THE LINEAR ABSTRACT MACHINE

Y. LAFONT

Department of Computing, Imperial College, London, United Kingdom SW7 2BZ

Abstract. Linear Logic [6] provides a refinement of functional programming and suggests a new implementation technique, with the following features:
- a synthesis of strict and lazy evaluation,
- a clean semantics of side effects,
- no garbage collector.

Introduction

Let us consider some questions arising in the area of functional programming.

Update

Pure functional programming is elegant but not very efficient. Usually, references are added to make practical languages, but the transparency of the semantics is lost. In LISP, you do not need references to modify data. For example, the "function" *nconc* (physical concatenation of lists, Fig. 1) replaces the final *nil* of its first argument with its second one. By the way, it alters all data sharing the last cell of its first argument.

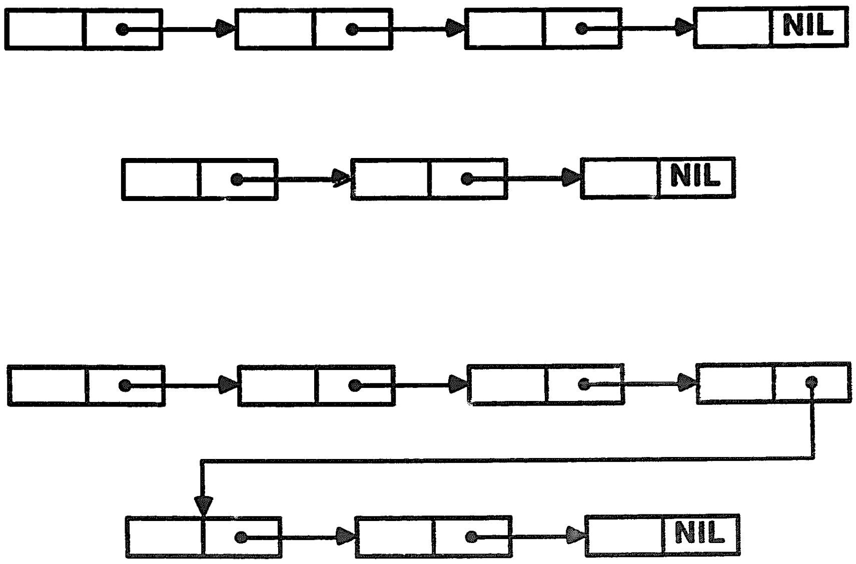

Fig. 1. *nconc*.

Is it possible to integrate those impure functions (that alter or destroy their arguments) into a declarative style?

Free cons

In some LISP dialects, the free list is accessible to the user so that your can manage your garbage collection yourself. Sometimes, it is clear that data are no longer useful, and it is reasonable to salvage space, but in general, you may be easily mistaken, with unfortunate consequences!

Is it possible to ensure that data will not be used later on?

Laziness

There are two main evaluation mechanisms of functional languages. The *strict* one (call by value) is often more efficient, but the *lazy* one (call by need) allows attractive infinite constructions.

Is it possible to mix *strictness* and *laziness* harmoniously?

You may tackle all these problems by means of ad hoc analysis tools, but they become very intricate when you allow higher-order functions.

What we propose is a typed calculus taking those "implementation details" into account. Doing this, we leave the area of functional programming, bringing in:

- a new kind of typed programming languages (*linear languages*)
- a new implementation technique (*Linear Abstract Machine*).

The underlying logic is the Intuitionistic Linear Logic of J.Y. Girard, as Intuitionistic Logic is the underlying logic of functional programming (Curry–Howard paradigm). Linear Logic [6] seems to answer many significant questions in computer science (see [5] for example).

Since Linear Logic may be unfamiliar to the reader, we give a short introduction to it (Sections 1 to 3). Then we present the Linear Abstract Machine, main subject of our paper (Sections 4 and 5), and we give a positive answer to the previous questions (Section 6). Linear λ-Calculus (Section 7) provides the basis of a programming language that can be implemented on that kind of machine. Quotations and technical details are carried over to appendices.

This paper is an extended version of [7].

1. Sequent Calculus

1.1. Gentzen Sequent Calculus

Less known than Natural Deduction, Gentzen Sequent Calculus [4, 9] is perhaps the most elegant presentation of Intuitionistic Logic. There we have a clear distinction between *structural* and *logical* rules (see Appendix A).

Structural rules are: exchange, identity (usually considered as an axiom, it can be restricted to atomic formulas), cut, contraction and weakening. The main property of Sequent Calculus is the cut elimination theorem (Hauptsatz): in the proof of any sequent $\Gamma\vdash A$, the cut rule can be eliminated.

From the Hauptsatz, one deduces consistency ($\vdash \bot$ cannot be proved) and the subformula property (every provable sequent $A_1, \ldots, A_n \vdash B$ has a proof that contains only subformulas of $A_1, \ldots, A_n$ and B).

If you forget the rules for the conjunction, you may hesitate between the following ones:

$$\frac{\Gamma\vdash A \qquad \Delta\vdash B}{\Gamma, \Delta\vdash A \wedge B} \quad \text{or} \quad \frac{\Gamma\vdash A \qquad \Gamma\vdash B}{\Gamma\vdash A \wedge B}$$

They are equivalent because you derive the second from the first:

$$\begin{array}{c} \dfrac{\Gamma\vdash A \qquad \Gamma\vdash B}{\Gamma, \Gamma\vdash A \wedge B} \\ \hline \vdots \\ \text{(contractions)} \\ \vdots \\ \hline \Gamma\vdash A \wedge B \end{array}$$

... and the first from the second:

$$\dfrac{\begin{array}{c} \Gamma\vdash A \\ \hline \vdots \\ \text{(weakenings)} \\ \vdots \\ \hline \Gamma, \Delta\vdash A \end{array} \qquad \begin{array}{c} \Delta\vdash B \\ \hline \vdots \\ \text{(weakenings)} \\ \vdots \\ \hline \Gamma, \Delta\vdash B \end{array}}{\Gamma, \Delta\vdash A \wedge B}$$

Note that a similar phenomenon happens with $\top$, which can be characterized by one of the following rules:

$$\overline{\vdash \top} \quad \text{or} \quad \overline{\Gamma\vdash \top}$$

1.2. Girard Sequent Calculus

If we remove contraction and weakening from Sequent Calculus, the previous rules are no longer equivalent, and correspond to distinct connectors.

We obtain Girard Sequent Calculus for Intuitionistic Linear Logic. The connectors are $\otimes$ (tensor product), $\mathbf{1}$ (tensor unit), $\multimap$ (linear implication), & (direct product), t (direct unit), $\oplus$ (direct sum) and $\mathbf{0}$ (direct zero)[1].

$$\frac{\Gamma, A, B, \Delta\vdash C}{\Gamma, B, A, \Delta\vdash C}\text{(exchange)} \qquad \frac{}{A\vdash A}\text{(identity)} \qquad \frac{\Gamma\vdash A \qquad \Delta, A\vdash B}{\Gamma, \Delta\vdash B}\text{(cut)}$$

(no contraction) (no weakening)

[1] In Classical Linear Logic [6], there is another disjunction (the tensor sum).

Logical rules:

$$\frac{\Gamma\vdash A \qquad \Delta\vdash B}{\Gamma,\Delta\vdash A\otimes B} \qquad \frac{\Gamma, A, B\vdash C}{\Gamma, A\otimes B\vdash C} \qquad \frac{}{\vdash \mathbf{1}} \qquad \frac{\Gamma\vdash A}{\Gamma, \mathbf{1}\vdash A}$$

$$\frac{\Gamma, A\vdash B}{\Gamma\vdash A\multimap B} \qquad \frac{\Gamma\vdash A \qquad \Delta, B\vdash C}{\Gamma, \Delta, A\multimap B\vdash C}$$

$$\frac{\Gamma\vdash A \qquad \Gamma\vdash B}{\Gamma\vdash A\& B} \qquad \frac{\Gamma, A\vdash C}{\Gamma, A\& B\vdash C} \qquad \frac{\Gamma, B\vdash C}{\Gamma, A\& B\vdash C} \qquad \frac{}{\Gamma\vdash \mathbf{t}}$$

$$\frac{\Gamma\vdash A}{\Gamma\vdash A\oplus B} \qquad \frac{\Gamma\vdash B}{\Gamma\vdash A\oplus B} \qquad \frac{\Gamma, A\vdash C \qquad \Gamma, B\vdash C}{\Gamma, A\oplus B\vdash C} \qquad \frac{}{\Gamma, \mathbf{0}\vdash A}$$

In Gentzen Sequent Calculus, a sequent $A_1,\dots,A_n\vdash B$ means that B is a consequence of $A_1\wedge\cdots\wedge A_n$. Here, it means that B is a consequence of $A_1\otimes\cdots\otimes A_n$.

Theorem 1.1 (Haupsatz for Intuitionistic Linear Logic). *In the proof of any sequent $\Gamma\vdash A$, the cut rule can be eliminated.*

The demonstration is similar to Gentzen's one, and even simpler because of the absence of contraction and weakening.

1.3. Examples of proofs

From now on, *linear* means *linear intuitionistic.*

We have a straightforward translation from linear formulas to intuitionistic ones, mapping $\otimes$ and $\&$ to $\wedge$ (conjunction); $\mathbf{1}$ and $\mathbf{t}$ to $\top$ (true); $\multimap$ to $\Rightarrow$ (implication); $\oplus$ to $\vee$ (disjunction) and $\mathbf{0}$ to $\perp$ (false).

If a linear formula is provable, its translation is a provable intuitionistic formula. For example, the linear formula $(A\& B)\multimap A$ (where A and B are atomic formulas) admits the following proof:

$$\frac{\dfrac{\dfrac{}{A\vdash A}}{A\& B\vdash A}}{\vdash (A\& B)\multimap A}$$

Of course, its translation $(A\wedge B)\Rightarrow A$ is provable. But the converse is false: The linear formula $(A\otimes B)\multimap A$ is not provable although its translation $(A\wedge B)\Rightarrow A$ is still provable.

Let us show that $(A\otimes B)\multimap A$ is not provable. A *cut free* proof of $\vdash (A\otimes B)\multimap A$ ends inevitably like this:

$$\frac{\dfrac{\dfrac{\vdots}{A, B\vdash A}}{A\otimes B\vdash A}}{\vdash (A\otimes B)\multimap A}$$

But without weakening, it is clearly impossible to deduce $A, B\vdash A$.

We have the following distributivity property:

$$A \otimes (B \oplus C) \equiv (A \otimes B) \oplus (A \otimes C).$$

Here $A \equiv B$ means that both $A \vdash B$ and $B \vdash A$ are provable. Of course, it is not true if you replace $\otimes$ by &. It is very easy to build a cut-free proof in a bottom-up fashion, for example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{A \vdash A} \quad \overline{B \vdash B}}{A, B \vdash A \otimes B}}{A, B \vdash (A \otimes B) \oplus (A \otimes C)} \quad \dfrac{\dfrac{\overline{A \vdash A} \quad \overline{C \vdash C}}{A, C \vdash A \otimes C}}{A, C \vdash (A \otimes B) \oplus (A \otimes C)}}{A, B \oplus C \vdash (A \otimes B) \oplus (A \otimes C)}}{A \otimes (B \oplus C) \vdash (A \otimes B) \oplus (A \otimes C)}$$

1.4. Classical Linear Logic

In [6], you will not find Intuitionistic but Classical Linear Logic. At first sight, Classical Linear Logic is to Intuitionistic Linear Logic what Classical Logic is to Intuitionistic Logic. For example, you have negation as a primitive connector, and the excluded middle law. However, Classical Linear Logic is constructive whereas Classical Logic is not!

We are now convinced that Classical Linear Logic is the most promising direction, but the intuitionistic case is probably more accessible and it was the first implemented.

2. Categorical Combinatory Logic

2.1. The intuitionistic case

Categorical Combinatory Logic (Appendix B) [11, 12, 2, 3] provides an alternative presentation of Intuitionistic Logic. A categorical combinator $\varphi : A \to B$ is the representation of a proof of $A \vdash B$ (one formula on the left side). In a way, Categorical Combinatory Logic is more elementary than Sequent Calculus. Its suitability to implementation is shown in [1, 16].

2.2. Linear combinators

The linear combinators are the basic operators for the theory of symmetric monoidal closed categories with finite products and coproducts (Appendix C):
Sequential compositors:

$$\dfrac{\varphi : A \to B \qquad \psi : B \to C}{\psi \circ \varphi : A \to C} \qquad \dfrac{}{\mathbf{id} : A \to A}$$

Parallel compositors:

$$\frac{\varphi:A\to B \qquad \psi:C\to D}{\varphi\otimes\psi:A\otimes C\to B\otimes D} \qquad \frac{}{\mathbf{1}:\mathbf{1}\to\mathbf{1}}$$

Arrange combinators:

$$\frac{}{\mathbf{assl}:A\otimes(B\otimes C)\leftrightarrow(A\otimes B)\otimes C:\mathbf{assr}}$$

$$\frac{}{\mathbf{insl}:A\leftrightarrow\mathbf{1}\otimes A:\mathbf{dell}} \qquad \frac{}{\mathbf{exch}:A\otimes B\leftrightarrow B\otimes A:\mathbf{exch}}$$

These combinators are invertible (we write $\varphi:A\leftrightarrow B:\psi$ for $\varphi:A\to B$ and $\psi:B\to A$). The names are acronyms: **assl** (associate left), **assr** (associate right), **insl** (insert left), **dell** (delete left) and **exch** (exchange).
Logical combinators:

$$\frac{\varphi:A\otimes B\to C}{\mathbf{cur}(\varphi):A\to B\multimap C} \qquad \frac{}{\mathbf{app}:(A\multimap B)\otimes A\to B}$$

$$\frac{\varphi:A\to B \qquad \psi:A\to C}{\langle\varphi,\psi\rangle:A\to B\&C} \qquad \frac{}{\mathbf{fst}:A\&B\to A} \qquad \frac{}{\mathbf{snd}:A\&B\to B} \qquad \frac{}{\langle\rangle:A\to t}$$

$$\frac{}{\mathbf{inl}:A\to A\oplus B} \qquad \frac{}{\mathbf{inr}:B\to A\oplus B} \qquad \frac{\varphi:A\to C \qquad \psi:B\to C}{\{\varphi|\psi\}:A\oplus B\to C} \qquad \frac{}{\{\}:\mathbf{0}\to A}$$

Proposition 2.1. *The two systems (Sequent Calculus and Categorical Combinatory Logic) are equivalent: For every combinator $A\to B$ there is a proof of $A\vdash B$, and for every proof of $A_1,\ldots,A_n\vdash B$ there is a combinator $A_1\otimes\cdots\otimes A_n\to B$.*

The proof is almost straightforward, the only problematic rules being

$$\frac{\Gamma, A\vdash C \qquad \Gamma, B\vdash C}{\Gamma, A\oplus B\vdash C} \qquad \frac{}{\Gamma, \mathbf{0}\vdash A}$$

But you can use the linear implication to send Γ to the right side.

For example, the proof of $A\otimes(B\oplus C)\vdash(A\otimes B)\oplus(A\otimes C)$ corresponds to the following combinator:

$$\mathbf{app}\circ(\{\mathbf{cur}(\mathbf{inl}\circ\mathbf{exch})|\mathbf{cur}(\mathbf{inr}\circ\mathbf{exch})\}\otimes\mathbf{id})\circ\mathbf{exch}:$$
$$A\otimes(B\oplus C)\to(A\otimes B)\oplus(A\otimes C)$$

2.3. *About the products*

A tensor product is clearly weaker than a cartesian one. For example, it allows permutation (**exch**), but no access (**fst**, **snd**). The main point is that $\otimes$ has a closure ($\multimap$), whereas the cartesian (or direct) product & has not. In fact, $A\otimes B$ can be considered as a type of strict pairs, and $A\&B$ as a type of lazy pairs. Indeed, **fst** and **snd** are not strict in their arguments (only one of them will be used), but the linear application **app** needs both the function and its argument.

3. The modality

It is possible to define recursive data types as in the intuitionistic case (e.g., concrete data types of ML).

3.1. *Inductive data types*

We have the usual equation defining the type of natural numbers:

$$\mathbf{Nat} = \mathbf{1} \oplus \mathbf{Nat}.$$

This is clearly enough in a programming language where general recursive programs are allowed. But for a logical system, we need an explicit primitive iteration operator, and Nat is characterized by the following rules:

$$\frac{}{\mathbf{zero} : \mathbf{1} \to \mathbf{Nat}} \qquad \frac{}{\mathbf{succ} : \mathbf{Nat} \to \mathbf{Nat}} \qquad \frac{\varphi : \mathbf{1} \to A \qquad \psi : A \to A}{\mathbf{iter}(\varphi, \psi) : \mathbf{Nat} \to A}$$

The reader may find explicit rules for the following inductive data types:

$$\mathbf{List}(A) = \mathbf{1} \oplus (A \otimes \mathbf{List}(A)),$$

$$\mathbf{BinaryTree}(A) = A \oplus \mathbf{1} \oplus (\mathbf{BinaryTree}(A) \otimes \mathbf{BinaryTree}(A)).$$

3.2. *Of course!*

If you replace $\oplus$ by & in the previous equations, you obtain the (dual) notion of projective data type. The modality $!A$ (read "of course A") may be characterized by the following equation:

$!A$ is a free coalgebra over A:

The corresponding rules are

$$\frac{}{\mathbf{read} : !A \to A} \qquad \frac{}{\mathbf{kill} : !A \to \mathbf{1}} \qquad \frac{}{\mathbf{dupl} : !A \to !A \otimes !A}$$

$$\frac{\varphi : A \to B \qquad \varepsilon : A \to \mathbf{1} \qquad \delta : A \to A \otimes A}{\mathbf{make}(\varphi, \varepsilon, \delta) : A \to !B}$$

The modality is exactly what we need to recover what is lost in Linear Logic: contraction and weakening. The similarity with $\mathbf{BinaryTree}(A)$ is obvious, but that definition is not complete (see Appendix D). However, it is enough for our purpose.

$!A$ is a free coalgebra over A:

Proposition 3.1. *Every $\varphi : !A \to B$ can be lifted to a combinator* $\mathbf{lift}(\varphi) : !A \to !B$.

Moreover, we have the following proposition.

Proposition 3.2. *The modality maps direct products to tensor products*:

$$!\mathbf{t} \equiv \mathbf{1}, \qquad !(A \& B) \equiv !A \otimes !B.$$

To prove this proposition, we have to construct the following combinators:

$$\mathbf{subl} : !\mathbf{t} \leftrightarrow \mathbf{1} : \mathbf{crys}, \qquad \mathbf{crac} : !(A \& B) \leftrightarrow !A \otimes !B : \mathbf{glue}.$$

The constructions are detailed in Appendix E.

3.3. Intuitionistic Logic recovered

With the modality *of course*, it is possible to recover the expressive power of Intuitionistic Logic. More precisely, we have an embedding of Intuitionistic Logic into Linear Logic. Hence, the linear connectors appear as more primitive than the intuitionistic ones.

The embedding maps an intuitionistic formula A to a linear one $|A|$:

- $|A| = A$ when A is an atomic formula;
- $|A \wedge B| = |A| \& |B|$ and $|\top| = \mathbf{t}$;
- $|A \Rightarrow B| = !|A| \multimap |B|$;
- $|A \vee B| = !|A| \oplus !|B|$ and $|\bot| = \mathbf{0}$.

Theorem 3.3. *An intuitionistic formula A is provable (in Categorical Combinatory Logic) if and only if its translation $|A|$ is provable (in Linear Categorical Combinatory Logic).*

To prove that A is provable when $|A|$ is provable, we simply extend the translation of Subsection 1.3 ($!A$ has the same interpretation as A).

For the opposite, we need the following lemma.

Lemma 3.4. *For every categorical combinator $\varphi : A \to B$, there is a linear one $|\varphi| : !|A| \to |B|$.*

The proof is a straightforward induction using Propositions 3.1 and 3.2. For example, if $\varphi : A \to B$ gives $|\varphi| : !|A| \to |B|$, and $\psi : B \to C$ gives $|\psi| : !|B| \to |C|$, then $|\psi \circ \varphi| = |\psi| \circ \mathbf{lift}(|\varphi|) : !|A| \to |C|$.

Theorem 3.3 means that Linear Logic with the modality has the power of Intuitionistic Logic. As a consequence, functional languages are implementable on the Linear Abstract Machine (Section 5). However, this implementation is not realistic because of the translation from categorical combinators to linear ones, which generates huge code.

3.4. Sequent rules for the modality

In fact, Girard considered the modality as a new connector characterized by the following Sequent rules:

$$\frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B} \qquad \frac{\Gamma \vdash B}{\Gamma, !A \vdash B} \qquad \frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B} \qquad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A}$$

$!\Gamma$ represents a sequence $!A_1, \ldots, !A_n$ in the last rule, which has apparently no nice translation into Categorical Combinators. By the way, our rules cannot easily be expressed in Sequent Calculus, but our modality is stronger (Girard's rules are derivable). Moreover, we do not introduce new concepts (modulo recursion, the modality is definable from direct and tensor product). We can say that our modality is an implementation of Girard's one.

4. The theory of execution

4.1. *Atomic and primitive combinators*

We start from a given graph whose points are called *atomic types* and whose arrows are called *atomic combinators*. For example, the atomic types may be the states, and the atomic combinators the basic actions of robots controlled by our computer.

A tensor product of atomic types is called a *primitive type* and a (sequential and parallel) composition of arrange and atomic combinators is called a *primitive combinator* (see Section 2). Clearly, domains and codomains of primitive combinators are primitive types. The effects of programs will be primitive combinators.

4.2. *Canonical combinators*

For every type A, we single out a class of combinators $\mu : X \to A$ (where X varies over primitive types) that we call *canonical combinators* of type A:[2]

- if A is atomic, the only canonical combinator of type A is $\mathbf{id} : A \to A$;
- the canonical combinators of type $A \otimes B$ are the $\mu \otimes \nu : X \otimes Y \to A \otimes B$ where $\mu : X \to A$ and $\nu : Y \to B$ are canonical combinators;
- the only canonical combinator of type $\mathbf{1}$ is $\mathbf{1}$;
- for any other type A, the canonical combinators of type A are the $\gamma \circ \mu : X \to A$ where $\gamma : Y \to A$ is a constructor (any combinator of the form $\mathbf{cur}(\varphi)$, $\langle \varphi, \psi \rangle$, $\langle\,\rangle$, $\mathbf{inl}$, $\mathbf{inr}$) and $\mu : X \to Y$ is canonical.[3]

For example, the canonical combinators of type $A \multimap B$ are the $\mathbf{cur}(\varphi) \circ \mu : X \to A \multimap B$ where $\varphi : Y \otimes A \to B$, and $\mu : X \to Y$ is canonical.

The *data* handled by programs will be canonical combinators.

4.3. *Execution relation*

We define an *execution relation* $\mu \rightsquigarrow^{\varphi}_{\alpha} \nu$ when $\varphi : A \to B$, $\mu : X \to A$, $\nu : Y \to B$ and $\alpha : X \to Y$, where μ, ν are canonical and α is primitive.

[2] Note the similarity with the canonical terms of [15].

[3] This definition is not well founded since Y is not, in general, a subformula of A. It does not matter: what is defined inductively is the relation "μ is a canonical combinator of type A".

$$\begin{tikzcd}
A \arrow[r, "\varphi"] & B \\
X \arrow[u, "\mu"] \arrow[r, "\alpha"'] & Y \arrow[u, "\nu"']
\end{tikzcd}$$

This relation will have the following meaning: "The program φ applied to the datum μ gives the datum ν, with the effect α". The notion of effect was not introduced deliberately: It comes naturally when you try to define an operational semantics of linear combinators.

The execution relation is defined inductively:

$$\frac{(\alpha \text{ is an atomic combinator})}{\mathbf{id} \rightsquigarrow^{\alpha}_{\alpha} \mathbf{id}} \qquad \frac{\mu \rightsquigarrow^{\varphi}_{\alpha} \mu' \qquad \mu' \rightsquigarrow^{\psi}_{\beta} \mu''}{\mu \rightsquigarrow^{\psi \circ \varphi}_{\beta \circ \alpha} \mu''} \qquad \frac{}{\mu \rightsquigarrow^{\mathbf{id}}_{\mathbf{id}} \mu}$$

$$\frac{\mu \rightsquigarrow^{\varphi}_{\alpha} \mu' \qquad \nu \rightsquigarrow^{\psi}_{\beta} \nu'}{\mu \otimes \nu \rightsquigarrow^{\varphi \otimes \psi}_{\alpha \otimes \beta} \mu' \otimes \nu'} \qquad \frac{}{1 \rightsquigarrow^{1}_{1} 1}$$

$$\frac{}{\lambda \otimes (\mu \otimes \nu) \rightsquigarrow^{\mathbf{assl}}_{\mathbf{assl}} (\lambda \otimes \rhd) \otimes \nu} \qquad \frac{}{(\lambda \otimes \mu) \otimes \nu \rightsquigarrow^{\mathbf{assr}}_{\mathbf{assr}} \lambda \otimes (\mu \otimes \nu)}$$

$$\frac{}{\mu \rightsquigarrow^{\mathbf{insl}}_{\mathbf{insl}} 1 \otimes \mu} \qquad \frac{}{1 \otimes \mu \rightsquigarrow^{\mathbf{dell}}_{\mathbf{dell}} \mu} \qquad \frac{}{\mu \otimes \nu \rightsquigarrow^{\mathbf{exch}}_{\mathbf{exch}} \nu \otimes \mu}$$

$$\frac{(\gamma \text{ is a constructor})}{\mu \rightsquigarrow^{\gamma}_{\mathbf{id}} \gamma \circ \mu} \qquad \frac{\lambda \otimes \mu \rightsquigarrow^{\varphi}_{\alpha} \nu}{(\mathbf{cur}(\varphi) \circ \lambda) \otimes \mu \rightsquigarrow^{\mathbf{app}}_{\alpha} \nu}$$

$$\frac{\mu \rightsquigarrow^{\varphi}_{\alpha} \nu}{\langle \varphi, \psi \rangle \circ \mu \rightsquigarrow^{\mathbf{fst}}_{\alpha} \nu} \qquad \frac{\mu \rightsquigarrow^{\psi}_{\alpha} \nu}{\langle \varphi, \psi \rangle \circ \mu \rightsquigarrow^{\mathbf{snd}}_{\alpha} \nu}$$

$$\frac{\mu \rightsquigarrow^{\varphi}_{\alpha} \nu}{\mathbf{inl} \circ \mu \rightsquigarrow^{\{\varphi|\psi\}}_{\alpha} \nu} \qquad \frac{\mu \rightsquigarrow^{\psi}_{\alpha} \nu}{\mathbf{inr} \circ \mu \rightsquigarrow^{\{\varphi|\psi\}}_{\alpha} \nu}$$

You can easily check the following proposition.

Proposition 4.1. *If $\mu \rightsquigarrow^{\varphi}_{\alpha} \nu$, then the corresponding square commutes (see Appendix* C.3):

$$\begin{tikzcd}
A \arrow[r, "\varphi"] & B \\
X \arrow[u, "\mu"] \arrow[r, "\alpha"'] \arrow[ur, phantom, "="] & Y \arrow[u, "\nu"']
\end{tikzcd}$$

To execute a program φ on a datum μ, you apply the execution rules in a bottom up fashion until you find a result ν and an effect α such that $\mu \rightsquigarrow^{\varphi}_{\alpha} \nu$. This is obviously deterministic, but it is not clear if it terminates. The main result is the following theorem.

Theorem [illegible] (Termination). *For every $\varphi : A \to B$ and μ a canonical combinator of type A, there is a canonical combinator ν of type B and a primitive combinator α such that $\mu \leadsto_{\alpha}^{p} \nu$.*

This theorem is proved in Appendix F. ν and α are uniquely determined by φ and μ. Since ν is the result and α the effect, we make an abuse of notation[4], writing $\nu = \varphi\mu$ and $\alpha = \partial\varphi\mu$. It is then possible to reformulate the execution rules in a nice way, for example,

$$(\varphi \circ \psi)\mu = \varphi(\psi\mu), \qquad \partial(\varphi \circ \psi)\mu = \partial\varphi(\psi\mu) \circ \partial\psi\mu.$$

5. The Linear Abstract Machine

The *Linear Abstract Machine* (LAM) is a cousin of the *Categorical Abstract Machine* [1]. In particular, it uses the same basic ingredients: code, environment and stack. It is a sequential implementation of the execution rules (Section 4).

5.1. Environment and code

The environment is a representation of a canonical combinator:

- the canonical combinator $\mu \otimes \nu$ is represented by a pair (u, v) (strict pair) and 1 is represented by $(\,)$;
- the canonical combinator $\gamma \circ \mu$ is represented by a pair $\gamma.u$ where γ is a piece of code.

The code is a list of primitive instructions. Sequential composition becomes concatenation (in opposite order), but parallel composition has to be sequentialized:

- $\varphi \circ \psi$ becomes ψ @ φ (concatenation) and **id** becomes [ ] (empty list);
- $\varphi \otimes \psi$ $(=(\mathbf{id} \otimes \psi) \circ (\varphi \otimes \mathbf{id}))$ becomes **[Splitl]** @ φ @ **[Consl;Splitr]** @ ψ @ **[Consr]**, and 1 $(=\mathbf{id})$ becomes [ ].

Splitl pushes the second component of the environment and gets to the first one, whereas **Consl** reconstructs the pair. **Splitr** and **Consr** are symmetric instructions. Of course, the sequence **[Consl;Splitr]** can be abbreviated into a single instruction **Swap** that exchanges the environment with the top of the stack.

For the other combinators, the translation is straightforward: **assl** becomes **[Assl]**, . . . , **cur**(φ) becomes **[Cur**(φ)**]**, $\langle\varphi, \psi\rangle$ becomes **[Pair**(φ, ψ)**]**, . . . , $\{\varphi \mid \psi\}$ becomes **[Altv**(φ, ψ)**]**, . . .

5.2. Execution

We can now specify the machine as shown in Table 1. We use the notation $x :: [x_1; \ldots; x_n]$ for the list $[x; x_1; \ldots; x_n]$.

[4] Here, the combinator φ is considered as a program taking an input of type A and returning an output of type B. On the other hand, the canonical combinators μ and ν are considered as data of types A and B respectively.

Table 1

Linear Abstract Machine					
before			after		
code	environment	stack	code	environment	stack
Split :: C	(u, v)	S	C	u	$v :: S$
Consl :: C	u	$v :: S$	C	(u, v)	S
Splitr :: C	(u, v)	S	C	v	$u :: S$
Consr :: C	v	$u :: S$	C	(u, v)	S
Assl :: C	$(u, (v, w))$	S	C	$((u, v), w)$	S
Assr :: C	$((u, v), w)$	S	C	$(u, (v, w))$	S
Insl :: C	u	S	C	$((), u)$	S
Dell :: C	$((), u)$	S	C	u	S
Exch :: C	(u, v)	S	C	(v, u)	S
$\gamma :: C$	u	S	C	$\gamma.u$	S
App :: C	$(\text{Cur}(C').u, v)$	S	C'	(u, v)	$C :: S$
Fst :: C	$\text{Pair}(C', C'').u$	S	C'	u	$C :: S$
Snd :: C	$\text{Pair}(C', C'').u$	S	C''	u	$C :: S$
Alt(C', C'') :: C	Inl.u	S	C'	u	$C :: S$
Altv(C', C'') :: C	Inr.u	S	C''	u	$C :: S$
[ ]	u	$C :: S$	C	u	S

5.3. *Possible extensions*

It is of course possible to add machine numbers, with primitive instructions for arithmetics, for example as shown in Table 2. In fact, because of the linear constraints, primitive instructions to duplicate or erase numbers will be necessary as well.

It is also possible to add primitive instructions for external effects, e.g., inputs and outputs as shown in Table 3. Those effects (especially the outputs) are similar to the atomic combinators we considered in the theoretical model of Section 4.

Table 2

before		after	
code	environment	code	environment
Num(p) :: C	$()$	C	p
Sum(p) :: C	(p, q)	C	$p+q$

Table 3

before		after		
code	environment	code	environment	effect
Input :: C	$()$	C	c	c is received from the keyboard
Output :: C	c	C	$()$	c is sent to the screen

5.4. *Examples of execution*

If α and β are instructions with some effect (for example, outputs) the program **fst** $\circ \langle \alpha, \beta \rangle$ is executed without the effect of β (see Table 4).

What happens now if we execute the program $\alpha \otimes \beta$? The answer is demonstrated in Table 5. Because we choose to implement parallel composition in a specific order, α is executed before β. But that order is not specified by the program: we have a form of arbitrary interleaving! In fact, if T is the (atomic) type representing a terminal[5], $\alpha \otimes \beta$ is of type $T \otimes T \rightarrow T \otimes T$. It needs two terminals! Technically, we replace the datum **id** (which contains no information) by the address (or port) of the terminal as shown in Table 6. There is no trouble since effects are executed on independent terminals.

5.5. *Looping code*

For the modality we do not need specific instructions. Indeed, at the implementation level, we have no reason to forbid looping code, so we use the recursive

Table 4

instruction	environment	stack	effect
	id		
Pair([α], [β])			
	Pair([α], [β]).**id**		
Fst			
	id	[ ]	
α			α
	id	[ ]	

Table 5

instruction	environment	stack	effect
	id$\otimes$**id**		
Splitl			
	id	**id**	
α			α
	id	**id**	
Consl			
	id$\otimes$**id**		
Splitr			
	id	**id**	
β			β
	id	**id**	
Consr			
	id$\otimes$**id**		

[5] A *console*, not a terminal object in a category!

Table 6

instruction	environment	stack	effect
	$port_1 \otimes port_2$		
Splitl			
	$port_1$	$port_2$	
α			α on $port_1$
	$port_1$	$port_2$	
Consl			
	$port_1 \otimes port_2$		
Splitr			
	$port_2$	$port_1$	
β			β on $port_2$
	$port_2$	$port_1$	
Consr			
	$port_1 \otimes port_2$		

definitions:

$$!A = A \& (1 \& (!A \otimes !A)),$$

$$\mathbf{read} = \mathbf{fst} : !A \to A,$$

$$\mathbf{kill} = \mathbf{fst} \circ \mathbf{snd} : !A \to 1, \qquad \mathbf{dupl} = \mathbf{snd} \circ \mathbf{snd} : !A \to !A \otimes !A,$$

$$\mathbf{make}(\varphi, \varepsilon, \delta) = \pi \quad \text{where } \pi = \langle \varphi, \langle \varepsilon, (\pi \otimes \pi) \circ \delta \rangle \rangle.$$

Therefore, the combinator **make**(_, _, _) is compiled into looping code.

6. Main features

Here we answer the questions posed in the Introduction.

6.1. Side effects

Nothing forbids us to add instructions for side effects, that is, internal modification of data. For example, you can introduce a predefined type of *difference lists* whose objects are represented by a pair of pointers (the head and the end of the list), and a primitive instruction for physical concatenation (as *nconc*).

That, of course, you can do in functional programming too, but then you expose yourself to *perverse* side effects since physical replacement may affect other parts of the environment. Nothing of the sort will happen here, for a very simple reason: there is no sharing!

6.2. Memory allocation

Consider the memory allocation of the Categorical Abstract Machine (Fig. 2). The environment is a graph whose nodes may be shared or orphan. It is even possible to have isolated structures. Because there are shared nodes, it is not safe

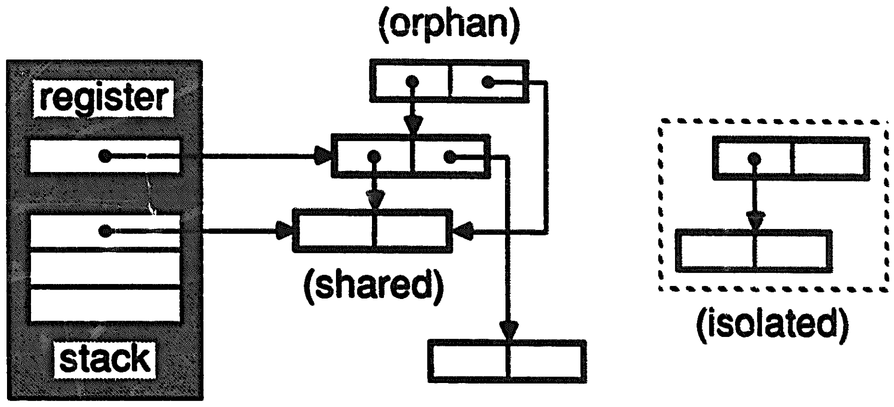

Fig. 2. The Categorical Abstract Machine.

to salvage a cell after an access. Because there are orphan nodes, it is useful to recover all unreachable cells when no free cell is available (garbage collection).

With the Linear Abstract Machine, the environment is a tree (without shared or orphan nodes). More precisely, there is a tree for the register and for each component of the stack (Fig. 3). Because there is no shared node, it is safe to salvage a cell after an access. Consider the typical example of **Splitr** (Fig. 4). After the access, the cell returns to the paradise of free *cons*! Access is not the only case: when you execute a destructor (**App**, **Fst**, **Snd**, **Altv**(C', C'')), you have to salvage the cell of the closure. Because there is no orphan node, the memory is full only in hopeless situations.

Finally, the Linear Abstract Machine is a machine without garbage collector. In other words, garbage collection is part of the job of the basic instructions.

6.3. Evaluation strategy

Connectors $\multimap$, &, $\oplus$ could be described as lazy, because the corresponding data are not evaluated: a constructor (**Cur**(C), **Pair**(C', C''), **Inl** or **Inr**) builds a closure which is evaluated by a destructor (**App**, **Fst**, **Snd** or **Altv**(C', C'')).

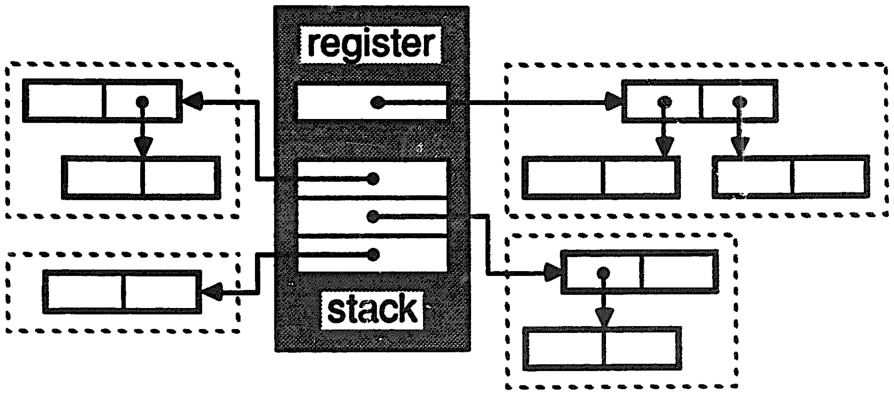

Fig. 3. The Linear Abstract Machine.

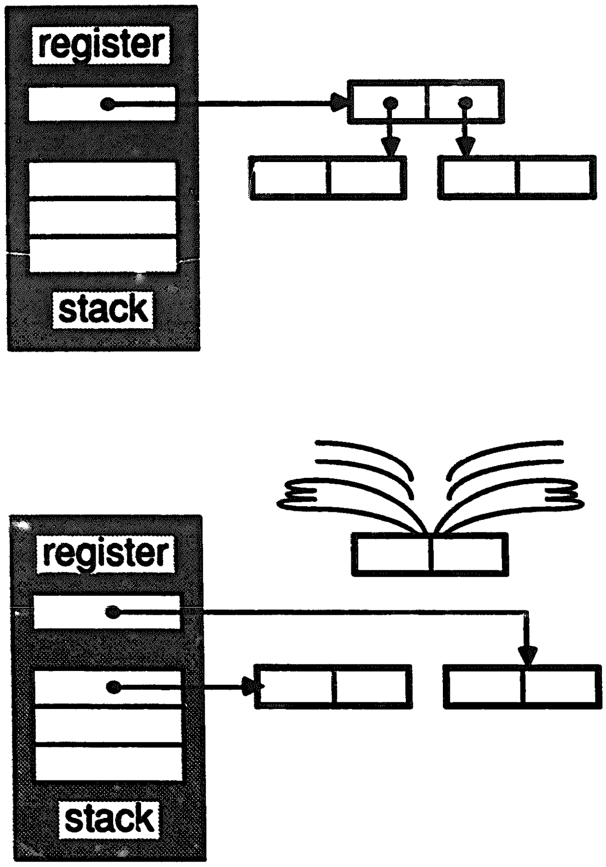

Fig. 4. Linear Abstract Machine working.

- A datum of type $A \multimap B$ is a $\mathbf{Cur}(\varphi).u$ where φ corresponds to a combinator $X \otimes A \rightarrow B$ and u is the environment of type X: it is a closure.
- A datum of type $A \& B$ is a $\mathbf{Pair}(\varphi, \psi).u$ where φ and ψ correspond to combinators $X \rightarrow A$ and u is the environment of type X.
- A datum of type $A \oplus B$ is a constructor **Inl** or **Inr** with a datum of type A or B: it is the usual representation of coproducts.

$A \otimes B$ is the type of strict pairs, and $A \& B$ is the type of lazy pairs. Of course, they correspond to different uses: strict structures are well adapted to permutation (e.g., sorting) whereas lazy ones are adapted to access (e.g., menus). In short, strictness and laziness coexist harmoniously in the same type system. The evaluation strategy ensures that only necessary computations are performed.

7. λ-Calculus

7.1. *Linear constraints*

The absence of contraction and weakening in Sequent Calculus corresponds to linear constraints in λ-calculus. We use λ-Calculus with explicit pair and conditional:

term:	*variable*	*pattern*:	*variable*
	constant		( )
	(*term*, *term*)		(*pattern*, *pattern*)
	λ *pattern*.*term*		
	term term		
	if *term* **then** *term* **else** *term*		

In a closed term, a variable must occur twice: once in a pattern, where it is bound, and once in the scope of this pattern, The conditional follows a special rule since the two members of the alternative must share the environment.

- $\mathit{Free}\, x = \{x\}$ (x variable),
- $\mathit{Free}\, c = \emptyset$ (c constant),
- $\mathit{Free}(M, N) = \mathit{Free}\, M \cup \mathit{Free}\, N$ ($\mathit{Free}\, M$ and $\mathit{Free}\, N$ must be disjoint),
- $\mathit{Free}(\lambda P.M) = \mathit{Free}\, M \setminus \mathit{Free}\, P$ ($\mathit{Free}\, P$ must be a subset of $\mathit{Free}\, M$),
- $\mathit{Free}(MN) = \mathit{Free}\, M \cup \mathit{Free}\, N$ ($\mathit{Free}\, M$ and $\mathit{Free}\, N$ must be disjoint),
- $\mathit{Free}(\textbf{if } M \textbf{ then } N \textbf{ else } N') = \mathit{Free}\, M \cup \mathit{Free}\, N$ ($\mathit{Free}\, M$ and $\mathit{Free}\, N$ must be disjoint, $\mathit{Free}\, N$ and $\mathit{Free}\, N'$ must be equal).

For example $\lambda x.x$ is a correct (closed) term, but $\lambda x.xx$ is not (independently of typing rules).

7.2. Typing rules

The typing rules are essentially the same as for usual λ-calculus. You simply replace the cartesian product by ⊗ and arrow by ⊸. The boolean type is **Bool** = **1**⊕**1**.

$$\frac{}{x : A \vdash x : A} \qquad \frac{x_i : A_i \vdash M : A \qquad y_j : B_j \vdash N : B}{x_i : A_i, y_j : B_j \vdash (M, N) : A \otimes B} \qquad \frac{}{\vdash () : \mathbf{1}}$$

$$\frac{x_i : A_i \vdash P : A \qquad x_i : A_i, y_j : B_j \vdash M : B}{y_j : B_j \vdash \lambda P.M : A \multimap B}$$

$$\frac{x_i : A_i \vdash M : A \multimap B \qquad y_j : B_j \vdash N : A}{x_u : A_i, y_j : B_j \vdash MN : B}$$

$$\frac{}{\vdash \mathbf{true} : \mathbf{Bool}} \qquad \frac{}{\vdash \mathbf{false} : \mathbf{Bool}}$$

$$\frac{x_i : A_i \vdash M : \mathbf{Bool} \qquad y_j : B_j \vdash N : A \qquad y_j : B_j \vdash N' : A}{x_i : A_i, y_j : B_j \vdash \textbf{if } M \textbf{ then } N \textbf{ else } N' : A}$$

It is possible to add rules for the direct product and the direct sum, but it does not essentially enriches the calculus since the corresponding constructions can be encoded in pure Linear λ-Calculus (with conditional and recursion).

7.3. A programming language

Pure Linear λ-Calculus is far simpler than pure λ-Calculus: reduction always terminates in linear time! However, it is possible to add recursive constructions to make a real programming language.

Conclusion

We recapitulate the panorama of Intuitionistic Linear Logic versus Intuitionistic Logic in Table 7. A more detailed study, with implementations is in [10]. We are now carrying out a similar study for the Classical Linear Logic of [6].

Table 7

logic	Intuitionistic Logic	Intuitionistic Linear Logic
structural rules	exchange, identity, cut, contraction, weakening	exchange, identity, cut (no contraction, no weakening)
connectors	$\wedge, \top, \Rightarrow, \vee, \bot$	$\otimes, \mathbf{1}, \multimap, \&, \mathbf{t}, \oplus, \mathbf{0}$ (+ modality)
categorical model	cartesian closed category (with finite coproducts)	symmetric monoidal closed category (with finite products and coproducts)
machine	Categorical Abstract Machine	Linear Abstract Machine
calculus	λ-Calculus	Linear λ-Calculus

Appendix A. Gentzen Sequent Calculus for Intuitionistic Logic

A sequent $A_1, \ldots, A_n \vdash B$ means that the formula B is a consequence of the hypotheses $A_1, \ldots, A_n$. In the following rules, Γ and Δ are sequences of formulas.
Structural rules:

$$\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\,\text{(exchange)},$$

$$\frac{}{A \vdash A}\,\text{(identity)}, \qquad \frac{\Gamma \vdash A \qquad \Delta, A \vdash B}{\Gamma, \Delta \vdash B}\,\text{(cut)},$$

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\,\text{(contraction)}, \qquad \frac{\Gamma \vdash B}{\Gamma, A \vdash B}\,\text{(weakening)}.$$

Logical rules:

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \qquad \frac{\Gamma, A \vdash C}{\Gamma, A \wedge B \vdash C} \qquad \frac{\Gamma, B \vdash C}{\Gamma, A \wedge B \vdash C} \qquad \frac{}{\vdash \top}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \frac{\Gamma \vdash A \qquad \Delta, B \vdash C}{\Gamma, \Delta, A \Rightarrow B \vdash C}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma, A \vdash C \qquad \Delta, B \vdash C}{\Gamma, \Delta, A \vee B \vdash C} \qquad \frac{}{\bot \vdash A}$$

Appendix B. Categorical combinators

$$\frac{}{\mathbf{id}: A \to A} \qquad \frac{\varphi : A \to B \quad \psi : B \to C}{\psi \circ \varphi : A \to C}$$

$$\frac{\varphi : A \to B \quad \psi : A \to C}{\langle \varphi, \psi \rangle : A \to B \wedge C} \qquad \frac{}{\mathbf{fst}: A \wedge B \to A} \qquad \frac{}{\mathbf{snd}: A \wedge B \to B}$$

$$\frac{}{\langle\,\rangle : A \to \mathbf{t}}$$

$$\frac{\varphi : A \wedge B \to C}{\mathbf{cur}(\varphi) : A \to B \Rightarrow C} \qquad \frac{}{\mathbf{app}: (A \Rightarrow B) \wedge A \to B}$$

$$\frac{}{\mathbf{inl}: A \to A \vee B} \qquad \frac{}{\mathbf{inr}: B \to A \vee B} \qquad \frac{\varphi : A \to C \quad \psi : B \to C}{\{\varphi | \psi\} : A \wedge B \to C}$$

$$\frac{}{\{\,\} : \bot \to A}$$

Appendix C. Categorical Linear Logic

C.1. Terminology

A symmetric monoidal category is a category $\mathscr{C}$ with a bifunctor $\otimes : \mathscr{C} \times \mathscr{C} \to \mathscr{C}$ and an object $\mathbf{1} \in \mathscr{C}$ such that

$$X \otimes (Y \otimes Z) \cong (X \otimes Y) \otimes Z, \qquad X \cong \mathbf{1} \otimes X, \qquad X \otimes Y \cong Y \otimes X.$$

$\cong$ denotes a *natural isomorphim*. These natural isomorphisms must satisfy the MacLane–Kelly equations (see Appendix C.3). A symmetric monoidal category $\mathscr{C}$ is *closed* if the functor $X \mapsto X \otimes A$ has a right adjoint $Y \mapsto A \multimap Y$ for every $A \in \mathscr{C}$:

$$\mathbf{Hom}(X \otimes A, Y) \cong \mathbf{Hom}(X, A \multimap Y).$$

A *categorical model* of Linear Logic is just a symmetric monoidal closed category $(\mathscr{C}, \otimes, \mathbf{1}, \multimap)$ with finite *products* $(\&, \mathbf{t})$ and *coproducts* $(\oplus, \mathbf{0})$.

C.2. Some categorical models

A category with finite products is a symmetric monoidal category; hence, a cartesian closed category with finite coproducts is a categorical model of Linear Logic[6]. We call those models *degenerate* because there is no distinction between tensor and direct product. In **Set**, for example, $\otimes$ (and $\&$) is the cartesian product, $\oplus$ is the disjoint union, and $X \multimap Y = Y^X$.

[6] This justifies the translation of Subsection 1.3.

A more interesting example is the category of modules over a fixed ring: $\otimes$ is the tensor product and $X \multimap Y = \mathbf{Hom}(X, Y)$. Here there is no distinction between direct product (&) and direct sum ($\oplus$).

The category **Top** of topological spaces is not cartesian closed, but is a categorical model of Linear Logic: $X \otimes Y$ is the set $X \times Y$ with the finest topology that makes sections $x \mapsto (x, y)$ and $y \mapsto (x, y)$ continuous. $\mathbf{1}$ is the usual one-point space. $X \multimap Y$ is the space of continuous maps $X \mapsto Y$ with the pointwise convergence topology. & is the usual cartesian product and $\oplus$ is the disjoint union. Tensor and direct product are different, but $\mathbf{1}$ and $\mathbf{t}$ are the same.

Another example is the category of pointed sets: An object is a set with a distinguished element $\perp$ and a morphism is a function preserving $\perp$. The tensor product of X and Y is $X \times Y$ where you identify all pairs containing a $\perp$, and the tensor unit is $\{\perp, \top\}$. $X \multimap Y$ is the set $\mathbf{Hom}(X, Y)$ with the constant bottom function as $\perp$. The direct product is the cartesian product, and the direct sum is the disjoint union where you identify the two $\perp$'s. $\mathbf{t}$ and $\mathbf{0}$ are the same.

C.3. *Equations*

The definitions in Appendix C.1 yield the following equational theory.

Category axioms:

$$(\varphi \circ \psi) \circ \chi = \varphi \circ (\psi \circ \chi), \qquad \mathbf{id} \circ \varphi = \varphi, \qquad \varphi \circ \mathbf{id} = \varphi.$$

Functoriality of the tensor product:

$$(\varphi \circ \varphi') \otimes (\psi \circ \psi') = (\varphi \otimes \psi) \circ (\varphi' \otimes \psi'), \qquad \mathbf{id} \otimes \mathbf{id} = \mathbf{id} : A \otimes B \to A \otimes B,$$

$$\mathbf{1} = \mathbf{id} : \mathbf{1} \to \mathbf{1}.$$

Naturality of arrange combinators:

$$\begin{tikzcd}
A \otimes (B \otimes C) \arrow[r, "\mathbf{assl}"] \arrow[d, "\varphi \otimes (\psi \otimes \chi)"'] \arrow[dr, phantom, "="] & (A \otimes B) \otimes C \arrow[d, "(\varphi \otimes \psi) \otimes \chi"] & \cdots \\
A' \otimes (B' \otimes C') \arrow[r, "\mathbf{assl}"'] & (A' \otimes B') \otimes C' &
\end{tikzcd}$$

Inverse equations:

$$\mathbf{assl} \circ \mathbf{assr} = \mathbf{id} : (A \otimes B) \otimes C \to (A \otimes B) \otimes C,$$

$$\mathbf{assr} \circ \mathbf{assl} = \mathbf{id} : A \otimes (B \otimes C) \to A \otimes (B \otimes C),$$

$$\mathbf{insl} \circ \mathbf{dell} = \mathbf{id} : \mathbf{1} \otimes A \to \mathbf{1} \otimes A, \qquad \mathbf{dell} \circ \mathbf{insl} = \mathbf{id} : A \to A,$$

$$\mathbf{exch} \circ \mathbf{exch} = \mathbf{id} : A \otimes B \to A \otimes B.$$

MacLane-Kelly equations:

$$\begin{tikzcd}
A\otimes(B\otimes(C\otimes D)) \arrow[r, "\mathbf{assl}"] \arrow[d, "\mathbf{id}\otimes\mathbf{assl}"] & (A\otimes B)\otimes(C\otimes D) \arrow[r, "\mathbf{assl}"] \arrow[d, phantom, "="] & ((A\otimes B)\otimes C)\otimes D \\
A\otimes((B\otimes C)\otimes D) \arrow[rr, "\mathbf{assl}"] & & (A\otimes(B\otimes C))\otimes D \arrow[from=1-3, "\mathbf{assl}\otimes\mathbf{id}"']
\end{tikzcd}$$

$$\begin{tikzcd}
 & A\otimes B \arrow[dl, "\mathbf{insl}"'] \arrow[dr, "\mathbf{insl}\otimes\mathbf{id}"] \arrow[d, phantom, "="] & \\
\mathbf{1}\otimes(A\otimes B) \arrow[rr, "\mathbf{assl}"] & & (\mathbf{1}\otimes A)\otimes B
\end{tikzcd}$$

$$\begin{tikzcd}
A\otimes(B\otimes C) \arrow[r, "\mathbf{assl}"] \arrow[d, "\mathbf{id}\otimes\mathbf{exch}"] & (A\otimes B)\otimes C \arrow[r, "\mathbf{exch}"] \arrow[d, phantom, "="] & C\otimes(A\otimes B) \arrow[d, "\mathbf{assl}"'] \\
A\otimes(C\otimes B) \arrow[r, "\mathbf{assl}"] & (A\otimes C)\otimes B \arrow[r, "\mathbf{exch}\otimes\mathbf{id}"] & (C\otimes A)\otimes B
\end{tikzcd}$$

Closure equations:

$$\mathbf{app}\circ(\mathbf{cur}(\varphi)\otimes\psi)=\varphi\circ(\mathbf{id}\otimes\psi), \qquad \mathbf{cur}(\varphi)\circ\psi=\mathbf{cur}(\varphi\circ(\psi\otimes\mathbf{id})),$$

$$\mathbf{cur}(\mathbf{app})=\mathbf{id}:A\multimap B\to A\multimap B.$$

Product equations:

$$\mathbf{fst}\circ\langle\varphi,\psi\rangle=\varphi, \qquad \mathbf{snd}\circ\langle\varphi,\psi\rangle=\psi,$$

$$\langle\varphi,\psi\rangle\circ\chi=\langle\varphi\circ\chi,\psi\circ\chi\rangle, \qquad \langle\mathbf{fst},\mathbf{snd}\rangle=\mathbf{id}:A\&B\to A\&B,$$

$$\langle\,\rangle\circ\varphi=\varphi, \qquad \langle\,\rangle=\mathbf{id}:\mathbf{t}\to\mathbf{t}.$$

Coproduct equations:

$$\{\varphi|\psi\}\circ\mathbf{inl}=\varphi, \qquad \{\varphi|\psi\}\circ\mathbf{inr}=\psi,$$

$$\chi\circ\{\varphi|\psi\}=\{\chi\circ\varphi|\chi\circ\psi\}, \qquad \{\mathbf{inl}|\mathbf{inr}\}=\mathbf{id}:A\oplus B\to A\oplus B,$$

$$\varphi\circ\{\,\}=\varphi, \qquad \{\,\}=\mathbf{id}:\mathbf{0}\to\mathbf{0}$$

Appendix D. The co-monad of course!

We use the modality $!A$ when we need an arbitrary number of copies of a datum of type A. But the rules of Section 3 do not specify that we have copies of the same datum. Since $\otimes$ is *not* a cartesian product, we cannot require, for example, that $\mathbf{dupl}:!A\to !A\otimes !A$ is the diagonal map. But things can be expressed in a roundabout way.

A type X with two morphisms $\varepsilon : X \to 1$ and $\delta : X \to X \otimes X$ is called a (commutative) *co-monoid* if it satisfies the following axioms:

$$\begin{tikzcd}
 & X \arrow[dl, "\delta"'] \arrow[dr, "\mathrm{insl}"] \arrow[d, phantom, "="] & \\
X \otimes X \arrow[rr, "\varepsilon \otimes \mathrm{id}"'] & {} & 1 \otimes X
\end{tikzcd}$$

(unit)

$$\begin{tikzcd}
 & X \arrow[dl, "\delta"'] \arrow[dr, "\delta"] & \\
X \otimes X \arrow[rr, "\mathrm{exch}"'] & & X \otimes X
\end{tikzcd}$$

(commutativity)

$$\begin{tikzcd}
 & X \arrow[dl, "\delta"'] \arrow[dr, "\delta"] & \\
X \otimes X \arrow[d, "\mathrm{id} \otimes \delta"'] & = & X \otimes X \arrow[d, "\delta \otimes \mathrm{id}"] \\
X \otimes (X \otimes X) \arrow[rr, "\mathrm{assl}"] & & (X \otimes X) \otimes X
\end{tikzcd}$$

(associativity)

Now we can give the correct definition of the modality: $!A$ is the *co-free co-monoid* co-generated by A. Hence, the correct rules are

$$\frac{}{\mathbf{read} : !A \to A} \qquad \frac{}{\mathbf{kill} : !A \to 1} \qquad \frac{}{\mathbf{dupl} : !A \to !A \otimes !A}$$

($!A$, **dupl**, **kill** is a co-monoid)

$$\frac{\varphi : A \to B \quad \varepsilon : A \to 1 \quad \delta : A \to A \otimes A \qquad (A, \delta, \varepsilon \text{ is a co-monoid})}{\mathbf{make}(\varphi, \varepsilon, \delta) : A \to !B}$$

We add the condition that $\mathbf{make}(\varphi, \varepsilon, \delta)$ is the only morphism $\pi : A \to !B$ such that

$$\mathbf{read} \circ \pi = \varphi, \qquad \mathbf{kill} \circ \pi = \varepsilon, \qquad \mathbf{dupl} \circ \pi = (\pi \otimes \pi) \circ \delta.$$

In the degenerate case (see Appendix C.2), the only co-monoids are the X, ε, δ where ε is the canonical arrow, and δ is the diagonal. That means that degenerate models are models of the modality, with $!A = A$.

However the most interesting result is the following categorical version of Theorem 3.3.

Theorem D.1. *If $\mathscr{C}$ is a symmetric monoidal closed category with finite products and modality, then the co-Kleisli category associated to the co-monad ! is cartesian closed*[7].

The proof uses the following result, which is a categorical version of Proposition 3.2.

Proposition D.2. *There is a canonical isomorphism*: $!(A \& B) \cong !A \otimes !B$.

[7] For these notions of *monad* and *Kleisli category*, see [14] for example.

Appendix E. Some combinators

$$\mathbf{insr} = \mathbf{exch} \circ \mathbf{insl} : A \to A \otimes 1, \qquad \mathbf{delr} = \mathbf{dell} \circ \mathbf{exch} : A \otimes 1 \to A,$$

$$\mathbf{mix} = \mathbf{asr} \circ ((\mathbf{assl} \circ (\mathbf{id} \otimes \mathbf{exch}) \circ \mathbf{assr}) \otimes \mathbf{id}) \circ \mathbf{assl} :$$
$$(A \otimes B) \otimes (C \otimes D) \to (A \otimes C) \otimes (B \otimes D),$$

$$\frac{x : !A \to B}{\mathbf{lift}(x) = \mathbf{make}(x, \mathbf{kill}, \mathbf{dupl}) : !A \to !B} \qquad \frac{x : A \to B}{!x = \mathbf{lift}(x \circ \mathbf{read}) : !A \to !B}$$

$$\mathbf{subl} = \mathbf{kill} : !\mathrm{t} \to 1, \qquad \mathbf{crys} = \mathbf{make}(\langle\,\rangle, \mathbf{id}, \mathbf{insl}) : 1 \to !\mathrm{t},$$

$$\mathbf{crac} = (!\mathbf{fst} \otimes !\mathbf{snd}) \circ \mathbf{dupl} : !(A \& B) \to !A \otimes !B,$$

$$\mathbf{glue} = \mathbf{make}(\langle \mathbf{delr} \circ (\mathbf{read} \otimes \mathbf{kill}), \mathbf{dell} \circ (\mathbf{kill} \otimes \mathbf{read})\rangle, \mathbf{dell} \circ (\mathbf{kill}$$
$$\otimes \mathbf{kill}), \mathbf{mix} \circ (\mathbf{dupl} \otimes \mathbf{dupl})) : !A \otimes !B \to !(A \& B).$$

Appendix F. Proof of the termination theorem

We prove Theorem 4.2. If $\varphi : A \to B$ is a combinator, and $\mathscr{C}$ a set of canonical combinators of type B, we write $\varphi\mu \downarrow \mathscr{C}$ when μ is a canonical combinator of type A and there is a $\nu \in \mathscr{C}$ and a primitive combinator α such that $\mu \rightsquigarrow^{\varphi}_{\alpha} \nu$. For every formula A, we construct inductively a set $\mathrm{Cv}(A)$ (*convergence* of A) of canonical combinators of type A:

- $\mathrm{Cv}(A) = \{\mathbf{id}\}$ (A is atomic),
- $\mathrm{Cv}(1) = \{\mathbf{1}\}$,
- $\mathrm{Cv}(A \otimes B) = \{\mu \otimes \nu \mid \mu \in \mathrm{Cv}(A), \nu \in \mathrm{Cv}(B)\}$,
- $\mathrm{Cv}(A \multimap B) = \{\mathbf{cur}(\varphi) \circ \mu \mid \varphi : X \otimes A \to B, (\forall \nu \in \mathrm{Cv}(A)) \varphi(\mu \otimes \nu) \downarrow \mathrm{Cv}(B)\}$,
- $\mathrm{Cv}(A \& B) = \{\langle \varphi, \psi \rangle \circ \mu \mid \varphi : X \to A, \psi : X \to B, \varphi\mu \downarrow \mathrm{Cv}(A), \psi\mu \downarrow \mathrm{Cv}(B)\}$,
- $\mathrm{Cv}(\mathrm{t}) = \{\langle\,\rangle \circ \mu\}$,
- $\mathrm{Cv}(A \oplus B) = \{\mathbf{inl} \circ \mu \mid \mu \in \mathrm{Cv}(A)\} \cup \{\mathbf{inr} \circ \mu \mid \mu \in \mathrm{Cv}(B)\}$,
- $\mathrm{Cv}(0) = \emptyset$.

By induction on *combinators*, and on *canonical combinators*, you check successively the following lemmas.

Lemma F.1. *For every combinator $\varphi : A \to B$ and $\mu \in \mathrm{Cv}(A)$, $\varphi\mu \downarrow \mathrm{Cv}(B)$.*

Lemma F.2. *For every A, $\mathrm{Cv}(A)$ is the set of all the canonical combinators of type A.*

The termination theorem is a consequence of these two lemmas.

Acknowledgment

The author is especially grateful to Jean-Yves Girard and Gérard Huet for their encouragement, and to Samson Abramsky for his helpful corrections and suggestions.

References

[1] G. Cousineau, P.L. Curien and M. Mauny, The categorical abstract machine, in: J.P. Jouannaud, ed., *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science **201** (Springer, Berlin, 1985) 50-64.

[2] P.L. Curien, Categorical combinatory logic, in: W. Brauer, ed., *Proc. ICALP '85*, Lecture Notes in Computer Science **194** (Springer, Berlin, 1985) 130–139.

[3] P.L. Curien, *Categorical Combinators, Sequential Algorithms and Functional Programming* (Pitman, London, 1986).

[4] E. Szabo, ed., *The Collected Papers of Gerhard Gentzen* (North-Holland, Amsterdam, 1969).

[5] J.Y. Girard, Linear logic and parallelism, in: *Proc. School on Semantics of Parallelism, IAC* (CNR, Roma, 1986).

[6] J.Y. Girard, Linear logic, *Theoret. Comput. Sci.* **50** (1987) 1–102.

[7] J.Y. Girard and Y. Lafont, Linear logic and lazy computation, in: *Proc. TAPSOFT '87, Vol. 2*, Lecture Notes in Computer Science **250** (Springer, Berlin, 1987) 52–66.

[8] G.M. Kelly, On MacLane's conditions for coherence of natural associativities, *J. Algebra* **1** (1964) 397–402.

[9] S.C. Kleene, *Introduction to Meta-mathematics* (North-Holland, Amsterdam, 1952).

[10] Y. Lafont, Logiques, catégories et machines, Thèse de Doctorat, Université de Paris 7, 1988.

[11] J. Lambek, Deductive systems and categories, *Math. Systems Theory* (1968).

[12] J. Lambek, From lambda-calculus to cartesian closed categories, in: J.P. Seldin and J.R. Hindley eds., *To H.B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism* (Academic Press, New York, 1980).

[13] J. Lambek and P.J. Scott, *Introduction to Higher Order Categorical Logic*, Cambridge Studies in Advanced Mathematics (Cambridge University Press, London, 1986).

[14] S. MacLane, *Categories for the Working Mathematician*, Graduate Texts in Mathematics **5** (Springer, Berlin, 1971).

[15] P. Martin-Löf, *Intuitionistic Type Theory*, Studies in Proof Theory, Lecture Notes (Bibliopolis, Napoli, 1984).

[16] M. Mauny and A. Suarez, Implementing functional languages in the categorical abstract machine, in: *Proc. Lisp and Functional Programming Conf.* (ACM, Boston, MA, 1986).

[17] M.E. Szabo, *Algebra of Proofs*, Studies in Logic and the Foundations of Mathematics **88** (North-Holland, Amsterdam, 1978).