qquad \frac{\Xi, x{:}A \,;\, \Gamma \vdash c}{\Xi \,;\, \Gamma \vdash \mu\, \lfloor x \rfloor \,.\, c : ?A^{\perp}}\ ?$$

Figure 4: Polarised L

The other typing rules for polarised L are given in Figure 4. They are, actually, textually identical to the rules of linear L, except that now A and B stand for positive types. Polarised L is really only a matter of constraining the variables to have positive types.

The shift are not simple coercions between negative and positive types: they have a real computational significance. Indeed $\uparrow A$ is a bigger type than A: if A contains only values v,

$\uparrow A$ contains any computation which *evaluates to* v. For instance, 1 only contains the value $()$, but $\uparrow 1$ contains computations like $\big(\lambda().\ \Uparrow()\big)\ ()$. In fact, polarised L is a linear variant of Levy's call-by-push-value (CBPV) language [Lev01]. With $\uparrow A$ and $\downarrow N$ playing the role, respectively, of FA and UN.

Like in CBPV, computations can be chained, with the expression $t \,\mathrm{to}\, x.\, u$, so that the value computed by t is bound to x in u, in a manner reminiscent of monadic composition.

$$t \,\mathrm{to}\, x.\, u = \mu\alpha.\ \big\langle\, t \,\big|\, \mu\Uparrow x.\ \langle\, u \,|\, \alpha \,\rangle \,\big\rangle$$

with the same typing rule as in [Lev01] (up to linearity):

$$\dfrac{\Xi\ ;\ \Gamma \vdash t : \uparrow A \qquad \dfrac{\dfrac{\Xi\ ;\ \Delta, x{:}A \vdash u : N}{\Xi\ ;\ \Delta, \alpha{:}N^\perp, x{:}A \vdash \langle\, u \,|\, \alpha \,\rangle}\ \mathrm{cut}\,\alpha}{\Xi\ ;\ \Delta, \alpha{:}N^\perp \vdash \mu\Uparrow x.\ \langle\, u \,|\, \alpha \,\rangle : \downarrow A^\perp}\ \downarrow}{\dfrac{\dfrac{\Xi\ ;\ \Gamma, \Delta, \alpha{:}N^\perp \vdash \big\langle\, t \,\big|\, \mu\Uparrow x.\ \langle\, u \,|\, \alpha \,\rangle \,\big\rangle}{\Xi\ ;\ \Gamma, \Delta \vdash \mu\alpha.\ \big\langle\, t \,\big|\, \mu\Uparrow x.\ \langle\, u \,|\, \alpha \,\rangle \,\big\rangle : N}\ \mu}{\Xi\ ;\ \Gamma, \Delta \vdash t \,\mathrm{to}\, x.\, u : N}\ \mathrm{definition}}\ \mathrm{cut}$$

Together with the reduction rule:

$$\langle\, \Uparrow v \,\mathrm{to}\, x.\, u \,|\, \alpha \,\rangle \rightsquigarrow \big\langle\, u[\, x \backslash v \,] \,\big|\, \alpha \,\big\rangle$$

Dually, the type $\downarrow N$ represents the type of suspended computations. A suspended computation differs from regular computation in that they can be stored in a value. Suspending a computation corresponds to an operation well-known to the functional programmer: building a closure. Indeed, a closure is nothing but packing a computation (typically in form of a code pointer) together with the environment necessary for the computation to be resumed later. That is, turning a computation into a value.

A feature shared by CBPV and polarised L, is that functions are computations $A \multimap N$. They are not made into closure unless they are suspended into the type $\big\downarrow(A \multimap N)$. If closures are considered expensive to make, which they often are, this property can be useful as functions of multiple arguments $A \multimap B \multimap C \multimap N$ do not need intermediate closures.

As a matter of fact, the Rust programming language has, for efficiency purposes, so-called *stack closures*, which are, in fact, not closures by that definition. Stack closures are functions which can be used only as argument of another function, and be called, but *not* be stored in a value. From the point of view of polarised L, this would correspond to having variables of negative type, with restricted usage.

Suspended values, again inspired by CBPV, can be turned back into computation with a force combinator, which is adjoint to return:

$$\mathrm{force}\ t = \mu\alpha.\ \langle\, t \,|\, \Uparrow\alpha \,\rangle$$

Typed as

$$\dfrac{\dfrac{\Xi\ ;\ \Gamma \vdash_v t : \downarrow N \qquad \dfrac{\dfrac{}{\Xi\ ;\ \alpha{:}N^\perp \vdash_v \alpha : N^\perp}\ \mathrm{id}}{\Xi\ ;\ \alpha{:}N^\perp \vdash \Uparrow\alpha : \uparrow N^\perp}\ \uparrow}{\Xi\ ;\ \Gamma, \alpha{:}N^\perp \vdash \langle\, t \,|\, \Uparrow\alpha \,\rangle}\ \mathrm{cut}}{\dfrac{\Xi\ ;\ \Gamma \vdash \mu\alpha.\ \langle\, t \,|\, \Uparrow\alpha \,\rangle : N}{\Xi\ ;\ \Gamma \vdash \mathrm{force}\ t : N}\ \mu}\ \mathrm{definition}$$

And with reduction rule:

$$\langle\, \mathrm{force}\ \Downarrow t \,|\, \alpha \,\rangle \rightsquigarrow \langle\, t \,|\, \alpha \,\rangle$$

4.2. **Translations.** Despite the call-by-value slant of polarised L, both call-by-value and call-by-name λ-calculus can be embedded in polarised L. Again, all of the definitions are macros. For simplicity we will only give encoding of linear λ-calculus, but the intuitionistic version is not very different. We use the definitions of $\lambda x.\, t$ and $t\ u$ defined in Figure 1, like the other connectives they have only changed inasmuch as the polarisation of the operands of the arrow type: the arrow $A \multimap N = A^{\perp} \parr N$ has positive domain and negative codomain.

4.2.1. *Call-by-name λ-calculus.* Call-by-name λ-calculus is obtained by interpreting all type as being negative. As a consequence, all variables in the context must be shifted which modifies the variable rule, and the arrow is encoded as $\downarrow N \multimap M$:

$$\dfrac{\dfrac{}{\Gamma, x{:}{\downarrow}N \vdash_v x : {\downarrow}N}\ \text{id}}{\Gamma, x{:}{\downarrow}N \vdash \text{force}\ x : N}\ \text{force}$$

$$\dfrac{\Gamma \vdash t : {\downarrow}N \multimap M \qquad \dfrac{\Delta \vdash u : N}{\Delta \vdash_v \Downarrow u : {\downarrow}N}\ {\downarrow}}{\Gamma, \Delta \vdash t\ \Downarrow u : M}\ \text{app}$$

$$\dfrac{\Gamma, x{:}{\downarrow}N \vdash t : M}{\Gamma \vdash \lambda x.\, t : {\downarrow}N \multimap M}\ \lambda$$

In call-by-name λ-calculus, the arguments of functions are suspended so that their computation happens at use point: when the variables are used and forced. This translation, which is mostly forced by the choice that all types are interpreted as negative, happens to correspond closely to simple implementations, for instance Krivine's abstract machine.

4.2.2. *Call-by-value λ-calculus.* Dually, in call-by-value λ-calculus all of the types are interpreted as positive. Since a λ-term is a computation, not a value, the type of the terms – rather than the hypotheses as in call by name – must be shifted. Also, the encoding of functions is a little more involved: $\Big\downarrow\big(A \multimap {\uparrow}B\big)$. Again, this translation follows straightforwardly from the choice that every type is positive.

$$\dfrac{\dfrac{}{\Gamma, x{:}A \vdash_v x : A}\ \text{id}}{\Gamma, x{:}A \vdash \Uparrow x\ : {\uparrow}A}\ {\uparrow}$$

$$\dfrac{\Delta \vdash u : {\uparrow}A \qquad \dfrac{\Gamma \vdash t : {\uparrow}\Big\downarrow\big(A \multimap {\uparrow}B\big) \qquad \dfrac{\dfrac{\dfrac{}{f{:}\Big\downarrow\big(A \multimap {\uparrow}B\big) \vdash_v f : \Big\downarrow\big(A \multimap {\uparrow}B\big)}\ \text{id}}{f{:}\Big\downarrow\big(A \multimap {\uparrow}B\big) \vdash \text{force}\ f : A \multimap {\uparrow}B}\ \text{force} \quad \dfrac{}{x{:}A \vdash_v x : A}\ \text{id}}{x{:}A, f{:}\Big\downarrow\big(A \multimap {\uparrow}B\big) \vdash \text{force}\ f\ x : {\uparrow}B}\ \text{app}}{\Gamma, x{:}A \vdash t\,\text{to}\,f.\,\text{force}\ f\ x : {\uparrow}B}\ \text{chain}}{\Gamma, \Delta \vdash u\,\text{to}\,x.\,t\,\text{to}\,f.\,\text{force}\ f\ x : {\uparrow}B}\ \text{chain}$$

$$\dfrac{\dfrac{\dfrac{\Gamma, x{:}A \vdash t : {\uparrow}B}{\Gamma \vdash \lambda x.\, t : A \multimap {\uparrow}B}\ \lambda}{\Gamma \vdash_v \Downarrow\lambda x.\, t : \Big\downarrow\big(A \multimap {\uparrow}B\big)}\ \text{thunk}}{\Gamma \vdash \Uparrow\Downarrow\lambda x.\, t\ : {\uparrow}\Big\downarrow\big(A \multimap {\uparrow}B\big)}\ {\uparrow}$$

As in any call-by-value calculus, there is a non-canonical choice in the order of evaluation. We observe it in the translation of application: we chose to evaluate the argument before the function, but the reverse works just as well and behaves differently in presence of effects.

Notice that, contrary to the call-by-name λ-calculus, the encoding of call-by-value λ-calculus introduces a closure around every abstraction. This aspect is discussed, in the context of abstract machines, in [Ler90, Chapter 3]. Closures are usually expensive, hence we may want to eliminate intermediate closures in expressions of the form

$$\Uparrow\Downarrow\lambda x.\, \Uparrow\Downarrow\lambda y.\, t$$

When it is applied to two arguments. It can be done by partial evaluation like in Section 3.5, because the application to two arguments u and v:

$$\left\langle v \,\mathrm{to}\, y.\ \left(u \,\mathrm{to}\, x.\ (\Uparrow\Downarrow\lambda x.\, \Uparrow\Downarrow\lambda y.\, t\ )\ \mathrm{to}\, f.\, \mathrm{force}\ f\ y\right)\ \mathrm{to}\, g.\, \mathrm{force}\ g\ x \,\middle|\, \alpha \right\rangle$$

Is convertible to

$$\langle\, v \,\mathrm{to}\, y.\, u \,\mathrm{to}\, x.\, t \,|\, \alpha \,\rangle$$

It requires some care to avoid code duplication, however. Compared to the solution of the ZINC abstract machine [Ler90, Chapter 3], this optimisation only applies to statistically detectable situation, whereas the ZINC tries dynamically to avoid intermediate closures (by checking the number of available arguments on the stack). So partial evaluation of polarised L is more efficient (as it forgoes dynamic tests), but does not apply as often.

4.2.3. *Call-by-value linear* L. Like λ-calculus, linear L can be translated into polarised L. There is the same dichotomy as for λ-calculus with one translation interpreting each type as positive and another as negative.

Unlike λ-calculus, types having all the same polarity causes difficulty because of the cut rule, and the μ rule. So, sadly, these are not macro-translations. To be fair, apart from the cut and μ rules, the terms are macro-translated, although their types are not. We will focus on the multiplicative fragment, the rest follows straightforwardly. We write $[\![A]\!]$ for the translation of type A.

In the encoding where all types are positive, the product $A\otimes B$ can be simply interpreted as $[\![A]\!]\otimes[\![B]\!]$, and 1 as 1. We can see, already, that this calculus will have a call-by-value feel. Like in call-by-value λ-calculus, hypotheses have their bare type, but the type of the active term is shifted. Here is the identity rule

$$\frac{\dfrac{}{x{:}[\![A]\!] \vdash_v x : [\![A]\!]}\ \mathrm{id}}{x{:}[\![A]\!] \vdash \Uparrow x : \uparrow[\![A]\!]}\ \uparrow$$

When translating the type $A^\perp \parr B^\perp$, we must construct a continuation of $[\![A\otimes B]\!] = [\![A]\!]\otimes[\![B]\!]$, so basically we have no alternative but to choose $[\![A]\!]^\perp \parr [\![B]\!]^\perp$, but since it's not a positive type, it needs to be shifted: $[\![A^\perp \parr B^\perp]\!] = \downarrow\left([\![A]\!]^\perp \parr [\![B]\!]^\perp\right)$. Likewise, $[\![\perp]\!] = \downarrow\perp$.

Since cut and μ involve dualisation, they need to relate $[\![A]\!]$ and $[\![A^\perp]\!]$. The property we deduce from the definition is: $[\![A^\perp]\!] = \downarrow[\![A]\!]^\perp$ or $\uparrow[\![A^\perp]\!] = [\![A]\!]^\perp$. The cut rule must be oriented to recognise which of its operand has an extra shift.

$$\frac{\dfrac{\Gamma \vdash t : \uparrow[\![A]\!]}{\Gamma \vdash_v \Downarrow t : \downarrow\uparrow[\![A]\!]}\ \mathrm{thunk} \qquad \Delta \vdash u : \uparrow\downarrow[\![A]\!]^\perp}{\Gamma,\Delta \vdash \langle\, \Downarrow t \,|\, u \,\rangle}\ \mathrm{cut}$$

The μ rule is worse, as it must be duplicated depending on whether it must add a shift or not to the selected hypothesis.

$$\dfrac{\dfrac{\dfrac{\Gamma, x{:}[\![A]\!] \vdash c}{\Gamma \vdash_v \mu\Uparrow x\,.\,c : \downarrow [\![A]\!]^\perp}\downarrow}{\Gamma \vdash \Uparrow\mu\Uparrow x\,.\,c : \uparrow\downarrow [\![A]\!]^\perp}\uparrow}{\Gamma \vdash \Uparrow\mu\Uparrow x\,.\,c : \uparrow [\![A^\perp]\!]}\ \text{definition} \qquad \dfrac{\dfrac{\Gamma, x{:}[\![A]\!] \vdash c}{\Gamma \vdash \mu x.\,c : [\![A]\!]^\perp}\mu}{\Gamma \vdash \mu x.\,c : \uparrow [\![A^\perp]\!]}\ \text{definition}$$

The translation of introduction rules, on the other hand, are quite unproblematic. Pairing is obtained by first computing the values of the two components and then returning the pair of the obtained values. Like for call-by-value λ-calculus, the order of evaluation is non-canonical.

$$\dfrac{\Delta \vdash v : \uparrow [\![B]\!] \qquad \dfrac{\Gamma \vdash u : \uparrow [\![A]\!] \qquad \dfrac{\dfrac{\dfrac{}{y{:}[\![A]\!] \vdash_v y : [\![A]\!]}\,\text{id} \quad \dfrac{}{x{:}[\![B]\!] \vdash_v x : [\![B]\!]}\,\text{id}}{x{:}[\![B]\!], y{:}[\![A]\!] \vdash_v (y,x) : [\![A]\!] \otimes [\![B]\!]}\otimes}{x{:}[\![B]\!], y{:}[\![A]\!] \vdash \Uparrow (y,x) : \uparrow\big([\![A]\!] \otimes [\![B]\!]\big)}\uparrow}{\Gamma, x{:}[\![B]\!] \vdash u \,\mathrm{to}\, y.\, \Uparrow (y,x) : \uparrow\big([\![A]\!] \otimes [\![B]\!]\big)}\,\text{chain}}{\Gamma, \Delta \vdash v \,\mathrm{to}\, x.\, u \,\mathrm{to}\, y.\, \Uparrow (y,x) : \uparrow\big([\![A]\!] \otimes [\![B]\!]\big)}\,\text{chain}$$

A product continuation waits for the computation to be finished and continues with the two components

$$\dfrac{\dfrac{\dfrac{\Gamma, x{:}[\![A]\!], y{:}[\![A]\!] \vdash c}{\Gamma \vdash \mu\,(x,y)\,.\,c : [\![A]\!]^\perp \parr [\![B]\!]^\perp}\parr}{\Gamma \vdash_v \Downarrow\mu\,(x,y)\,.\,c : \downarrow\big([\![A]\!]^\perp \parr [\![B]\!]^\perp\big)}\,\text{thunk}}{\Gamma \vdash \Uparrow\Downarrow\mu\,(x,y)\,.\,c : \uparrow\downarrow\big([\![A]\!]^\perp \parr [\![B]\!]^\perp\big)}\uparrow$$

The introduction rules of nullary connectives 1 and $\perp$ are straightforward.

The small discrepancy which prevents this translation to be only done with macro is due to the fact that linear L is a single-sided sequent calculus. In a two-sided sequent calculus, the cut rule is already asymetric and their are two μ-binders: one for the left-hand side variables and one for the right-hand side variables. The call-by-value translation of a two-sided linear L has right-hand types translated to positive types, and left-hand types to negative types. This is loosely the same as the call-by-value restriction in the original system L paper [CH00].

4.2.4. *Call-by-name linear* L. The negative translation of linear L follows along the same lines. It has a call-by-name flavour, as witnessed by the identity rule

$$\dfrac{\dfrac{}{x{:}\downarrow N \vdash_v x : \downarrow N}\,\text{id}}{x{:}\downarrow N \vdash \mathrm{force}\ x : N}\,\text{force}$$

Hypotheses are shifted in the context, and forced at use point. A product $N \otimes M$ is encoded as $\uparrow\big(\downarrow [\![N]\!] \otimes \downarrow [\![M]\!]\big)$, so that a pair contains suspended computations:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash u : [\![N]\!]}{\Gamma \vdash_v \Downarrow u : \downarrow [\![N]\!]}\,\text{thunk} \quad \dfrac{\Delta \vdash v : [\![M]\!]}{\Delta \vdash_v \Downarrow v : \downarrow [\![M]\!]}\,\text{thunk}}{\Gamma, \Delta \vdash_v (\Downarrow u, \Downarrow v) : \downarrow [\![N]\!] \otimes \downarrow [\![M]\!]}\otimes}{\Gamma, \Delta \vdash \Uparrow(\Downarrow u, \Downarrow v) : \uparrow\Big(\downarrow [\![N]\!] \otimes \downarrow [\![M]\!]\Big)}\uparrow$$

While a pair continuation, whose type is $[\![N^\perp \parr M^\perp]\!] = \uparrow [\![N]\!]^\perp \parr \uparrow [\![M]\!]^\perp$, binds the two (suspended) components of a product:

$$\frac{\Gamma, x{:}\downarrow[\![N]\!], y{:}\downarrow[\![M]\!] \vdash c}{\Gamma \vdash \mu\,(x,y)\,.\,c : \uparrow[\![N]\!]^\perp \parr \uparrow[\![M]\!]^\perp}\ \parr$$

For the cut rule, we proceed like in the call-by-value case: we have either $[\![N^\perp]\!] = \uparrow[\![N]\!]^\perp$ or $\downarrow[\![N^\perp]\!] = [\![N]\!]^\perp$. The cut rule is oriented such that cutting with a pair continuation amounts to forcing the outermost pair constructor.

$$\frac{\Gamma \vdash u : \uparrow[\![N]\!]^\perp \quad \dfrac{\Delta \vdash v : [\![N]\!]}{\Delta \vdash_v \Downarrow v : \downarrow[\![N]\!]}\ \text{thunk}}{\Gamma, \Delta \vdash \langle\, u \,|\, \Downarrow v \,\rangle}\ \text{cut}$$

Like in the call-by-value translation, the μ rule is duplicated:

$$\frac{\dfrac{\Gamma, x{:}\downarrow[\![N]\!] \vdash c}{\Gamma \vdash_v \mu x.\, c : \uparrow[\![N]\!]^\perp}\ \downarrow}{\Gamma \vdash \mu x.\, c : [\![N^\perp]\!]}\ \text{definition}$$

$$\frac{\dfrac{\dfrac{}{y{:}\downarrow[\![N^\perp]\!] \vdash y : \downarrow[\![N^\perp]\!]}\ \text{id}}{y{:}\downarrow[\![N^\perp]\!] \vdash \text{force}\ y : [\![N^\perp]\!]}\ \text{force} \quad \dfrac{\dfrac{\Gamma, x{:}\downarrow[\![N]\!] \vdash c}{\Gamma \vdash \mu x.\, c : \uparrow[\![N]\!]^\perp}\ \mu}{\Gamma \vdash \mu x.\, c : \uparrow\downarrow[\![N^\perp]\!]}\ \text{definition}}{\Gamma \vdash \mu x.\, c\,\text{to}\,y.\,\text{force}\ y : [\![N^\perp]\!]}\ \text{chain}$$

The evaluation strategy of the negative translation of linear L, evaluating the outermost constructor on demand, is essentially the same behaviour as lazy programming language such as Haskell (except that lazy programming languages have a call-by-need strategy, hence suspended computations must be shared rather than duplicated).

Like in the call-by-value case, if we translated a two-sided sequent calculus, we would obtain a macro-translation. And it would correspond to the call-by-name calculus of [CH00].

5. Dependent types

Polarised L, with its type-based distinction of values and computations, gives an answer to the question of what it means for type to have effects in them: the answer is, they should not exist. If we see the type $\prod_{x:A} N$ as a generalisation of the type $A \multimap N$, it is clear that x only stands for values.

The obvious limitation is that even pure computation are ostensibly forbidden in types, preventing proofs by computation which are quite popular in modern dependent-type-theory-based proof assistants [Bou97]. Another, somewhat separate, issue which is not addressed in this last section is that of the computation *of* types (aka *strong eliminta-tion*), which would be necessary to prove, for instance, the equivalence of $A \oplus B$ and $\sum_{x:1\oplus 1} \langle\, x \,|\, \{\mu\,(1.())\,.\,A\,,\, \mu\,(2.())\,.\,B\}\,\rangle$. The latter will not even be a valid type.

Despite these limitation, this modest proposal for a dependently typed linear logic is already a fairly expressive logic which includes dependent elimination – as described in Section 5.2.

5.1. **Weak dependent types.** A first approach to extend linear L or polarised L with dependent types is to leverage the remark that the duplicable context behaves like a natural deduction context. We could therefore define a dependent product $\prod_{x:A} N$ like in natural deduction. This dependent product would generalise $!A \multimap N$ and we would retain a separate, non-dependent, linear arrow $A \multimap N$. Such a system would be along the lines of linear LF [CP96], except in sequent calculus form rather than a natural deduction.

There are no particular difficulty with this approach, but it has severe limitations. The most important limitation is that such a system cannot be extended to dependent elimination. The system we propose in this section has a dependent product which generalises the linear arrow. In Section 5.2, we enrich it with dependent elimination.

A key point of our presentation of L so far, is that $\mu\alpha.\ \langle\, t \,|\, \alpha \,\rangle$ is essentially the same as just t, this underlies, in particular, our encoding of pattern-matching, and of linear λ-calculus. This is problematic if types depend on linear variables. Indeed, if t has type N, which depend on some linear variable x, we need α to have type $N^\perp$ which also depends on x. But variables have to be split between t and α so x can only go on one side of the cut.

This is where the polarised discipline helps: as variables represent only values, we can restrict type to contain values which are harmless in that they cannot perform effects by themselves. So, it is innocuous to allow variable duplication in types. In this proposal, it is manifested by a third context – usually denoted Θ – in the typing judgement of value, which represents variables accessible from the type, but not from the value.

This new typing context affects principally the identity and cut rules. Indeed another way one can think about the typing context is that it allows variables to have any type.

$$\frac{\Xi, \Theta \vdash A : \star}{\Xi \ ;\ \Theta \ ;\ x{:}A \vdash_v x : A}\ \text{id} \qquad \frac{\Xi \ ;\ \Theta \ ;\ \Gamma \vdash_v t : A \quad \Xi \ ;\ \Theta, \Delta \vdash u : A^\perp}{\Xi \ ;\ \Theta, \Gamma, \Delta \vdash \langle\, t \,|\, u \,\rangle}\ \text{cut}$$

In the cut rule, the variables which are necessary to make sense of the types in Γ and A but occur in the computation u are kept in the typing context of the value t. In the identity rule, the duplicable context and the typing context are joined: types do not make a difference between linear and duplicable variables.

The full system is given in Figure 5. It makes the simplifying hypothesis that the type of duplicable variables does not depend on linear variables. When we write Γ, Δ, it is implied that types in Δ may depend on variables of Γ but types of Γ may not depend on variables of Δ. Also, all context which appear in the premise of a typing rule is supposed to make sense, hence in the cut and tensor rule, it is understood that Γ and Δ do not depend on each other, and in the introduction rule for $?A^\perp$, the context Γ does not depend on x. These independence constraints are omitted for the sake of readability. Apart from the dependencies between bindings in the context, the context is not assumed to have a particular ordering. In particular, the context does not have a linear structure, in contrast with common practice in dependently typed natural deduction. Notice that we use a dependent product $\prod_{x:A} N$ which generalises the linear arrow, rather than a more symmetric generalisation of $N \parr M$, this is simply by lack of a notation for the latter. We assume that duality commutes with conversion[1].

As a consequence of this presentation, values do not reference the linear variables which occur in their types. This is a form of uniformity which values have to conform to: in the

[1]There is no way, in Figure 5, to actually have values in types. So of course, substitution – which is the identity – commutes with dualisation. This assumption should rather be seen as a constraint on extensions of the system.

Reduction

$$
\begin{array}{lcl}
\langle\, t \,|\, \mu x.\, c \,\rangle & \rightsquigarrow & c[\, x \backslash t \,] \\
\langle\, \mu \Uparrow x \,.\, c \,|\, \Uparrow t \,\rangle & \rightsquigarrow & c[\, x \backslash t \,] \\
\langle\, (t,u) \,|\, \mu\,(x,y)\,.\, c \,\rangle & \rightsquigarrow & c[\, x \backslash t, y \backslash u \,] \\
\langle\, () \,|\, \mu().\, c \,\rangle & \rightsquigarrow & c \\
\langle\, 1.t \,|\, \{\mu\,(1.x)\,.\, c_1 \,,\, \mu\,(2.y)\,.\, c_2\} \,\rangle & \rightsquigarrow & c_1[\, x \backslash t \,] \\
\langle\, 2.t \,|\, \{\mu\,(1.x)\,.\, c_1 \,,\, \mu\,(2.y)\,.\, c_2\} \,\rangle & \rightsquigarrow & c_2[\, y \backslash t \,] \\
\langle\, \lfloor t \rfloor \,|\, \mu\,\lfloor x \rfloor\,.\, c \,\rangle & \rightsquigarrow & c[\, x \backslash t \,]
\end{array}
$$

Typing

$$\frac{\Xi, \Theta \vdash A : \star}{\Xi \;;\; \Theta \;;\; x{:}A \vdash_v x : A}\ \mathrm{id} \qquad \frac{\Xi \;;\; \Theta \;;\; \Gamma \vdash_v t : A \quad \Xi \;;\; \Theta, \Delta \vdash u : A^\perp}{\Xi \;;\; \Theta, \Gamma, \Delta \vdash \langle\, t \,|\, u \,\rangle}\ \mathrm{cut}$$

$$\frac{\Xi \vdash A : \star}{\Xi, x{:}A \;;\; \Theta \;;\; \cdot \vdash_v x : A}\ \mathrm{id'} \qquad \frac{\Xi \;;\; \Gamma, x{:}A, \Psi \vdash c}{\Xi \;;\; \Gamma \vdash \mu x.\, c : A^\perp}\ \mu$$

$$\frac{\Xi \;;\; \Theta \;;\; \Gamma, x{:}A \vdash c}{\Xi \;;\; \Theta \;;\; \Gamma \vdash_v \mu \Uparrow x \,.\, c : \downarrow A^\perp}\ \downarrow \qquad \frac{\Xi \;;\; \Gamma \vdash_v t : A}{\Xi \;;\; \Gamma \vdash \Uparrow t : \uparrow A}\ \uparrow$$

$$\frac{\Xi \;;\; \Theta \;;\; \Gamma \vdash_v t : A \quad \Xi \;;\; \Theta \;;\; \Delta \vdash_v u : B[\, x \backslash t \,] \quad \Xi, \Theta \vdash \sum_{x:A} B : \star}{\Xi \;;\; \Theta \;;\; \Gamma, \Delta \vdash_v (t,u) : \sum_{x:A} B}\ \otimes$$

$$\frac{\Xi \;;\; \Gamma, x{:}A, y{:}B \vdash c \quad \Xi, \Gamma \vdash \prod_{x:A} B^\perp : \star}{\Xi \;;\; \Gamma \vdash \mu\,(x,y)\,.\, c : \prod_{x:A} B^\perp}\ \parr$$

$$\frac{}{\Xi \;;\; \Theta \;;\; \cdot \vdash_v () : 1}\ 1 \qquad \frac{\Xi \;;\; \Gamma \vdash c}{\Xi \;;\; \Gamma \vdash \mu().\, c : \bot}\ \bot$$

$$\frac{\Xi \;;\; \Theta \;;\; \Gamma \vdash_v t : A}{\Xi \;;\; \Theta \;;\; \Gamma \vdash_v 1.t : A \oplus B}\ \oplus\mathrm{l} \qquad \frac{\Xi \;;\; \Gamma, x{:}A \vdash c_1 \quad \Xi \;;\; \Gamma, y{:}B \vdash c_2}{\Xi \;;\; \Gamma \vdash \{\mu\,(1.x)\,.\, c_1 \,,\, \mu\,(2.y)\,.\, c_2\} : A^\perp \& B^\perp}\ \&$$

$$\frac{\Xi \;;\; \Theta \;;\; \Gamma \vdash_v u : B}{\Xi \;;\; \Theta \;;\; \Gamma \vdash_v 2.u : A \oplus B}\ \oplus\mathrm{r}$$

No rule for 0

$$\frac{}{\Xi \;;\; \Gamma \vdash \{\} : \top}\ \top$$

$$\frac{\Xi \;;\; \Theta \;;\; \cdot \vdash_v t : A}{\Xi \;;\; \Theta \;;\; \cdot \vdash_v \lfloor t \rfloor : !A}\ !$$

$$\frac{\Xi, x{:}A \;;\; \Gamma \vdash c}{\Xi \;;\; \Gamma \vdash \mu\,\lfloor x \rfloor\,.\, c : ?A^\perp}\ ?$$

Types

$$\frac{}{\Theta \vdash 1 : \star} \qquad \frac{}{\Theta \vdash \bot : \star} \qquad \frac{\Theta \vdash A : \star \quad \Theta, x{:}A \vdash B : \star}{\Theta \vdash \sum_{x:A} B : \star} \qquad \frac{\Theta \vdash A : \star \quad \Theta, x{:}A \vdash N : \star}{\Theta \vdash \prod_{x:A} N : \star}$$

$$\frac{}{\Theta \vdash 0 : \star} \qquad \frac{}{\Theta \vdash \top : \star} \qquad \frac{\Theta \vdash A : \star \quad \Theta \vdash B : \star}{\Theta \vdash A \oplus B : \star} \qquad \frac{\Theta \vdash N : \star \quad \Theta \vdash M : \star}{\Theta \vdash N \& M : \star}$$

$$\frac{\Theta \vdash N : \star}{\Theta \vdash \downarrow N : \star} \qquad \frac{\Theta \vdash A : \star}{\Theta \vdash \uparrow A : \star} \qquad \frac{\Theta \vdash A : \star}{\Theta \vdash !A : \star} \qquad \frac{\Theta \vdash N : \star}{\Theta \vdash ?N : \star}$$

Figure 5: Weak dependent L

types of values, term variables behave a little like the type variables of the Hindley-Milner type system.

Maybe surprisingly, as the only substitution occur in the premises of the tensor rule, the typing rules, in weak dependent L, of linear λ-abstraction and application correspond to standard rules for dependently typed λ-calculus – except linear.

Linear λ abstraction $\lambda x.\, t$ which is defined, as before, as $\mu\,(x,\alpha)\,.\,\langle\, t \,|\, \alpha \,\rangle$ demonstrates the essential use of the typing context:

$$\dfrac{\dfrac{\Xi\ ;\ \Gamma, x{:}A \vdash t : N \quad \dfrac{\Xi, \Gamma, x{:}A \vdash N^{\perp} : \star}{\Xi\ ;\ \Gamma, x{:}A\ ;\ \alpha{:}N^{\perp} \vdash_v \alpha : N^{\perp}}\ \mathrm{id}}{\Xi\ ;\ \Gamma, x{:}A, \alpha{:}N^{\perp} \vdash \langle\, t \,|\, \alpha \,\rangle}\ \mathrm{cut}}{\Xi\ ;\ \Gamma \vdash \mu\,(x,\alpha)\,.\,\langle\, t \,|\, \alpha \,\rangle : \prod_{x:A} N}\ \parr$$

The typing derivation of the application $t\ u = \mu\alpha.\,\langle\, t \,|\, (u,\alpha) \,\rangle$ indeed performs a substitution in the body of $\prod_{x:A} N$. The well-formedness conditions of types are omitted for brevity:

$$\dfrac{\dfrac{\Xi\ ;\ \Gamma \vdash t : \prod_{x:A} N \qquad \dfrac{\Xi\ ;\ \Gamma\ ;\ \Delta \vdash_v u : A \quad \overline{\Xi\ ;\ \Gamma\ ;\ \alpha{:}N^{\perp}[\,x\backslash u\,] \vdash_v \alpha : N^{\perp}[\,x\backslash u\,]}\ \mathrm{id}}{\Xi\ ;\ \Gamma\ ;\ \Delta, \alpha{:}N^{\perp}[\,x\backslash u\,] \vdash_v (u,\alpha) : \sum_{x:A} N^{\perp}}\ \otimes}{\Xi\ ;\ \Gamma, \Delta, \alpha{:}N^{\perp}[\,x\backslash u\,] \vdash \left\langle\, t \,\middle|\, (u,\alpha) \,\right\rangle}\ \mathrm{cut}}{\Xi\ ;\ \Gamma, \Delta \vdash \mu\alpha.\,\left\langle\, t \,\middle|\, (u,\alpha) \,\right\rangle : N[\,x\backslash u\,]}\ \mu$$

Weak dependent L is not very expressive, nonetheless the various implementations of LF have demonstrated that this weak kind of dependent types can already be quite useful [Pfe91, AHMP92, HHP93, PS99]. It is also pleasant that we could derive a natural definition for a dependently typed *linear* λ-calculus from design choices which are largely technical, and were not meant to force this definition.

5.2. **Dependent elimination.** If weak dependent L can encode dependently typed linear λ-calculus, as it happens, this does not extend to regular λ-calculus, as we shall demonstrate momentarily. This is because it lacks so-called dependent elimination. Going back to the always quite representative multiplicative fragment, let us consider the following statement, in ML-like syntax:

```
let (x,y) = u in v
```

or in the syntax of L:

$$\mu\alpha.\,\left\langle\, u \,\middle|\, \mu\,(x,y)\,.\,\langle\, v \,|\, \alpha \,\rangle \,\right\rangle$$

In weak dependent L, the type of v cannot depend on x or y as it has the dual type of α's, but α is not in the scope of x and y.

Concretely, let us suppose that we have a positive type $u = v$ whose formation rule is given by:

$$\dfrac{\Xi\ ;\ \Gamma \vdash u : A \quad \Xi\ ;\ \Gamma \vdash v : A}{\Xi, \Gamma \vdash u = v : \star}$$

And introduction rule

$$\dfrac{\Xi, \Theta \vdash u = u : \star}{\Xi\ ;\ \Theta\ ;\ \Gamma \vdash_v \mathrm{refl} : u = u}$$

We will not need the dual type.

With such a type it would be desirable, but is not possible in weak dependent L, to prove the statement

$$\prod_{x:A\otimes B}\Big\uparrow \sum_{y:A}\sum_{z:B} x = (y,z)$$

That every value in $A \otimes B$ is a pair.

Dependent elimination allows, in let (x,y) = u in v to link u with (x,y) in the type of v. Dependent elimination comes in two main brands: Paulin's style [PM93], as used by Coq, and Coquand's style [Coq92], as used by Agda. In Paulin elimination, the type of v may contain references to (x,y) which are transformed into occurrences of u in the type of the whole term. Coquand's elimination does the same to the whole typing context. Since, in L, elimination is introduced over commands, which do not have a distinguished type, dependent elimination in dependent L will resemble Coquand elimination most.

Looking closely at the representation of elimination in L

$$\mu\alpha.\, \langle\, u \,|\, \mu\,(x,y)\,.\,\langle\, v \,|\, \alpha\,\rangle\,\rangle$$

we can observe that dependent elimination must be split in two parts – in contrast with natural deduction where elimination is a single operation. First, in $\mu\,(x,y)\,.\,\langle\, v \,|\, \alpha\,\rangle$, the types of v and α may depend on (x,y) and this dependency must be hidden as x and y are going out of scope. And in a second time – when cutting with u – u must be linked into the type.

This leads to the idea of introducing a distinguished variable, which we write $\bullet$ which stands for the value against which the current computation will be cut[2]. Notice that by definition, since variables only stand for values and values are cut against computation, the *cut variable* $\bullet$ can only appear in the typing judgement of computations, not of values, nor of commands.

In the typing judgements of computations, we use names such as $N_\bullet$ or $\Gamma_\bullet$ as reminders that the type or context can contain the cut variable. Here is the cut rule:

$$\frac{\Xi\ ;\ \Theta\ ;\ \Gamma \vdash_v t : A \quad \Xi\ ;\ \Theta, \Delta_\bullet \vdash u : A^\perp}{\Xi\ ;\ \Theta, \Gamma, \Delta_\bullet[\,\bullet\backslash t\,] \vdash \langle\, t \,|\, u\,\rangle}\ \text{cut}$$

Notice how, since the duplicable context is shared between the value and the computation, it does not make sense for it to contain the cut variable. The cut rule eliminates the cut variable by replacing it with the value t as the semantics of the cut variable mandates.

The cut variable is introduced by the μ-binders, notably in the introduction rule for $N \parr M$:

$$\frac{\Xi\ ;\ \Gamma, x{:}A, y{:}B, \Gamma_\bullet[\,\bullet\backslash(x,y)\,] \vdash c \quad \Xi, \Gamma \vdash \prod_{x:A} B^\perp}{\Xi\ ;\ \Gamma, \Gamma_\bullet \vdash \mu\,(x,y)\,.\,c : \prod_{x:A} B^\perp}\ \parr$$

So, indeed, c is typed as if the hypothetical cut value was (x,y). In this rule $\prod_{x:A} B^\perp$ does not mention the cut variable. Just as in the cut rule, the type $A^\perp$ does not mention the cut variable. Types of computation are duals of types of values which do not have the cut variable. All of the mentions of type variable are in the context, which is not a problem since the current continuation can be hosted there. Indeed here is the rule for dependent

[2]Credit where credit is due: this idea was originally formulated by Hugo Herbelin in an unpublished draft. The context was an intuitionistic dependent type theory in sequent calculus form. It is mentionned at the very end of [Her05, Chapter 3]

elimination of pairs:

$$\dfrac{\Xi\ ;\ \Gamma, x{:}A, y{:}B \vdash v : N_\bullet[\,\bullet\backslash(x,y)\,] \quad \dfrac{\Xi, \Gamma, x{:}A, y{:}B \vdash N_\bullet[\,\bullet\backslash(x,y)\,]^\perp : \star}{\Xi\ ;\ \Gamma, x{:}A, y{:}B\ ;\ \alpha{:}N_\bullet[\,\bullet\backslash(x,y)\,]^\perp \vdash_v \alpha : N_\bullet[\,\bullet\backslash(x,y)\,]^\perp}\ \mathrm{id}}{\dfrac{\Xi\ ;\ \Gamma, x{:}A, y{:}B, \alpha{:}N_\bullet[\,\bullet\backslash(x,y)\,]^\perp \vdash \langle\, v \,|\, \alpha \,\rangle}{\Xi\ ;\ \Gamma, \alpha{:}N_\bullet{}^\perp \vdash \mu\,(x,y)\,.\,\langle\, v \,|\, \alpha \,\rangle : A^\perp \parr B^\perp}\ \parr}\ \mathrm{cut}$$

Which can be further combined with a value u to obtain a rule akin to dependent elimination in natural deduction (omitting the type well-formedness constraint):

$$\dfrac{\Xi\ ;\ \Gamma\ ;\ \Delta \vdash_v u : A \otimes B \quad \begin{array}{c}\Xi\ ;\ \Gamma, x{:}A, y{:}B \vdash v : N_\bullet[\,\bullet\backslash(x,y)\,] \\ \vdots \\ \Xi\ ;\ \Gamma, \alpha{:}N_\bullet{}^\perp \vdash \mu\,(x,y)\,.\,\langle\, v \,|\, \alpha \,\rangle : A^\perp \parr B^\perp\end{array}\ \mathrm{cut}}{\dfrac{\Xi\ ;\ \Gamma, \Delta, \alpha{:}N_\bullet{}^\perp[\,\bullet\backslash u\,] \vdash \left\langle\, u \,\middle|\, \mu\,(x,y)\,.\,\langle\, v \,|\, \alpha \,\rangle \right\rangle}{\Xi\ ;\ \Gamma, \Delta \vdash \mu\alpha.\,\left\langle\, u \,\middle|\, \mu\,(x,y)\,.\,\langle\, v \,|\, \alpha \,\rangle \right\rangle : N_\bullet[\,\bullet\backslash u\,]}\ \mu}\ \mathrm{cut}$$

The full system can be found in Figure 6. Let us illustrate dependent L further by showing the rules for dependent product of duplicable terms. That is, the typing derivation for $\lambda\,\lfloor x \rfloor\,.\,t$ which is defined (see Section 3.1) as $\mu\,(\lfloor x \rfloor\,, \alpha)\,.\,\langle\, t \,|\, \alpha \,\rangle = \mu\,(\beta, \alpha)\,.\,\langle\, \beta \,|\, \mu\,\lfloor x \rfloor\,.\,\langle\, t \,|\, \alpha \,\rangle\,\rangle$ (using shortcut rules for variables as in Section 3.1, the shortcut omit type well-formedness verification):

$$\dfrac{\dfrac{\dfrac{\dfrac{\Xi, x{:}A\ ;\ \cdot \vdash t : N_\bullet[\,\bullet\backslash\lfloor x \rfloor\,]}{\Xi, x{:}A\ ;\ \alpha{:}N_\bullet{}^\perp[\,\bullet\backslash\lfloor x \rfloor\,] \vdash \langle\, t \,|\, \alpha \,\rangle}\ \mathrm{cut}\,\alpha}{\Xi\ ;\ \alpha{:}N_\bullet{}^\perp \vdash \mu\,\lfloor x \rfloor\,.\,\langle\, t \,|\, \alpha \,\rangle : ?A^\perp}\ ?}{\Xi\ ;\ \beta{:}!A, \alpha{:}N_\bullet{}^\perp[\,\bullet\backslash\beta\,] \vdash \left\langle\, \beta \,\middle|\, \mu\,\lfloor x \rfloor\,.\,\langle\, t \,|\, \alpha \,\rangle \right\rangle}\ \mathrm{cut}\,\beta}{\Xi\ ;\ \cdot \vdash \mu\,(\beta, \alpha)\,.\,\left\langle\, \beta \,\middle|\, \mu\,\lfloor x \rfloor\,.\,\langle\, t \,|\, \alpha \,\rangle \right\rangle : \prod_{\beta:!A} N_\bullet[\,\bullet\backslash\beta\,]}\ \otimes$$

As it happens this is not exactly the rule of dependent product in λ-calculus. Namely, all the free variables in a type are of the form $\lfloor x \rfloor$ while it is not necessarily the case of bound variables. Remember, however, from Section 3.2 how intuitionistic conjunction and disjunction could be encoded as $!A \otimes !B$ and $!A \oplus !B$ respectively. Encoding of intuitionistic positive connectives require a strict interleaving of linear positive connectives and the duplicability modality. This is essentially the same phenomenon which is reflected in the duplicable dependent product rule.

As a last example, let us return to the equality example of the beginning of the present section. We promised a proof derivation for:

$$\prod\nolimits_{x:A\otimes B} \Big\uparrow \sum\nolimits_{y:A} \sum\nolimits_{z:B} x = (y,z)$$

This sort of statement is quite precisely what dependent elimination provides. A derivation of this statement can be found in Figure 7 Note the necessary step which uses the cut variable, without the cut variable the derivation is no longer valid (in weak dependent L, the cut rule would lead to $\cdot\ ;\ \cdot\ ;\ \cdot \vdash \mathrm{refl} : x = (y,z)$, which is certainly an underivable sequent).

There is a limit to dependent L, however: it does not enjoy subject reduction. This is due to the fact that commutative cuts are simply reductions in L. Commutative conversion are not usually correct in dependently typed languages with dependent elimination (though there are much more many correct commutative conversion with Coquand elimination than with Paulin elimination).

REDUCTION

$$
\begin{array}{lcl}
\langle\, t \,|\, \mu x.\, c \,\rangle & \rightsquigarrow & c[\, x \backslash t \,] \\
\langle\, \mu \Uparrow x \,.\, c \,|\, \Uparrow t \,\rangle & \rightsquigarrow & c[\, x \backslash t \,] \\
\langle\, (t,u) \,|\, \mu\,(x,y)\,.\, c \,\rangle & \rightsquigarrow & c[\, x \backslash t, y \backslash u \,] \\
\langle\, () \,|\, \mu().\, c \,\rangle & \rightsquigarrow & c \\
\langle\, 1.t \,|\, \{\mu\,(1.x)\,.\, c_1 \,,\, \mu\,(2.y)\,.\, c_2\} \,\rangle & \rightsquigarrow & c_1[\, x \backslash t \,] \\
\langle\, 2.t \,|\, \{\mu\,(1.x)\,.\, c_1 \,,\, \mu\,(2.y)\,.\, c_2\} \,\rangle & \rightsquigarrow & c_2[\, y \backslash t \,] \\
\langle\, \lfloor t \rfloor \,|\, \mu\,\lfloor x \rfloor\,.\, c \,\rangle & \rightsquigarrow & c[\, x \backslash t \,]
\end{array}
$$

TYPING

$$\frac{\Xi, \Theta \vdash A : \star}{\Xi \;;\; \Theta \;;\; x{:}A \vdash_v x : A}\ \mathrm{id} \qquad \frac{\Xi \;;\; \Theta \;;\; \Gamma \vdash_v t : A \quad \Xi \;;\; \Theta, \Delta_\bullet \vdash u : A^\perp}{\Xi \;;\; \Theta, \Gamma, \Delta_\bullet[\,\bullet \backslash t\,] \vdash \langle\, t \,|\, u \,\rangle}\ \mathrm{cut}$$

$$\frac{\Xi \vdash A : \star}{\Xi, x{:}A, \Psi \;;\; \Theta \;;\; \cdot \vdash_v x : A}\ \mathrm{id'} \qquad \frac{\Xi \;;\; \Gamma, x{:}A, \Gamma_\bullet[\,\bullet \backslash x\,] \vdash c}{\Xi \;;\; \Gamma, \Gamma_\bullet \vdash \mu x.\, c : A^\perp}\ \mu$$

$$\frac{\Xi \;;\; \Gamma, x{:}A \vdash c}{\Xi \;;\; \Theta \;;\; \Gamma \vdash_v \mu \Uparrow x \,.\, c : \downarrow A^\perp}\ \downarrow \qquad \frac{\Xi \;;\; \Gamma \vdash_v t : A}{\Xi \;;\; \Gamma \vdash \Uparrow t \,:\, \uparrow A}\ \uparrow$$

$$\frac{\Xi \;;\; \Theta \;;\; \Gamma \vdash_v t : A \quad \Xi \;;\; \Theta \;;\; \Delta \vdash_v u : B[\, x \backslash t \,] \quad \Xi, \Theta \vdash \sum_{x:A} B : \star}{\Xi \;;\; \Theta \;;\; \Gamma, \Delta \vdash_v (t,u) : \sum_{x:A} B}\ \otimes$$

$$\frac{\Xi \;;\; \Gamma, x{:}A, y{:}B, \Gamma_\bullet[\,\bullet \backslash (x,y)\,] \vdash c \quad \Xi, \Gamma \vdash \prod_{x:A} B^\perp}{\Xi \;;\; \Gamma, \Gamma_\bullet \vdash \mu\,(x,y)\,.\, c : \prod_{x:A} B^\perp}\ \parr$$

$$\frac{}{\Xi \;;\; \Theta \;;\; \cdot \vdash_v () : 1}\ 1 \qquad \frac{\Xi \;;\; \Gamma, \Gamma_\bullet[\,\bullet \backslash ()\,] \vdash c}{\Xi \;;\; \Gamma, \Gamma_\bullet \vdash \mu().\, c : \bot}\ \bot$$

$$\frac{\Xi \;;\; \Theta \;;\; \Gamma \vdash_v t : A}{\Xi \;;\; \Theta \;;\; \Gamma \vdash_v 1.t : A \oplus B}\ \oplus\mathrm{l} \qquad \frac{\Xi \;;\; \Gamma, x{:}A, \Gamma_\bullet[\,\bullet \backslash 1.x\,] \vdash c_1 \quad \Xi \;;\; \Gamma, y{:}B, \Gamma_\bullet[\,\bullet \backslash 2.y\,] \vdash c_2}{\Xi \;;\; \Gamma, \Gamma_\bullet \vdash \{\mu\,(1.x)\,.\, c_1 \,,\, \mu\,(2.y)\,.\, c_2\} : A^\perp \& B^\perp}\ \&$$

$$\frac{\Xi \;;\; \Theta \;;\; \Gamma \vdash_v u : B}{\Xi \;;\; \Theta \;;\; \Gamma \vdash_v 2.u : A \oplus B}\ \oplus\mathrm{r}$$

$$\text{No rule for } 0 \qquad \frac{}{\Xi \;;\; \Gamma, \Gamma_\bullet \vdash \{\} : \top}\ \top$$

$$\frac{\Xi \;;\; \Theta \;;\; \cdot \vdash_v t : A}{\Xi \;;\; \Theta \;;\; \cdot \vdash_v \lfloor t \rfloor : !A}\ ! \qquad \frac{\Xi, x{:}A \;;\; \Gamma_\bullet[\,\bullet \backslash \lfloor x \rfloor\,] \vdash c}{\Xi \;;\; \Gamma_\bullet \vdash \mu\,\lfloor x \rfloor\,.\, c : ?A^\perp}\ ?$$

Figure 6: Dependent L with dependent elimination

As we have seen above, elimination of a value of tensor type can be given the following (specialised) type:

$$\frac{\Xi \;;\; \Gamma \;;\; \Delta \vdash_v u : A \otimes B \quad \Xi \;;\; \Gamma, x{:}A, y{:}B \vdash v : C_\bullet[\,\bullet \backslash (x,y)\,] \multimap N_\bullet[\,\bullet \backslash (x,y)\,]}{\Xi \;;\; \Gamma, \Delta \vdash \mu\alpha.\, \langle\, u \,\big|\, \mu\,(x,y)\,.\, \langle\, v \,|\, \alpha \,\rangle \,\rangle : C_\bullet[\,\bullet \backslash u\,] \multimap N_\bullet[\,\bullet \backslash u\,]}$$

With $\Xi \;;\; \Gamma, \Delta \;;\; \Pi \vdash_v t : C_\bullet[\,\bullet \backslash u\,]$ we have

$$\Xi \;;\; \Gamma, \Delta, \Pi \vdash \Big(\mu\alpha.\, \langle\, u \,\big|\, \mu\,(x,y)\,.\, \langle\, v \,|\, \alpha \,\rangle \,\rangle\Big)\ t : N_\bullet[\,\bullet \backslash u\,]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\ ;\ \cdot\ ;\ y{:}A \vdash_v y : A}\ \mathrm{id} \quad \dfrac{\dfrac{}{\cdot\ ;\ \cdot\ ;\ z{:}B \vdash_v z : B}\ \mathrm{id} \quad \dfrac{}{\cdot\ ;\ \cdot\ ;\ \cdot \vdash_v \mathrm{refl} : (y,z) = (y,z)}}{\cdot\ ;\ \cdot\ ;\ z{:}B \vdash_v (z, \mathrm{refl}) : \sum_{z':B} (y,z) = (y,z')}\otimes}{\cdot\ ;\ \cdot\ ;\ y{:}A, z{:}B \vdash_v \big(y, (z, \mathrm{refl})\big) : \sum_{y':A}\sum_{z':B} (y,z) = (y',z')}\otimes}{\cdot\ ;\ y{:}A, z{:}B \vdash \Uparrow\big(y, (z, \mathrm{refl})\big) \ : \ \Big\uparrow \sum_{y':A}\sum_{z':B} (y,z) = (y',z')}\uparrow}{\cdot\ ;\ y{:}A, z{:}B, \alpha{:}\Big(\Big\uparrow \sum_{y':A}\sum_{z':B} (y,z) = (y',z')\Big)^{\perp} \vdash \Big\langle \Uparrow\big(y, (z, \mathrm{refl})\big) \,\Big|\, \alpha \Big\rangle}\ \mathrm{cut}\,\alpha}{\cdot\ ;\ \alpha{:}\Big(\Big\uparrow \sum_{y:A}\sum_{z:B} \bullet = (y,z)\Big)^{\perp} \vdash \mu\,(y,z)\,.\,\Big\langle \Uparrow\big(y, (z, \mathrm{refl})\big) \,\Big|\, \alpha \Big\rangle : A^{\perp} \parr B^{\perp}}\ \parr}{\cdot\ ;\ x{:}A \otimes B, \alpha{:}\Big(\Big\uparrow \sum_{y:A}\sum_{z:B} x = (y,z)\Big)^{\perp} \vdash \Big\langle\, x \,\Big|\, \mu\,(y,z)\,.\,\Big\langle \Uparrow\big(y, (z, \mathrm{refl})\big) \,\Big|\, \alpha \Big\rangle\Big\rangle}\ \mathrm{cut}\,x}{\cdot\ ;\ x{:}A \otimes B \vdash \mu\alpha.\,\Big\langle\, x \,\Big|\, \mu\,(y,z)\,.\,\Big\langle \Uparrow\big(y, (z, \mathrm{refl})\big) \,\Big|\, \alpha \Big\rangle\Big\rangle : \Big\uparrow \sum_{y:A}\sum_{z:B} x = (y,z)}\ \mu}{\cdot\ ;\ \cdot \vdash \lambda x.\,\mu\alpha.\,\Big\langle\, x \,\Big|\, \mu\,(y,z)\,.\,\Big\langle \Uparrow\big(y, (z, \mathrm{refl})\big) \,\Big|\, \alpha \Big\rangle\Big\rangle : \prod_{x:A\otimes B} \Big\uparrow \sum_{y:A}\sum_{z:B} x = (y,z)}\ \lambda$$

Figure 7: Derivation of an equality result with dependent elimination

However, the application $(\mu\alpha.\,\langle\, u \,|\, \mu\,(x,y)\,.\,\langle\, v \,|\, \alpha\,\rangle\rangle)\ \ t$ reduces (against an arbitrary context) to $\mu\alpha.\,\langle\, u \,|\, \mu\,(x,y)\,.\,\langle\, v \,|\, (t,\alpha)\,\rangle\rangle$ (see Section 3.5). And, unfortunately, the latter is not necessarily well-typed. For it to be well-typed we need a stronger statement on t:

$$\forall v\Psi, (\Xi\ ;\ \Gamma\ ;\ \Psi \vdash v : A \otimes B) \to (\Xi\ ;\ \Gamma, \Psi\ ;\ \Pi_\bullet[\,\bullet\backslash v\,] \vdash t : C_\bullet[\,\bullet\backslash v\,])$$

If t has such types, then in particular $\Xi\ ;\ \Gamma, x{:}A, y{:}B\ ;\ \Pi_\bullet[\,\bullet\backslash(x,y)\,] \vdash t : C_\bullet[\,\bullet\backslash(x,y)\,]$, which will be the proof obligation since dependent elimination takes care of evolving the $\Pi_\bullet$ context through the proof.

Not all values are as such, however, and subject reduction may indeed fail. To fix subject reduction, we would need to rely on a more involved mechanism than simple substitutions to specialise the context. Some sort of equality constraints would probably work.

However, the lack of subject reduction is not necessarily bad. Indeed, if u, above, is of the form (a,b) (which every closed u is), then $\mu\alpha.\,\langle\,(a,b)\,|\,\mu\,(x,y)\,.\,\langle\, v \,|\, (t,\alpha)\,\rangle\rangle$ further reduces to $\mu\alpha.\,\big\langle\, v[\,x\backslash a, y\backslash b\,] \,\big|\, (t,\alpha)\,\big\rangle$ which has indeed the appropriate type. So, at least for closed terms, a failure of subject reduction can be fixed by additional reductions. This sort of type safety property can be proved using Krivine realisability [MM09]. Such a proof is left to future work, so let it be the last act of this article to state the following conjecture:

Conjecture 5.1 (Type safety)**.** For any closed term $\cdot\ ;\ \cdot \vdash t : N$ of negative type, the normal form c of the command $\langle\, t \,|\, \alpha\,\rangle$ has type $\cdot\ ;\ \alpha{:}N^{\perp} \vdash c$.

Conclusion

Linear L, together with its polarised and dependent flavours, is a rich toolkit which models a wide variety of phenomena in programming languages. It is very expressive, with many features from programming languages being macro-expressible in linear and polarised L.

A testimony to the robustness of the diverse variants is that the macros – λ-abstraction and deep pattern matching – defined for linear L are left unchanged in polarised and dependent L. Only their typing evolves. Though polarised L enforces a strong call-by-value style restriction, which forces translations from programming language to be more complex (but still macros) as we have finer control on their behaviour. This expressiveness makes linar L and its variants good candidates both as intermediate languages and as programming languages.

Dependent L gives a novel account of dependent types, where positive connectives – hence dependent elimination – are an essential part of the design. The limitations of dependent L clearly have to be addressed in the future. In particular, the lack of strong elimination. In a sense the types of dependent L are values, but strong elimination mandates that they be computations. This is still unclear what it would mean in this context, however. It is nonetheless desirable to expand the expressiveness of dependent L to have a family of system of the same kind as PTS, which requires strong elimination.

To give a more realistic account of programming languages (with or without dependent types), the logical types should be extended with inductive types. In the context of one-sided sequent calculus, the dual of an inductive type would be a co-inductive type. Inductive types would be positive, and hence introduced by values and eliminated by recursive fixed points (presumably defined by pattern-matching). Dually, co-inductive types are introduced by recursive fixed points and eliminated by values. This would validate the co-pattern point of view, heralded for instance in [APTS13].

References

[AHMP92] Arnon Avron, Furio Honsell, Ian A. Mason, and Robert Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354, December 1992.

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[APTS13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '13*, page 27, New York, New York, USA, 2013. ACM Press.

[AR99] V.Michele Abrusci and Paul Ruet. Non-commutative logic I: the multiplicative fragment. *Annals of Pure and Applied Logic*, 101(1):29–64, November 1999.

[BKR99] Nick Benton, Andrew Kennedy, and George Russell. Compiling standard ML to Java bytecodes. *ACM SIGPLAN Notices*, 34(1):129–140, January 1999.

[Bou97] Samuel Boutin. Using reflection to build efficient and certified decision procedures. *Theoretical Aspects of Computer Software*, 1281:515–529, 1997.

[BW96] Nick Benton and Philip Wadler. Linear logic, monads and the lambda calculus. *Logic in Computer Science, 1996. LICS'*, 1996.

[CH00] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. *ACM SIGPLAN Notices*, 35(9):233–243, September 2000.

[CMM10] Pierre-Louis Curien and Guillaume Munch-Maccagnoni. The duality of computation under focus. In Cristian Calude and Vladimiro Sassone, editors, *Theoretical Computer Science*, IFIP Advances in Information and Communication Technology, pages 165–181. Springer Berlin Heidelberg, 2010.

[Coq92] Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, 1992.

[CP96] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Logic in Computer Science, 1996.*, 72:1–72, 1996.

[DK13] Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming - ICFP '13*, page 429, 2013.

[DL06] Roy Dyckhoff and Stéphane Lengrand. LJQ: a strongly focused calculus for intuitionistic logic. In *Logical Approaches to Computational Barriers*, 2006.

[EMgS09] Jeff Egger, Rasmus Mø gelberg, and Alex Simpson. Enriching an effect calculus with linear types. *Computer Science Logic*, 2009.

[Fel90] Matthias Felleisen. On the expressive power of programming languages. In Neil Jones, editor, *ESOP'90*, number 432 in Lecture Notes in Computer Science, pages 134–151. Springer Berlin Heidelberg, 1990.

[Gir96] JY Girard. Proof-nets: the parallel syntax for proof-theory. *Lecture Notes in Pure and Applied Mathematics*, pages 1–28, 1996.

[Har12] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, Cambridge, 2012.

[Her05] Hugo Herbelin. *C'est maintenant qu'on calcule: au cœur de la dualité*. Habilitation, Université Paris 11, 2005.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[KB11] Neelakantan R. Krishnaswami and Nick Benton. A semantic model for graphical user interfaces. *ACM SIGPLAN Notices*, 46(9):45, September 2011.

[Lau02] Olivier Laurent. *Étude de la polarisation en logique*. PhD thesis, Université Aix-Marseille 2, 2002.

[LDM10] Stéphane Lengrand, Roy Dyckhoff, and James McKinna. A Focused Sequent Calculus Framework for Proof Search in Pure Type Systems. *Logical Methods in Computer Science*, 2010.

[Ler90] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report, Inria, 1990.

[Lev01] Paul Blain Levy. *Call-By-Push-Value*. PhD thesis, Queen Mary, University of London, 2001.

[Mar95] Simon Marlow. *Deforestation for Higher-Order Functional Programs*. PhD thesis, University of Glasgow, 1995.

[MM09] Guillaume Munch-Maccagnoni. Focalisation and classical realisability. In *Computer Science Logic*, number June, pages 409–423, 2009.

[MM13] Guillaume Munch-Maccagnoni. *Syntax and Models of a non-Associative Composition of Programs and Proofs*. PhD thesis, Université Paris Diderot - Paris 7, 2013.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. *Logical frameworks*, 1991.

[Plo75] Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq rules and properties. In Marc Bezem and JanFriso Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer Berlin Heidelberg, 1993.

[PS98] Simon Peyton Jones and André Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, September 1998.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf—a meta-logical framework for deductive systems. *Automated Deduction—CADE-16*, 1632:202–206, 1999.

[See89] RAG Seely. Linear logic, *-autonomous categories and cofree coalgebras. In *Categories in Computer Science and Logic*. American Mathematical Society, 1989.

[Wad03] Philip Wadler. Call-by-value is dual to call-by-name. *ACM SIGPLAN Notices*, 38(9):189–201, September 2003.

[Zei08] Noam Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 153(1-3):66–96, April 2008.