Interaction Nets

Yves Lafont
CNRS
Laboratoire d'Informatique de l'Ecole Normale Supérieure *

Abstract

We propose a new kind of programming language, with the following features:

- a simple graph rewriting semantics,
- a complete symmetry between constructors and destructors,
- a type discipline for deterministic and deadlock-free (microscopic) parallelism.

Interaction nets generalise Girard's *proof nets* of linear logic and illustrate the advantage of an *integrated logic* approach, as opposed to the *external* one. In other words, we did not try to design a logic describing the behaviour of some given computational system, but a programming language for which the type discipline is already (almost) a logic.

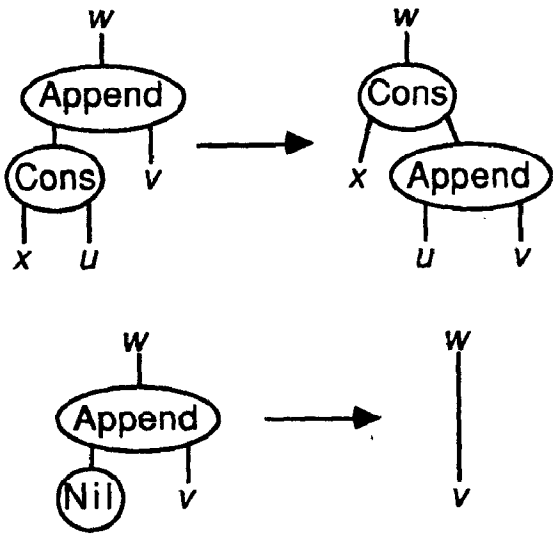
In fact, we shall scarcely refer to logic, because we adopt a naïve and pragmatic style. A typical application we have in mind for this language is the design of interactive softwares such as editors or window managers.

1 Principles of Interaction

Throughout this text, *net* means *undirected graph with labelled vertices*, also called *agents*. For each label, also called *symbol*, a finite set of *ports* has been fixed:

*Address: 45 rue d'Ulm, 75230 Paris Cedex 05, France.
Electronic mail: lafont@dmi.ens.fr

We shall consider rewrite rules :



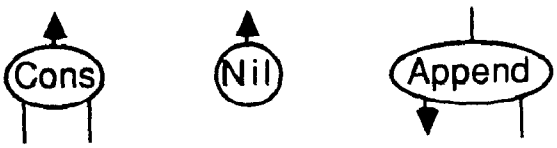
Here, rewriting is just a convenient language to express a very concrete notion of *interaction*, which we shall make precise by requiring some properties of rules. The first one is in fact imposed by our option of nets (as opposed to trees or directed graphs):

1. (linearity)

   Inside a rule, each variable occurs exactly twice, once in the left member and once in the right one.

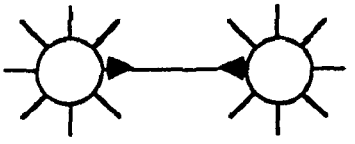
Consequently, explicit duplication and erasing symbols are required for algorithms such as *unary multiplication* (figure 1).

To express our second constraint, we must first distinguish a *principal port* for each symbol:

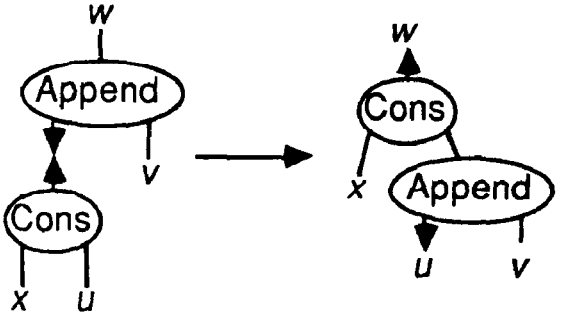


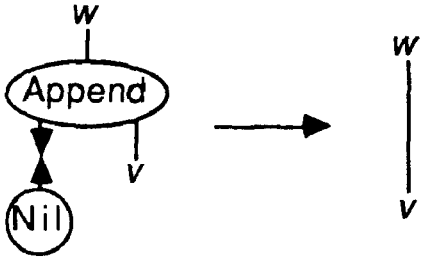
2. (binary interaction)

   Agents interact through their principal port only, which means that left members of rules are restricted to the following form:

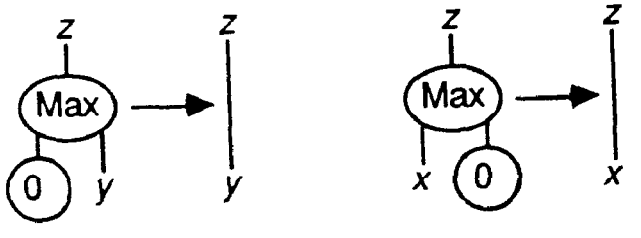


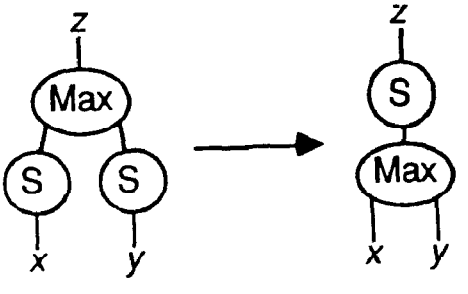
Indeed the rules for *append* satisfy our constraint:





But for example, *unary maximum* cannot be expressed as follows:





An extra symbol is needed (figure 2). In other words, we have to choose which argument is looked first (local sequentiality), and algorithms such as *parallel or* are excluded.

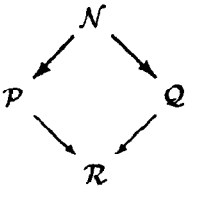
A pair of agents which are connected by their principal port is called *alive*, because some rule — maybe several, maybe none — is supposed to reduce it. Clearly, a third constraint is necessary to ensure deterministic computation:

3. (no ambiguity)

   There is *at most one* rule for each pair of distinct symbols S, T, and no rule for S, S.

The three conditions are enough to get the following (easy) property:

Proposition 1 *(strong confluence)*
If $\mathcal{N}$ reduces in one step to $\mathcal{P}$ and $\mathcal{Q}$, with $\mathcal{P} \neq \mathcal{Q}$, then $\mathcal{P}$ and $\mathcal{Q}$ reduce in one step to a common $\mathcal{R}$.



Indeed, by conditions 2 and 3, rules apply to disjoint pairs of agents, and cannot interfere with each other. Usual complications are avoided by condition 1. In fact, interactions are purely *local* and can be performed concurrently: proposition 1 expresses that the relative order of concurrent reductions is completely irrelevant.

So far, nothing ensures that all alive pairs of agents are reducible, but it is a reasonable requirement, and indeed, it will be the case of *typed* nets. Consequently, if the right member of a rule contains some alive pair, we should be able to reduce it, and the following condition is natural:

4. (optimisation)

   Right members of rules contain no alive pair.

Even with this extra condition, termination is not ensured. The simplest counterexample is the *turnstile* (figure 3).

To illustrate the flexibility of nets for programming, we exhibit two simple examples: *concatenation of difference-lists* and *polish parsing.*

Concatenation of lists is performed in *linear time* with respect to its first argument. *Constant time* concatenation is possible with *difference-lists*: the idea consists in plugging the front of the second argument at the end of the second one. This requires two steps (figure 4) with an extra symbol, as in the case of *unary maximum.*

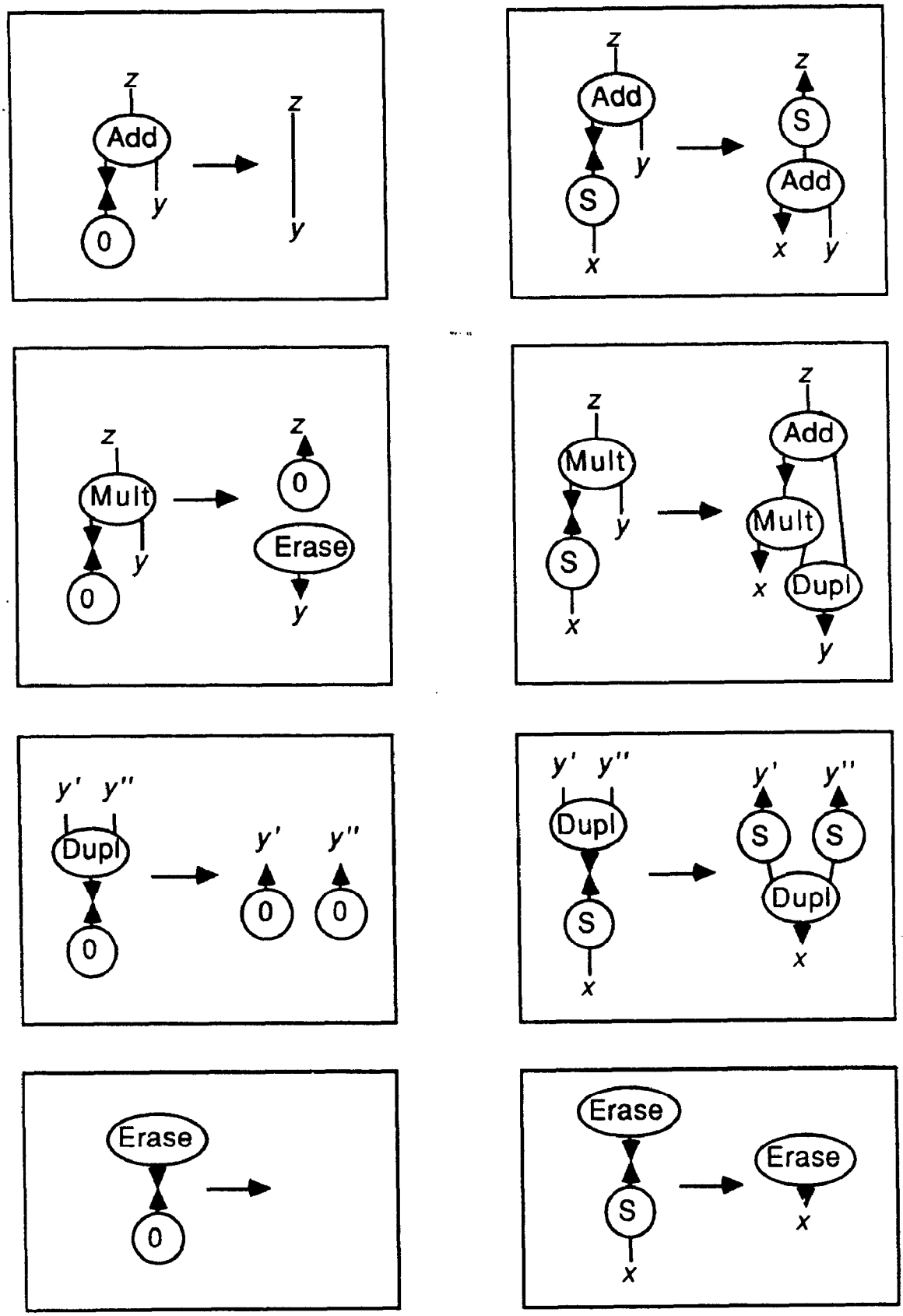


Figure 1: explicit duplication and erasing for unary multiplication

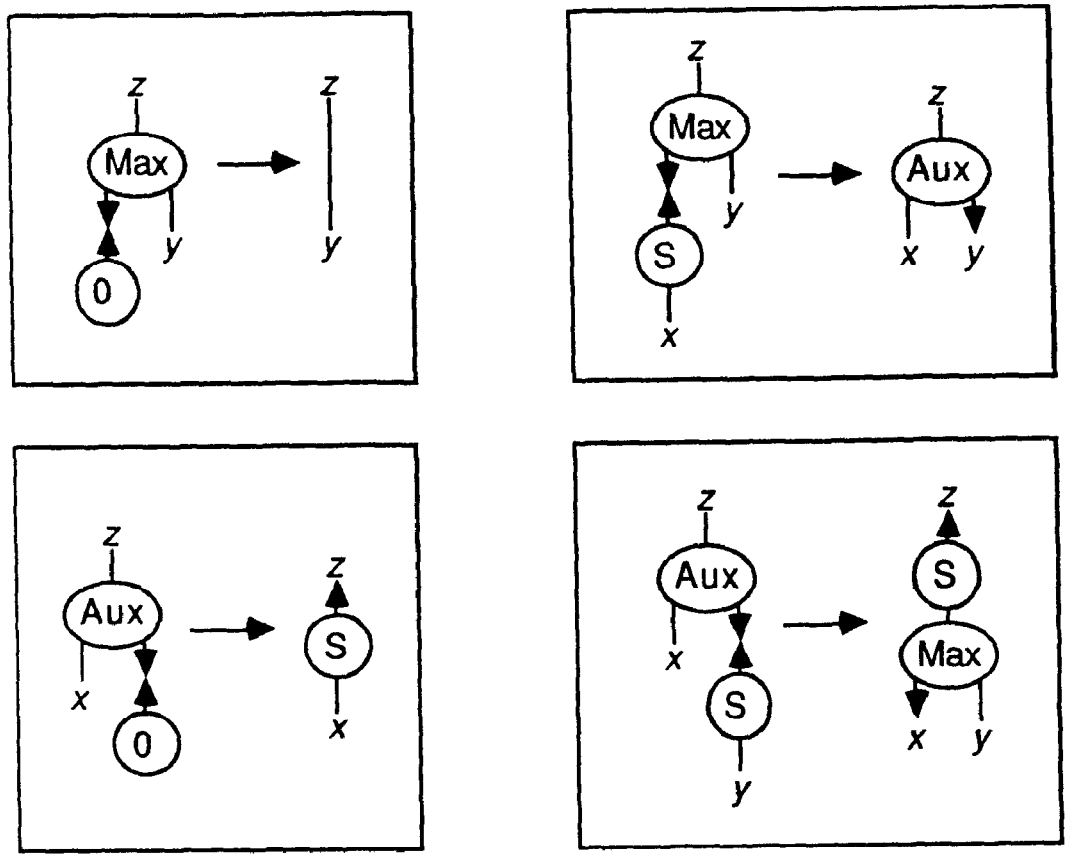


Figure 2: extra symbol for unary maximum

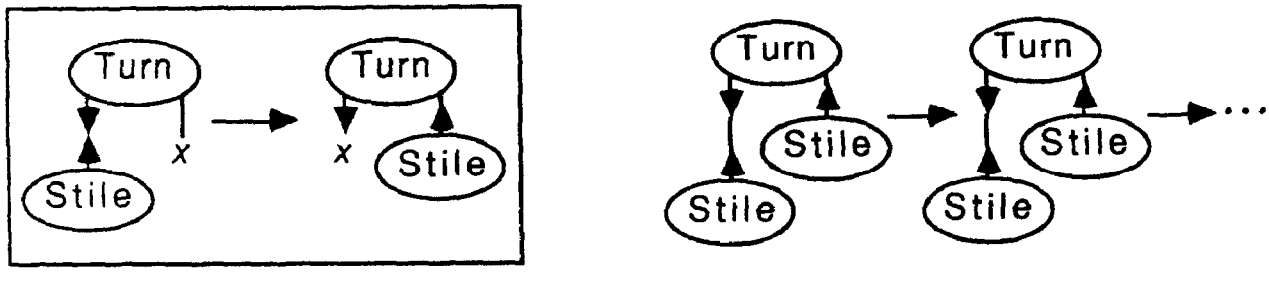


Figure 3: infinite computation with turnstile

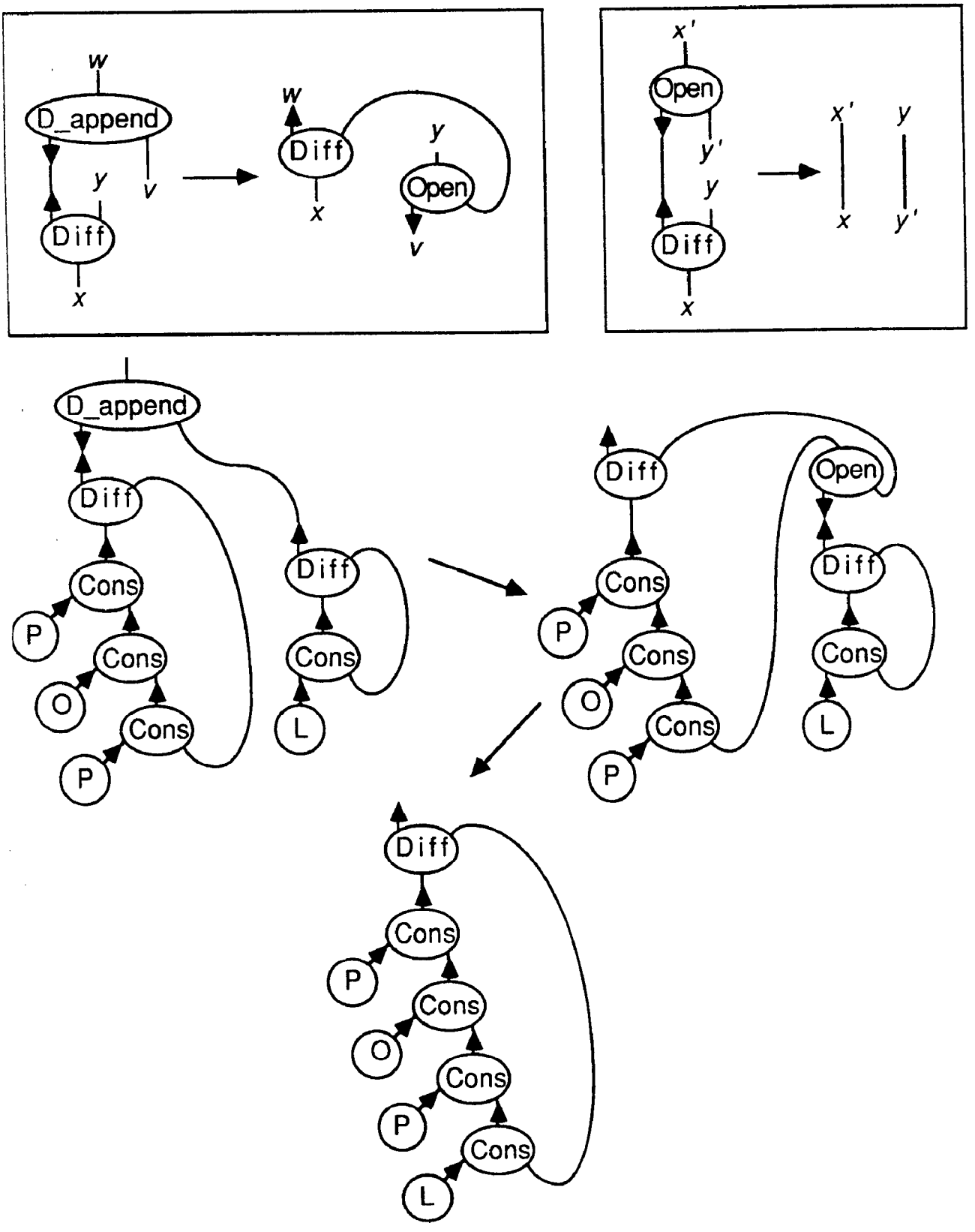


Figure 4: concatenation of difference-lists

Our second example is a parser for very simple arithmetic expressions in polish notation:

$$\begin{array}{rl} \textit{expression}: & + \textit{ expression expression} \\ | & 1 \end{array}$$

The parser takes an infinite stream of symbols as input, and returns a tree as well as a new stream (the rest) as outputs (figure 5).

Finally, note that the expressive power of interaction nets is not limited: *Turing machines* or *SK-reduction*, for example, can be easily simulated within interaction nets.

2 A Type Discipline

We are going to strengthen the conditions of section 1 so that for each alive pair of agents, some rule applies. Introducing rules for all pairs of symbols is not conceivable: how the devil would Cons interact with Nil, or Parse with Append? Moreover this would be inconsistent with condition 3. So we are led to limit valid configurations by means of *typing*.

We introduce *constant types* atom, list, nat, d_list, stream, tree, ... For each symbol, ports must be typed as input (τ^-) or output (τ^+):

A net is *well typed* if inputs are connected to outputs of the same type. A rule is well typed if:

- symbols in the left member *match*, which means that their principal ports have opposite types,
- the right member is well typed (the types of variables being given by the left member).

So we have new conditions for typed interaction:

5. (typing)

   Rules are well typed.

6. (completeness)

   There is a rule for each pair of matching symbols.

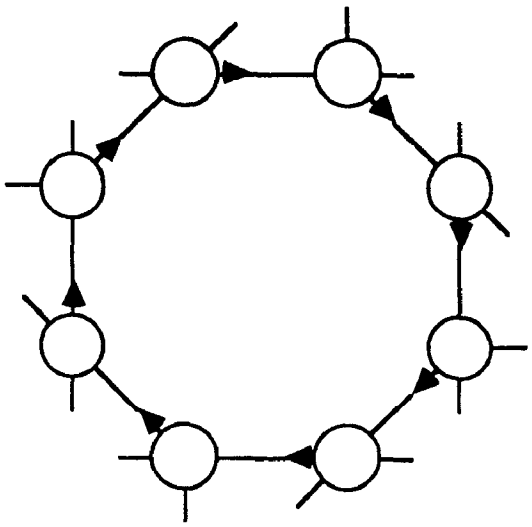
All examples in section 1 are easily typed. The choice of an input/output denomination is purely conventional: it does not matter if you call input what I call output, and conversely, but we must agree on matching. In other words, the notions of *constructor* (symbol with a positively typed principal port, like Cons) and *destructor* (symbol with a negatively typed principal port, like Append) are symmetrical in our system.

So far, typing ensures *local correctness* of computations, but we shall see that a notion of *global correctness* is necessary to prevent *deadlock*.

Proposition 2 *(stopping cases)*
Let $\mathcal{N}$ be well typed, finite, nonempty, with free variables $x_1, \ldots, x_n$. If $\mathcal{N}$ is irreducible then one of the following conditions holds:

i) some x_i is connected to a principal port, or to another variable,

ii) $\mathcal{N}$ contains a vicious circle:



Indeed, starting from any point, you can follow principal ports until you reach a variable, or you loop! Case (i) simply means that $\mathcal{N}$ is ready to interact with its environment, but case (ii) is pathological. In fact, by condition 2, we have clearly:

Proposition 3 *(deadlock)*
A vicious circle stays forever.

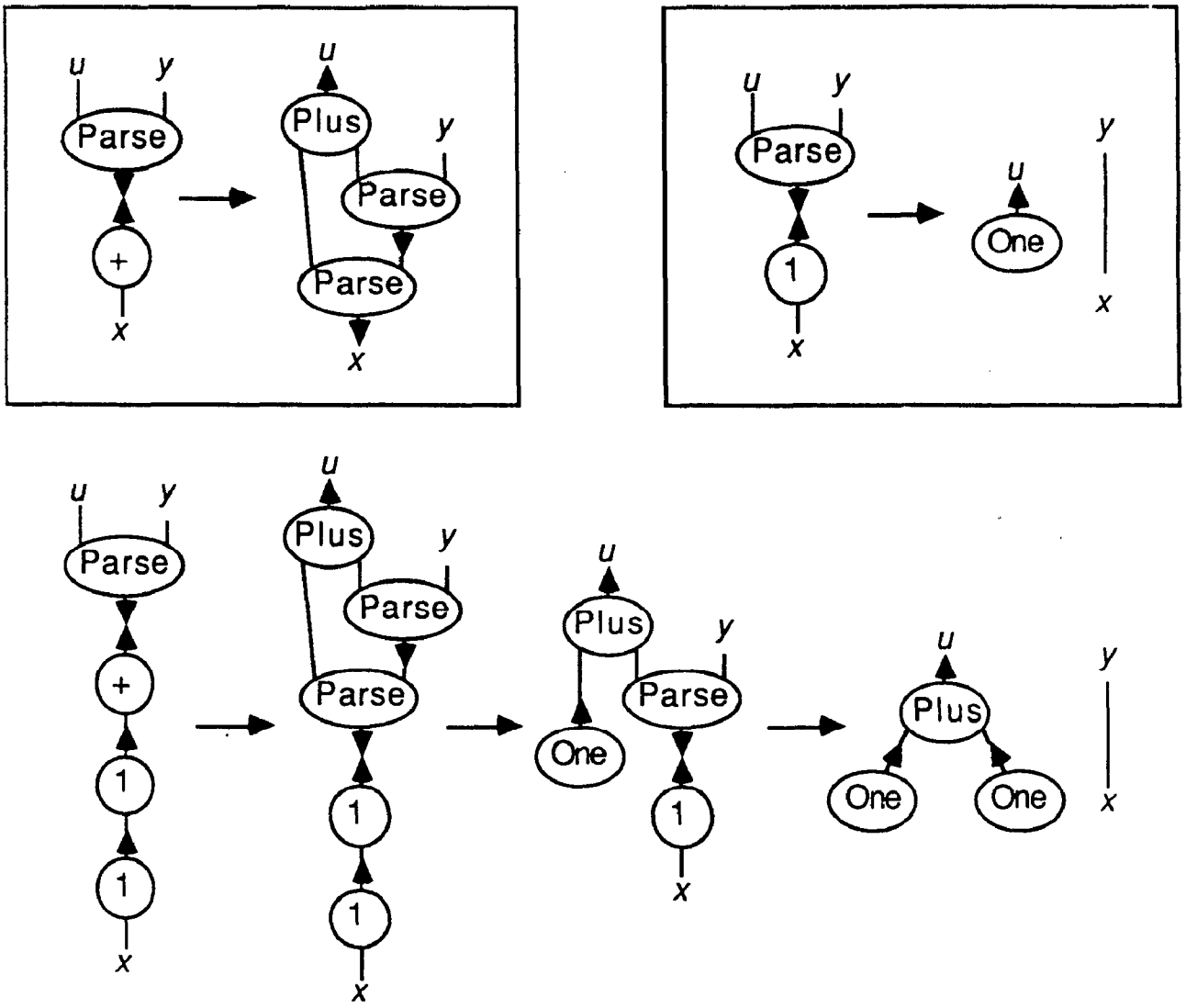
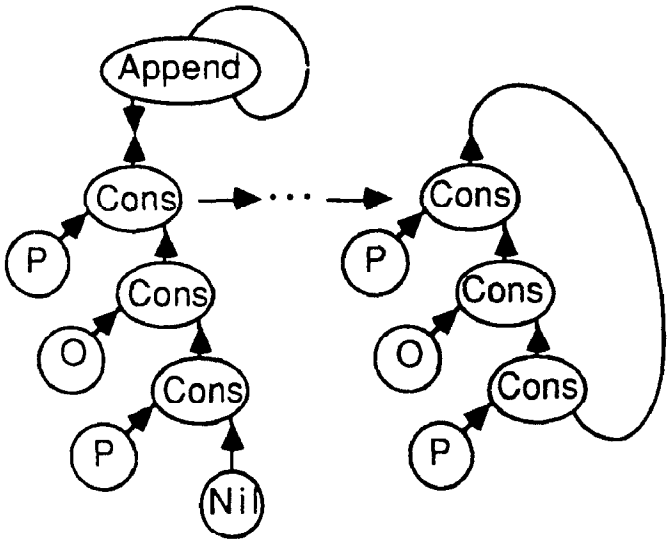
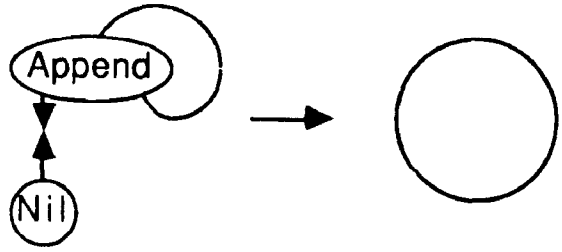


Figure 5: polish parsing

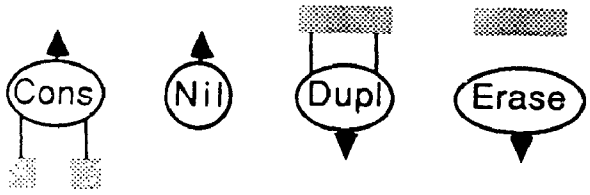
The trouble with vicious circles is that they can appear unexpectedly during a computation:



By the way, we should also consider the degenerated case:

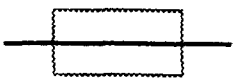


We need some extra condition to prohibit creation of such configurations. Forbidding cycles is unthinkable, because our examples (figures 8, 4 and 11) would collapse, but it is possible to distinguish between good and bad cycles. For that purpose, we assume that a *partition* on auxiliary ports is given for each symbol:

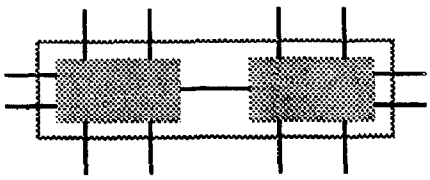


A net is called *simple* if it can be obtained by using only the following operations:

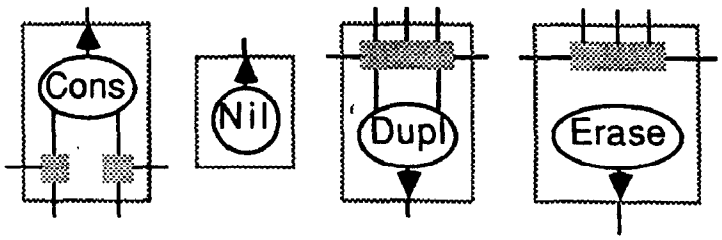
- LINK (an edge):



- CUT (a single connection between two nets):

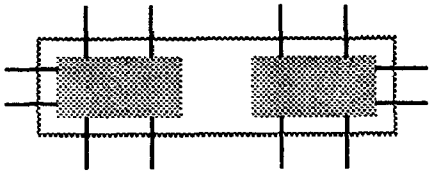


- GRAFT (connecting a new agent with nets, according to its partition):



We also introduce a larger class of *semi-simple* nets by allowing two extra operations:

- EMPTY (an empty net)
- MIX (juxtaposing two nets):



To get some intuition, consider the special case of symbols with *discrete* partitions (all classes are singletons):

Proposition 4 *(topological interpretation)*
In the discrete case, a net is simple when it is a connected graph without cycle, and it is semi-simple when it has no cycle.

In the general case, a straightforward induction gives:

Proposition 5 *(static correctness)*
A semi-simple net (and so, a simple one) contains no vicious circle.

Of course, the converse is false: we are not just worried about actual deadlocks but about potential ones.

A rule is *simple* if, when variables have been grouped according to partitions in the left members, the right member becomes simple (figure 7). With a suitable choice of partitions for symbols (figure 6) all rules in this paper are actually simple[1].

[1] As far as deadlock is concerned, *semi-simple* rules could be allowed, but the author has no interesting example. In fact, it is still not clear which concept, *simplicity* or *semi-simplicity*, should be used

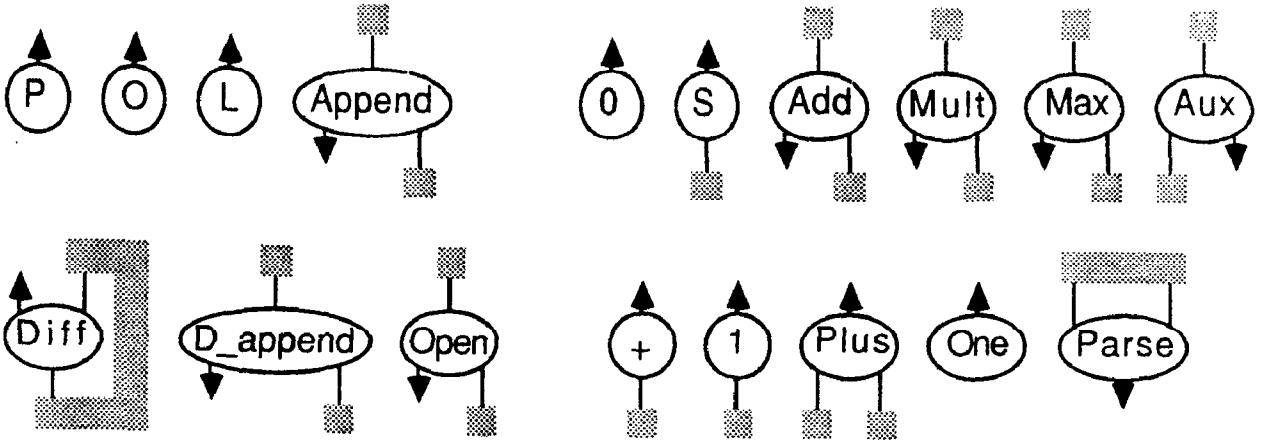


Figure 6: suitable partitions

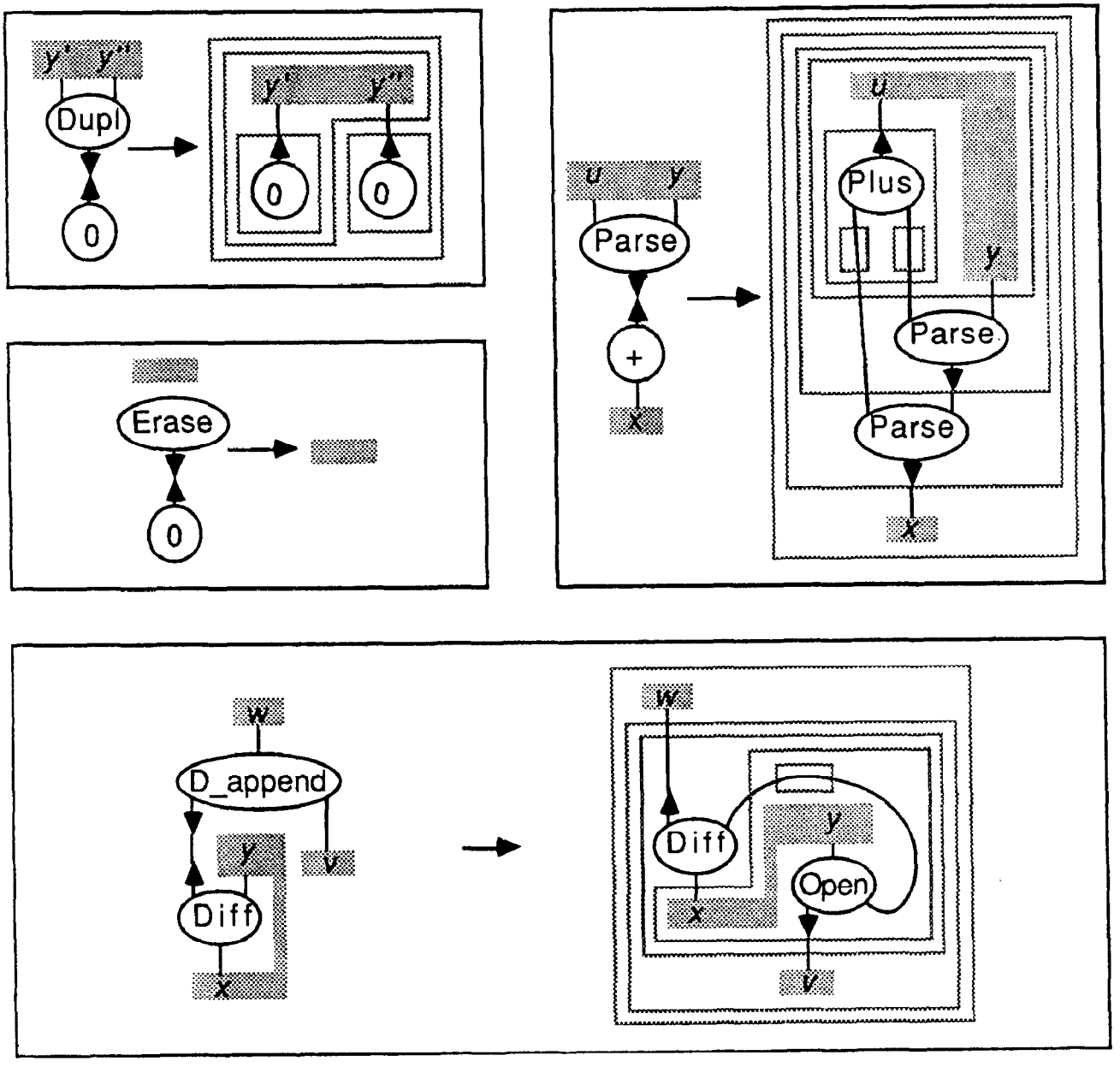


Figure 7: simple rules

The following result is essentially an adaptation of the *cut elimination theorem* for multiplicative linear logic (see appendix):

Proposition 6 *(invariance)*
Simple nets are closed under reduction by simple rules.

So our last constraint is:

7. (simplicity)
   Rules are simple.

With this condition, proposition 5 and 6 have the following consequence:

Proposition 7 *(dynamic correctness)*
A simple net will never deadlock.

3 A Programming Language

In this section, we describe a *concrete syntax* for interaction programming. A program contains three parts:

- *type* declaration (a list of identifiers),
- *symbol* declaration (identifiers with typing and partitions),
- interaction rules (the core of the program).

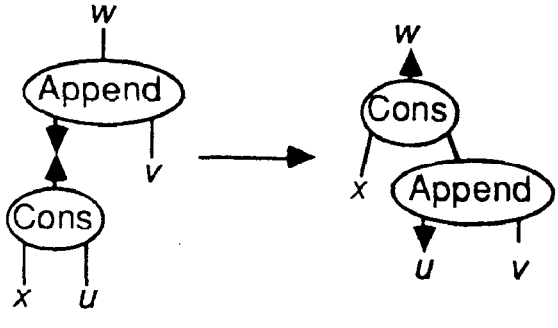
For each symbol, the type of its principal port is given first:

symbol Cons : $\mathsf{list}^+; \mathsf{atom}^-, \mathsf{list}^-$
Nil : list^+
Append : $\mathsf{list}^-; \mathsf{list}^-, \mathsf{list}^+$

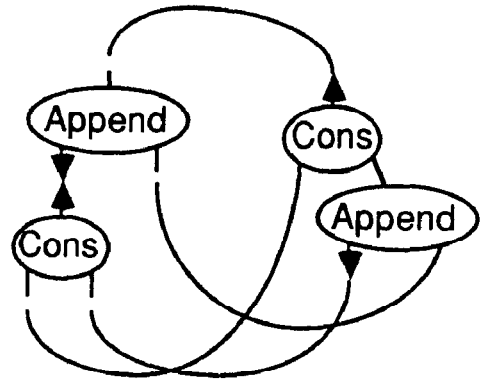
Non-discrete partitions are specified by means of curly brackets.

symbol Dupl : $\mathsf{nat}^-; \{\mathsf{nat}^+, \mathsf{nat}^+\}$
Erase : $\mathsf{nat}^-; \{\}$

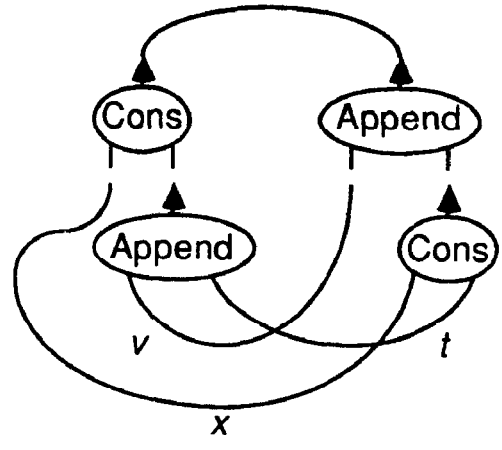
Notation for rules is a bit disconcerting, but very natural. Consider the following example:



We join variables between left and right members:



Putting principal ports up and auxiliary ones down, we obtain two trees with links between leafs:



So interaction is written as follows:

$$\mathsf{Cons}\,[x, \mathsf{Append}(v,t)] \quad)\langle \quad \mathsf{Append}\,[v, \mathsf{Cons}(x,t)]$$

Note that the left and right sides of)⟨ have nothing to do with the left and right members of the initial rule. It requires a bit of training to become acquainted with this syntax (figures 8, 9, 10 and 11).

This language can be implemented efficiently on a sequential machine. Basically, nets are encoded by means of cells and pointers, and alive pairs are stored in a stack. Because of the linearity, there is no need for a *garbage collector* (as in [Lafont88a]).

Here we described only the kernel of our language, but it should be interfaced with external devices such as keyboards or displays: for example, the output of a keyboard can be seen as an infinite stream of agents with characters as symbols (as in figure 5). Furthermore some extensions such as *polymorphic typing* and *modularity* are certainly needed to get a high level programming language.

type nat

symbol $0 : \mathsf{nat}^+$
$\mathsf{S} : \mathsf{nat}^+; \mathsf{nat}^-$
$\mathsf{Add} : \mathsf{nat}^-; \mathsf{nat}^-, \mathsf{nat}^+$
$\mathsf{Mult} : \mathsf{nat}^-; \mathsf{nat}^-, \mathsf{nat}^+$
$\mathsf{Dupl} : \mathsf{nat}^-; \{\mathsf{nat}^+, \mathsf{nat}^+\}$
$\mathsf{Erase} : \mathsf{nat}^-; \{\}$
$\mathsf{Max} : \mathsf{nat}^-; \mathsf{nat}^-, \mathsf{nat}^+$
$\mathsf{Aux} : \mathsf{nat}^-; \mathsf{nat}^-, \mathsf{nat}^+$

$$
\begin{array}{rcl}
0 & \rangle\langle & \mathsf{Add}\,[y, y] \\
\mathsf{S}\,[\mathsf{Add}(y, t)] & \rangle\langle & \mathsf{Add}\,[y, \mathsf{S}(t)]
\end{array}
$$

$$
\begin{array}{rcl}
0 & \rangle\langle & \mathsf{Mult}\,[\mathsf{Erase}, 0] \\
\mathsf{S}\,[\mathsf{Mult}(y', \mathsf{Add}(y'', z))] & \rangle\langle & \mathsf{Mult}\,[\mathsf{Dupl}(y', y''), z]
\end{array}
$$

$$
\begin{array}{rcl}
0 & \rangle\langle & \mathsf{Dupl}\,[0, 0] \\
\mathsf{S}\,[\mathsf{Dupl}(x', x'')] & \rangle\langle & \mathsf{Dupl}\,[\mathsf{S}(x'), \mathsf{S}(x'')]
\end{array}
$$

$$
\begin{array}{rcl}
0 & \rangle\langle & \mathsf{Erase} \\
\mathsf{S}\,[\mathsf{Erase}] & \rangle\langle & \mathsf{Erase}
\end{array}
$$

$$
\begin{array}{rcl}
0 & \rangle\langle & \mathsf{Max}\,[y, y] \\
\mathsf{S}\,[x] & \rangle\langle & \mathsf{Max}\,[\mathsf{Aux}(x, z), z]
\end{array}
$$

$$
\begin{array}{rcl}
0 & \rangle\langle & \mathsf{Aux}\,[x, \mathsf{S}(x)] \\
\mathsf{S}\,[y] & \rangle\langle & \mathsf{Aux}\,[\mathsf{Max}(y, t), \mathsf{S}(t)]
\end{array}
$$

Figure 8: unary arithmetics

type atom, list

symbol P : atom$^+$
O : atom$^+$
L : atom$^+$
Cons : list$^+$; atom$^-$, list$^-$
Nil : list$^+$
Append : list$^-$; list$^-$, list$^+$

Cons $[x, \text{Append}(v,t)]$ $\rangle\langle$ Append $[v, \text{Cons}(x,t)]$
Nil $\rangle\langle$ Append $[v, v]$

Figure 9: lists

type list, d_list

symbol Diff : d_list$^+$; {list$^-$, list$^+$}
D_append : d_list$^-$; d_list$^-$, d_list$^+$
Open : d_list$^-$; list$^-$, list$^+$

Diff $[x, y]$ $\rangle\langle$ D_append $[\text{Open}(t, y), \text{Diff}(x, t)]$
Diff $[x, y]$ $\rangle\langle$ Open $[y, x]$

Figure 10: difference-lists

type stream, tree

symbol + : stream$^+$; stream$^-$
1 : stream$^+$; stream$^-$
Plus : tree$^+$; tree$^-$, tree$^-$
One : tree$^+$
Parse : stream$^-$; {tree$^+$, stream$^+$}

+ $[\text{Parse}(v, \text{Parse}(w, y))]$ $\rangle\langle$ Parse $[\text{Plus}(v, w), y]$
1 $[x]$ $\rangle\langle$ Parse $[\text{One}, x]$

Figure 11: parsing

Conclusion

Our proposal can be compared with existing programming paradigms. As in *functional programming*, we have a strong type discipline and a deterministic semantics based on a Church-Rosser property, but the functional paradigm (like intuitionistic logic) assumes an essential asymmetry between inputs and outputs, which is incompatible with parallelism and unconvenient for writing interactive softwares.

Our rules are clearly reminiscent of clauses in *logic programming*, especially in the use of variables (see the example of difference-lists), and our proposal could be related to PARLOG or GHC. There are also some similarities with *data-flow languages* and the CCS-CSP family, but as far as we know, the concepts of *principal port* (which is critical for determinism) and *semi-simplicity* (which prevents deadlock) has never been considered in such systems.

In the appendix, we explain how this work relates to linear logic. Our first contribution was much more in the lineage of functional programming, with an emphasis on questions of lazyness and memory allocation [Girafont,Lafont88a]. On the other hand, [Lafont87] can be considered as an embryo of interaction nets, although the right framework was not discovered at that time. The first idea of generalising multiplicative connectors of linear logic appears in [Girard88] (partitions are considered in [Regnos]) and led to the *Geometry of interaction* [Girard89,Girard89a].

We are now working on a true implementation of the language to develop real examples in a practical programming environment.

Appendix: Linear Logic

Let us write $\vdash A_1, \ldots, A_n$ if there is a *simple* net with free variables $x_1, \ldots, x_n$ of types $A_1, \ldots, A_n$. By definition of *simplicity*, we have the following rules:

$$\frac{\vdash \Gamma, A, B, \Delta}{\vdash \Gamma, B, A, \Delta}\ \text{EXCHANGE}$$

$$\frac{}{\vdash A, A^{op}}\ \text{LINK} \qquad \frac{\vdash A, \Gamma \quad \vdash A^{op}, \Delta}{\vdash \Gamma, \Delta}\ \text{CUT}$$

The exchange rule expresses that the order of variables is irrelevant (simple nets are not necessarily planar graphs). So far, those are rules of *linear logic* [Girard87] (see also in [Giraflor] for a short introduction) but here we have plenty of *logical rules* corresponding to the symbol declaration part of our programs, for example:

$$\frac{\vdash \mathsf{atom}^+, \Gamma \quad \vdash \mathsf{list}^+, \Delta}{\vdash \mathsf{list}^+, \Gamma, \Delta}\ \text{Cons} \qquad \frac{}{\vdash \mathsf{list}^+}\ \text{Nil}$$

$$\frac{\vdash \mathsf{list}^+, \Gamma \quad \vdash \mathsf{list}^-, \Delta}{\vdash \mathsf{list}^-, \Gamma, \Delta}\ \text{Append}$$

$$\frac{\vdash \mathsf{nat}^-, \mathsf{nat}^-, \Gamma}{\vdash \mathsf{nat}^-, \Gamma}\ \text{Dupl} \qquad \frac{\vdash \Gamma}{\vdash \mathsf{nat}^-, \Gamma}\ \text{Erase}$$

Basically, interaction consists in *cut elimination*:

$$\frac{\dfrac{}{\vdash \mathsf{list}^+}\ \text{Nil} \qquad \dfrac{\vdash \mathsf{list}^+, \Gamma \quad \vdash \mathsf{list}^-, \Delta}{\vdash \mathsf{list}^-, \Gamma, \Delta}\ \text{Append}}{\vdash \Gamma, \Delta}\ \text{CUT}$$

$$\downarrow$$

$$\frac{\vdash \mathsf{list}^+, \Gamma \quad \vdash \mathsf{list}^-, \Delta}{\vdash \Gamma, \Delta}\ \text{CUT}$$

The concrete syntax $\mathsf{Nil}\ )\!(\ \mathsf{Append}\ [v, v]$ is of course more concise!

By adding *polymorphic typing*, we integrate "official" rules of linear logic such as:

$$\frac{\vdash A, \Gamma \quad \vdash B, \Delta}{\vdash A \otimes B, \Gamma, \Delta}\ \text{Times}$$

but not the following one:

$$\frac{\vdash A, \Gamma \quad \vdash B, \Gamma}{\vdash A \,\&\, B, \Gamma}\ \text{With}$$

Indeed, our proposal generalises the so-called *multiplicative* fragment of linear logic, for which the notion *proof net* works very well, but with very limited dynamics (everything reduces in linear time). On the contrary, our type system does not ensure termination, although it would be interesting to isolate terminating subsystems.

References

[Girard87] J.Y. Girard, Linear logic, *TCS* **50** (1987) 1-102.

[Girard88] J.Y. Girard, Multiplicatives, in *Rendiconti del seminario matematico dell'università e politecnico di Torino*, special issue on logic and computer science (1988).

[Girard89] J.Y. Girard, Towards a geometry of interaction, in *Conference on categories, computer science and logic*, Contempory Mathematics, *AMS* **92** (1989).

[Girard89a] J.Y. Girard, Geometry of interaction 1: interpretation of system F, in *ASL meeting* (North-Holland, Padova, 1989).

[Girafont] J.Y. Girard & Y. Lafont, Linear Logic and Lazy Computation, in *TAPSOFT '87, vol. 2, LNCS* **250** (Springer-Verlag, Pisa) 52-66.

[Giraflor] J.Y. Girard, Y. Lafont, P. Taylor, *Proofs and Types*, Cambridge Tracts in Theoretical Computer Science (Cambridge University Press, 1989).

[Lafont87] Y. Lafont, Linear Logic Programming, in *Workshop on Programming Logic* (Göteborg, 1987) 209-220.

[Lafont88] Y. Lafont, Logiques, Catégories et Machines, thèse de doctorat (Université de Paris VII, 1988).

[Lafont88a] Y. Lafont, The Linear Abstract Machine, *TCS* **59** (1988) 157-180.

[Regnos] V. Danos & L. Régnier, The structure of multiplicatives, typescript (1988).