# Interaction Nets

Simon Gay
Queens'

Diploma in Computer Science 1991

| | |
|---|---|
| **Name** | Simon Gay |
| **College** | Queens' |
| **Project Title** | Interaction Nets |
| **Examination** | Diploma in Computer Science 1991 |
| **Word Count** | Approximately 11200 |
| **Originator** | Dr. A. Pitts |
| **Supervisor** | F.J.M. Davey |

**Aims of Project**

To implement an interactive interpreter for the Interaction Nets language; also to investigate extensions to the basic language, understand the foundation of the language in Girard's Linear Logic, and develop examples of programming in it.

**Work Completed**

The original aims of the project have been realised, with the exception of a couple of things (modularity, investigation of performance) which were in the category of areas which it might have been interesting to look at.

**Special Difficulties**

None.

## Acknowledgements

I would like to thank Dr. Andrew Pitts for coming up with the original idea for a project based on Yves Lafont's Interaction Nets. It turned out to be an extremely interesting and enjoyable project, allowing both practical and theoretical investigation, and opening up the field of Linear Logic which should be a good topic for future work.

I am also grateful to Francis Davey for being a helpful and encouraging supervisor. I have benefited from several discussions with him on Linear Logic and type theory. He also persuaded me to look at polymorphism in Interaction Nets, which became a very fruitful exercise.

## Contents

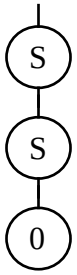
# 0 Introduction

This document describes a project based on the ideas in [1], in which Lafont proposes the language of Interaction Nets based on the proof nets of Girard's Linear Logic [2]. The project consisted of implementing an interactive interpreter for the language, experimenting with extensions to it, developing examples of practical programming, and investigating the theoretical framework of Linear Logic and how Interaction Nets fits into it.

Firstly, the language is described, and illustrated with some simple examples. Then Linear Logic is presented via Sequent Calculus, and the connection with Interaction Nets is explained. Three classes of extension to the language are described, dealing with providing an analogue of macros in addition to functions, providing functions with side-effects, and polymorphic typing. Next, the implementation is discussed, and some example programs presented. Finally there are some further thoughts on possible future directions. A more complete description of the facilities provided by the system, formal syntax definitions, an extract from the actual program, and a copy of the original proposal can be found in the appendices.

# 1 The Interaction Nets Language

Interaction Nets is a programming language based on graph reduction. Programming consists of specifying the ways in which graphs can be constructed, and defining rewriting rules. The graphs are known as **nets**; a net is an undirected graph with labelled vertices, satisfying certain other conditions. In the pure Interaction Nets language there are no built-in definitions, so for example numbers must be defined by the programmer. One way to do this is to use a unary representation and consider nets such as



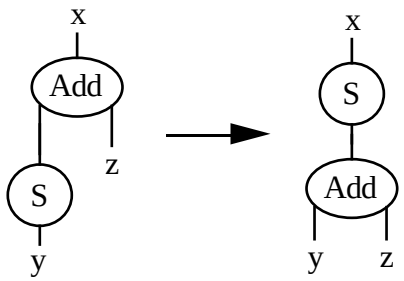

where suitable **symbol definitions** must be given for S and 0 to allow them to be connected up in this way. Observe that a net is allowed to have 'dangling edges' which are only connected at one end. Such edges can be considered as free variables of the net and are usually labelled. In fact, it is also possible to have edges which are connected at neither end; such edges are necessary in order to describe some rewriting rules.

Each **agent** (vertex) in a net is an instance of some symbol. A symbol definition associates the name of a symbol with a specification of how an agent with that name can be connected to other agents. An agent has a number of **ports** to which edges can be attached. Ports are typed, a type specification consisting of a type name and a direction. The directions are **input**, represented by - , and **output**, represented by **+** . Two ports can only be connected if they have the same type and opposite directions. For the arithmetic example, the declarations

```
type      Nat;
symbol    0 : Nat +
          S : Nat + , Nat -;
```

introduce a type Nat and define symbols 0 and S, giving 0 a single port of type Nat + and S two ports of types Nat + and Nat - .

To do arithmetic with numbers represented in this way, **interaction rules** (graph rewriting rules) such as

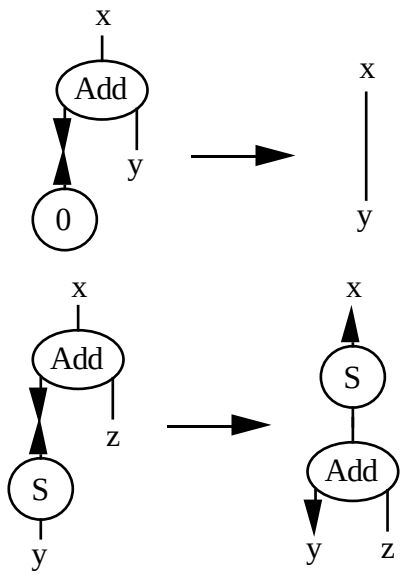



can be used, where Add is defined by

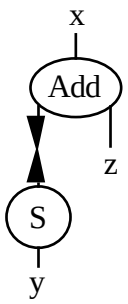
```
symbol    Add : Nat - , Nat - , Nat +;
```

Interaction rules are local; in an interaction, two adjacent agents are replaced by a net. One port of each agent is identified as the **principal port**, and two agents can only interact if they are connected by their principal ports. Graphically, principal ports are

distinguished by arrows. In a symbol definition, the first port listed is the principal port; the above definitions correctly specify the principal port of each symbol. By convention, the rest of the ports are listed in the order obtained by moving anticlockwise round the agent. Given a set of symbol definitions, a complete set of interaction rules includes just one rule for each pair of symbols whose principal ports can be connected together. For 0, S and Add, the rules are




Diagrams like this are the most natural way to understand interaction rules, but a textual syntax is also used. In fact, the current system only works with textual descriptions. The form of type and symbol definitions is described above, and is easily readable. It is essentially the same as the notation described by Lafont. The textual description of a net is obtained by labelling all its edges, including both free edges and internal edges, and listing the agents. Each agent appears in the list as its name followed by the labels on the edges emanating from it, in brackets. The edge labels are listed in the order in which the corresponding ports appear in the symbol definition. So the net




is described by

Add(u,z,x),S(u,y)

or

S(u,y),Add(u,z,x) .

An edge which has a free variable at each end is described by the names of the variables separated by a - .

The empty net, represented graphically by

is described textually by the single character @ .

In this notation, an interaction rule consists of a pair of net descriptions separated by an arrow. The net on the left of the arrow must consist of a pair of agents connected only by their principal ports. This is different from Lafont's notation for rules, which is described in Appendix A; it is intended to be more readable than his. The rules for addition are defined by

```
rule    Add(u,y,x),0(u) -> x-y
        Add(u,y,x),S(u,z) -> S(x,w),Add(z,y,w);
```

With the definitions given so far, the Interaction Nets system can be used to do simple calculations. For example, to calculate 2+2, the command

```
net     Add(u,v,x),S(u,a),S(a,b),0(b),S(v,c),S(c,d),0(d);
```

loads the given net into the system, and the command

```
reduce
```

applies interaction rules until there are no more **active links** (principal ports connected together) and then produces the output

```
S(x,y),S(y,z),S(z,u),S(u,v),0(v)
```

The language of Interaction Nets is based on Linear Logic; nets generalise the proof nets of Linear Logic. The main implication of this connection is that there are linearity constraints on nets and interaction rules. In a net, all free variables must be distinct, that is, all free edges must have distinct labels. In a rule, the two sides must have precisely the same free variables. To see the impact that linearity has in practice, consider defining multiplication for unary arithmetic. Similarly to addition, the necessary symbol definition is

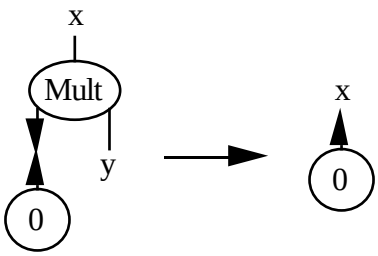
```
symbol          Mult : Nat - , Nat - , Nat - ;
```

and rules are needed to capture the usual recursive equations

$0.x = 0$

$(y+1).x = y.x + x$ .

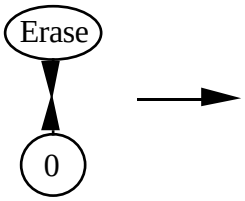
The obvious rule for the first equation is

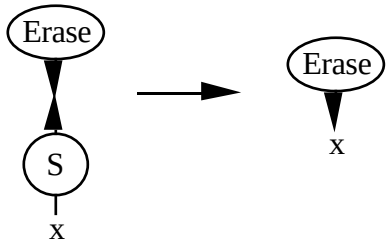



but this is wrong because the left side has more free variables than the right side. The problem is that the multiplication function uses its second argument non-linearly; if the first argument is zero, the second argument is discarded. To express the equations in the form of linear interaction rules, this discarding of the second argument must be made explicit by means of an agent Erase which is defined by

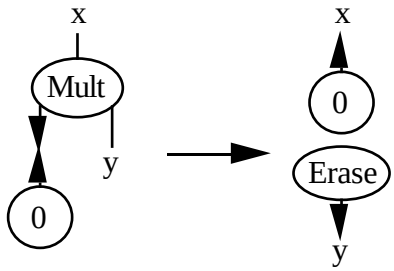
```
symbol          Erase : Nat - ;
```

and interacts according to the rules



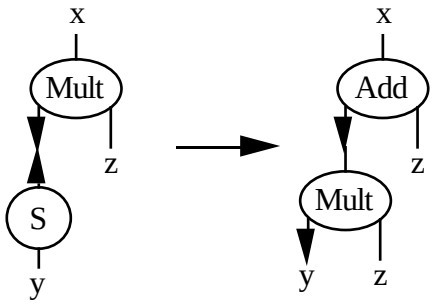



The correct rule for the first multiplication equation can now be given:




and this can be seen to be linear.

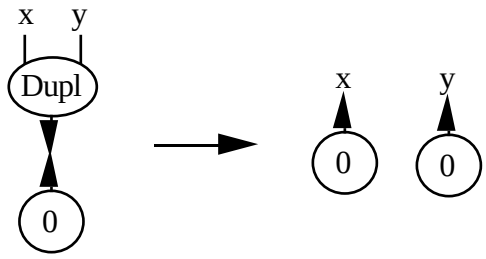
The second equation for multiplication is non-linear in a different way: part of one of the arguments is duplicated. The obvious interaction rule is

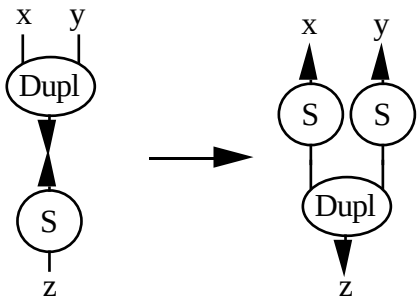



which again violates linearity. The solution this time is to make the duplication explicit by means of the definition

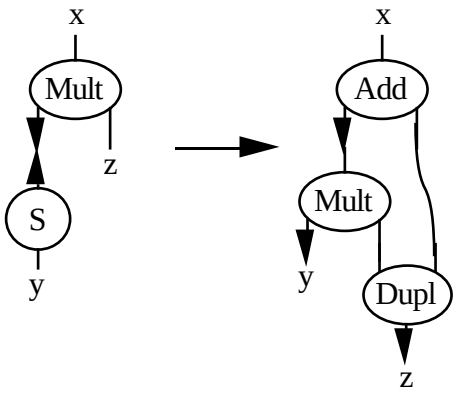
symbol Dupl : Nat - , Nat + , Nat + ;

and rules





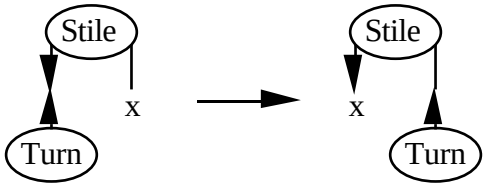

which allow the correct form of the second rule for Mult to be given:

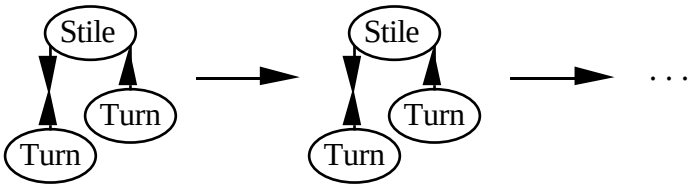



Linearity is really about control of resources. Explicit duplication and deletion mean that an implementation of Interaction Nets needs no garbage collector, since no sharing of substructures is allowed.

Interaction rules must also be well-typed in that as well as the two nets in a rule being well-typed, a free variable must be attached to ports of the same type and direction on both sides of the rule. If the right hand side of a rule contains an isolated edge, the variables on thet edge must be attached in the left hand side of the rule to ports of the same type and opposite directions. It is clear that with these constraints on interaction rules, well-typed nets remain well-typed after any reductions.

It is possible to define rules which lead to non-terminating computations, such as Lafont's ‘turnstile’ example:




but further constraints on nets can ensure that when a sequence of reductions terminates, the result can be interpreted as a meaningful answer rather than a deadlock situation. When reduction stops, the net reached has no active links. Consider starting from some agent and following principal ports around the net. Eventually, either a free variable is reached, or the path loops. If it is always the first case which arises, then the net represents an 'answer' and the free variables are the 'handles' which were attached

to the original problem. The net is ready to interact with its environment - if additional agents are attached to the free variables, reduction might be able to continue. But if the net contains cycles, the situation is a deadlock, and it is these cases which should be eliminated. One possibility would be to forbid the construction of any net containing cycles, but this is unnecessarily restrictive as it makes it impossible to define rules which use duplication, such as the second rule for unary multiplication. However, it is possible to distinguish betweem innocuous and pathological cycles - the programmer knows that a cycle containing a Dupl agent will be broken after a number of applications of the rules for Dupl. The solution is to divide the ports of each agent into partitions which are used in the following way. If a net is built up from the empty net by successively adding agents, which may or may not be connected to existing agents, then at any time the net consists of a number of connected components. Connecting two or more ports of a new agent to a single component forms a cycle in the net. The rule is that this can only be done if all the ports being connected to the same component are in the same partition. This guarantees that nets containing bad cycles cannot be constructed; and, since the same constraint applies to the net on the right hand side of an interaction rule, that no bad cycles can be introduced by reductions.

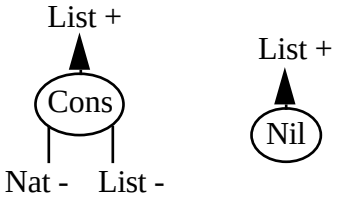
The default is for partitions to be discrete; non-discrete partitions are specified in a symbol definition by enclosing the relevant ports in curly brackets. So the correct definition for Dupl is

```
symbol          Dupl : Nat - , {Nat + , Nat +} ;
```

This means that a partition must consist of a group of adjacent ports, so the order of the ports must be chosen in such a way that this is true. Also, when describing a net containing cycles to the Interaction Nets system, the agent which makes a cycle legal (the agent with a non-discrete partition) must be the last agent in the cycle to be listed, otherwise the system cannot check the partitioning.

Lafont specifies the ‘optimisation’ condition that the right-hand side of an interaction rule should not contain any active links; if it did, they could be reduced at definition-time to save work when reducing nets later. This condition is not enforced in the present system, because if the natural form of a rule contains active links, leaving them in makes the program clearer. Also, reducing such active links at definition-time could lead to non-termination in the parser.

Because they are used in a later example, it is worth mentioning here that lists can be represented in Interaction Nets by means of the agents




(for lists of natural numbers); it is straightforward to write rules for list operations such as appending, as for example in the following sample run.

```
$ inets -pure
Interaction Nets

# type Nat List;

# symbol 0:Nat +
S : Nat + , Nat -
Nil : List +
Cons : List + , Nat - , List -
Append : List - , List - , List + ;
```

```
# rule Append(u,y,x),Nil(u) -> x-y
Append(u,y,x),Cons(u,v,w) -> Cons(x,v,u),Append(w,y,u);

# net Append(a,b,x),Cons(a,c,d),0(c),Nil(d),Cons(b,e,f),S(e,g),0(g),Nil(f);

# reduce
Cons(x,x1,y),0(x1),Cons(y,z,u),S(z,v),0(v),Nil(u)
```

This completes the description of the basic language. Some less trivial examples, illustrating various programming techniques, are discussed in section 5. But now, the theoretical foundations of Interaction Nets will be described.

## 2 Linear Logic and Interaction Nets

In this section Linear Logic is presented, and the connection with Interaction Nets explained. The exposition of Sequent Calculus, Linear Sequent Calculus and proof nets follows but expands that in [2]. The explicit description of how Interaction Nets fits in is new, although from hints in various papers it seems certain that Lafont has developed it previously.

### 2.1 Sequent calculus

Logical axioms and rules of inference can be presented by in Sequent Calculus, where a **sequent** $A_1,..,A_n \vdash B_1,..,B_n$ with the $A_i$ and $B_i$ logical formulae means that $B_1 \vee B_2 \vee .. \vee B_n$ is the consequence of $A_1 \wedge A_2 \wedge .. \wedge A_n$ (where $\wedge$ and $\vee$ in this context are 'meta-connectives', not part of logic); consequence, that is, in the sense of syntactic entailment (hence the use of the single turnstile $\vdash$ ); there is no notion of 'truth'. A system in which the right-hand side of a sequent can contain several formulae is a **classical** system, whereas restricting sequents to be of the form $A_1,..,A_n \vdash B$ yields an **intuitionistic** system. This is the distinction between the philosophy that it is meaningful to prove $A \vee B$ without specifying which has been proved, and the intuitionist philosophy which only admits constructive proofs. In the rules below $\Gamma$, $\Delta$,.. stand for sequences of formulae, and are used to represent the context in which a rule is being used. The interpretation of a rule of Sequent Calculus is that the sequent below the horizontal line can be deduced from those above. Thus Sequent Calculus rules are statements about how valid proofs may be constructed, for example "If from the assumptions $\Gamma$, I can prove A, and from the assumptions $\Delta$, I can prove B, then from the assumptions $\Gamma,\Delta$, I can prove $A \wedge B$ by 'anding' together the proofs of A and B".

When the rule for $\wedge$ described above is put into Sequent Calculus, it takes the form

$$\frac{\Gamma \vdash A,\Delta \quad \Gamma' \vdash B,\Delta'}{\Gamma,\Gamma' \vdash A \wedge B,\Delta,\Delta'}$$

and there is also a slightly different rule

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

There are also rules for introducing $\wedge$ on the left-hand side of sequents:

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

which formalise the idea that if $A \wedge B$ is known then both A and B are known individually. The rules for $\vee$ are similar:

$$\frac{\Gamma,A \vdash \Delta \quad \Gamma',B \vdash \Delta'}{\Gamma,\Gamma',A \vee B \vdash \Delta,\Delta'} \qquad \frac{\Gamma,A \vdash \Delta \quad \Gamma,B \vdash \Delta}{\Gamma,A \vee B \vdash \Delta}$$

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

in fact, $\vee$ being dual to $\wedge$, the rules are the same as those for $\wedge$ but with left and right interchanged. There is some redundancy in this formulation of the logical rules, which will be discussed later.

There is also the identity axiom

$$\frac{}{A \vdash A}\ \mathrm{Id}$$

and the **cut** rule

$$\frac{\Gamma \vdash A,\Delta \quad \Gamma', A \vdash \Delta'}{\Gamma,\Gamma' \vdash \Delta,\Delta'}\ \mathrm{Cut}$$

In addition to these logical rules, the use of certain structural rules is made explicit. These rules control how hypotheses are used in a proof. The exchange rules express the fact that formulae may be written in any order:

$$\frac{\Gamma,A,B,\Gamma' \vdash \Delta}{\Gamma,B,A,\Gamma' \vdash \Delta}\ \mathcal{L}\mathrm{X} \qquad \frac{\Gamma \vdash \Delta,A,B,\Delta'}{\Gamma \vdash \Delta,B,A,\Delta'}\ \mathcal{R}\mathrm{X}$$

The weakening rule on the left allows additional hypotheses to be introduced, which may not necessarily be used in the proof (if A is not listed in $\Gamma$):

$$\frac{\Gamma \vdash \Delta}{\Gamma,A \vdash \Delta}\ \mathcal{L}\mathrm{W}$$

and on the right, in the classical case, allows deductions in the style of 'If I can prove that 2+2=4, then I can prove that either 2+2=4 or the moon is made of green cheese':

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash A,\Delta}\ \mathcal{R}\mathrm{W}$$

The contraction rule on the left states that a hypothesis need only be introduced into a proof once, and can then be used any number of times:

$$\frac{\Gamma,A,A \vdash \Delta}{\Gamma,A \vdash \Delta}\ \mathcal{L}\mathrm{C}$$

and on the right, that proving $A \vee A$ is no better than proving A:

$$\frac{\Gamma \vdash A,A,\Delta}{\Gamma \vdash A,\Delta}\ \mathcal{R}\mathrm{C}$$

We will not discuss quantifiers here, but it is reasonable to ask how negation and implication fit into this scheme. In particular, one might expect to introduce $\Rightarrow$ together with rules

$$\frac{\Gamma,A \vdash B,\Delta}{\Gamma \vdash A \Rightarrow B,\Delta}\ \mathcal{R}\Rightarrow \qquad \frac{\Gamma \vdash A,\Delta \quad \Gamma', B \vdash \Delta'}{\Gamma,\Gamma', A \Rightarrow B \vdash \Delta,\Delta'}\ \mathcal{L}\Rightarrow$$

$$\frac{\Gamma \vdash A,\Delta \quad \Gamma' \vdash A \Rightarrow B,\Delta'}{\Gamma,\Gamma' \vdash B,\Delta,\Delta'} \Rightarrow\text{-Elim}$$

First of all, observe that the rules for operating on the right- and left-hand sides of sequents are essentially the same, with an exchange of $\wedge$ and $\vee$. This repetition can be avoided by introducing negation, so that each atomic proposition comes in the two forms A and $\neg$A, and using asymmetrical sequents, replacing $A_1,..,A_n \vdash B_1,..,B_n$ by $\vdash A_1^{\perp},..A_n^{\perp},B_1,..,B_n$ . This still fits in with intuition: 'I can always prove $\neg A_1 \vee \neg A_2 \vee .. \vee \neg A_n \vee B_1 \vee .. \vee B_n$' is equivalent to 'If I can prove $A_1 \wedge A_2 \wedge .. \wedge A_n$ then I can prove $B_1 \vee B_2 \vee .. \vee B_n$ '. Negation is extended to compound formulae by de Morgan's laws: $\neg(A \wedge B) = \neg A \vee \neg B$, $\neg(A \vee B) = \neg A \wedge \neg B$. And by definition, $\neg\neg A = A$. Putting the rules of Sequent Calculus into asymmetrical form, the identity axiom becomes

$$\frac{}{\vdash A,\neg A}\ \text{Id}$$

the cut rule

$$\frac{\vdash A,\Gamma \quad \vdash \neg A,\Delta}{\vdash \Gamma,\Delta}\ \text{Cut}$$

(note that what were the contexts on opposite sides of a turnstile can now be combined into a single context on the right); the structural rules

$$\frac{\vdash \Gamma,A,B,\Delta}{\vdash \Gamma,B,A,\Delta}\ \text{X} \qquad \frac{\vdash \Gamma}{\vdash A,\Gamma}\ \text{W} \qquad \frac{\vdash A,A,\Gamma}{\vdash A,\Gamma}\ \text{C}$$

and the logical rules

$$\frac{\vdash A,\Gamma}{\vdash A \vee B,\Gamma} \vee 1 \qquad \frac{\vdash B,\Gamma}{\vdash A \vee B,\Gamma} \vee 2 \qquad \frac{\vdash A,B,\Gamma}{\vdash A \vee B,\Gamma} \vee 3$$

$$\frac{\vdash A,\Gamma \quad \vdash B,\Delta}{\vdash A \wedge B,\Gamma,\Delta} \wedge 1 \qquad \frac{\vdash A,\Gamma \quad \vdash B,\Gamma}{\vdash A \wedge B,\Gamma} \wedge 2$$

The number of logical rules can be reduced further. The rule $\wedge 1$ can be derived from $\wedge 2$ by the proof

$$\frac{\dfrac{\vdash A,\Gamma}{\vdash A,\Gamma,\Delta}\ * \qquad \dfrac{\vdash B,\Delta}{\vdash B,\Gamma,\Delta}\ *}{\vdash A \wedge B,\Gamma,\Delta} \wedge 2$$

in which the steps marked * consist of several applications of weakening and exchange. Conversely, the proof

$$\frac{\dfrac{\vdash A,\Gamma \qquad \vdash B,\Gamma}{\vdash A \wedge B,\Gamma,\Gamma} \wedge 2}{\vdash A \wedge B,\Gamma}\ *$$

where the step * consists of several applications of contraction and exchange, derives $\wedge 2$ from $\wedge 1$. For the $\vee$ rules, the proofs

$$\dfrac{\dfrac{\vdash A,\Gamma}{\vdash A,B,\Gamma}\,*}{\vdash A\vee B,\Gamma}\vee 3 \qquad \dfrac{\dfrac{\vdash B,\Gamma}{\vdash A,B,\Gamma}\,*}{\vdash A\vee B,\Gamma}\vee 3$$

derive the rules $\vee 1$ and $\vee 2$ from $\vee 3$; * is weakening, with an exchange in the first proof. Finally the proof

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash A,B,\Gamma}{\vdash A\vee B,B,\Gamma}\vee 1}{\vdash B,A\vee B,\Gamma}\,X}{\vdash A\vee B,A\vee B,\Gamma}\vee 2}{\vdash A\vee B,\Gamma}\,C$$

recovers $\vee 3$ from $\vee 1$ and $\vee 2$. Thus it is sufficient to have just two logical rules, $\vee 3$ and either $\wedge 1$ or $\wedge 2$.

Going back to the question of how to include implication, $A\Rightarrow B$ can be introduced as a shorthand for $\neg A\vee B$. In the asymmetrical form, the rule $\mathcal{R}\Rightarrow$ is

$$\dfrac{\vdash \neg A,B,\Gamma}{\vdash A\Rightarrow B,\Gamma}$$

which is just $\vee 3$, and similarly $\mathcal{L}\Rightarrow$ is an instance of $\wedge 1$. The elimination rule becomes

$$\dfrac{\vdash A,\Gamma \quad \vdash A\Rightarrow B,\Delta}{\vdash B,\Gamma,\Delta}$$

which is obtained from the existing rules by the proof

$$\dfrac{\dfrac{\dfrac{}{\vdash \neg B,B}\,Id \quad \vdash A,\Gamma}{\vdash A\wedge\neg B,B,\Gamma}\wedge 1 \qquad \vdash \neg A\wedge B,\Delta}{\vdash B,\Gamma,\Delta}\,Cut$$

This recovering of an elimination rule by using the cut rule is a general feature; the cut rule is an alternative to specifying elimination rules for each connective.

The Cut Elimination Theorem, due to Gentzen, states that occurrences of the cut rule can be eliminated from any Sequent Calculus proof which has no hypotheses (that is, from any proof of a tautology). A proof can be found in [5]. It will be seen later that cut elimination has an interesting computational interpretation.

### 2.2 Linear Sequent Calculus

Linear Logic will now be introduced, via Linear Sequent Calculus. Linear Logic was discovered by Girard, who noticed that in a certain model (coherence semantics) of ordinary logic, implication can be broken down into simpler operations. This view will not be discussed here; more details can be found in [2] and [3]. We just note that Linear Logic is lower-level than traditional logic in that it allows finer control of resources.

The appropriate step leading from Sequent Calculus to Linear Sequent Calculus is to discard the structural rules of weakening and contraction. The result is a system in which a proof must use each of its hypotheses exactly once; thus there is a sort of ‘conservation of propositions’ going on. The lack of weakening and contraction means that the alternative formulations of the logical rules for $\wedge$ and $\vee$ are no longer equivalent. To reflect the fact that there are now two essentially different ways of using

each connective, two conjunctions and two disjunctions are introduced. The **tensor product** (cumulative conjunction) ⊗ (times) combines contexts:

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

while the **direct product** (alternative conjunction) & (with) works within a single context:

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

The dual connectives ⅋ (par) for ⊗ and ⊕ (plus) for & have rules

$$\frac{\vdash A,B,\Gamma}{\vdash A⅋B,\Gamma}\ ⅋ \qquad \frac{\vdash A,\Gamma}{\vdash A\oplus B,\Gamma}\ \oplus 1 \qquad \frac{\vdash B,\Gamma}{\vdash A\oplus B,\Gamma}\ \oplus 2$$

and are related to the conjunctions by de Morgan's laws:

$$(A \otimes B)^{\perp} = A^{\perp} ⅋ B^{\perp} \qquad (A \,\&\, B)^{\perp} = A^{\perp} \oplus B^{\perp}$$
$$(A ⅋ B)^{\perp} = A^{\perp} \otimes B^{\perp} \qquad (A \oplus B)^{\perp} = A^{\perp} \,\&\, B^{\perp}$$

where $^{\perp}$ is linear negation. As before, the introduction of negation and the use of asymmetrical sequents are purely for convenience.

Incidentally, another way to look at the splitting of each intuitionistic connective into multiplicative and additive versions, pointed out by Davey (private communication) is that if both introduction and elimination rules are given the multiplicatives are introduced like intuitionistic connectives and the additives are eliminated like them.

Units can also be introduced: **1** for ⊗, ⊥ for ⅋, ⊤ for & and **0** for ⊕. They form two pairs of mutually linearly negated formulae, with

$$\mathbf{1}^{\perp} = \perp \qquad \perp^{\perp} = \mathbf{1} \qquad \top^{\perp} = \mathbf{0} \qquad \mathbf{0}^{\perp} = \top$$

and rules

$$\frac{\vdash \Gamma}{\vdash \perp,\Gamma}\ \perp \qquad \frac{}{\vdash \mathbf{1}}\ \mathbf{1} \qquad \text{(no rule for } \mathbf{0}\text{)} \qquad \frac{}{\vdash \top,\Gamma}\ \top$$

Discarding the contraction and weakening rules represents a significant loss of power in the logic. To recover it, contraction and weakening are brought back in a controlled form, via the introduction of the modality ! (of course) and its dual ? (why not):

$$(!A)^{\perp} = ?A^{\perp} \qquad (?A)^{\perp} = !A^{\perp}$$

with the structural rules appearing in the form of logical rules for the modalities:

$$\frac{\vdash \Gamma}{\vdash ?A,\Gamma}\ W? \qquad \frac{\vdash ?A,?A,\Gamma}{\vdash ?A,\Gamma}\ C? \qquad \frac{\vdash A,\Gamma}{\vdash ?A,\Gamma}\ D? \qquad \frac{\vdash A,?\Gamma}{\vdash !A,?\Gamma}\ !$$

where the W, C and D stand for contraction, weakening and dereliction.

### 2.3 Proof Nets

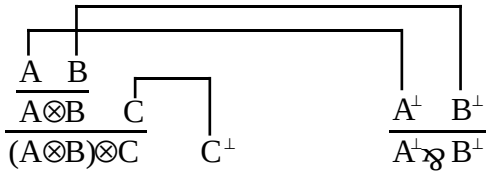
A proof written in Sequent Calculus contains a lot of redundancy, as contexts are rewritten without change each time a logical rule is used. The interesting features of a proof can displayed in the form of a proof net. Proof nets apply only to the so-called multiplicative fragment of Linear Logic, containing ⊗, 1, ⅋ and ⊥. These are the connectives for which the sequent rules have the property that the contexts in the

conclusion are the disjoint unions of those in the premise. The other connectives are the additives. The multiplicative rules add contexts together, while the additive rules do not!

In a proof net, negated pairs of propositions are linked together when introduced, uses of logical rules appear as before but without the contexts (the context is the rest of the proof net), and the cut rule appears as a line with nothing below it. Also, there is no need to explicitly indicate uses of the structural rule. For example, the proof

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash A, A^{\perp} \quad \vdash B, B^{\perp}}{\vdash A \otimes B, A^{\perp}, B^{\perp}} \otimes \qquad \vdash C, C^{\perp}}{\vdash (A \otimes B) \otimes C, A^{\perp}, B^{\perp}, C^{\perp}} \otimes}{\vdash A^{\perp}, B^{\perp}, (A \otimes B) \otimes C, C} \text{X}}{\vdash A^{\perp} \parr B^{\perp}, (A \otimes B) \otimes C, C^{\perp}} \parr$$

becomes

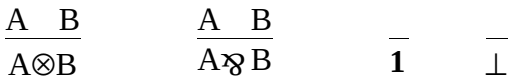



Formally, a proof net is a graph constructed from the components

link:


cut:


and logical rules:




according to the conditions that each formula must be the conclusion of exactly one rule and a premise of at most one rule (formulae which are not premises are conclusions of the proof net), and the rules of Linear Sequent Calculus:

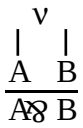
If A and $A^{\perp}$ are conclusions of the proof nets ν and ν' , then


is a proof net.

If A and B are conclusions of the proof nets ν and ν' , then


is a proof net.

If A and B are conclusions of the same proof net ν, then



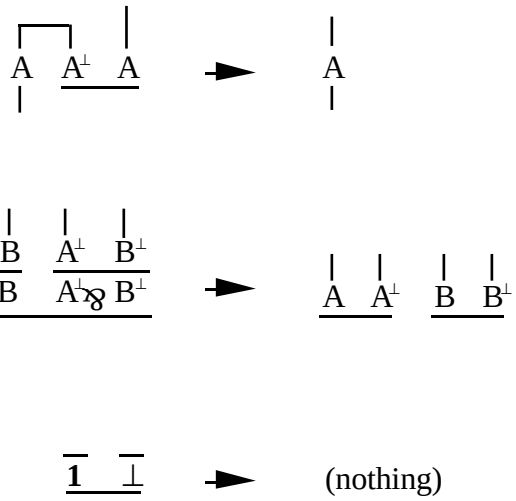

is a proof net.

If ν is a proof net, then

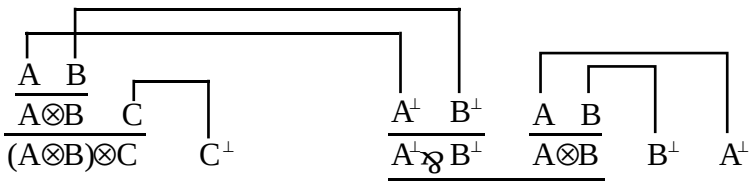

is a proof net.

And finally, $\overline{\mathbf{1}}$ is a proof net.

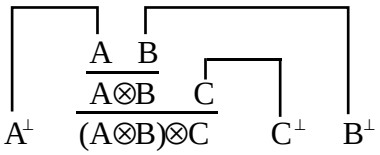
Cut elimination in a proof net consists of rewrites:




For example, the proof net




reduces in three steps to




It can already be seen that there is a similarity between proof nets and Interaction Nets. In the next section, the connection is described in detail.

### 2.4 From Linear Logic to Interaction Nets

A correspondence between proofs in a constructive logic and programs in a typed programming language can be set up via the Curry-Howard Isomorphism. Logical propositions are interpreted as types, and a proof of a proposition corresponds to a

program computing a value of the corresponding type. Cut elimination, which reduces a proof to a simpler proof (one without cuts) corresponds to evaluating the program to eliminate function calls. As cut elimination proceeds, extra information is accumulated on the computational side of the correspondence, leading to the value of the final term (rather than its type) being interpreted as the result of the program. In a traditional functional programming language based on lambda calculus, the correspondence with Intuitionistic Logic can be seen; cut elimination corresponds to beta-conversion.

In this way, the relationship between the multiplicative fragment of Linear Logic and Interaction Nets can be obtained. Types of ports (of agents) correspond to propositions, and symbol definitions correspond to logical rules. An agent in a net corresponds to the use of some logical rule in a proof; a net is a proof of the propositions corresponding to the types of its free variables. In fact, the net is a graphical representation of the corresponding proof, in the way that proof nets are. Interaction between agents corresponds to cut elimination; a complete set of interaction rules is a collection of convresions making a proof of the cut elimination theorem for the logic described by the type and symbol definitions possible. Of course, this proof may not work; the non-terminating turnstile example is a logic for which the cut elimination theorem fails.

Partitioning of the ports of an agent reflects the way in which logical rules can combine different contexts (as for $\otimes$) or work within a single context (as for $\parr$). In the rules for constructing proof nets, $\parr$ *must* connect twice to the same proof net, whereas a non-discrete partition of an agent's ports merely allows the possibility. This is related to the distinction drawn by Lafont [1] between *simple* and *semi-simple* nets. A simple net is constructed by using the rules LINK (an edge), CUT (a single connection between two nets) and GRAFT (connecting a new agent to nets, according to its partition). The partitioning rule in this case demands that all ports in a partition must be connected to the same net (which will be connected, viewed as a graph); in addition, an agent such as Erase can only be introduced without a connection if its non-principal ports are all in a partition together (an empty partition, of course). This means that when Erase interacts with 0 (say), what is left is not an empty net but a lone empty partition, a distinction which is not made in the present system. But some agents of the same 'shape', such as 0, do not have an empty partition. This difference is clearly similar to the difference between the Sequent Calculus rules for **1** and $\bot$; the rule for $\bot$ involves a context (which may be empty) while that for **1** does not. A semi-simple net can be constructed using the above rules, together with the extra rules EMPTY (an empty net) and MIX (juxtaposing two nets without a connection). The present system allows semi-simple nets. It is interesting to observe that the MIX rule for constructing nets corresponds to a structural rule

$$\frac{\vdash \Gamma \quad \vdash \Delta}{\vdash \Gamma, \Delta}$$

which is not a rule of Linear Sequent Calculus. The question of whether to include this rule is discussed in [3], and decided in the negative.

More generally, Interaction Nets can be seen as corresponding to the 'proof nets' of multiplicative logics (in which logical rules combine contexts), not just Linear Logic. While interaction rules are linear in terms of occurrences of free variables, when the types are considered an agent such as Dupl seems to correspond to contraction. It is not clear exactly what is going on here; maybe Dupl implicitly applies modalities to its arguments.

The Interaction Nets language as described up to now allows only atomic types, and a program in it describes a logic with no connectives. But the language has been extended to allow the definition of polymorphic type constructors, corresponding to logical connectives. This extension is described, with others, in the next section.

# 3 Additional Language Features

## 3.1 Pseudo-agents

Pseudo-agents are the Interaction Nets equivalent of macros in traditional programming languages. A pseudo-agent has a symbol definition as for an agent, and an expansion rule which is like an interaction rule but with only the pseudo-agent on the left hand side. The expansion rule specifies a net by which the pseudo-agent is replaced when it interacts with another agent. An example in which pseudo-agents are useful is defining the function Reverse for lists. If lists are introduced via

```
type      List;
symbol    Nil : List +
          Cons : List +, Int -, List - ;
```

then the usual efficient reversing algorithm can be implemented by

```
symbol    Rev : List -, List -, List + ;
rule      Rev(u, y, x), Nil(u) -> x-y
          Rev(u, y, x), Cons(u, v, w) -> Rev(w, s, x), Cons(s, v, y);
```

but then when using Rev, the auxiliary input must be given Nil to start with. The extra definitions

```
symbol    Reverse : List - , List +;
rule      Reverse(y, x) -> Rev(y, u, x),Nil(u);
```

introduce a pseudo-agent Reverse; when Reverse is about to interact with Cons or Nil, the expansion rule is applied to replace the Reverse with a Rev and the appropriate Nil.

Pseudo-agents do not add any expressive power to the language. If the expansion rule for a pseudo-agent does not contain the pseudo-agent itself on the right-hand side, then the pseudo-agent can be eliminated from a program by expanding it in all rules and nets. If a pseudo-agent does appear in its own expansion, eliminating it involves replacing it by an agent with a dummy principal port. For example, a pseudo-agent defined by

```
symbol    Foo : Int - , Int +;
rule      Foo(y, x) -> . . . Foo(u, v) . . . ;
```

can be replaced by

```
type      Signal;
symbol    Foo : Signal - , Int - , Signal - , Int +
          Go : Signal + ;
rule      Foo(t, y, z, x), Go(t) -> . . . Foo(z, u, s, v), Go(s) ;
```

with an occurrence of Foo(y, x) in a top-level net being replaced by

```
Foo(u, y, v, x), Go(u), Go(v) .
```

Despite this, pseudo-agents are valuable as a means of hiding auxiliary inputs, as in the Reverse example and other recursive algorithms, and for avoiding the use of Go signals which would tend to obscure certain algorithms otherwise.

## 3.2 Built-in agents and interaction rules

The basic language of Interaction Nets has been extended by the addition of built-in definitions to provide input/output, character handling, and integer arithmetic. The Interaction Nets system includes these extensions by default, but they can be turned off in order to use ‘pure’ Interaction Nets. This section describes the behaviour of the new built-in agents and interaction rules. The built-in definitions are processed as any other

definitions at system start-up time, but their internal representations are modified to make them behave in the desired way. Introducing built-in integers makes it much more difficult to check that a program has just one rule for each possible pair of interacting agents; checking sufficiency of rules has not been extended to cover the built-in agents, and in the current system the checking which was previously implemented has been disabled.

### 3.2.1 Characters

The following definitions are made:

```
type        Char ;
symbol      AChar : Char +
            EraseChar : Char -
            DuplChar : Char - , {Char + , Char +} ;
rule        EraseChar(u),AChar(u) -> @
            DuplChar(u,x,y),AChar(u) -> AChar(x),AChar(y) ;
```

An AChar agent is a prototype character. In fact the system behaves as if there is an agent for each ASCII character; the name of such an agent is a character in single quotes. So the effect is as if there were definitions

```
symbol      'A' : Char +
            'q' : Char +
            . . .
```

The interaction rules involving AChar agents work for all character agents in the way specified by the above definitions, with the additional property that DuplChar correctly duplicates the particular character it is given, ie the information as to which character is present is carried across the reduction.

### 3.2.2 Arithmetic

The following definitions are made (the type Int is used to distinguish the built-in integers from the type Nat of natural numbers defined by successors):

```
type        Int
            Bool ;
symbol      AnInt : Int +
            EraseInt : Int -
            DuplInt : Int - , {Int + , Int +}
            ABool : Bool +
            EraseBool : Bool -
            DuplBool : Bool - , {Bool + , Bool +}
            Add : Int - , Int - , Int +
            AddN : Int - , Int +
            Negate : Int - , Int +
            Equal : Int - , Int - , Bool +
            EqualsN : Int - , Bool +
            LessThan : Int - , Int - , Bool +
            MoreThanN : Int - , Bool + ;
```

As for characters, AnInt and ABool are prototypes; the system behaves as if there is an agent for each integer, with names ...,~2, ~1, 0, 1, 2,... , and agents called TRUE and FALSE for the booleans. The interaction rules are again prototypes. There are the obvious rules for erasing and duplication. The rules for addition are

```
rule        Add(u,v,x),AnInt(u) -> AddN(v,x)
            AddN(u,x),AnInt(u) -> AnInt(x) ;
```

where AddN is a prototype for agents called Add1, Add2,... which behave in the expected way : Add3(u,x),5(u) reduces to 8(x). The rule for Add is a prototype for infinitely many (actually only finitely many due to the modular arithmetic on the underlying machine) rules of the form

```
Add(u,v,x),1(u) -> Add1(v,x)
Add(u,v,x),~1(u) -> Add~1(v,x)
. . .
```

The rule for Negate is straightforward:

```
rule        Negate(u,x),AnInt(u) -> AnInt(x)
```

and again, this is a prototype for infinitely many rules. The equality and comparison tests are handled similarly to addition via the rules

```
rule        Equal(u,v,x),AnInt(u) -> EqualsN(v,x)
            EqualsN(u,x),AnInt(u) -> ABool(x)
            LessThan(u,v,x),AnInt(u) -> MoreThanN(v,x)
            MoreThanN(u,x),AnInt(u) -> ABool(x);
```

as demonstrated by the dialogue

```
# net Equal(u,v,x),3(u),4(v);

# reduce1
Equals3(x1,x),4(x1)

# reduce1
FALSE(x)

# net LessThan(u,v,x),~5(u),7(v);

# reduce1
MoreThan~5(x1,x),7(x1)

# reduce1
TRUE(x)
```

Using these symbols and rules, it is possible to define the other arithmetic operations. However, although it is possible to write

```
symbol      Foo : Bool - , Int - , Int + ;
rule        Foo(u,v,x),TRUE(u) -> x-v
            Foo(u,v,x),FALSE(u) -> Negate(v,x) ;
```

there is no way for a user-defined rule to involve, say, an AnInt agent and specify that the particular integer required depends on other AnInt agents. So although the rule for EraseInt does not have to be built-in, the rule for DuplInt does.

### 3.2.3 Input and Output

The definitions for input and output are

```
type        In
            Out ;
symbol      InStream : In +
            EraseInStream : In -
            OutStream : Out -
            EraseOutStream : Out +
            ReadChar : In - , In + , Char +
            ReadInt : In - , In + , Int +
            PrintChar : Char - , Out - , Out +
```

```
PrintAChar : Out + , Out -
PrintInt : Int - , Out - , Out +
PrintAnInt : Out + , Out - ;
```

Consider the rules for reading and printing characters; those for integers are similar. The way ReadChar works is by interacting with an input stream to produce a character and the rest of the input stream. The prototype rule is

```
rule        ReadChar(u,y,x),InStream(u) -> AChar(x),InStream(y) .
```

In the present system, all InStreams read from standard input, but in a more general system the InStream on the right of the above rule would be connected to the same file as the one on the left. The AChar on the right of the rule becomes, when this rule is used in a reduction, the next character on the input. The reason for making ReadChar output the rest of the input stream is to ensure that characters can be used in the order in which they are read.

Printing characters is similar but requires an auxiliary agent:

```
rule        PrintChar(u,y,x),AChar(u) -> PrintAChar(x,y)
            PrintAChar(u,y),OutStream(u) -> OutStream(y)
```

In the first rule, the PrintAChar becomes Print'A' , Print'B' etc, depending on the AChar on the left. When the second rule is used, the character in the PrintAChar is sent to the output stream, and as for ReadChar, the output stream is also returned. This ensures that characters can be written out in a known order.

### 3.2.4 Drawbacks

This approach to providing built-in agents is not necessarily the best one. The main problem is that when objects are not built by recursive use of constructors, an agent representing a function can no longer destructure its argument by pattern-matching. For integers, this means that equality testing and hence booleans must also be built-in, although the booleans, being a finite and small type, could perfectly well be defined by the user. Also, the introduction of pseudo-agents is necessary. Consider defining the factorial function for the built-in integers. There will be an agent Fact with Int input and output, which must test its argument for equality with zero. Thus when Fact is used, it must actually be an Equals0 agent which interacts with the input, and so Fact has to be a pseudo-agent.

Possibly the ideal solution to the problem of providing built-in integers would allow the usual S and 0 definitions to be used, with the system somehow recognising that a sequence of S agents ending in a 0 agent represents an integer and implementing integer functions directly. So a few standard integer functions could be implemented efficiently, but it would also be possible to define (for example) Fact in the usual unary way (but it would use built-in multiplication). Integers could also be printed and read as digits. No work has been done towards implementing such an improved scheme.

## 3.3 Polymorphic typing

Interaction Nets can benefit as much as any language from the introduction of a polymorphic type system. It was not immediately obvious what form polymorphic typing should take in Interaction Nets, but it was decided that the natural approach would be to provide facilities whereby the correspondence with Linear Logic could be extended from atomic propositions to logical connectives. The result is that type constructors can be defined, corresponding to logical connectives; they can be of arity one or two. Instead of annotating types with + or -, ^ is used to represent the linear negation $^{\perp}$. Negating a type constructor gives a dual constructor, corresponding to the dual connective. Type variables `a, `b, ... can be used in type expressions; they can also be negated by ^. Two ports can be connected if the type of one is the negation (as a

logical formula) of the type of the other. Building a net involves unifying the types of ports being connected, instantiating type variables if necessary.

For example, the definitions for polymorphic lists are

```
type        List : 1;
```

which declares List as a (postfix) type constructor of arity one, and

```
symbol      Nil : `a List
            Cons : `a List , `a^, `a^ List^ ;
```

where the type `a^ List^ represents an input of type `a List, being its dual. Defining

```
type        Int;
symbol      0 : Int
            S : Int , Int^ ;
```

means that a net such as

```
Cons(x, y, z),0(y),Nil(z)
```

can be constructed; in this net, the free variable x has type Int List, which the system indicates by displaying the net as

```
Cons(x : Int List , y, z),0(y),Nil(z)
```

The built-in definitions can be made polymorphic: EraseChar, EraseInt and so on can be replaced by a single Erase, defined by

```
symbol      Erase : `a ;
```

and similarly for duplication:

```
symbol      Dupl : `a^ , {`a , `a} ;
```

but separate rules still have to be defined for interactions with each possible agent.

With this type system, it is possible to express the multiplicative fragment of Linear Logic within Interaction Nets via the definitions

```
type        1
            T : 2;
```

which declare the unit type 1 and the (infix, associating to the left) type constructor T (for ⊗) of arity 2,

```
symbol      One : 1
            Bottom : 1^
            Times : `a T `b , `b^ , `a^
            Par : `a T^ `b , {`b^ , `a^ } ;
```

giving agents corresponding to the usual logical rules, and

```
rule        One(x),Bottom(x) -> @
            Times(x, y, z),Par(x, u, v) -> y-u , z-v ;
```

which enable the cut elimination theorem to be proved. Type variables on free edges represent atomic propositions; the identity axiom is used implicitly in the form of an edge, one end being immediately connected to an agent. Cuts between atomic propositions do not appear in the interaction net. The example proof net reduction of section 2.3 can be done in Interaction Nets as follows:

```
# net Times(x,y,p),Times(p,q,r),Par(s,q,r),Times(s,v,u);

# show_net
Times(x: `b T `a  T `c  ,y: `c^ ,v1),Times(v1,w,x1),Par(y1,w,x1),Times(y1,v: `a^
 ,u: `b^ )
```

```
# reduce
Times(x: `b T `a  T `c  ,y: `c^ ,v1),Times(v1,v: `a^ ,u: `b^ )
```

The type variables `a and `b have become reversed with respect to the propositions A and B, as a result of which one happens to be named first when the net is printed.

It is actually possible to represent proof nets in Interaction Nets in such a way that cuts between atomic propositions *can* be seen. A symbol definition

```
symbol          Id : `a , `a^ ;
```

is needed, and an interaction rule

```
rule            Id(y, x), Id(y, z) -> y-z ;
```

With these definitions, the introduction of A and $A^{\perp}$ is represented by the fragment of net Id(x, y), Id(z, y) where x is the A end (with type `a) and z is the $A^{\perp}$ end.(with type `a^) Cuts between A and $A^{\perp}$ then have to be eliminated using the interaction rule for Id. This scheme corresponds to the usual cut elimination process; doing without the cuts between atomic propositions implements the fast cut elimination mechanism described in [2]. 'Slow' cut elimination on the same proof net takes three steps, as mentioned before:

```
# net Times(x,b,a),Id(b,c),Id(y,c),Times(a,e,d),Id(e,g),Id(d,f),Id(h,f),
Id(i,g),Par(j,i,h),Times(j,l,k),Id(l,m),Id(k,n),Id(v,m),Id(u,n);

# show_net
Times(x: `a^ T `b^  T `c^  ,v1,w),Id(v1,x1),Id(y: `c ,x1),Times(w,y1,z),Id(y1,u1),
Id(z,v2),Id(w1,v2),Id(x2,u1),Par(y2,x2,w1),Times(y2,z1,u2),Id(z1,v3),Id(u2,w2),
Id(v: `b ,v3),Id(u: `a ,w2)

# reduce1
Times(x: `a^ T `b^  T `c^  ,v1,w),Id(v1,x1),Id(y: `c ,x1),Times(w,y1,z),Id(y1,u1),
Id(z,v2),Id(w1,v2),Id(x2,u1),Id(x2,y2),Id(w1,z1),Id(v: `b ,y2),Id(u: `a ,z1)

# reduce1
Times(x: `a^ T `b^  T `c^  ,v1,w),Id(v1,x1),Id(y: `c ,x1),Times(w,y1,z),Id(y1,u1),
Id(z,v2),Id(w1,u1),Id(w1,x2),Id(v: `b ,x2),Id(u: `a ,v2)

# reduce1
Times(x: `a^ T `b^  T `c^  ,v1,w),Id(v1,x1),Id(y: `c ,x1),Times(w,y1,z),Id(y1,u1),
Id(z,v2),Id(v: `b ,u1),Id(u: `a ,v2)
```

(this textual output is rather opaque, and is included to demonstrate that proof net reductions have actually been carried out by the system).

Polymorphic typing is another good reason for not requiring interaction rules for all possible interacting pairs of agents. Once Erase is polymorphic, it can potentially interact with (for example) the principal input of Add; maybe Erase interacting with inputs can be interpreted as an exception which propagates itself upwards, but what about Dupl? Most of the time, the programmer probably does not want to worry about such things. Also, checking the completeness of interaction rules in the polymorphic case is more difficult!

# 4 The System

The system provides an environment for experimenting with Interaction Nets programming. It allows programs to be written and executed - that is, it accepts type, symbol and rule definitions, allows nets to be entered, and will reduce a given net to normal form by applying the given interaction rules. The system checks typing, linearity and partitioning as it processes definitions. It also allows nets to be reduced step by step and examined at each stage, as an alternative to carrying out the whole sequence of reductions without stopping. Further facilities allow programs to be written to and read from files. A complete description of the user interface appears in an Appendix.

As well as implementing the basic Interaction Nets language, the system implements certain additional language features as described previously. The built-in definitions are optional, and can be turned on or off when the system is started up: entering the system with the command ‘inets’ includes the built-ins, while ‘inets -pure’ leaves them out.

## 4.1 Implementation of the basic system

The system is implemented in Modula-3, running in a Unix environment. It was decided to use an imperative language rather than a functional or declarative language, because the fundamental operation of carrying out a reduction by applying an interaction rule to a net is naturally viewed as updating a global state, and is most easily programmed in an imperative style. Modula-3 was felt to be the most suitable of the available imperative languages, as it is higher-level than C and provides useful constructions which Modula-2 omits, such as exception handling. Unix was the only available environment in which to use Modula-3, and also its wide range of program development tools gave it an advantage over Phoenix. In retrospect, these were the right decisions.

The actual program is structured as a number of separate modules, in order to divide it into sections providing different types of function and also to take advantage of the possibilities for separate compilation. By using the ‘make’ system of Unix, it is possible to ensure that only the minimum amount of recompilation is done when a change is made to the program.

A recursive descent parser is used to process type, symbol and rule definitions. This method was chosen because it is easy to implement and, since it detects syntax errors as soon as possible, can be made to give clear error messages. It was also straightforward to extend the parser to handle new syntax.

Nets are represented by structures of cells and pointers which reflect the way in which their agents are connected together. A reduction step involves copying the net on the right hand side of a rule and then moving a lot of pointers to build the copy into the existing net. Active links are stored on a stack.

It was mentioned previously that reducing a net need not introduce any garbage. However, Modula-3 includes a garbage collector, so the present implementation does not bother to explicitly free storage, letting the runtime system clean up instead.

The top level of the system is a very simple command interpreter. Its only interesting feature is that it can call itself recursively to read definitions and commands from a file.

## 4.2 Implementation of pseudo-agents

Expanding a pseudo-agent is essentially the same as applying an interaction rule; the only difference is that just one agent is replaced by a section of net, rather than an interacting pair of agents. The code for applying a rule was copied and modified to produce the code for expanding a pseudo-agent. It would probably have been possible

to produce a single procedure for both functions, but it was felt that this would make the programming unnecessarily difficult.

### 4.3 Implementation of built-in definitions

The prototype definitions are stored in the same way as any other definitions. But their representations are modified to cause the required special behaviour. All agents can contain a piece of extra information; for most agents there is none, but an AnInt agent, for example, holds the actual integer which it represents.

When an agent is read or printed, names must be allowed which reflect the internal data specific to that agent. To achieve this, a symbol can have a read-name function and a print-name function. The print-name function maps internal data to a textual name, and if present is used when printing an agent with that symbol definition. A read-name function maps in the opposite direction. When an agent is being read, and a name is encountered which is not the name of any symbol, the read-name functions are used in turn to attempt to interpret the name as a valid one for their symbol. If one of them succeeds, the appropriate agent is produced, with internal data given by the read-name function.

Finally, internal data of agents needs to be transferred across an interaction rule. To do this, a rule can have a list of functions which the internal data of the interacting agents to the internal data of each agent on its right hand side. These functions may cause side-effects, for example in the case of the built-in agents for input and output.

### 4.4 Implementation of polymorphism

The parser was modified to allow declaration of type constructors as well as atomic types, and reading of type structures. Port type specifications in symbol definitions have been replaced by tree structures, and the very simple test for matching of ports previously used was replaced by a type unification algorithm. This works by matching trees, instantiating type variables where necessary and keeping track of multiple instances of type variables. Instantiating a type variable is done by copying a pointer to the new value (when making the variable equal to an existing type) or building a new tree (when making the variable dual to an existing type). Free variables of nets are decorated with type information, because when type-checking while adding an agent, the type of a free variable can no longer be taken from a symbol definition (because it may have been instantiated further by the way in which the net has been connected up). This also allows the types of a net’s free variables to shown when the net is displayed.

A small point about syntax ought to be mentioned here. To make life easier for the parser, multiple symbol definitions in one ‘symbol’ command must be separated by full stops, in the polymorphic version of the system. This is not entirely satisfactory, but syntactic details are less important than the language extensions; and since the syntax of port types is also different, this slight quirk does not make moving programs between versions of the language significantly more troublesome.

# 5 Programming in Interaction Nets

In the example programs which follow, the port types of agents are not always specified before they are used in interaction rules; they can usually be deduced from the rules. The important part of a program is the collection of rules, which are always shown. The first three examples are for the simply typed version of Interaction Nets; the final one is polymorphic.

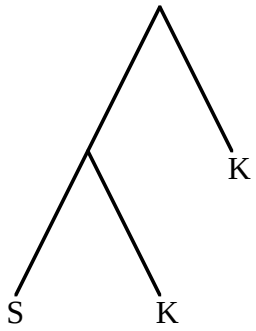
## 5.1 Combinator reduction

SK-combinators are a well-known alternative to λ-calculus as a notation for function application. They are described and motivated in [6] and [7], and have the advantage of doing without variables. The combinators are defined by

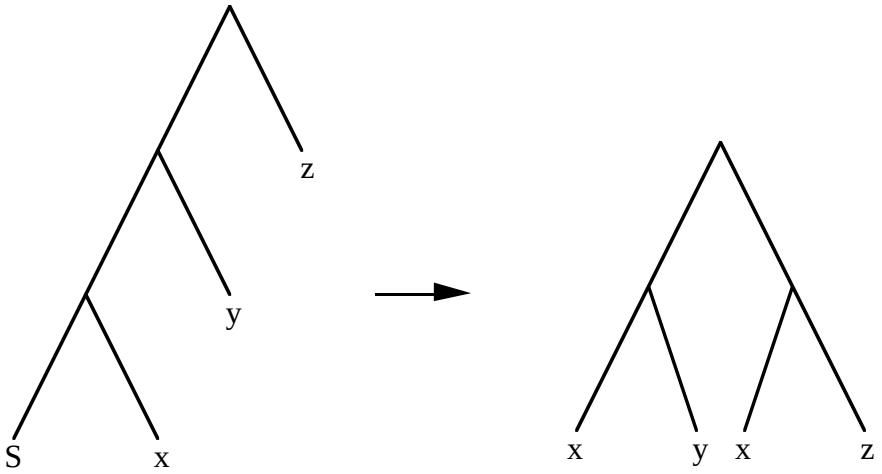
$$K = \lambda xy.x \ , \ S = \lambda xyz.x\ y\ (x\ z) \ .$$

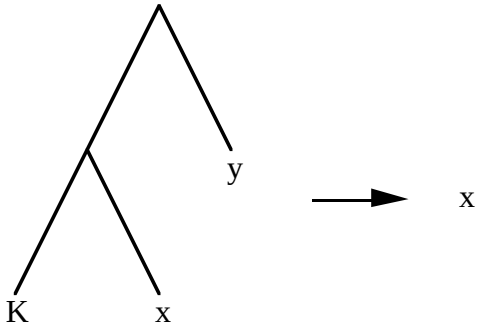
Since evaluation of combinator expressions can conveniently be carried out by means of graph rewriting rules, it is plausible that evaluation of combinator expressions in Interaction Nets should be fairly straightforward. This does indeed turn out to be the case; the infrastructure needed to carry out graph rewrites according to a given set of rules is present in the Interaction Nets interpreter, and it is only necessary to define appropriate symbols and rules.

A combinator expression can be represented naturally as a binary tree, with S s and K s at the leaves; for example the expression S K K (well known to equal I, the identity) takes the form:

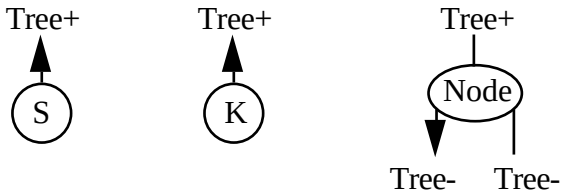



There are two rewriting rules, one for S and one for K :

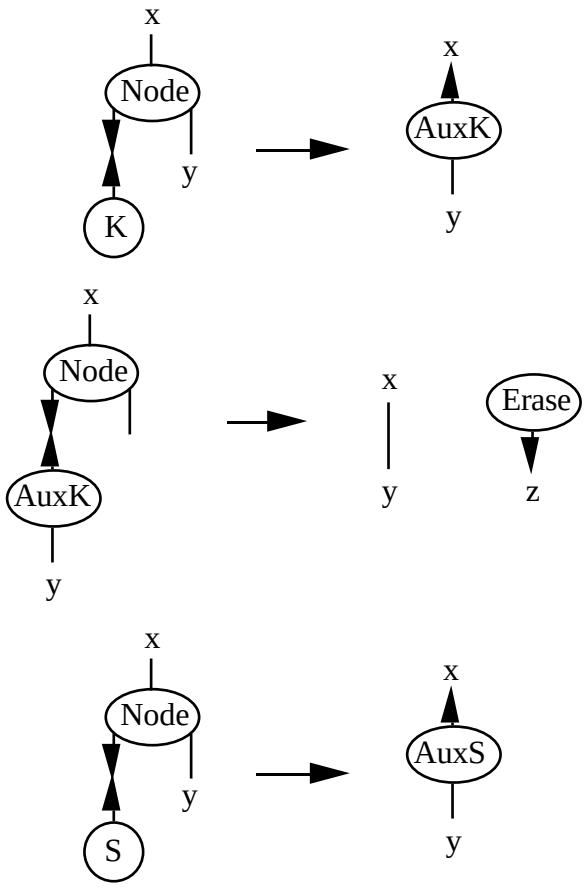


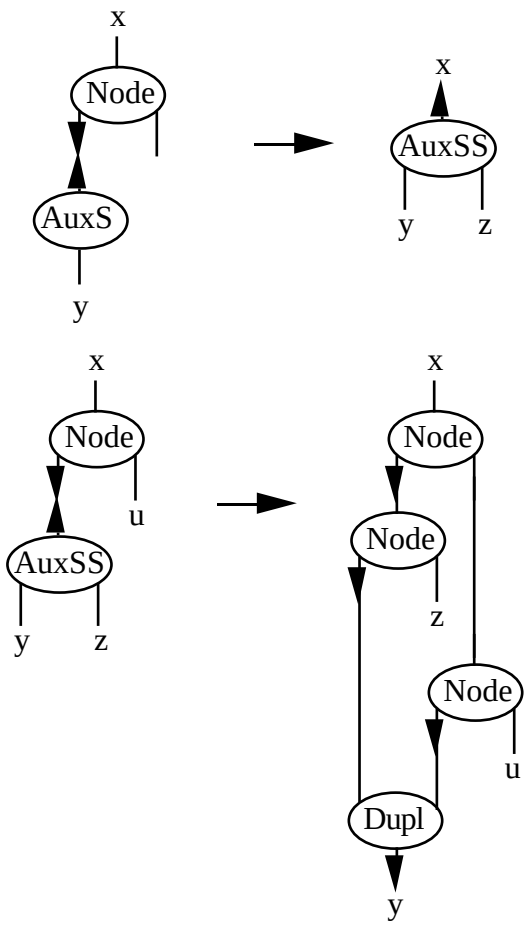



To translate all this into Interaction Nets, we define agents corresponding to S, K and the intermediate nodes:




and specify how S and K leaves interact with intermediate nodes. Auxiliary agents are needed since the original tree rewriting rules are not local (in the sense of Interaction Nets rules), and some erasing and duplication is needed because of the non-linear nature of S and K : S uses an argument twice, and K discards one. The appropriate interaction rules are as follows (the rules for erasing and duplication are not shown, as they are uninteresting):






Note that when a combinator expression represented in terms of these agents has been evaluated as far as possible, there may be some auxiliary agents in the final net; these can be interpreted as partially evaluated combinators of the appropriate variety.

This representation of combinator reduction inside Interaction Nets constitutes a demonstration that Interaction Nets have full computational power. It is well known that any computation can be specified as a combinator expression to be evaluated, so clearly any computation can be expressed in Interaction Nets via a translation into SK combinators. Of course, this is unlikely to be the most efficient way of doing the computation with Interaction Nets.

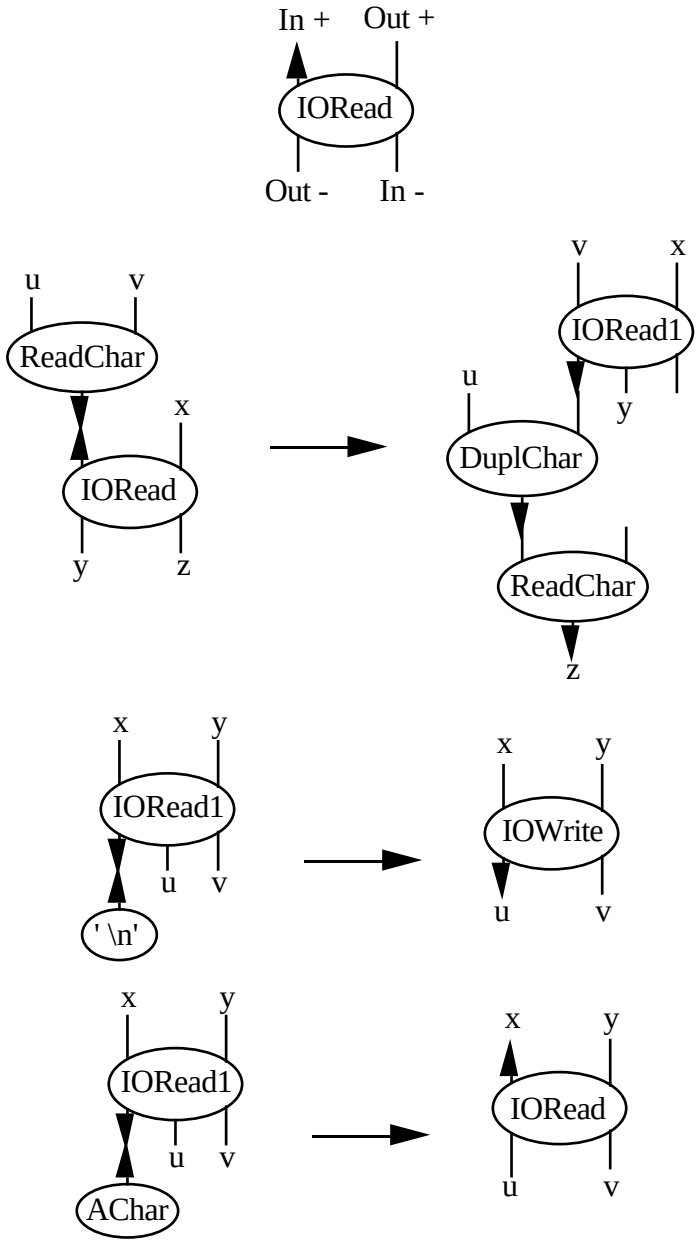
This means that we also have a way of compiling traditional functional programming languages into Interaction Nets, via compilation from the functional language to combinators (which is a standard technique). The use of additional combinators such as I, B and E (as described in [7]) is easily accomodated, as is the use of supercombinators optimised for the particular program being compiled. However, it should be possible to produce much better translations of functional programs into Interaction Nets than can be obtained by these methods.

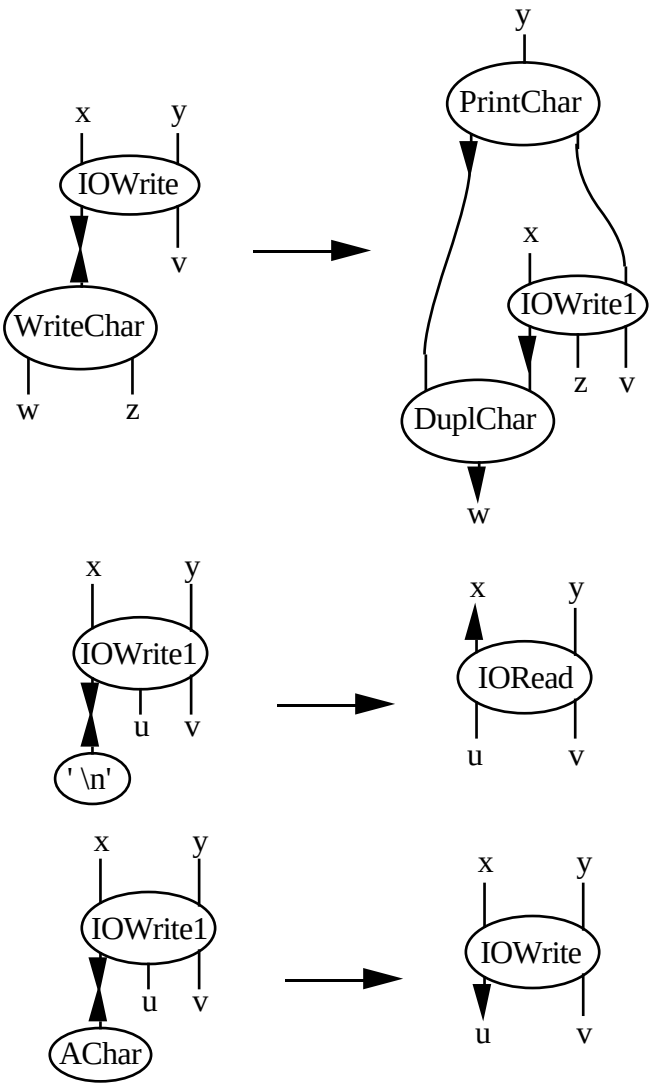
### 5.2 An interactive interpreter for combinators

A program has been written which implements a read-eval-print loop for combinatory expressions. It parses input consisting of S's, K's, ('s and )'s, and builds a tree representing a combinatory expression, as in 5.2 except that the nodes do not immediately evaluate the tree (their principal ports are outputs). When a complete line has been read, the tree is evaluated by converting the nodes into evaluating nodes with input principal ports, and the result is printed.

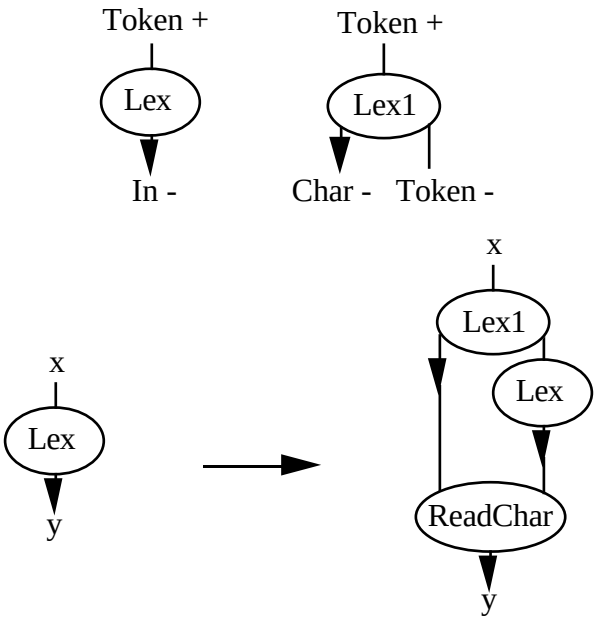
In order to ensure that input and output alternate in units of a line, access to the InStream and OutStream is controlled by an interlock unit, which has two ports

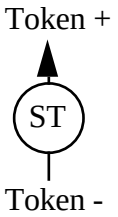
connected to the actual InStream and OutStream and two ports which look like an InStream and an OutStream to other agents. Only one of these ‘interface’ ports will be principal, and this changes when a newline character is read or printed. The interlock unit is an IORead, IORead1, IOWrite or IOWrite1 agent depending on its state. This is made to work by the following definitions (note that printing a character is done by the agent WriteChar, which then uses the built-in PrintChar):



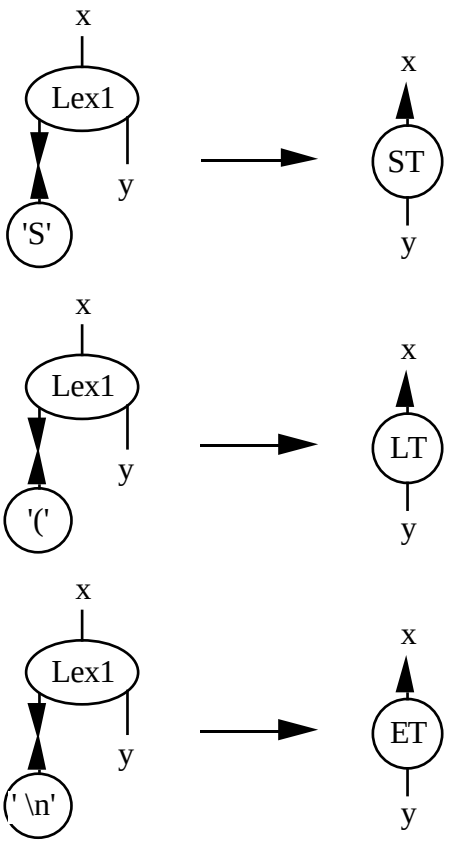



The stream of characters produced by ReadChar is lexically analysed into a stream of tokens representing S, K, [, ] or end-of-line. The agent Lex (actually a pseudo-agent) acts as a filter:



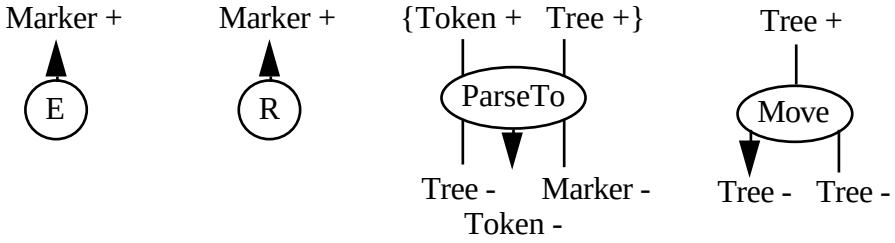


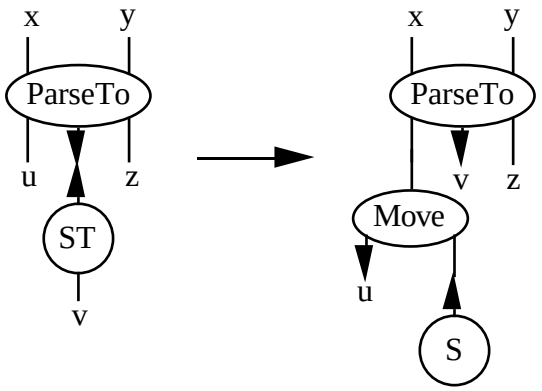

(ST for S Token; similarly KT, LT, RT, ET)



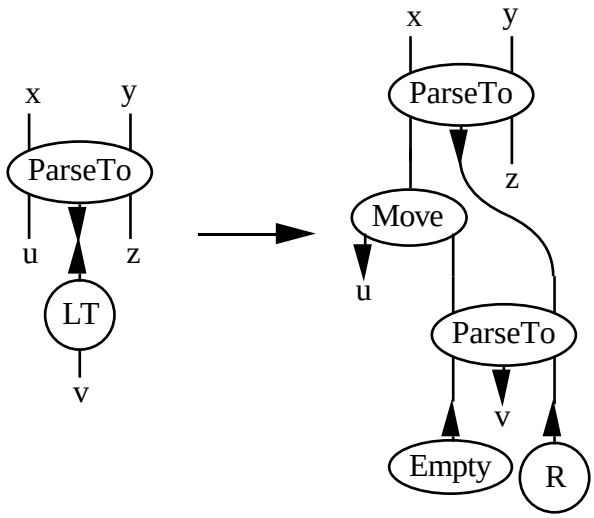

(similarly rules for K , ) )

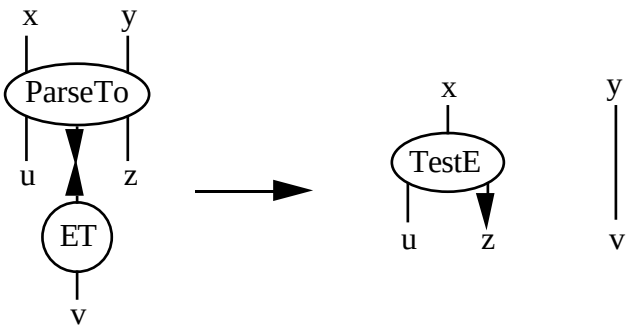
A stream of tokens is parsed into a tree structure by the agent ParseTo, which parses a stream of tokens up to either the next end-of-line or the next close bracket, given a tree structure already built up, and outputs a tree structure and the rest of the stream of tokens. A syntax error is signalled by producing a tree consisting of an Error agent. The agent Move is used to add new leaves to a combinator tree. The definitions are:

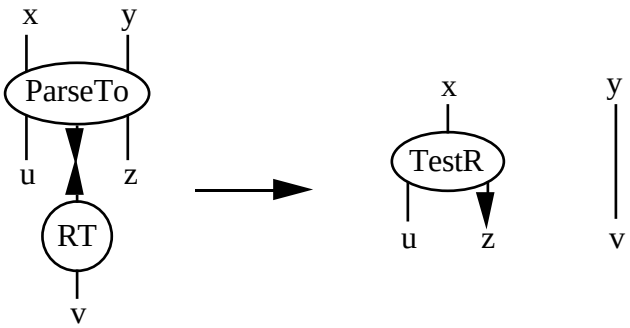


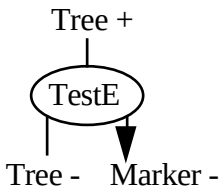


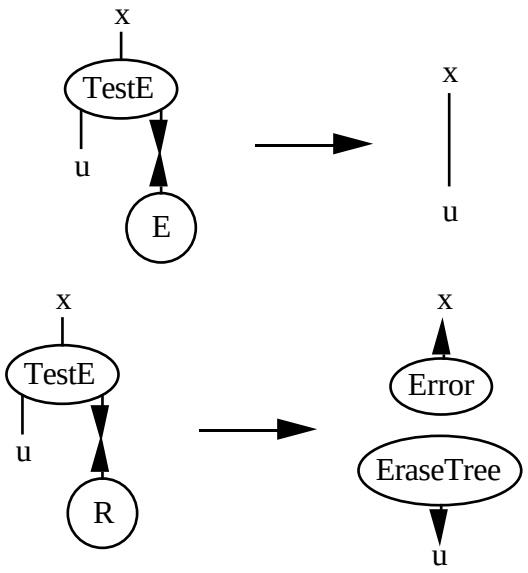

and the same for K



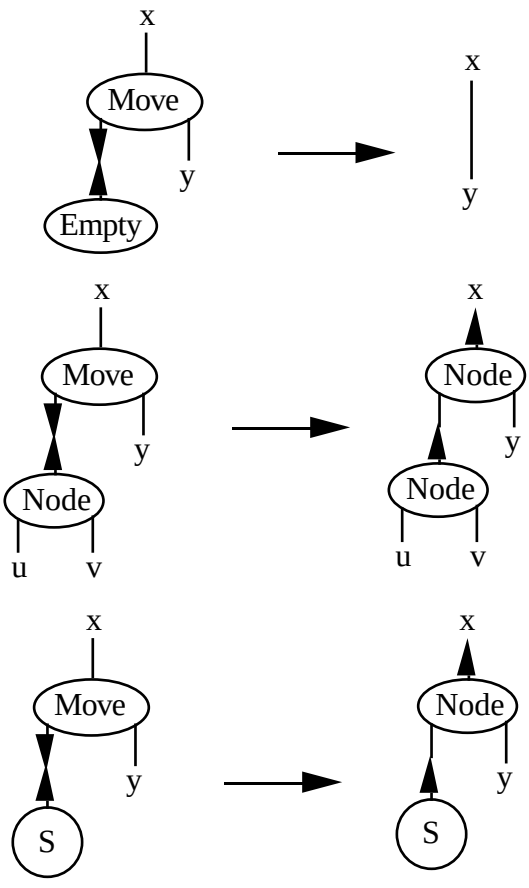



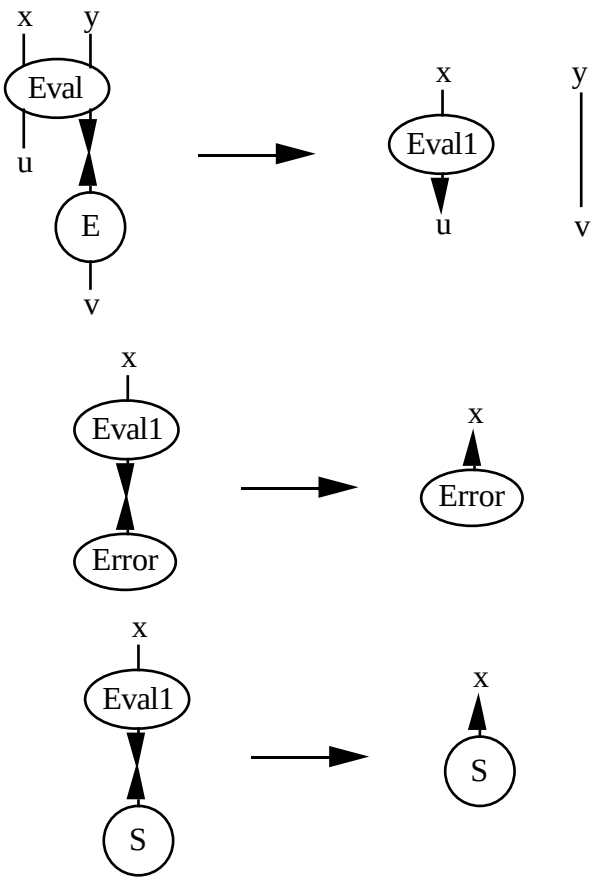



and similarly TestR

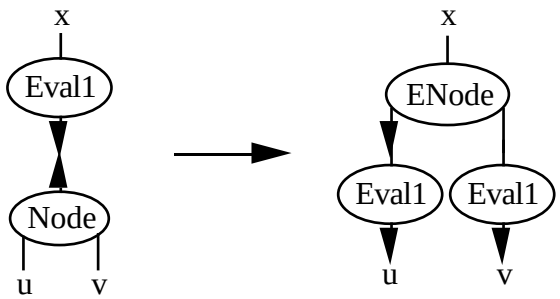



and the same for K

Evaluating a combinator tree consists of moving down it, replacing Node agents by ENode agents; an ENode agent interacts with S's and K's in the way described in 5.1. An Error tree must evaluate to an Error which can be picked up when the result is printed. Eval's inputs are connected to the stream of tokens and the tree built by ParseTo. The rules are:



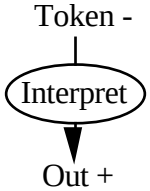

and the same for K

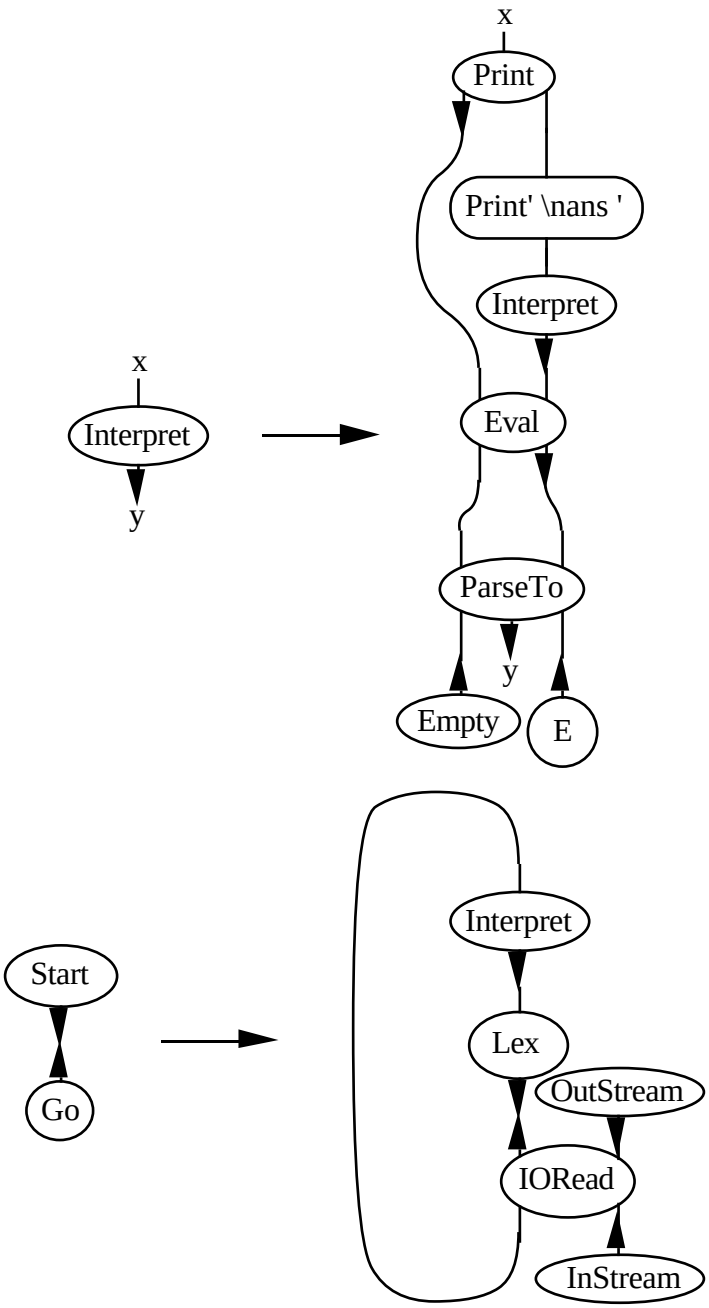



Depending on the order of reduction of active links, the ENodes may start evaluating the combinator tree before Eval1 has finished working on it. With the current use of a stack for active links, this is precisely what will happen.

A tree is printed by absorbing it with side-effects, remembering to print appropriate characters for partially evaluated combinators, and taking account of the fact that brackets may be necessary. The rules are omitted here as they are not very interesting.

Finally the lexical analyser, parser, evaluator and printer are tied up into a convenient top-level interface:






(the ‘agent’ labelled Print’\n ans’ stands for a sequence of WriteChar agents which together print a newline and ‘ans’) so that reducing the net

Start(x), Go(x)

enters the loop. An sample run follows (the reduced combinatory expressions are preceded by ‘ans’).

```
$ inets
Interaction Nets

# load ../nets/combinators

# net Start(x),Go(x);

# reduce
()
```

```
SKK
ans SKK
SKK(SKS)
ans SKS
SK(SKKK)SK
ans SK
```

The reason for the initial output of () is that the input stream picks up a newline as the first character read, which is interpreted as signifying an empty tree, evaluated and printed as () before the first 'ans' is printed. This glitch could be corrected somehow, but it hardly seems worthwhile for such a simple example.

It should be noted that the IORead agent used in the interpreter needs a symbol definition

```
symbol          IORead : {In + , Out -} , In - , Out + ;
```

in which the principal port is included in a partition, a situation not allowed in the original description of Interaction Nets. Indeed, with such a definition, deadlocked nets can be constructed, although in the combinator interpreter this does not happen because of the way the rules work. Logically speaking, including principal ports in partitions corresponds to using a rule

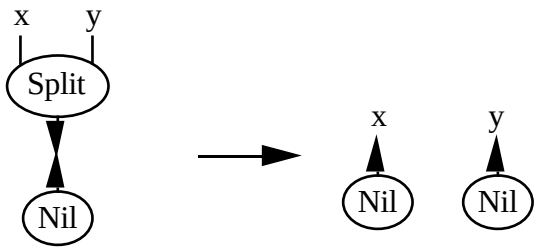
$$\frac{\vdash \Gamma, A, A^{\perp}}{\vdash \Gamma}$$

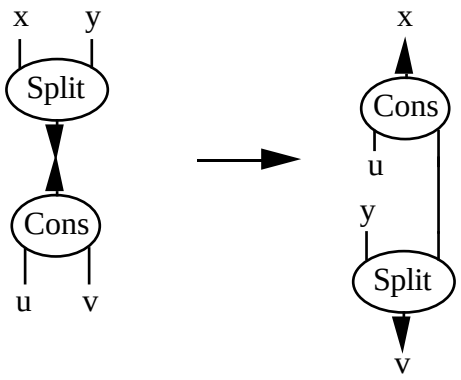
(a cut within a single context), which is not a rule of Sequent Calculus or Linear Sequent Calculus. So it is right to disallow this type of construction, and the correct approach to making input and output alternate neatly should be to modify the behaviour of the built-in agents appropriately.

SK-combinators being an extremely simple language, the lexical analyser in the above program is trivial, and the parser is almost trivial. However, the program contains the components of any interpreter, and by extending the sections appropriately it should be possible to interpret a more complex language, such as λ-calculus.

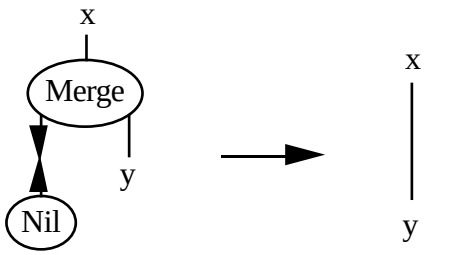
### 5.3 Mergesort

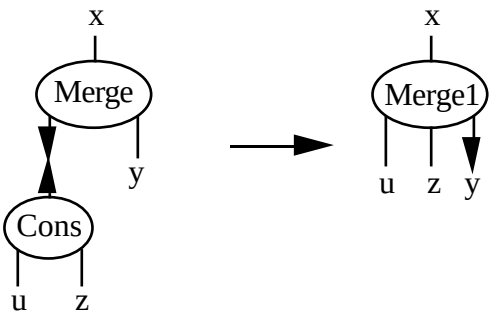
As an example of list programming in Interaction Nets, an implementation of mergesort follows. It has been written with the built-in integers, and comparison LessThan, in mind, but would work with a LessThan written for successor arithmetic. Mergesort consists of two components - splitting a list into halves, and merging two sorted lists into a single sorted list. For ease of coding, splitting is done by taking elements from the front of a list and putting them alternately into two lists. It is more usual to split a list into its first and second halves, but since the initial list is unordered it doesn't matter. The rules for splitting are

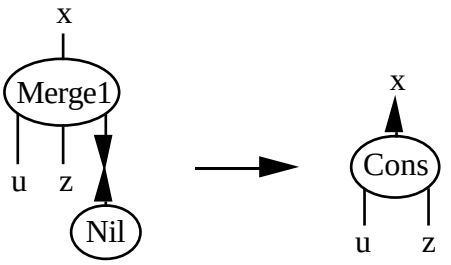


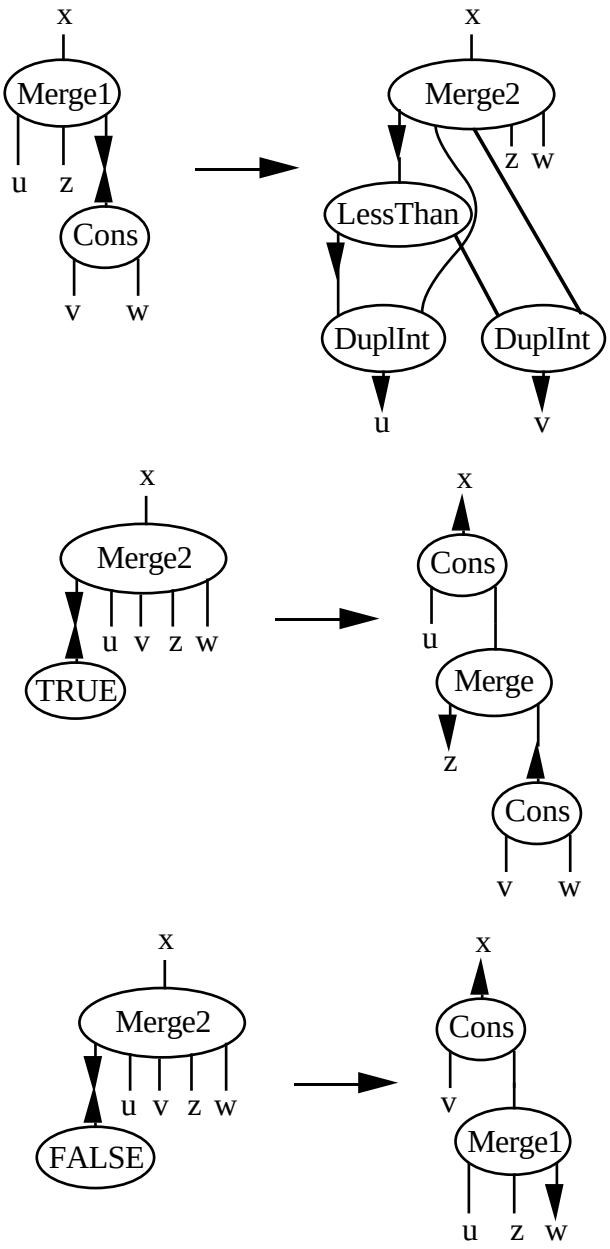



Merging sorted lists illustrates a feature of programming in Interaction Nets, which is to do with functions of several variables. An agent can only interact with one neighbour, so if it needs to see several inputs, it must interact with one of them and become an auxiliary agent depending on the interaction. The auxiliary agent can then interact with the next input, and so on. This is reminiscent of currying in traditional functional lanugages. For merging, the agent Merge tests whether its first input is Nil; if it is not, then the first element of the list is passed to Merge1. Merge1 test whether the second input to the Merge was Nil, and if not passes the first elements of the two input lists, and the result of comparing them, to Merge2. Merge2 selects the smaller element, and uses either Merge or Merge1 to handle the rest of the lists. The definitions are

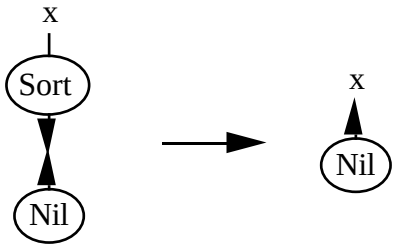


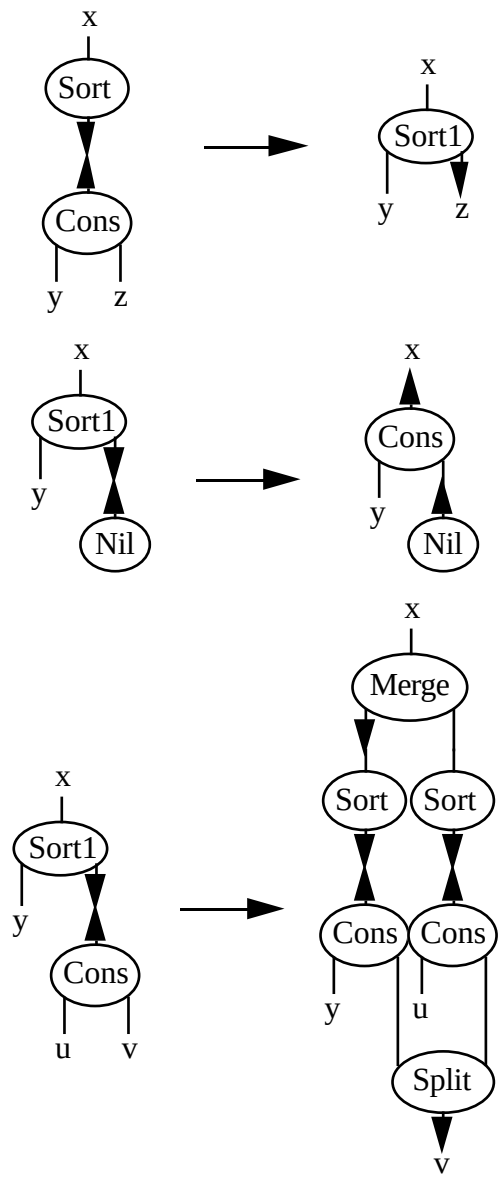






Finally, splitting and merging are combined to produce a sorting algorithm. The empty list and singleton lists need to be handled separately:



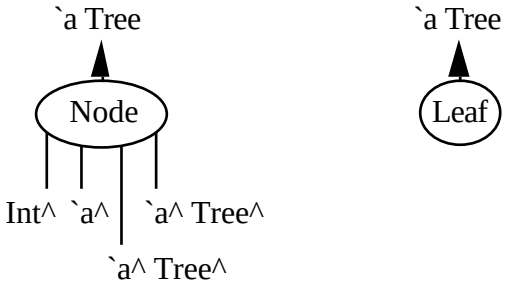



The last rule can be optimised if desired by reducing the Sort-Cons links. An example of mergesort in use follows.

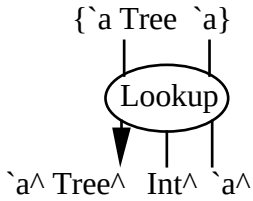
```
$ inets
Interaction Nets

# load ../nets/mergesort

# net Sort(y,x),Cons(y,a,b),4(a),Cons(b,c,d),2(c),Cons(d,e,f),1(e),
Cons(f,g,h),3(g),Nil(h);

# reduce
Nil(x1),1(y),Cons(x,y,z),Cons(z,u,v),4(w),Cons(v,x2,y1),Cons(y1,w,x1),3(x2),2(u
)
```

### 5.4 Binary search trees

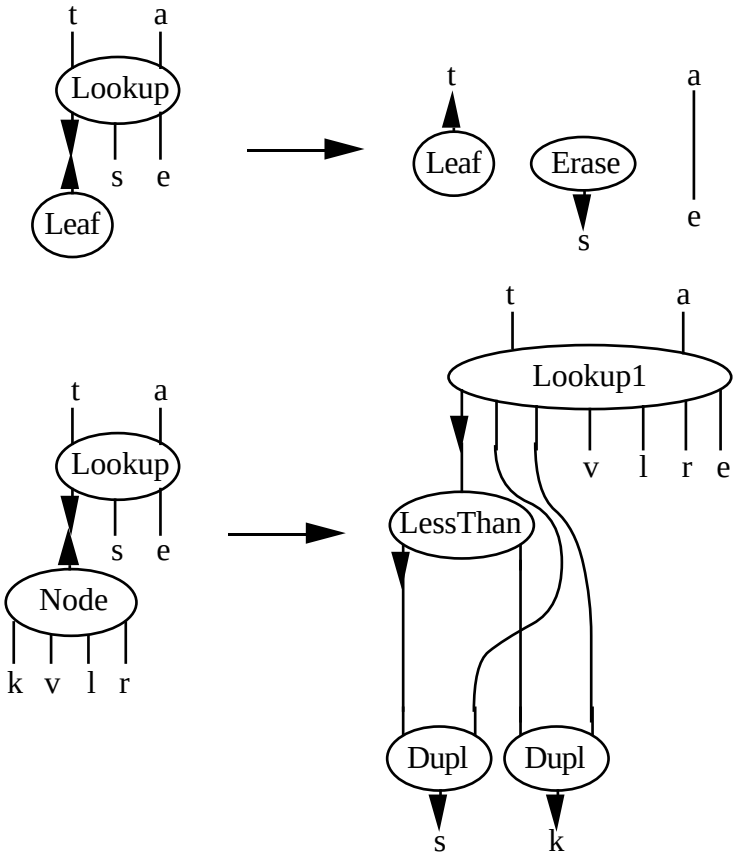
As a final example, consider the implementation of binary search trees, which may as well be made polymorphic. Using the agents

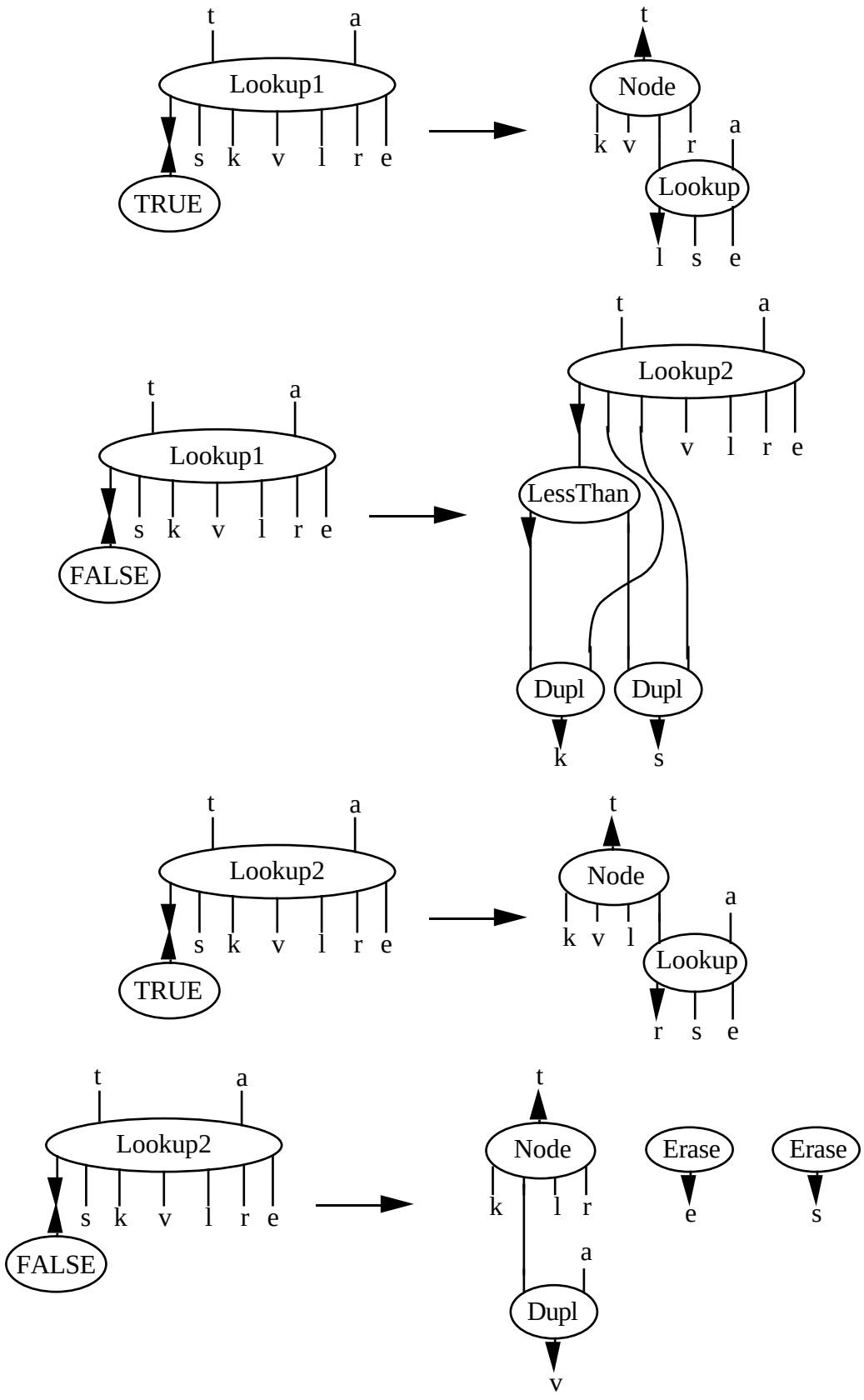



trees can be constructed, a node having an integer key, a value, and connections to two sub-trees. To look up a value by its key, the agent




is used; its inputs are the tree to be searched, the search key, and a value to return if the key is not found in the tree, and its outputs are the original tree and the value found. To perform the look-up, auxiliary agents are needed. Since it is necessary to interact with a node in order to test its key, the auxiliary agents must hold onto the information which was attached to a node so that the tree can be built up again. The ordering of the trees is with smaller keys to the left.The rules are






The free variables' names indicate their meanings - search, key, value, left, right, error. Extending Erase to handle trees is a simple matter. Updating a tree is very similar to looking up: given a key and value, either a new node must be created to replace a leaf, or the value at an existing node must be replaced by the new value. The top-level agent Insert only returns the new tree and needs the value to be inserted instead of an error

value, and the behaviour when a leaf is encountered or the key is found must be changed, but the structure of the search itself is the same. The actual rules for insertion are omitted.

# 6 Future directions

## 6.1 Parallelism

Since interaction rules are local, there is potential for parallel reduction of active links, without interference. In the absence of side-effects, the order of reduction is not important; any agents causing side-effects would have to be carefully designed. A parallel implementation would automatically exploit the possibilities for parallelism in algorithms such as Mergesort. A major difficulty might be how to store a net: a processor which is reducing a link must have access to the agents involved, so the net should be distributed; but what happens when two processors each have half of an active link? Agents would probably have to move around a network of processors. And where should the interaction rules be stored? It seems wasteful for every processor to keep a copy of the entire set of rules. Maybe processors could be specialised, each knowing about a small set of rules, in which case active links would have to move to processors which could reduce them.

## 6.2 Communicating processes

It might be possible to view agents as processes which can communicate via pipes (the edges). An agent could then perform computations internally, but would also know (via the interaction rules) how to co-operate with other types of process. The rules of constructing interaction nets would eliminate the possibility of deadlock in such networks.

## 6.3 More logic

Interaction Nets generalise proof nets, which work for the multiplicative fragment of Linear Logic. There are hints in [2] of an extension of proof nets to include the additive connectives. It would be interesting to investigate whether such extensions on the logical side could be interpreted in terms of a programming language. Also, [4] deals with quantifiers and extends proof nets to include them. Maybe useful extensions to Interaction Nets could be developed by looking at how quantifiers fit into Linear Logic.

## 6.4 Further language extensions

Looking from the opposite direction, there are various things which it would be useful to be able to express in the language. It is not possible in the present language to define a general mapping function for lists - this would require a concept of the type of an agent (for example, the type of an S agent is Int,Int^) and the ability to use ‘agent variables’ in interaction rules. Also, although Erase (or Dupl) can be given polymorphic ports, interaction rules still have to be defined for every agent which can be erased. Since all rules for Erase have the same form (to erase a structure, erase the top agent and put Erase agents on all its free variables), it would be convenient if such an interaction rule schema could be specified directly.

Another desirable language feature is modularity, which has not been investigated in this project. One approach might be to allow a module to export certain agents and guarantee that they will interact with other agents (having specified properties) to produce some result. For example, a sorting module could export an agent Sort which works on lists of integers, while hiding the definitions which implement a particular sorting algorithm.

Ideally, the language extensions corresponding to extensions of the underlying logic would turn out to be precisely those desirable from a programming point of view!

### 6.5 Compiling into Interaction Nets

A pure functional language (ML, say) with no built-in types (apart from the unit type) but doing everything by defining type constructors and using pattern-matching in function definitions, could probably be compiled into Interaction Nets in a fairly straightforward way. The only subtlety is that duplication and deletion of function arguments must be detected and made explicit. Combined with a parallel implementation of Interaction Nets, this could be an effective route to parallel evaluation of traditional functional programs.

### 6.6 A graphical user interface

It would be nice to be able to define interaction rules and nets graphically, and also to be able to view a net graphically at intermediate stages of reduction. Such a graphical interface could be used to provide a facility whereby the user could select active links to be reduced. Adding a graphical interface could easily form the basis of a project in its own right, even with good tools such as an object-oriented graphics system. It was decided at a very early stage that the present project would avoid this area.

# 7 Conclusions

The aims of the project have been realised to a satisfying extent. As has been noted at various points, the implementation does not conform exactly to Lafont's description; for example, the distinction between simplicity and semi-simplicity has not been incorporated into the system, and empty partitions are not interpreted correctly. The reason for this is that a (reasonably) full understanding of the logical basis of Interaction Nets has only been gained at a fairly late stage of the project. A new implementation could start with a good theoretical understanding, and modifications (such as to the partitioning rules) would not be introduced without considering the logical implications.

One area mentioned in the Proposal which has not been looked at is the impact of explicit duplication and erasing on performance. The problem is that there is nothing to compare it with. But it seems likely that the cost of duplication and erasing is no greater than the cost of garbage collection.

On the theoretical side, the logical foundations of Interaction Nets have surely been thoroughly worked out before by Lafont, Girard and others, but do not seem to be laid out explicitly in the literature. The logical investigations have been a significant part of the project, leading as they do to a nice way of adding polymorphism.

Finally, I always hoped that my project would introduce me to an interesting research area; this did indeed turn out to be the case, and from that point of view I certainly regard the exercise as having been a success.

## References

[1] Y.Lafont, Interaction Nets, in *POPL'90*, 95-108

[2] Y.Lafont, Introduction to Linear Logic, Lecture notes for the Summer School on Constructive Logics and Category Theory (Isle of Thorns, August 1988)

[3] J.Y.Girard, Linear Logic, *Theoretical Computer Science* **50** (1987) 1-102

[4] J.Y.Girard, Quantifiers in Linear Logic, in *SILFS'87*

[5] J.Y.Girard, Y.Lafont, P.Taylor, *Proofs and Types*, Cambridge Tracts in Theoretical Computer Science **7** (Cambridge University Press, 1989)

[6] J.Lambek, P.J.Scott, *Introduction to higher-order categorical logic*, Cambridge Studies in Advanced Mathematics **7** (Cambridge Univresity Press, 1986)

[7] M.J.C.Gordon, *Programming Language Theory and its Implementation*, Prentice Hall International Series in Computer Science (Prentice Hall International, 1988)

# Appendix A - Syntax of Interaction Nets

## A.1 Nets

A textual notation is used to describe nets. A net is specified by describing its agents and the connections between them. This is done by listing all the agents with their ports represented by variables, each variable occurring at most twice in the description. If a variable appears only once, then the port it represents is attached to a free variable. If a variable appears twice, then the two ports it represents are connected together. In this notation, an agent is decribed by its symbol name followed by the list of variables representing its ports, in brackets. Two variables connected together by an edge are represented by the notation x - y . The empty net is represented by "@". This syntax is defined by:

<net definition> ::= <net> “;”

<net> ::= “@” | <nonempty net>

<nonempty net> ::= <net part>

| <net part> “,” <nonempty net>

<net part> ::= <agent> | <edge>

<edge> ::= <variable name> “-” <variable name>

<agent> ::= <symbol name> “(“ <variable list> “)”

<variable list> ::= <variable name>

| <variable name> “,” <variable list>

<variable name> ::= sequence of alphanumerics

A second level of syntax has been devised which allows many variables to be eliminated by nesting symbols. Basically, foo(x,bar(y,z,-),w) is shorthand for foo(x,u,w),bar(y,z,u). Because a net is not a tree with a single root, it is still necessary to use the variable-matching notation in some places. This level of syntax is defined by modifying the above definition as follows:

<blank variable> ::= “-”

<agent> ::= <symbol name> “(” <argument list> “)”

<argument list> ::= <argument>

| <argument> “,” <argument list>

<argument> ::= <variable name>

| <blank variable>

| <agent>

The second level of syntax is not implemented in the current system.

## A.2 Types

In the basic system, a type declaration need only introduce the type name. So types are declared by listing their names, separated by white space and terminated by a semicolon. A type name consists of a sequence of letters and underscores. We have the following syntax definition:

<type name> ::= sequence of letters and underscores

<type declaration> ::= <type list> “;”

<type list> ::= <type name>

| <type name> <white space> <type list>

In the polymorphic version, the definitions become

<type name> ::= sequence of letters and underscores

<type declaration> ::= <type list> “;”

<type list> ::= <type constructor>

| <type constructor> <white space> <type list>

<type constructor> ::= <type name>

| <type name> “:” <arity>

<arity> ::= “1” | “2”

### A.3 Symbols

A symbol declaration consists of the name, and a list of ports with their types and directions, split into partitions. For example

Cons : list + , list - , list -

Nil : list + , {}

Dupl : list - , {list + , list + }

A symbol name consists of a sequence of characters, not including white space. We have the following definition:

<symbol definition> ::= <symbol list> “;”

<symbol list> ::= <symbol>

| <symbol> <white space> <symbol list>

<symbol> ::= <symbol name> “:” <partition list>

<port type> ::= <type name> (“+” | “-”)

<partition list> ::= <partition>

| <partition> “,” <partition list>

<partition> ::= <port type>

| “{” <port list> “}”

<port list> ::= empty

| <proper port list>

<proper port list> ::= <port type>

| <port type> “,” <proper port list>

<symbol name> ::= sequence of characters excluding white space

For the polymorphic version, the above needs to be modified as follows:

<symbol list> ::= <symbol>

| <symbol> “.” <symbol list>

<port type> ::= <type name> <negation>

| <port type> <constructor>

| <port type> <constructor> <port type>

| “(” <port type> “)”

<constructor> ::= <type name> <negation>

<negation> ::= “” | “^”

### A.4 Interaction rules

A rule consists of two nets, separated by -> . There may be matching of variables between the two halves of the rule; in fact, there often will be. But within each half of the rule, the syntax is the same as described above for nets. So the definition is:

<rule definition> ::= <rule list> “;”

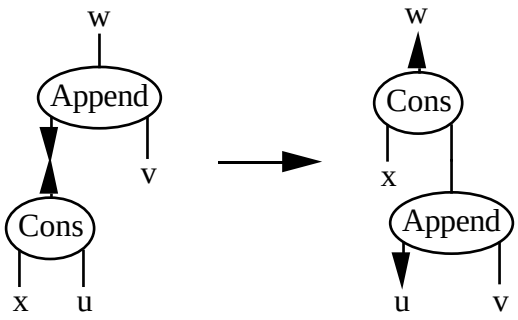
<rule list> ::= <rule>

| <rule> <white space> <rule list>

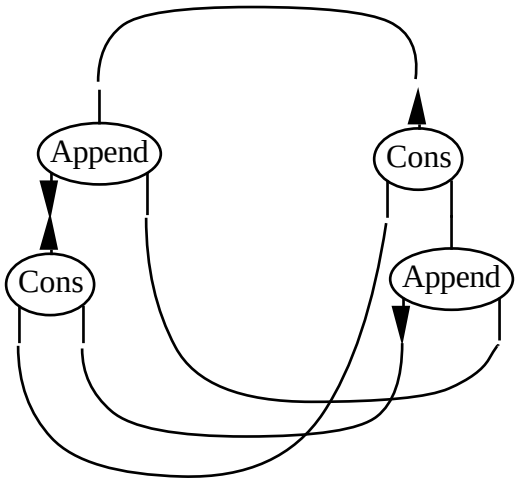
<rule> ::= <net> “->” <net>

### A.5 Lafont’s Rule Syntax

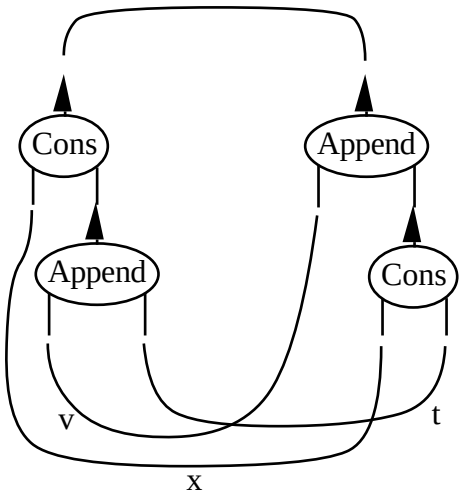
The following description of the originally proposed syntax for interaction rules is taken from [1]. Given a rule such as




free variables are joined between the two sides:




Then the graph is twisted around so that principal ports point up and auxiliary ports point down:




to give two trees with (labelled) links between leaves. The textual form of this rule is taken from the trees:

Cons[x, Append(v, t)] >< Append[v, Cons(x, t)]

The symbol >< represents two principal ports meeting. This syntax is not easy to work with (in particular, the left and right sides of the >< have nothing to do with the two sides of the original rule), so although it does get easier to understand with practice it was decided to devise a more obvious syntax for the present project.

## Appendix B - The User Interface

The user interface to the Interaction Nets system consists of a command interpreter which allows types, symbols and interaction rules to be defined, and nets to be reduced. There are also one or two diagnostic commands to assist with debugging. The prompt is a #. The commands currently supported, and their syntax, are as follows.

**Command** **Description**

*type* Declares types. The form of the command is the word 'type' folllowed by a space, followed by a number of type names, separated by spaces and terminated by a newline. The types thus declared are then available for defining symbols. If a type name is declared which is already known, then the declaration has no effect. There can never be duplicates in the list of types.

*list_types* Outputs the word 'type' followed by a list of the currently known types, separated by newlines and terminated by a semicolon.

*symbol* Defines symbols. The form of the command is the word 'symbol' folllowed by a number of symbol definitions, separated by spaces (in the polymorphic version, separated by full stops) and terminated by a newline. If a symbol is redefined, then all rules are deleted, since they may depend on the previous symbol definition.

*list_symbols* Outputs the word 'symbol' followed by a list of the currently known symbol definitions, separated by newlines (and full stops, in the polymorphic version) and terminated by a semicolon.

*rule* Defines rules. The form of the command is the word 'rule' followed by a number of rule definitions, separated by spaces and terminated by a newline. A rule definition consists of two net descriptions with an arrow ('->') between them. If a rule is redefined, the previous definition is overwritten.

*list_rules* Outputs the word 'rule' followed by a list of the currently known rule definitions, separated by newlines and terminated by a semicolon.

*clear* Removes all the current definitions.

*clear_symbols* Removes all the current symbol and rule definitions.

*clear_rules* Removes all the current rule definitions.

*net* Defines the 'current net' which can then be operated on. The form of the command is the word 'net' followed by a net description and a semicolon.

*show_net* Outputs the current net.

*dump_net* As an aid to debugging, this command outputs information about the internal representation of the current net.

*copy_net* Another debugging aid. This command builds a copy of the current net and displays it.

*reduce* Uses the currently known interaction rules to reduce the current net until it has no more active links. The final net is then displayed, and becomes the current net.

*reduce1* Does a single reduction step on the current net.

*load* Reads commands from a file. The form of this command is the word 'load' followed by a Unix filename. Commands are read from the file and executed as if they are being typed at the terminal. Error messages appear on the terminal.

*save* Outputs the current definitions to a file. The form of this command is the word 'save' followed by a Unix filename. The file is created if necessary; if it already exists, it will be overwritten. The current type, symbol and rule definitions are output to the file in such a way that they can be read back in using *load*, ie the same format is used as for *list_types, list_symbols* and *list_rules*.

*quit, finish, exit, bye* These commands all have exactly the same effect as a Control-D.

Note that the commands *list_types, list_symbols* and *list_rules* all produce their output in a form which can be read directly back in as type, symbol and rule definitions. The commands *type, symbol ,rule* and *net* all detect errors, and respond to them by displaying the line in which the error occurred, and attempting to indicate the region of the line which caused the error; this attempt is only successful in the case of errors which are detected while the line is being read, rather than afterwards. For example, syntax errors are pinpointed reliably, but if a rule is invalid because its two sides do not have the same free variables, then the error will be marked as occurring at the end of the line. No attempt is made to recover from errors. At an error, all the objects being defined in the current command, which have been completely defined by the time the error is detected, will be successfully defined. The object in which the error occurs is ignored, as are all subsequent definitions in the same command; in fact, any subsequent definitions in the same command are not even parsed.

An invocation of the user interface is terminated when it reads an end-of-file. So if it is receiving input from the terminal, a Ctrl-D will end the session.

## Appendix C - A Sample of the Program

The complete program consists of about 4500 lines, divided into several modules and their interfaces. Extracts from the module Nets, which is responsible for building and reducing nets, and its interface, are shown below. The interface is complete, but in the implementation, several uninteresting minor procedures and one or two major ones have been omitted. The procedures which reduce a net, and build an interaction rule from a pair of nets, are two major components which have been left in.

```
INTERFACE Nets;

IMPORT Symbols, Tables, List;

TYPE
  Rule <: REFANY;

  PortConn = RECORD
                target : REFANY; (* REF Item or Variable. *)
                port : REF PortConn; (* NIL if target is a variable. *)
                tname : TEXT; (* For use when displaying and copying nets. *)
              END;

  Component = List.T;

  Variable = Tables.E OBJECT c : Component; pc : PortConn;
               type : Symbols.TypeTree END;

  VarWithType = Tables.E OBJECT type : Symbols.TypeTree END;

  Agent = RECORD
            symbol : Symbols.SymDef;
            conns : List.T; (* List of PortConns. *)
            data : REFANY; (* Points to a record containing extra info. *)
            back : List.T; (* For double linking. *)
          END;

  Edge = RECORD
           p1, p2 : PortConn;
           back : List.T; (* For double linking. *)
         END;

  Item = RECORD
           agent : BOOLEAN;
           item : REFANY; (* Agent or Edge. *)
         END;

  ActiveLink = RECORD
                 a1, a2 : REF Agent;
               END;

  Net = REF RECORD
          vars : Tables.T;
          items : List.T; (* List of items. *)
          links : List.T; (* List  of active links. *)
          typevars : Tables.T;
```

```
        END;

  DataFn = PROCEDURE (first, second : REFANY) : REFANY;
    (* The type of functions for extra calculation in built-in rules. *)

  DataFnPack = RECORD
                 theFn : DataFn;
               END;

PROCEDURE ProtectRule(VAR rule : Rule);
(* Marks the given rule as protected. *)

PROCEDURE Protected(rule : Rule) : BOOLEAN;
(* Finds out whether rule is protected. *)

PROCEDURE ClearRules(VAR theRules : List.T; theSymbols : Tables.T);
(* Removes unprotected rules from the list, and reverts status of unprotected
symbols to Symbols.AgentType.Undef *)

PROCEDURE CreateNet() : Net;
(* Return an empty net. *)

PROCEDURE GetVars(net : Net) : Tables.T;
(* Returns the variables of the net. *)

PROCEDURE AddAgent(s : Symbols.SymDef; vars : List.T; VAR context : Tables.T;
          data : REFANY;
           VAR n : Net; checkParts : BOOLEAN) : BOOLEAN;
(* Adds a new agent, an instance of the symbol s with free variables given by
vars, to the net n.  Fails if the partitioning rules are violated. The
partitioning is only enforced if checkParts is true. *)

PROCEDURE AddEdge(v1, v2 : VarWithType; VAR n : Net);
(* Adds a new edge, with free variables v1 and v2, to the net n. *)

PROCEDURE Empty(n : Net) : BOOLEAN;
(* Tests for the empty net. *)

PROCEDURE Normal(n : Net) : BOOLEAN;
(* Tests whether the net is in normal form. *)

PROCEDURE BuildRule(left, right : Net) : Rule;
(* Puts two nets together into a rule, carrying out validity checks. Returns
NIL if the two nets do not form the two sides of a valid rule. *)

PROCEDURE AddFunction(f : DataFn; VAR rule : Rule);
(* Adds a function to the end of the list attached to the given rule. *)

PROCEDURE LeftSide(rule : Rule) : Net;
(* Extracts the left hand side of the rule.*)

PROCEDURE RightSide(rule : Rule) : Net;
(* Extracts the right hand side of the rule.*)

PROCEDURE FirstAgent(n : Net) : REF Agent;
(* Returns a pointer to the first agent of a net which is the lhs of a rule. *)

PROCEDURE SecondAgent(n : Net) : REF Agent;
```

```
(* Returns a pointer to the 2nd agent of a net which is the lhs of a rule. *)

PROCEDURE FindExpansion(refag : REF Agent; rules : List.T) : Rule;
(* Finds a pseudo-agent expansion for the given agent. *)

PROCEDURE FindRule(link : ActiveLink; rules : List.T;
                   VAR switch : BOOLEAN) : Rule;
(* Finds a rule from the list which can be applied to link. Sets switch if the
   agents in the rule are the opposite way round to the link. *)

PROCEDURE CopyNet(VAR net : Net) : Net;
(* Makes a copy of the net.*)

PROCEDURE Reduce(VAR net : Net; rules : List.T);
(* Performs one reduction on the given net. *)

END Nets.
```

```
MODULE Nets;

IMPORT Builtins, Tables, Symbols, Errors, List, Names, Wr, Stdio, Text;

REVEAL

  Rule = BRANDED REF RECORD
                   left, right : Net;
                   functions : List.T;
                  protect : BOOLEAN := FALSE;
                   context : Tables.T;
                 END;

PROCEDURE CreateNet() : Net =
(* Return an empty net, which has no items. *)
BEGIN
  RETURN NEW(Net, vars := Tables.Create(),
           items := NEW(List.T, first := NIL, tail := NIL), links := NIL,
          typevars := Tables.Create());
END CreateNet;

PROCEDURE RemoveActiveLink(agent1, agent2 : REF Agent; VAR l : List.T) =
(* Removes the active link between the two agents from the list. It's a system
   error if the link is not on the list. *)
VAR
  al : REF ActiveLink;
  ag1, ag2 : REF Agent;
BEGIN
  IF l = NIL THEN
    RAISE Errors.SystemError;
  END;
  al := l^.first;
  ag1 := al^.a1;
  ag2 := al^.a2;
  IF (ag1 = agent1 AND ag2 = agent2) OR (ag1 = agent2 AND ag2 = agent1) THEN
    l := l^.tail;
  ELSE
    RemoveActiveLink(agent1, agent2, l^.tail);
  END;
END RemoveActiveLink;

PROCEDURE BuildRule(left, right : Net) : Rule =
(* Puts two nets together into a rule, carrying out validity checks. Raises
an exception Errors.BuildError if the two nets do not form the two sides of a
 valid rule. Need to combine the contexts of the nets into a context for the
 rule. *)
VAR
  vl, l : List.T;
  len : CARDINAL;
  var1, var2, var3, var4 : Variable;
  p1, p2 : REF PortConn;
  ag1, ag2 : REF Agent;
  pdef1, pdef2 : Symbols.Port;
  leftsize, rightsize : CARDINAL;
  rule : Rule;
  truerule : BOOLEAN; (* Whether a rule or an expansion. *)
  context : Tables.T;
BEGIN
```

```
IF left^.typevars = NIL THEN
  context := right^.typevars;
ELSE
  context := left^.typevars;
  l := context;
  WHILE l^.tail # NIL DO
    l := l^.tail;
  END;
  l^.tail := right^.typevars;
END;
(* The lhs must have two items (for a rule)
   or one (for a pseudo-agent expansion) *)
len := List.Length(left^.items^.tail);
IF len = 1 THEN
  truerule := FALSE;
ELSIF len = 2 THEN
  truerule := TRUE;
ELSE
  RAISE Errors.NError(Errors.NErrorPackage
    {Errors.NErrorType.LeftWrongSize, "",""});
END;
(* Left member of a rule must have one active link. *)
IF truerule AND (List.Length(left^.links) # 1) THEN
  RAISE Errors.NError(Errors.NErrorPackage
    {Errors.NErrorType.LeftWrongLinks,"",""});
END;
(* The two sides must have the same number of free variables. *)
leftsize := Tables.Size(left^.vars);
rightsize := Tables.Size(right^.vars);
IF leftsize # rightsize THEN
  RAISE Errors.NError(Errors.NErrorPackage
    {Errors.NErrorType.WrongNumVars,"", ""});
END;
(* The agents on the left must only be connected by their principal ports.*)
IF truerule AND (List.Length(FirstAgent(left)^.conns)
   + List.Length(SecondAgent(left)^.conns) # leftsize + 2) THEN
  RAISE Errors.NError(Errors.NErrorPackage
    {Errors.NErrorType.TooManyConns,"", ""});
END;
(* Every free variable in the left member must occur in the right member,
   and the corresponding ports must be of the same type and direction; if
   there is an edge on the right, need to check two variables on the left.*)
vl := left^.vars;
WHILE vl # NIL DO
  var1 := NARROW(vl^.first, Variable);
  var2 := Tables.Lookup(var1.name, right^.vars);
  IF var2 = NIL THEN
   RAISE Errors.NError(Errors.NErrorPackage
     {Errors.NErrorType.ExtraVar, var1.name,""});
   (* Free variable occurred on left only. *)
  END;
  (* Now check ports. *)
  p1 := var1.pc.port;
  ag1 := var1.pc.target;
  p2 := var2.pc.port;
  IF p2 # NIL THEN (* Variable is on an agent. *)
    ag2 := var2.pc.target;
    pdef1 := Symbols.GetPortGlobal(WhichPort(p1, ag1), ag1^.symbol);
```

```
      pdef2 := Symbols.GetPortGlobal(WhichPort(p2, ag2), ag2^.symbol);
      IF NOT Symbols.Equal(Symbols.CreatePort(var1.type),
                  Symbols.CreatePort(var2.type), context) THEN
        RAISE Errors.NError(Errors.NErrorPackage
          {Errors.NErrorType.TypeNEq, var1.name,""});
       (* Types and directions do not match. *)
      END;
    ELSE (* It's on an edge. *)
      var3 := var2.pc.target;
      var4 := Tables.Lookup(var3.name, left^.vars);
      IF var4 = NIL THEN
        RAISE Errors.NError(Errors.NErrorPackage
          {Errors.NErrorType.ExtraEdgeVar, var3.name,""});
      (* Variable on edge in rhs does not occur in lhs *)
      END;
      p2 := var4.pc.port;
      ag2 := var4.pc.target;
      pdef1 := Symbols.GetPortGlobal(WhichPort(p1, ag1), ag1^.symbol);
      pdef2 := Symbols.GetPortGlobal(WhichPort(p2, ag2), ag2^.symbol);
      IF NOT Symbols.Match(Symbols.CreatePort(var1.type),
                  Symbols.CreatePort(var4.type), context) THEN
        RAISE Errors.NError(Errors.NErrorPackage
          {Errors.NErrorType.BadEdge,
            Text.Cat(var2.name, Text.Cat("-", var3.name)), ""});
      (* Edge in rhs links vars which don't match in lhs *)
      END;
    END;
    vl := vl^.tail;
  END;
  (* Can't define rules for a pseudo-agent or an expansion for a true agent.*)
  IF truerule THEN
    IF Symbols.GetStatus(FirstAgent(left)^.symbol) = Symbols.AgentType.Pseudo
      OR Symbols.GetStatus(SecondAgent(left)^.symbol) =
        Symbols.AgentType.Pseudo THEN
      RAISE Errors.NError(Errors.NErrorPackage
            {Errors.NErrorType.PseudRule, "", ""});
    ELSE
      Symbols.SetStatus(FirstAgent(left)^.symbol, Symbols.AgentType.Agent);
      Symbols.SetStatus(SecondAgent(left)^.symbol, Symbols.AgentType.Agent);
    END;
  ELSE
    IF Symbols.GetStatus(FirstAgent(left)^.symbol) =
      Symbols.AgentType.Agent THEN
      RAISE Errors.NError(Errors.NErrorPackage
            {Errors.NErrorType.AgExpan, "", ""});
    ELSE
      Symbols.SetStatus(FirstAgent(left)^.symbol, Symbols.AgentType.Pseudo);
    END;
  END;
  (* Now it's OK to build a rule. *)
  rule := NEW(Rule, left := left, right := right, context := context);
  IF rule = NIL THEN RAISE Errors.SystemError END;
  RETURN rule;
END BuildRule;

PROCEDURE PurgeComp(link : ActiveLink; comp : List.T) : List.T =
(* Removes occurrences of agents in link from comp. *)
VAR
```

```
  ri : REF Item;
BEGIN
  IF comp = NIL THEN
    RETURN NIL;
  END;
  ri := comp^.first;
  IF ri^.agent THEN
    IF ri^.item = link.a1 OR ri^.item = link.a2 THEN
      RETURN PurgeComp(link, comp^.tail);
    ELSE
      comp^.tail := PurgeComp(link, comp^.tail);
      RETURN comp;
    END;
  ELSE
    comp^.tail := PurgeComp(link, comp^.tail);
    RETURN comp;
  END;
END PurgeComp;

PROCEDURE Purge(link : ActiveLink; VAR comps : List.T) =
(* Removes empty components and occurrences of agents in link. *)
BEGIN
  IF comps = NIL THEN
    RETURN;
  END;
  comps^.first := PurgeComp(link, comps^.first);
  IF comps^.first = NIL THEN
    comps := comps^.tail;
    Purge(link, comps);
  ELSE
    Purge(link, comps^.tail);
  END;
END Purge;


PROCEDURE CopyNet(VAR net : Net) : Net =
(* Makes a copy of the given net. *)
VAR
  copy : Net;
  i, icopy : List.T;
  ri : REF Item;
BEGIN
  copy := CreateNet();
  Names.Reset(GetVars(net));
  i := net^.items^.tail;
  icopy := copy^.items;
  WHILE i # NIL DO
    ri := i^.first;
    icopy^.tail := NEW(List.T, first := NIL, tail := NIL);
    IF ri^.agent THEN
      icopy^.tail^.first := CopyAgent(copy, ri^.item, copy^.links, icopy);
    ELSE
      icopy^.tail^.first := CopyEdge(copy, ri^.item, icopy);
    END;
    ri := icopy^.tail^.first;
    icopy := icopy^.tail;
    i := i^.tail;
  END;
```

```
  RETURN copy;
END CopyNet;

PROCEDURE MatchAgent(ruleag, netag : REF Agent) : BOOLEAN =
(* Tests for matching of print names or symbol names.*)
VAR
  sym : Symbols.SymDef;
  pf : Symbols.PrintNameFn;
BEGIN
  IF ruleag = NIL THEN
    RETURN FALSE;
  END;
  sym := ruleag^.symbol;
  IF sym = netag^.symbol THEN
    pf := Symbols.GetPrintFn(sym);
    IF (pf # NIL) AND (ruleag^.data # NIL) THEN
      RETURN Text.Equal(pf(ruleag^.data), pf(netag^.data));
    ELSE
      RETURN TRUE;
    END;
  ELSE
    RETURN FALSE;
  END;
END MatchAgent;

PROCEDURE FindRule(link : ActiveLink; rules : List.T;
                   VAR switch : BOOLEAN) : Rule =
(* Finds a rule from the list which can be applied to link. Sets switch if the
   agents in the rule are the opposite way round to the link. *)
VAR
  theRule : Rule := NIL;
  netag1, netag2, ruleag1, ruleag2 : REF Agent;
BEGIN
  netag1 := link.a1;
  netag2 := link.a2;
  WHILE rules # NIL AND theRule = NIL DO
    ruleag1 := FirstAgent(LeftSide(rules^.first));
    ruleag2 := SecondAgent(LeftSide(rules^.first));
    IF (MatchAgent(ruleag1, netag1) AND MatchAgent(ruleag2, netag2)) THEN
      theRule := rules^.first;
      switch := FALSE;
    ELSIF   (MatchAgent(ruleag1, netag2) AND MatchAgent(ruleag2, netag1)) THEN
      theRule := rules^.first;
      switch := TRUE;
    END;
    rules := rules^.tail;
  END;
  RETURN theRule;
END FindRule;

PROCEDURE Expand(pseudoag : REF Agent; VAR net : Net; rules : List.T) =
(* Expands a pseudo-agent. *)
(* Code omitted; it's very similar to ReduceLink . *)

PROCEDURE ReduceLink(rlink : REF ActiveLink; VAR net : Net; rules : List.T) =
(* Performs a single reduction step on the given net. *)
VAR
  copy, left, right : Net;
```

```
  rule : Rule;
  lastagent : BOOLEAN; (* Identifies which agent's ports are being scanned. *)
  switch : BOOLEAN;
  rconn, lconn : List.T; (* To step round ports. *)
  ilist, ip, start, end : List.T;
  refag, ra1, ra2, ag1, ag2 : REF Agent;
  v : TEXT;
  matchvar : Variable;
  rpc : REF PortConn;
  e, refedge : REF Edge;
  ri, refi : REF Item;
  data1, data2 : REFANY; (* Data fields of agents in link, in the order in
    which the agents occur in the rule. *)
BEGIN
  rule := FindRule(rlink^, rules, switch); (* Find a matching rule. switch is
    set if the agents in the rule and the link are opposite ways round. The
    use of switch has to be compatible with FindRule - look at order of agents
    or of agents in link, in the rule. *)
  IF rule = NIL THEN
    RETURN; (* Can't happen unless using reduce1 with rules missing. *)
  END;
  left := LeftSide(rule);
  right := rule^.right;
  copy := CopyNet(right); (* Copy to bind into net. *)
  lastagent := FALSE;
  rconn := FirstAgent(left)^.conns^.tail; (* Connection in rule. *)
  ag1 := rlink^.a1;
  ag2 := rlink^.a2;
  IF switch THEN
    refag := rlink^.a2;
    data1 := rlink^.a2^.data;
    data2 := rlink^.a1^.data;
  ELSE
    refag := rlink^.a1;
    data1 := rlink^.a1^.data;
    data2 := rlink^.a2^.data;
  END;
  EvalDataFns(copy, rule^.functions, data1, data2);
  lconn := refag^.conns^.tail; (* Connection in net. *)
  LOOP (* Step round ports of lhs of rule, and of agents in link, in step. *)
    IF rconn = NIL THEN (* Change agents, or terminate. *)
      IF lastagent THEN
        EXIT;
      END;
      rconn := SecondAgent(left)^.conns^.tail;
      IF switch THEN
        refag := rlink^.a1;
      ELSE
        refag := rlink^.a2;
      END;
      lconn := refag^.conns^.tail;
      lastagent := TRUE;
    ELSE
      rpc := rconn^.first;
      v := NARROW(rpc^.target, Variable).name;
        (* Name of variable on port in lhs of rule. *)
      matchvar := Tables.Lookup(v, copy^.vars);
      IF matchvar = NIL THEN
```

```
      RAISE Errors.SystemError;
    END;
    rpc := lconn^.first;
    IF rpc^.port # NIL THEN (* An agent. *)
      rpc^.port^.target := matchvar.pc.target;
      rpc^.port^.port := matchvar.pc.port;
      IF matchvar.pc.port # NIL THEN (* Another agent. *)
        matchvar.pc.port^.target := rpc^.target;
        matchvar.pc.port^.port := rpc^.port;
        ra1 := rpc^.target;
        ra2 := matchvar.pc.target;
        IF rpc^.port^.port = List.First(ra2^.conns) AND
           matchvar.pc.port^.port = List.First(ra1^.conns) THEN
          (* A new active link. *)
          net^.links := NEW(List.T, tail := net^.links, first :=
            NEW(REF ActiveLink,a1 := ra1, a2 := ra2));
        END;
      ELSE (* An edge. *)
        NARROW(matchvar.pc.target, Variable).pc.target := rpc^.target;
        NARROW(matchvar.pc.target, Variable).pc.port := rpc^.port;
        NARROW(matchvar.pc.target, Variable).c := NIL;
          (* So no dud component refs. *)
        matchvar.pc.target := NIL;
      END;
    ELSE (* A variable. *)
      IF matchvar.pc.port # NIL THEN (* An agent. *)
        NARROW(rpc^.target, Variable).pc := matchvar.pc;
        matchvar.pc.port^.target := rpc^.target;
        matchvar.pc.port^.port := NIL;
      ELSE (* An edge. *)
        matchvar.name := NARROW(rpc^.target, Variable).name;
        matchvar.type := NARROW(rpc^.target, Variable).type;
        matchvar.c := NARROW(rpc^.target, Variable).c;
        Tables.Delete(matchvar.name, net^.vars);
        net^.vars := NEW(List.T, first := matchvar, tail := net^.vars);
      END;
    END;
    rpc^.port := NIL;
    rpc^.target := NIL;  (* Disconnect old port. *)
    (* Now move to next port. *)
    rconn := rconn^.tail;
    lconn := lconn^.tail;
  END;
END; (* LOOP *)
(* Now get rid of dud edges from copy. *)
ip := copy^.items;
ilist := ip^.tail;
WHILE ilist # NIL DO
  ri := ilist^.first;
  IF NOT ri.agent THEN
    e := ri.item;
    IF NARROW(e^.p1.target, Variable).pc.target = NIL OR
      NARROW(e^.p2.target, Variable).pc.target = NIL THEN
      e^.back^.tail := ilist^.tail;
      IF ilist^.tail # NIL THEN
        refi := ilist^.tail^.first;
        IF refi^.agent THEN
          refag := refi^.item;
```

```
          refag^.back := e^.back;
        ELSE
          refedge := refi^.item;
          refedge^.back := e^.back;
        END;
      END;
    ELSE
      ip := ip^.tail;
    END;
  ELSE
    ip := ip^.tail;
  END;
  ilist := ilist^.tail;
END;
(* Now remove the first agent in the link...*)
end := ag1^.back; (* Last node before first agent. *)
end^.tail := end^.tail^.tail; (* First node after first agent, or NIL. *)
IF end^.tail # NIL THEN
  ri := end^.tail^.first;
  IF ri^.agent THEN
    refag := ri^.item;
    refag^.back := end;
  ELSE
    refedge := ri^.item;
    refedge^.back := end;
  END;
END;
(* ...and replace the second by copy. *)
end := ag2^.back; (* Last node before second agent. *)
start := end^.tail^.tail; (* First node after second agent, or NIL. *)
end^.tail := copy^.items^.tail;
IF copy^.items^.tail = NIL THEN (* Just remove the second agent. *)
  ip := end;
  end^.tail := start;
ELSE
  ip^.tail := start;
  ri := end^.tail^.first;
  IF ri^.agent THEN
    refag := ri^.item;
    refag^.back := end;
  ELSE
    refedge := ri^.item;
    refedge^.back := end;
  END;
END;
IF start # NIL THEN
  ri := start^.first;
  IF ri^.agent THEN
    refag := ri^.item;
    refag^.back := ip;
  ELSE
    refedge := ri^.item;
    refedge^.back := ip;
  END;
END;
net^.links := List.AppendD(copy^.links, net^.links); (* In case there are
  active links in copy, ie in the rhs of the rule. *)
END ReduceLink;
```

```
PROCEDURE Reduce(VAR net : Net; rules : List.T) =
(* Reduces an active link, or expands a pseudo-agent. *)
VAR
  rlink : REF ActiveLink;
BEGIN
  IF net^.links = NIL THEN
    RETURN;
  END;
  rlink := net^.links^.first; (* Find an active link to reduce. *)
  net^.links := net^.links^.tail;
  IF Symbols.GetStatus(rlink^.a1^.symbol) = Symbols.AgentType.Pseudo THEN
    Expand(rlink^.a1, net, rules);
  ELSIF Symbols.GetStatus(rlink^.a2^.symbol) = Symbols.AgentType.Pseudo THEN
    Expand(rlink^.a2, net, rules);
  ELSE
    ReduceLink(rlink, net, rules);
  END;
END Reduce;

BEGIN
END Nets.
```

# Appendix D - The Proposal

## Background

In [Lafont] a new programming language is proposed, in which the state of a computation is represented by a graph and a program consists of graph rewriting rules. More specifically there is the idea of an agent, which is a labelled object which has various ports enabling it to be connected to other agents. A net is formed by connecting up agents in this way. One port of each agent is identified as the principal port; when two principal ports are connected together we have an active link, and it is along active links that interaction (graph rewriting) takes place. An interaction rule consists of two nets (in which some of the ports may be connected to free variables rather than other ports) where the first net contains an active link and the second net is used in a rewriting step to replace the first. In addition, ports are typed, and identified as inputs or outputs. There are various restrictions on the form of nets and interaction rules, which ensure that a computation cannot deadlock.

The language of interaction nets is based to some extent on Girard's Linear Logic, in which each hypothesis is used exactly once in a proof. The manifestation of this linearity in interaction nets is the condition that in an interaction rule, each free variable in the rule appears exactly once on each side of the rule. Implications of this are that deletion of parts of nets must be explicit, information which is to be used twice must be explicitly duplicated, and that an implementation of the language needs no garbage collector.

Lafont's paper also makes some suggestions as to what additional features might be desirable to make the language more high-level, such as adding modularity, polymorphic typing and facilities to enable a net to interact with its environment.

## Description of work planned

One aim of the project will be to produce an interactive system which allows programs in the interaction nets language to be defined and executed. This system must certainly enable a user to define agents and interaction rules and supply a net to be reduced. In addition it should check that the given nets and interaction rules satisfy the conditions described in [Lafont], and allow programs to be modified and stored in and retrieved from files. So this system will be an interactive interpreter for interaction nets. This part of the project will also involve devising a syntax for textual representation of nets and interaction rules, as that proposed in [Lafont] may not be the most suitable.

The interpreter will then provide a framework for implementing extensions to the language. It would be interesting to add some form of modularity and polymorphic typing, and possibly to provide built-in agents and interaction rules for certain functions - for example the necessary primitives to enable arithmetic to be carried out without doing everything by successors, or to enable an interaction nets program to input and output data.

The project will also involve developing examples of programming with interaction nets. In order to demonstrate that the computational power of interaction nets is not limited, other models of computation such as SK-combinators, lambda calculus or Turing machines could be implemented in interaction nets. It would also be interesting to develop examples illustrating the strengths and weaknesses of the interaction nets approach, and to investigate the impact of explicit duplication and deletion on program performance.

Finally it would be nice to gain some understanding of the theoretical framework into which interaction nets fit.

The program will be implemented in C, Modula-2 or Modula-3, depending on which seems to be the most suitable.

### Resources required

The only resource needed is whichever high-level language system is chosen, either on Phoenix or the Unix machines. Non-standard resources will not be necessary.

### Starting point

[Lafont] is the only material available on interaction nets. In addition to this paper I have been studying some other papers on linear logic. I have also started to write small programs in interaction nets, including a simple combinator reducer.

### Work plan

The program can be split into the following modules: the Front End, which accepts commands from the user and calls on other modules to execute them; the Translator, which translates textual descriptions of nets, agents and interaction rules into the internal representation and also translates in the opposite direction for the purpose of displaying the result of a net reduction; the Reducer, which reduces a net according to the interaction rules; and the Checker, which performs type-checking and checking of the other conditions on nets and interaction rules.

A minimal working system consists of a core Front End, the Translator, and the Reducer. These modules will be implemented first to produce a system which works but has no checking. The Checker will then be added. Additional features can then be added: the Front End can be given more functionality, which may result in new modules such as one for file input/output; the language can be extended, which involves extending the Translator and the Checker.

The Translator can be tested by looking at the internal representations it generates from given textual descriptions, and vice versa. It's two aspects can also be tested as the inverses of each other. The Reducer can be tested by giving it hand-built internal representations of nets and interaction rules; or, given a working Translator, by using the Translator to give data to it and get the results back. The Front End can be tested independently of the other modules, by having it make dummy calls to them. The Checker can also be tested independently, but again this would be made easier by using the Translator.

Development of example programs in interaction nets will proceed alongside developing the program modules, and these programs can then also be used for testing the system as it is developed. More ideas for language extensions may well be generated while writing programs.

Thus there are several stages which represent some degree of success: a minimal system with no checking; a system with checking but a minimal front end; a system with an extended front end; a complete system including language extensions.

### Timetable

10.12.90 - 20.12.90 Specify data structures for representing nets, agents and interaction rules. Define the syntax to be used for interaction nets. Define a minimal Front End and implement it as a skeleton with calls to other modules. Produce a description of the syntax and a user guide for the first version of the Front End.

up to 25.1.91 Design, implement and test the Translator. Produce a description of the design.

28.1.91 - 20.2.91 As above for the Reducer. During these stages, simple interaction nets programs will be developed and used for testing.

21.2.91 - 28.2.91 Prepare Progress Report. At this stage there should be a minimal working system.

4.3.91 - 22.3.91 As above for the Checker. Depending on how quickly work proceeds, it may be possible to do some of this before Progress Report time.

25.3.91 - 12.4.91 Plan language extensions to be implemented, and extensions to Front End. Produce a description of the language extensions and a new user guide for the Front End.

15.4.91 - 3.5.91 Implement language and Front End extensions. Some improvements to the Front End may already have been made in order to make the previous work easier, but this will bring it up to a final version.

6.5.91 - 7.6.91 Preparing for exams; slow down work on project but think about more example programs such as simulating other models of computation.

17.6.91 - 28.6.91 Complete example programs and illustrations of advantages and disadvantages of programming with interaction nets. Several examples will probably have been accumulated during the course of the project.

1.7.91 - 31.7.91 Writing up dissertation. This will draw on the various documents produced during the year.

## References

[Lafont] "Interaction Nets", Yves Lafont, POPL'90.