Introduction to interaction nets

Yves Lafont
CNRS - Institut de Mathématiques de Luminy
163 avenue de Luminy, Case 907, 13288 Marseille Cedex 9
E-mail: lafont@iml.univ-mrs.fr

November 26, 1998

Models of computation

Although computation has been practiced for millennia, general abstract models of computation only appeared in the thirties. The most famous ones are *Turing machines*, which prefigure computers, and *Church lambda-calculus*, which led to a whole family of programming languages, called functional languages, such as *LISP* and *ML* (see inserts).

There are many other models, but all lead to the same notion of *computable function*. Most mathematicians and computer scientists agree on this notion and consider that any function which is computable by some mechanical process is also computable by a Turing machine. This is *Church Thesis*, which allows to assert and prove rigorously negative results such as the indecidability of the tenth Hilbert Problem: there is no algorithm deciding if a given Diophantine equation has a solution.

Turing machines are also used to define finer notions which express the feasibility of computation: functions which are computable in polynomial time, in polynomial space, etc. Nevertheless, this model is based on a purely sequential notion of computation, with a processor and a memory playing differentiated roles.

Contrasting with this conception, we have the model of *cellular automata*, which is based on the following principle: the state of a cell a time $n+1$ depends only on the state of this cell and of its neighbors at time n. This is of course a model of parallel computation, since all cells play similar roles and contribute collectively to the progress of computation. Note that, for the good running of operations, it is essential that all cells change at the same time; in other words, the transitions are local, but synchronous. This global temporality is not quite satisfying from a purely aesthetic viewpoint. More prosaically, it implies severe technological limitations on the design of parallel computers: need for a common clock, difficulties to synchronize nets of distant machines, etc.

Principles of interaction

For the time being, let us forget those models, and try to design one that would be general enough, but with good properties. First, one can see a computation as a kind of discrete physical process. If the space of computation is 1-dimensional, as in the case of Turing machines, the state of computation at a given time will be represented by a word over a finite alphabet. In general, it will be a *net* consisting of *cells* which are labelled by a finite number of *symbols*. We assume that all cells carrying the same symbol have the same number of *ports*, and that ports are connected pairwise.

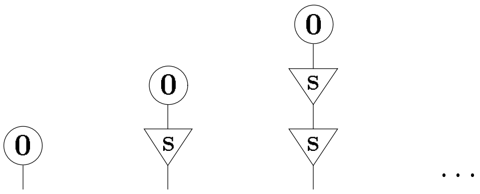
The dynamics of computation will be expressed by a finite number of local transitions. A priori, such a transition is a rewrite rule whose left member is a connected net with a certain number of *free ports*, and whose right member is a net with the same number of free ports. To get a truly local transition, we restrict ourselves to the case where the left member consists of two cells connected once. To avoid conflicts between those *interaction rules*, we fix one *principal port* for each cell, the other ports being called *auxiliary*, and we require that, in the left member of each rule, the two cells are connected by their principal ports. Finally, we allow at most one rule for each pair of symbols.

In this way, if two rules apply to the same net, one can execute both of them, one after the other or simultaneously, since they apply necessarily to disjoint subnets. Therefore, there is no need here for a global temporality: the transitions are local and asynchronous.

Practical work

In order to clearly understand how this works, let us try to program elementary arithmetical operations. If one does not worry about efficiency, it is convenient to adopt unary numeration: 0, s0 for 1, ss0 for 2, etc. So we start with the following two symbols:

In general, one draws a cell as a triangle with the principal port on one of its vertices and the auxiliary ports on the opposite edge. When there is only one port, it is necessarily principal, and one draws the cell as a circle. The natural numbers are represented by the following nets:



Such a net is *irreducible*: all principal ports are connected to auxiliary ports, except one which is free, and so, ready to interact with its environment.

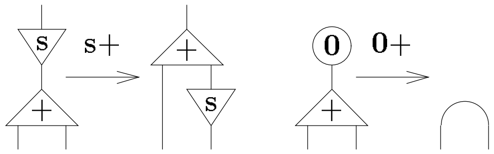
To program addition and multiplication, consider the equations which define them:

$$\mathrm{s}x + y = \mathrm{s}(x + y), \qquad 0 + y = y,$$

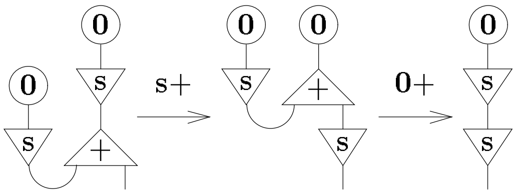
$$\mathrm{s}x \times y = (x \times y) + y, \qquad 0 \times y = 0.$$

In both cases, it is a definition by induction on the first argument, and that is where the principal port must be. The two auxiliary ports corresponding to the second argument and to the result of the operation:

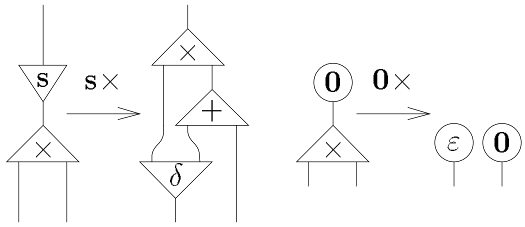
For addition, we get the following two rules:



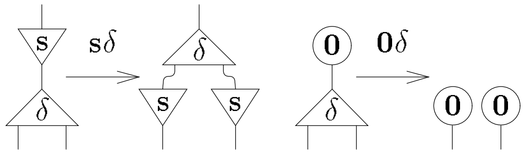
And here is how we compute $1 + 1$:

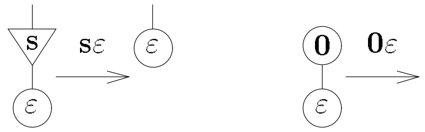


For multiplication, it is a bit more complicated, because the second argument y is used twice in a rule, not at all in the other one. So we must introduce a duplicator and a eraser:



The mechanisms of duplication and erasing are expressed by the following rules:





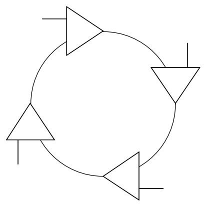
The reader can play at computing 2×2, and notice that rules can often be executed in parallel.

Programs and proofs

There is no widespread programming language based on this notion of interaction, but one can already glimpse the advantages of this approach by comparing it with functional programming, which is related to it. First, the interpretation of rules is transparent: there is no ambiguity related, for instance, to the scope of variables or to the order of execution. Furthermore, the management of memory is taken into account by the program itself, by means of erasing rules. More interesting: those nets represent interactive processes, with interleaved inputs and outputs, not only functions. Finally, a good parallel computer should be able to take advantage of the local and asynchronous character of transitions.

Originally, the *interaction nets* have not been designed as a programming language, not even as a general model of computation. The point was only to understand the mechanism of reduction of proofs in the *linear logic* of Jean-Yves Girard. This logic is a refinement of classical logic where the multiplicity of each hypothesis is taken into account. This allows to represent proofs in a geometrical form: the *proof nets*.

To be correct, the latter must satisfy a global condition which, in its simplest formulation due to Vincent Danos and Laurent Regnier, stipulates that certain graphs derived from the net itself are connected and acyclic. It happens that this condition of purely logical origin has a quite interesting interpretation in the framework of interaction nets. It prevents the creation of *vicious circles*, which are deadlock configurations of the following type:

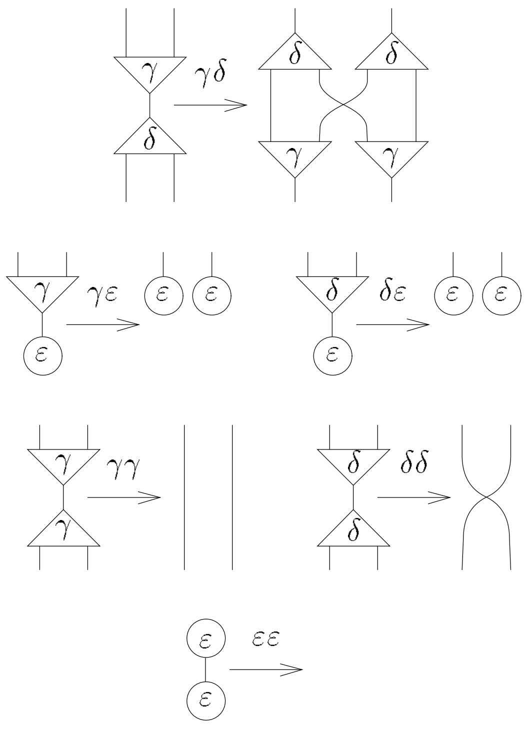


Since certain rules, such as $\mathbf{s}\times$ in the above example, may create cycles in a net, it is not at all obvious to foresee the creation of such a vicious circle. In fact, it is the acyclicity condition which is used here, and no extralogical justification is known for the condition of connectedness, but anyway, it is a beautiful example of transfer of concept between logic and computer science.

Interaction combinators

We have already written a program by means of interaction rules. One can think of many other, more or less complicated, ones. For instance, it is not hard to simulate a Turing machine, and even cellular automata, with interaction rules. One question arises naturally: is there any *universal* program, in which it is possible to translate any other program written with interaction rules? We must restrict ourselves to translations preserving the fine structure of computation: otherwise, any universal Turing machine would do.

The answer to this question is positive, but the most surprising thing is the simplicity of the solution. Indeed, three *combinators* γ, δ and ε suffice, with the following six rules:



One says that the first three rules, which involve two distinct symbols, are *commutations*, and the other three ones, which involve the same symbol twice, are *annihilations*. Those rules are reminiscent of the ones introduced by John Lamping in a very different perspective: the point was to implement the *shared reduction* of lambda-calculus.

The universality property means that, for a given program, it is possible to build, for each symbol α with n ports, a net of combinators $\Phi(\alpha)$ with n free ports, in such a way that, for each rule $R \to S$ of the program, the net $\Phi(R)$, that is R where each symbol has been replaced by its translation, reduces to $\Phi(S)$. In other words, each rule of the program is translated into a sequence of interaction between combinators. The latter may be considered as the elementary rules of computation.

The proof of this non trivial result is inspired by linear logic, and also by biology! Indeed, one starts by interpreting the whole program as a net of combinators Γ, and then one transforms this Γ into a net $!\Gamma$ which contains no δ. The latter is in some sense the genetic code of the program: it can be duplicated, erased, or executed. The translation of a symbol contains essentially the number of this symbol, as well as the genetic code, which is shared by all symbols.

Although the nets of combinators look very much like the *multiplicative* nets, which are the simplest proof nets, the links with logic are still mysterious.

Concrete machines

The interaction nets are an abstract model of computation, certainly less abstract than lambda-calculus, but still abstract. It seems indeed hard to think of any physical, chemical, or even biological process, that would *directly* implement the interaction rules of any program. The main difficulty comes from the fact that the geometry of the net changes in an unpredictable way.

So it is tempting to look for other modes of execution of nets, which would not explicitly involve the mechanism of reduction. It is with this aim in view that Jean-Yves Girard developed the *geometry of interaction* for proof nets. The idea consists in interpreting a net as a nilpotent operator on a Hilbert space, and the execution as a process of iteration of this operator. In fact, the Hilbertian structure is not really essential here: the point is to have enough space.

It happens that in the case of interaction combinators, the geometry of interaction has a particularly simple form: it provides a one-to-one mapping between the nets of combinators and the *reversible two-stack machines* over an alphabet with 2 letters. Those are variants of Turing machines, where the ribbon is replaced by stacks. The reversibility means that from the configuration, that is the state and the content of stacks, at a given time, it is always possible to recover the previous configuration. This mapping is very appealing, but for the time being, the resulting mode of execution is not quite convincing: the benefit of parallelism is lost, and furthermore, Vincent Danos and Laurent Regnier showed that execution may take exponential time in cases where reduction is linear.

A completely different approach consists in restricting oneself to *hard interaction*, where the geometry of nets is unchanged by reduction. In other words, the right member of each interaction rule has the same form as its left member: only symbols and positions of the two principal ports may change. It is of course a very strong requirement, and indeed, none of the above rules is of this type. The physical implementation of this kind of interaction seems more tractable, but it is a very low-level language: all the difficulty is to know how a program such as those considered above can be compiled into this language.

The french version of this paper, entitled “Les réseaux d'interaction”, will appear in *Images des Mathématiques, CNRS.*

Turing machines

A Turing machine is a kind of stylized calculator. There is no attempt to describe the mechanism of the processor; one only knows that it takes a finite number of states. The memory, on the other hand, must be potentially infinite: it is an unlimited ribbon with cells containing letters of a finite alphabet. At a given time, the read-write head gives access to a single cell.

The behavior of the machines is completely determined by its *transition table*, which gives for each states q and each letter a the instruction that must be executed when a is read in state q. Except for halting, all instructions are of the following form: write a' (instead of a), move the read-write head to the left (or to the right), and go to state q'.

Such a table can be considered as a program written in a very low-level language. Although it is a bit tedious, it is possible to build the table of a *universal* Turing machine, which can simulate any other Turing machine, in the same way that a computer can simulate any calculator.

Lambda-calculus

Lambda-calculus is a notation system for functions. If U is a term, $\lambda x.U$ stands for the function which maps x to U (in general, U contains one or several occurrences of the variable x). If U and V are terms, $U\,V$ stands for the application of the function U to the argument V. It is convenient to write $U\,V\,W$ for $(U\,V)\,W$: it allows to see U as a function with 2 arguments.

A single rule of computation suffices: the term $(\lambda x.U)\,V$ reduces to $U[V/x]$, that is U where all occurrences of the variable x have been replaced by the argument V. In fact, the process or replacement is slightly more subtile: one may only replace the occurrences of x which are *free* in U. In its *untyped* version, lambda-calculus is very powerful, since it allows to apply any function to any argument. Moreover, computation may not terminate.

There is no specific notation for primitive arguments, but the natural numbers can be represented by the functions $\underline{0} = \lambda f.\lambda x.x$, $\underline{1} = \lambda f.\lambda x.f\,x$, $\underline{2} = \lambda f.\lambda x.f\,(f\,x)$, etc. Addition and multiplication are respectively represented by the terms $A = \lambda p.\lambda q.\lambda f.\lambda x.p\,f\,(q\,f\,x)$ and $M = \lambda p.\lambda q.\lambda f.p\,(q\,f)$. For instance, $A\,\underline{2}\,\underline{3}$ reduces to $\underline{5}$, and $M\,\underline{2}\,\underline{3}$ reduces to $\underline{6}$. Church Thesis says that any *computable* function $\varphi : \mathbf{N}^k \to \mathbf{N}$ is represented by a term of lambda-calculus.

Exercise: what operation is represented by the term $\lambda p.\lambda q.p\,q$?

For more information ...

- on *linear logic*: J.-Y. Girard, "Linear logic: its syntax and semantics", in "Advances in Linear Logic", London Mathematical Society Lecture Note Series 222, pp 1–42, Cambridge University Press, 1995 ;
- on *proof-nets*: Y. Lafont, "From Proof-Nets to Interaction Nets", idem, pp 225–247 ;
- on *geometry of interaction*: V. Danos & L. Regnier, "Proof-Nets and the Hilbert space", idem, pp 307–328 ;
- on *interaction nets* and *interaction combinators*: Y. Lafont, "Interaction Combinators", Information and Computation 137 (1), pp 69–101, Academic Press, 1997.