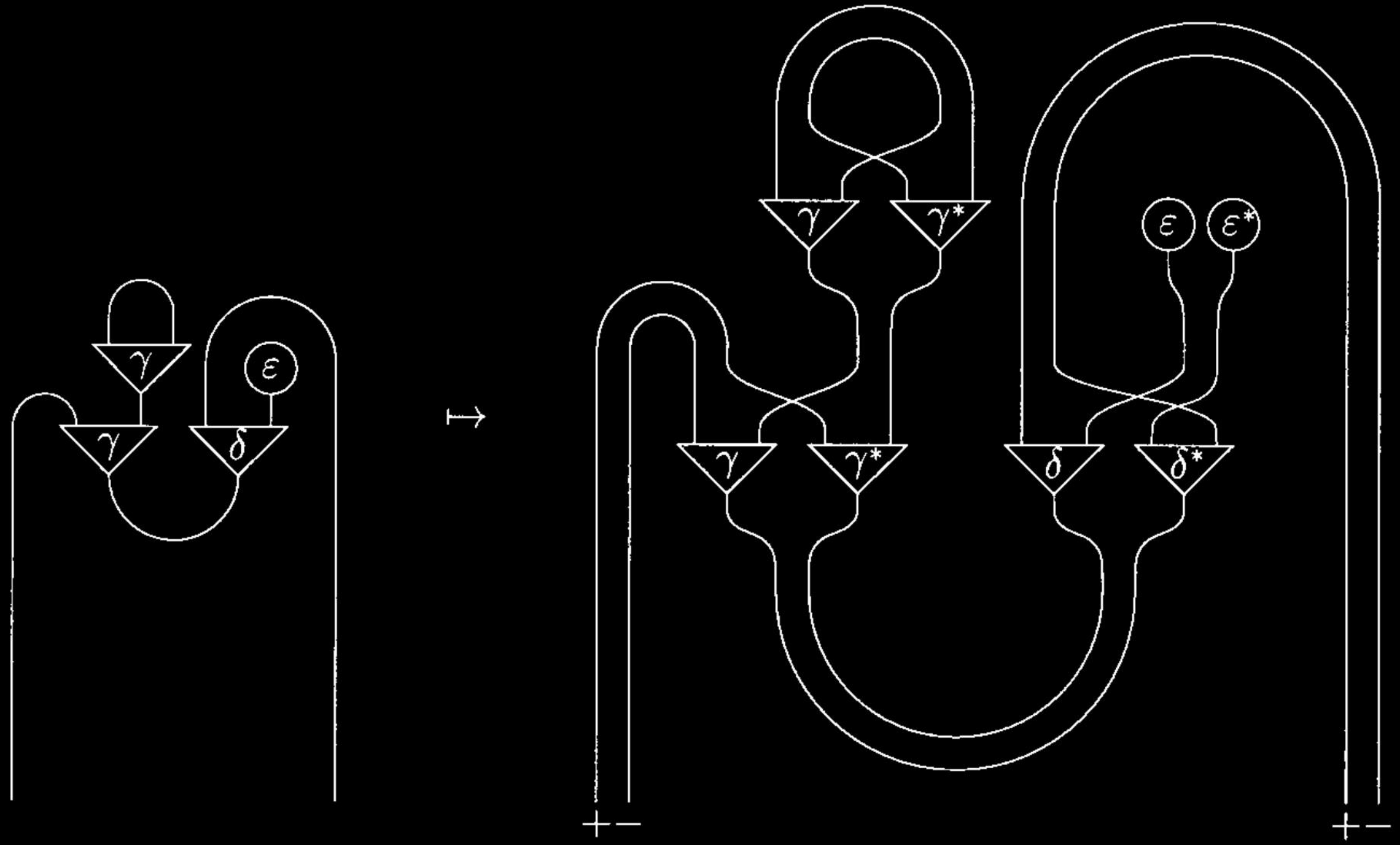


γ
γ
ε
δ
↦
γ
γ*
ε
ε*
γ
γ*
δ
δ*
+ −
+ −

Interaction Nets

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

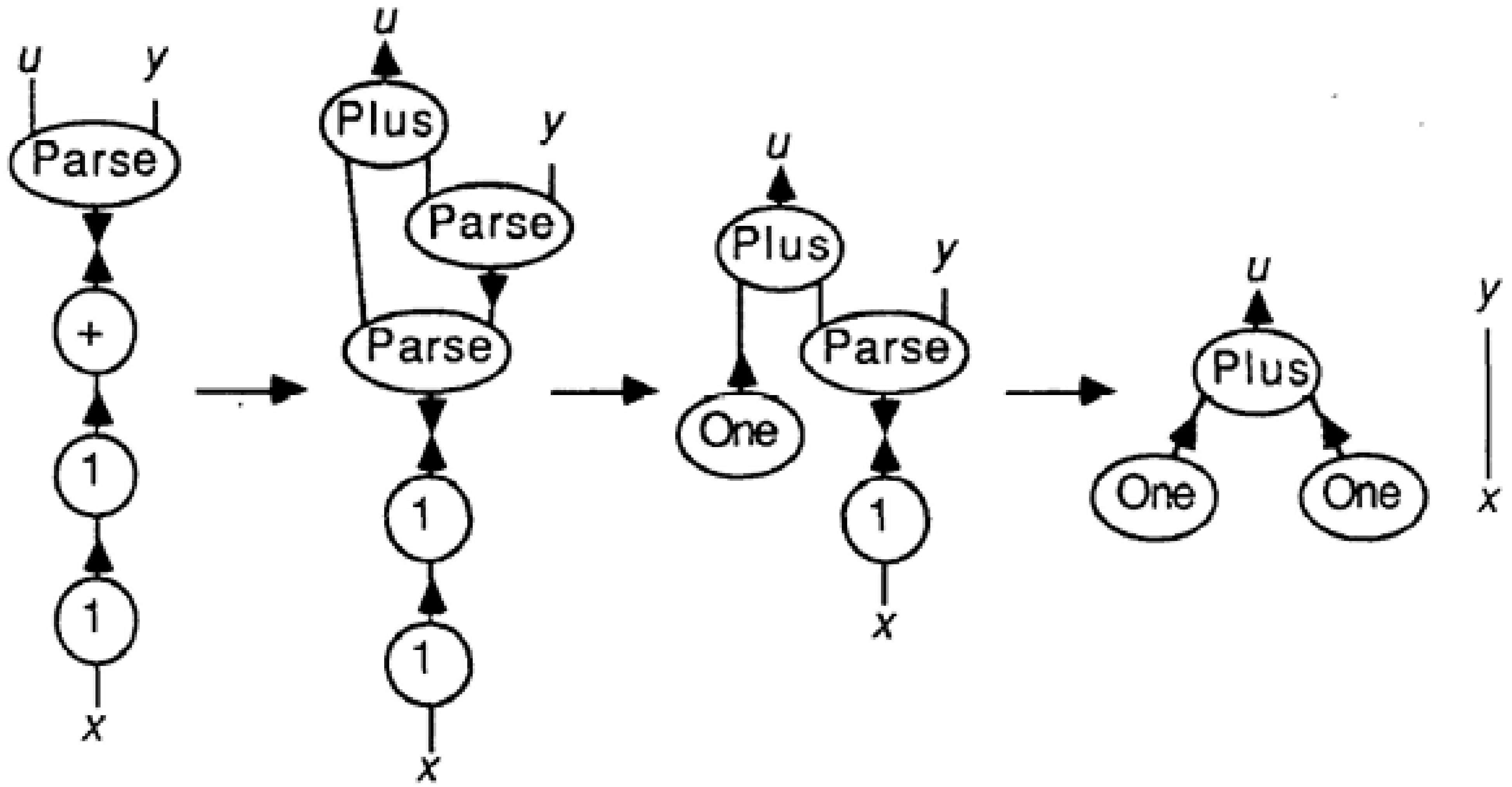


Figure 5: polish parsing

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.

Rewriting languages: intuition is
find and replace

Symmetry means no caller/callee relationship between functions/operators, no privileged direction.

Inherently parallel computation model.

Linear logic: you can't use a proposition multiple times (without specific operators).
Propositions are consumable resources.

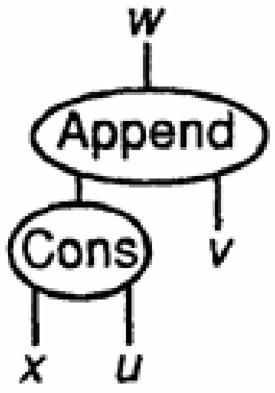
Principles of Interaction

Definitions

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:

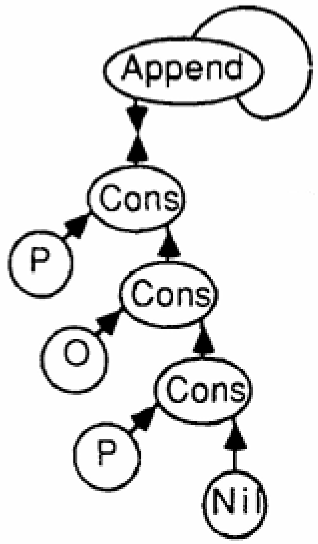
Agents**, labelled with symbols, have **ports

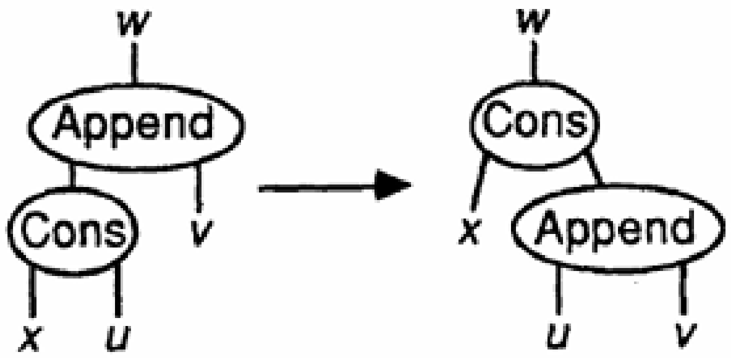


***Variables** can stand in for parts of a net*

*Agents can be connected into **nets***

Undirected!



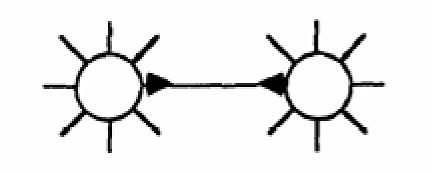


Rewriting rules** express **interaction

Rule properties

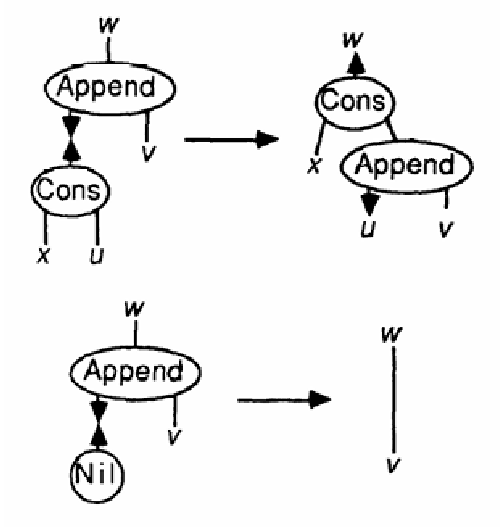
Binary interaction

Agents have a single **principal port**. Interactions only occur through these.

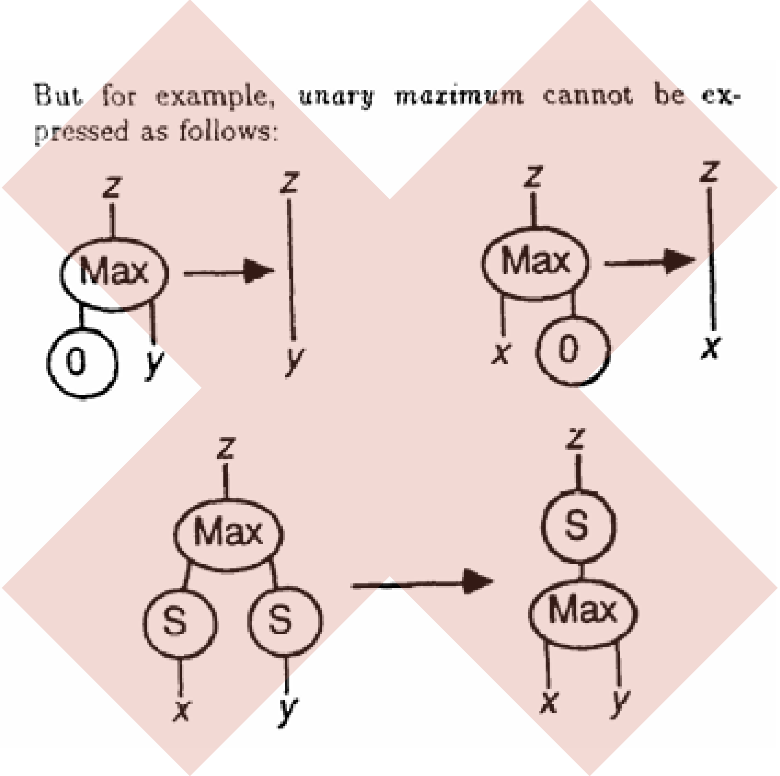


Left side of a rule is always of this form.

The **cons** and **append** rewrite rules look like this.



But… this means local sequentiality.
We have to pick one argument to look at first.



__S__ here means __Successor__
0 → S is 1
0 → S → S → S → S is 4
All integers are expressed this way.

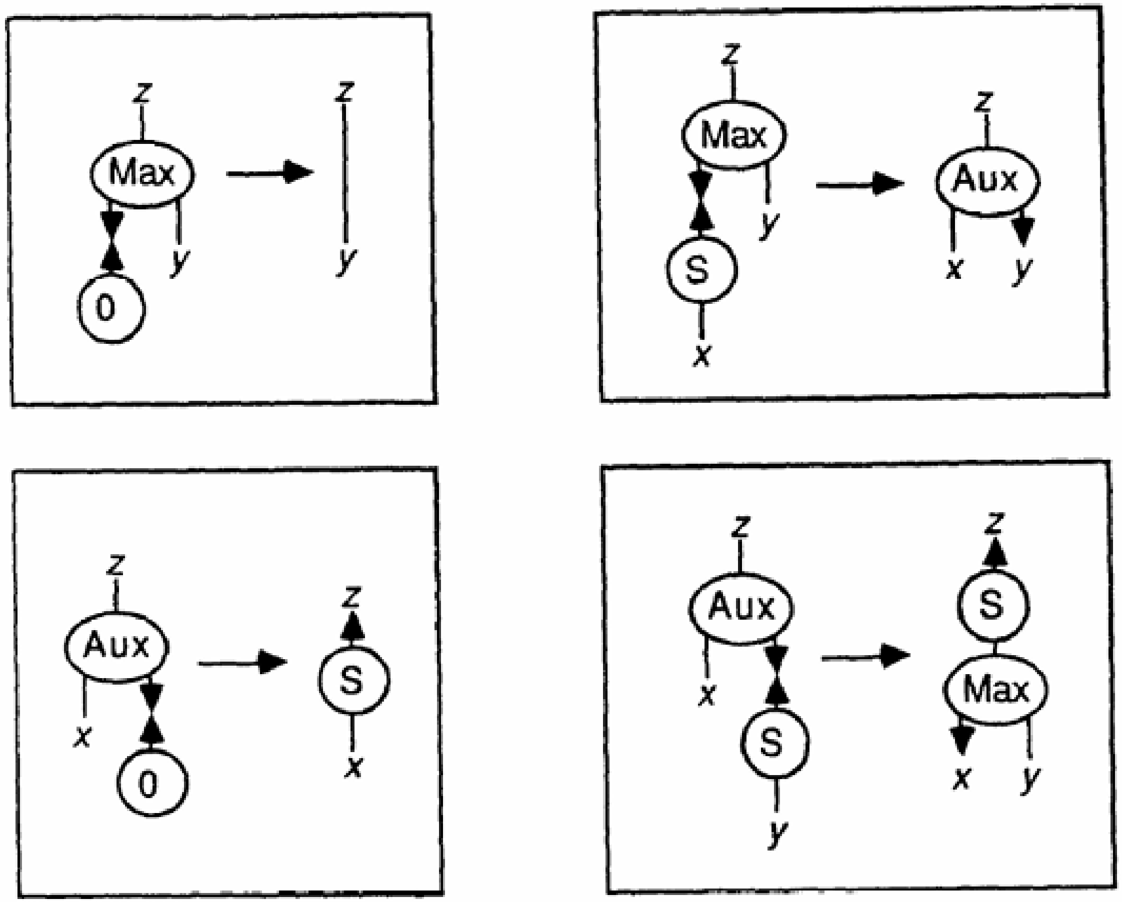
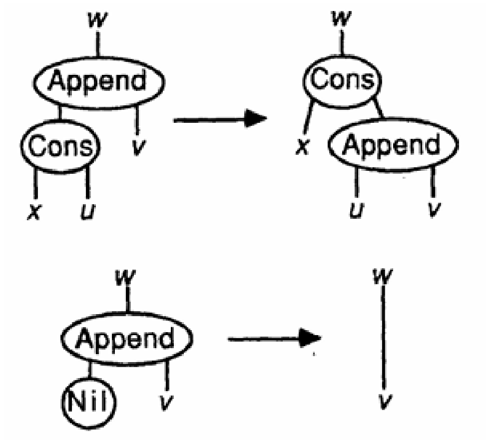


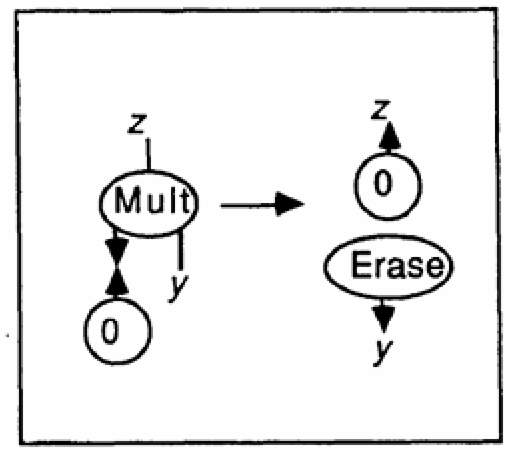
Figure 2: extra symbol for unary maximum

Linearity

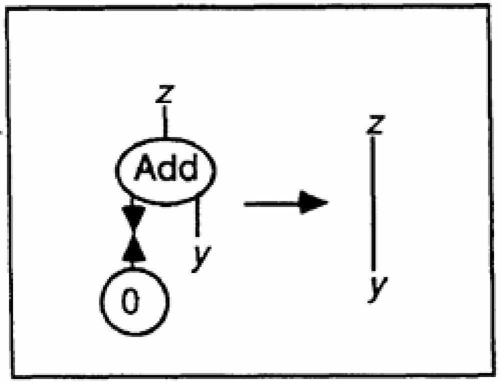


Each variable occurs
once on the left side of a rule,
once on the right.

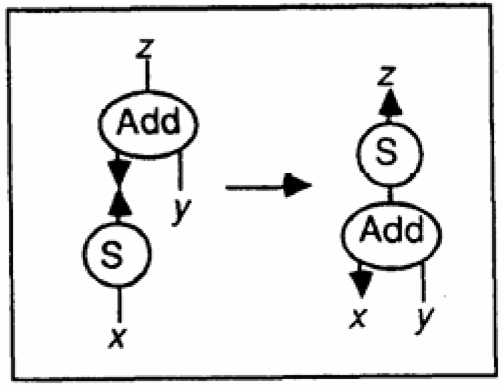
This means we need
explicit ***duplicate*** *and* ***erase*** *symbols*
for algorithms such as
unary multiplication.



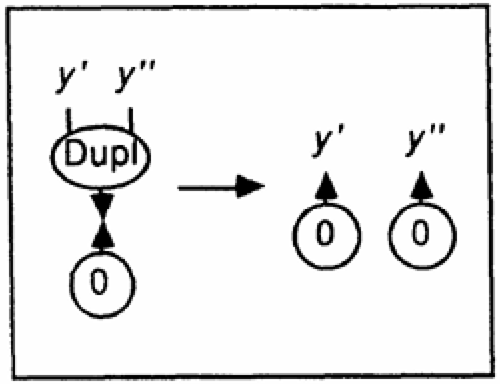
y + 0 = 0



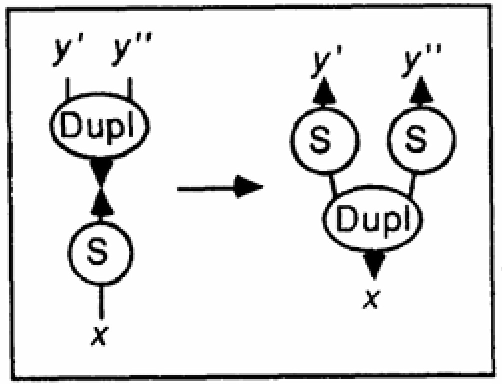
(x + 1) + y =
(x + y) + 1

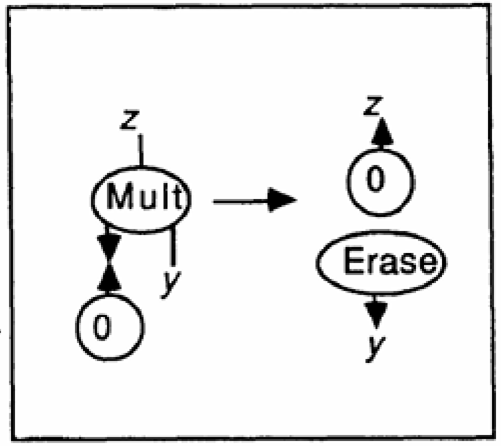


duplication of zero

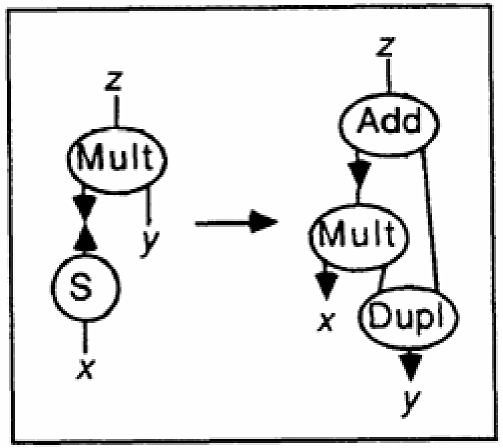


duplication of successor

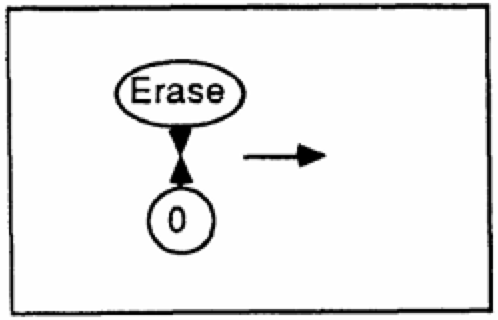




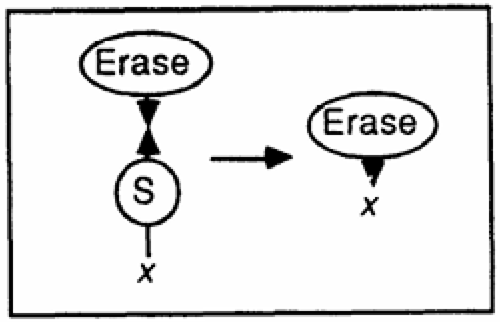
*y * 0 = 0*



*(x + 1) * y = (x * y) + y*

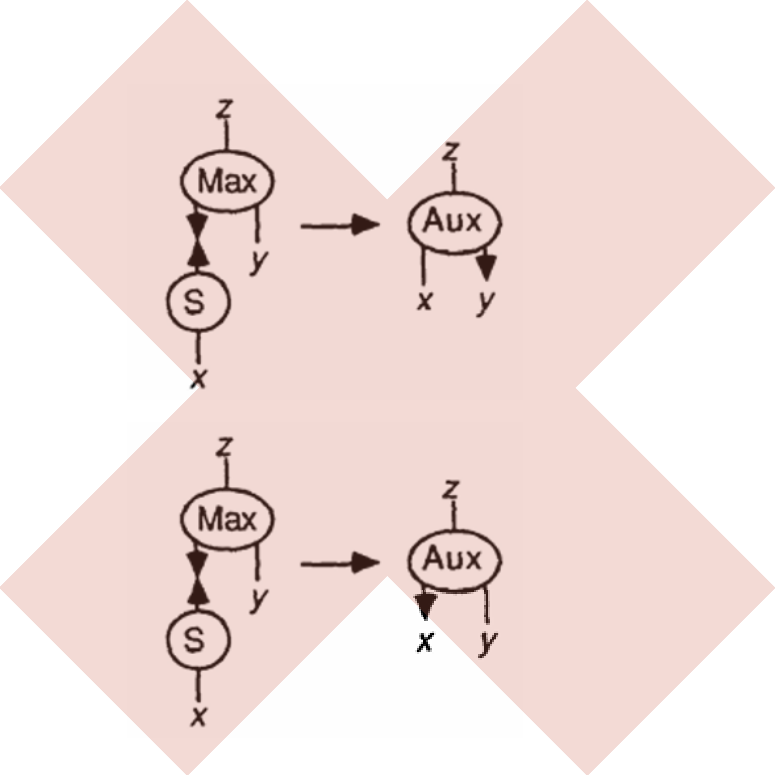


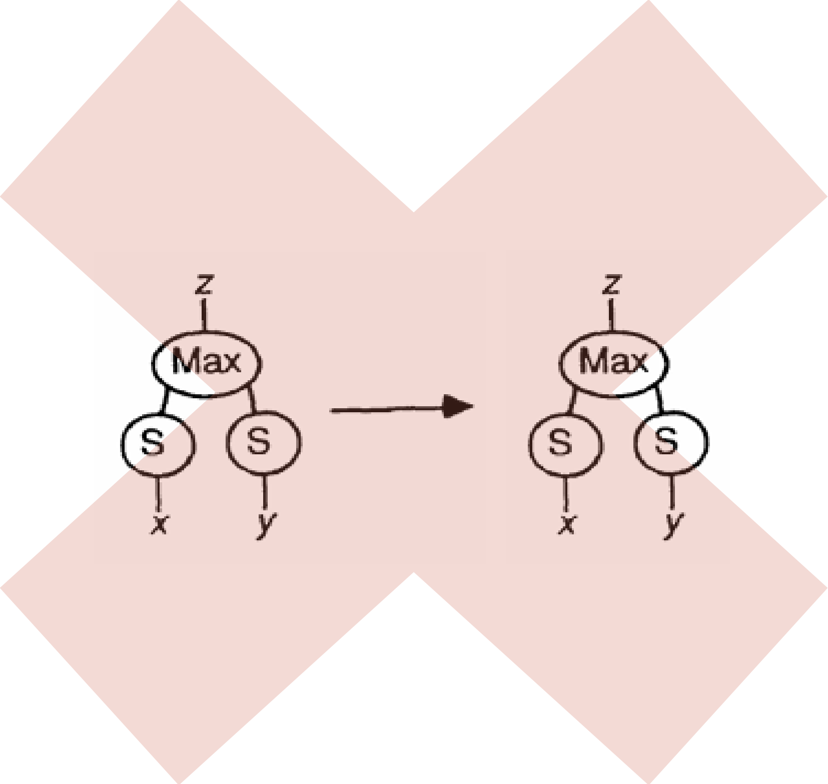
erasure of zero



erasure of successor

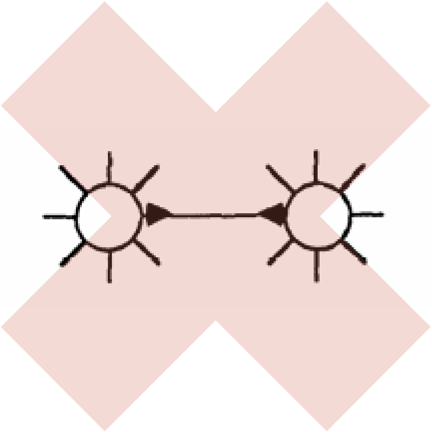
No ambiguity





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

Optimisation



Right side of rules contain no **alive pairs.**

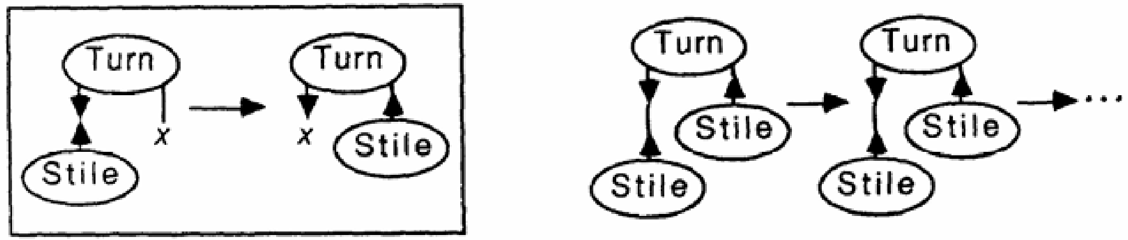


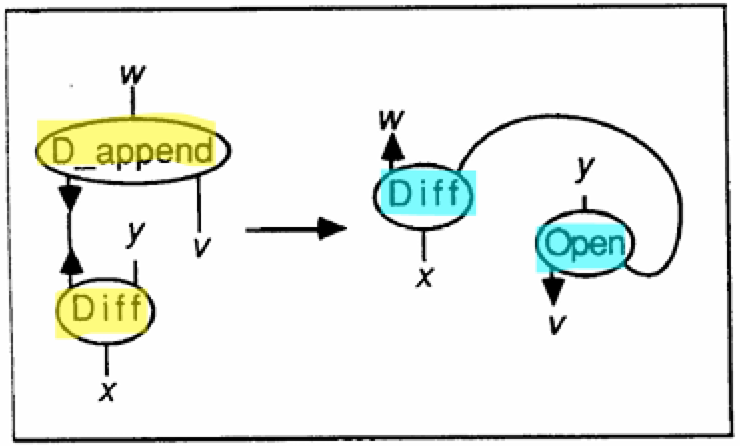
Figure 3: infinite computation with turnstile

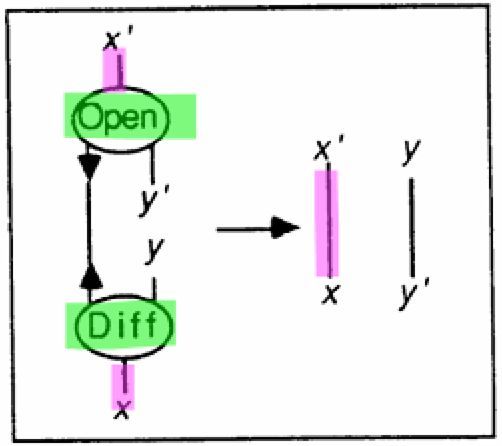
Note that this still allows **non-terminating** computation.

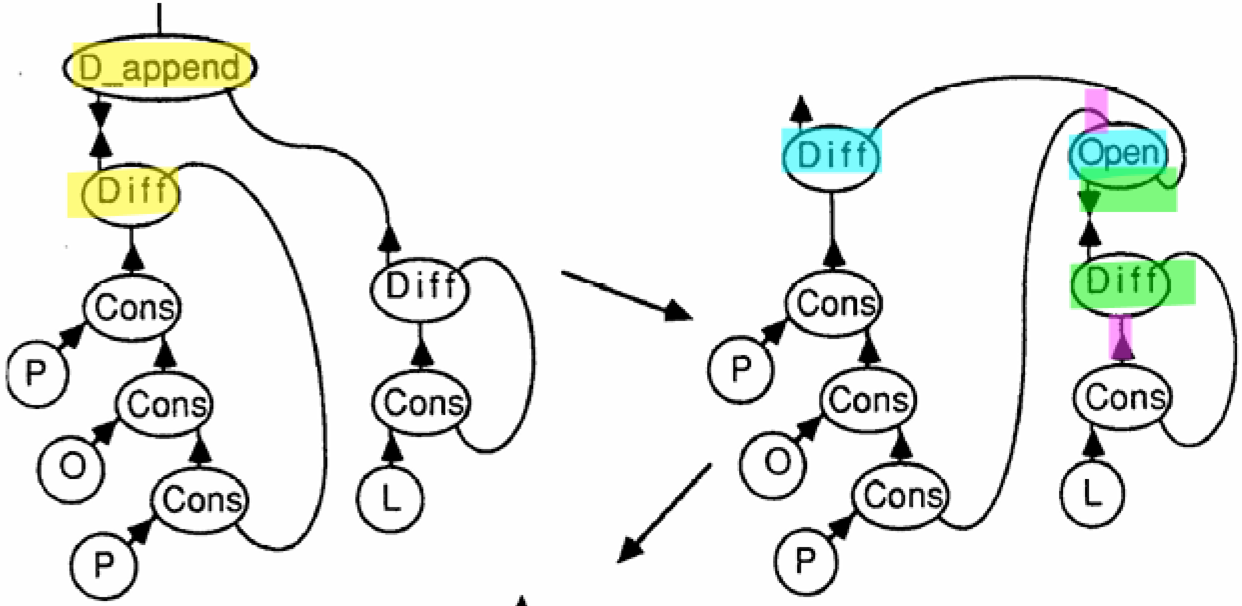
Example:

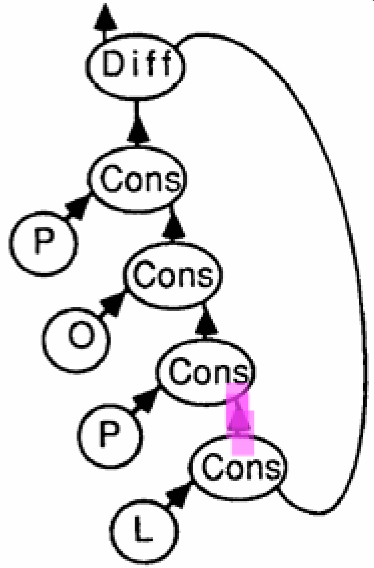
concatenation of difference-lists

Some **new rules**









*Difference lists can be concatenated in **constant time**, compared to **linear time** for normal lists.

Example:

Polish notation parser

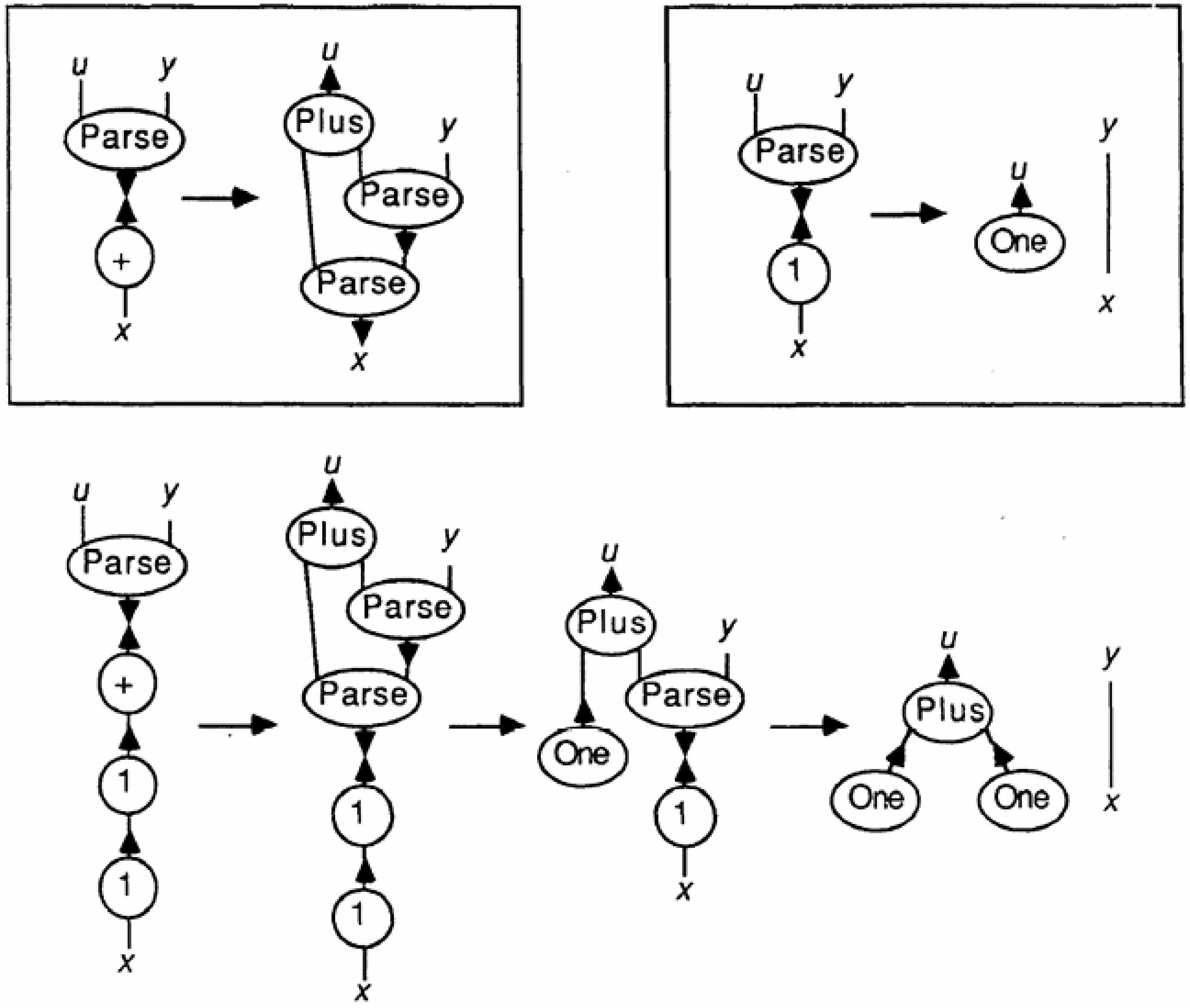


Figure 5: polish parsing

Converts **1 1 +**
into **One Plus One**

A Type Discipline

Constant types

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).

Interacting ports must have matching pairs of outputs (i.e. **list+** and **list-**)

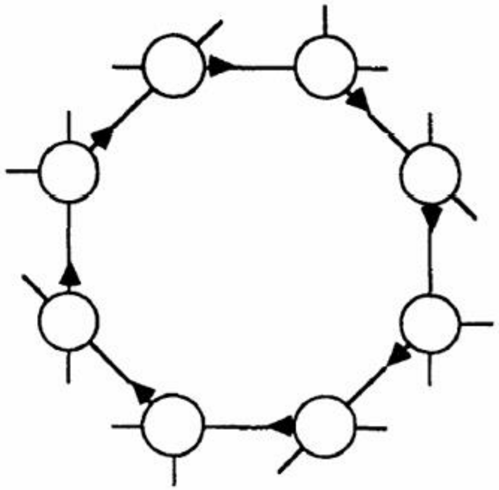
Input/output denomination only matters for matching, not actual function

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.

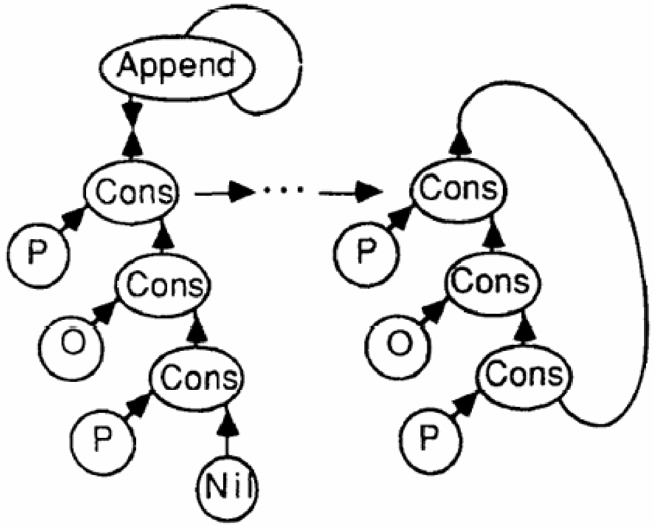
ii) $\mathcal{N}$ contains a vicious circle:



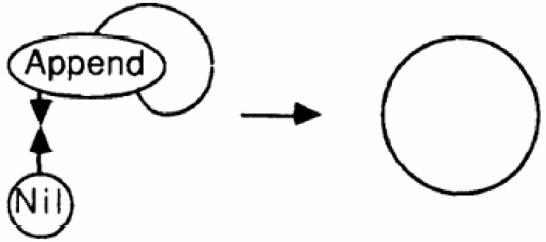
Indeed, starting from any point, you can follow principal ports until you reach a variable, or you loop! Case (i) simply means that $\mathcal{N}$ is ready to interact with its environment, but case (ii) is pathological. In fact, by condition 2, we have clearly:

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

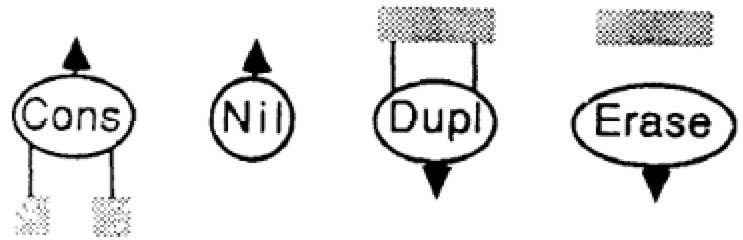
The trouble with vicious circles is that they can appear unexpectedly during a computation:



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



Partitions



i.e. non-principal ports

A partition of the **auxiliary ports** must be given for each symbol.

Ports in the **same partition** must connect to the **same net**.

Ports in **separate partitions** must connect to **different nets**.

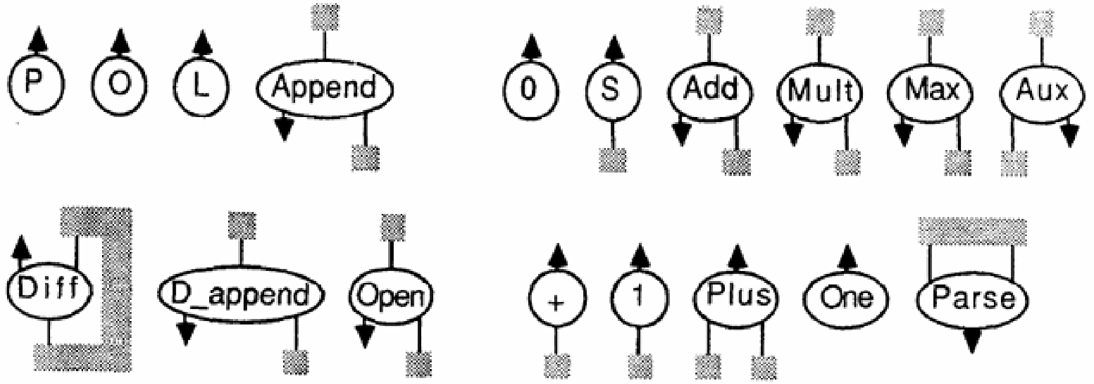
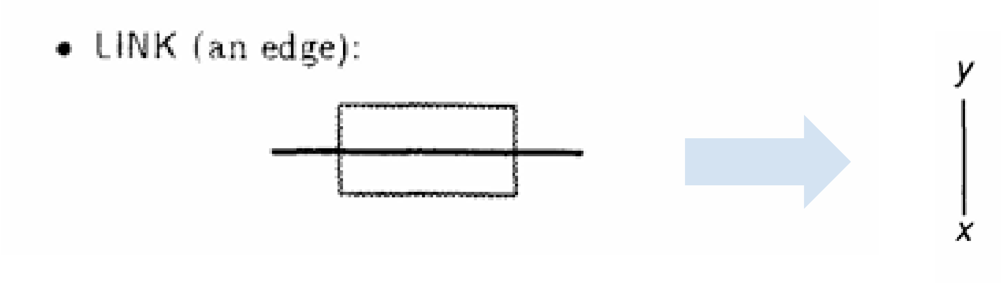


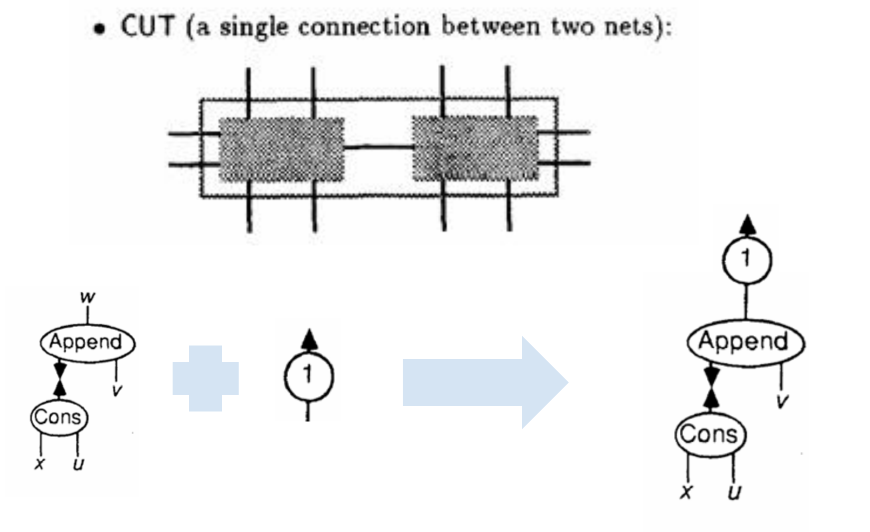
Figure 6: suitable partitions

A net will be called ***simple***, and **free of vicious circles**, if it is constructed with the following **operations**...

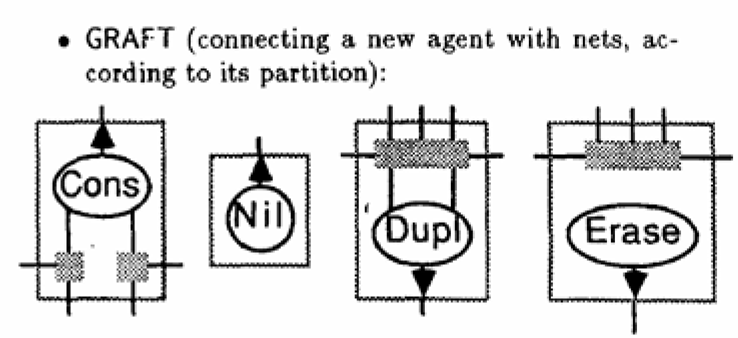
Operations



Base case: create an edge with two free variables



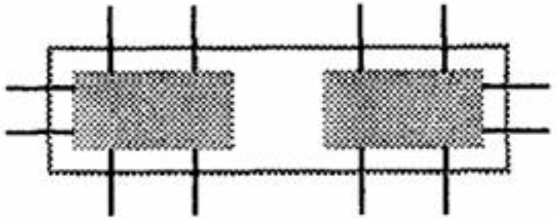
Connects free variables in separate nets



Each partition of the agent must connect to a **different net** (or be left free)

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

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



ok.

Programming Language Syntax

- *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:

$$\textbf{symbol}\quad \begin{array}{l} \mathsf{Cons} : \mathsf{list}^{+} ; \mathsf{atom}^{-}, \mathsf{list}^{-} \\ \mathsf{Nil} : \mathsf{list}^{+} \\ \mathsf{Append} : \mathsf{list}^{-} ; \mathsf{list}^{-}, \mathsf{list}^{+} \end{array}$$

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

$$\textbf{symbol}\quad \begin{array}{l} \mathsf{Dupl} : \mathsf{nat}^{-} ; \{\mathsf{nat}^{+}, \mathsf{nat}^{+}\} \\ \mathsf{Erase} : \mathsf{nat}^{-} ; \{\} \end{array}$$

	Haskell	OCaml	SML	Python	Inpla8	Inpla8r
n-queens 12	**0.23**	0.44	0.60	3.79	0.55	0.44
ack(3,11)	2.37	0.57	**0.42**	-	0.90	0.72
fib 38	1.61	**0.15**	0.27	9.27	0.46	0.46
bsort 20000	5.03	6.47	2.39	20.02	2.41	**1.56**
isort 20000	2.15	1.48	0.60	8.83	**0.33**	0.35
qsort 260000	0.36	0.22	0.27	10.33	0.15	**0.11**
msort 260000	0.38	0.17	0.29	11.09	0.14	**0.13**

https://github.com/inpla/inpla

Conclusion

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.


ε