Theory and Formal Methods 1994: Proceedings of the Second Imperial College Department of Computing Workshop on Theory and Formal Methods, Imperial College Press, 1995.

COMBINATORS FOR INTERACTION NETS

SIMON GAY
Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, United Kingdom
E-mail: sjg3@doc.ic.ac.uk

ABSTRACT

The completeness property of SK combinators is well known. We consider an analogous property in the context of Lafont's Interaction Nets, a graph rewriting system based on classical linear logic. The result is a set of combinators which is complete in the sense of being able to encode all possible patterns of interaction.

1 Introduction

It is a well-known result that any combinatory algebra containing the elements S, K and I with rewriting rules

$$Sxyz \to xz(yz) \quad Kxy \to x \quad Ix \to x$$

is *complete* in the following sense: for any expression $f(x)$ built from combinators and a variable x, there is a combinatory term f^* such that for all terms T, $f^*T \to f(T)$. The definition of f^* is

- $k^* = Kk$ if k does not contain x
- $x^* = I$
- $(gh)^* = Sg^*h^*$.

In fact, it turns out that $I = SKK$ and so I can be eliminated from the system, but this is not crucial. The point is that between them, the combinators can

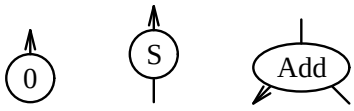
express all possible ways of routing an argument to the place at which it can be operated on. K discards its second argument; I leaves its argument where it is; and S copies its third argument, sending the copies to its other two arguments.

Combinators form a one-dimensional rewriting system, in which groups of adjacent objects interact and are replaced by new configurations. The aim of this paper is to study completeness in a setting in which interacting objects can be connected into more general networks than simple linear strings. The setting we will consider is that of Lafont's Interaction Nets[1] [5], which deals with a certain style of graph rewriting. The motivation for this study came from work on Interaction Nets as a programming language, which triggered a search for an analogy with the combinatory theory.

The rest of this paper is divided into four sections. Section 2 forms an introduction to Interaction Nets, showing how the style of interaction differs from general graph rewriting. Section 3 presents a set of nodes and rewriting rules which enable a result to be proved which corresponds to combinatory completeness. Section 4 extends the system of nodes to allow an encoding of recursive graph rewrites (in which a node may appear on both the left and right sides of a rewriting rule). This is achieved in a way analogous to the representation of recursive function definitions in terms of a Y combinator. The resulting system of nodes could be used as the basis of an implementation of Interaction Nets, just as combinator systems can implement functional programming languages. Finally, Section 5 contains some comments relating our system of nodes to the use of interaction nets for other applications.

2 Interaction Nets

Interaction Nets was introduced by Lafont [5] as a programming language based on the proof nets of Linear Logic [1]. To write a program, one first specifies a collection of *nodes* or *agents*. For example, we might introduce the nodes

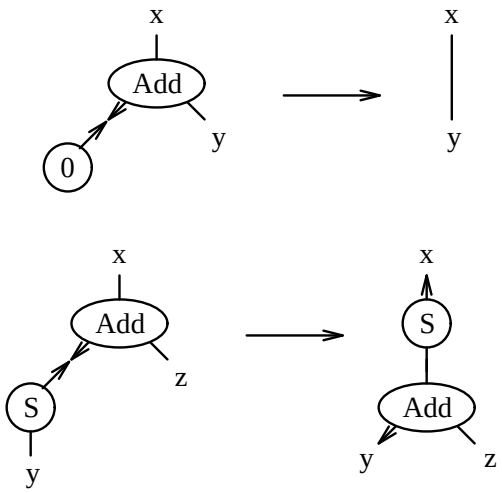


to enable us to program addition of natural numbers. Here, the node S represents successor and Add represents addition. Nodes can be connected into

[1] In this paper, we use *Interaction Nets* (capitalised) as the name of a programming language, and *interaction nets* (uncapitalised) as a plural phrase describing certain kinds of rewritable graph.

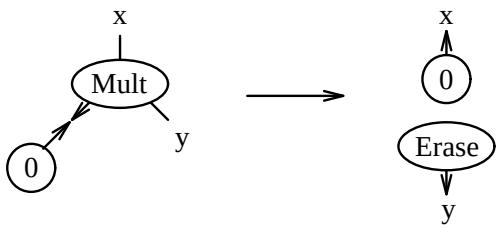
graphs or *nets*. For example, the net

represents the number 2. Note that a node can only be connected to other nodes by its *ports*, which are indicated graphically by unconnected edges. One port of each node is distinguished by an arrow; this is the *principal port*, and the others are *auxiliary ports*. Graph rewriting happens when two principal ports meet, so the next part of a program specifies rewriting rules for the various pairs of nodes (only one for each pair). For addition, the *interaction rules* are

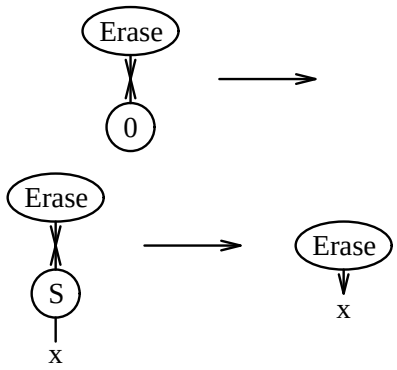


which correspond to the usual recursive definition of addition on unary numerals. The lower ports of Add behave as inputs and the upper one as an output. Interaction rules must have a pair of nodes on the left, but any net can appear on the right, subject to the restriction that there must be the same number of *free ports* (unconnected ports) on the left and the right. The free ports are labelled to indicate how the net on the right is to replace the pair of nodes on the left when the rewrite happens in the middle of a surrounding net. Each label on the left must appear exactly once on the right, and each label can only be attached to one port. This restriction comes into play when we define a node

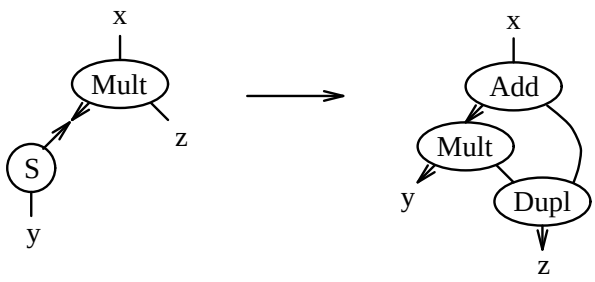
for multiplication. When multiplying by zero



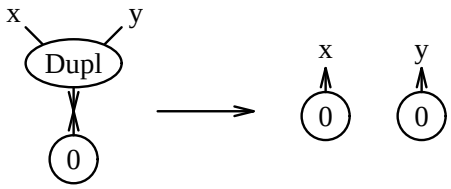
the second argument is not needed, and has to be explicitly removed with an Erase node defined by

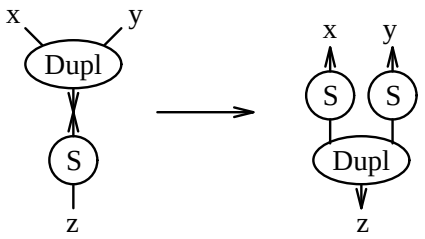


When multiplying by a successor



the second argument must appear as an argument to Add and also as an argument to the recursive appearance of Mult. This is achieved by using a node Dupl, whose definition





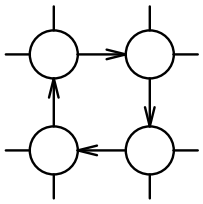
allows it to duplicate a net consisting of 0 and S nodes.

Having set up the nodes and interaction rules, a program can be executed by taking a net (usually with one or more free ports) and applying interaction rules to it until there are no more instances of a pair of principal ports meeting. The reduction relation on nets is confluent (trivially—there can be no critical pairs) but not strongly normalising. Interactions are purely local, and if there are several possible interactions in a net they can be done independently, and potentially in parallel.

In the arithmetic example, we did not give interaction rules for S with S, Add with Mult, etc. Thinking of S and 0 as data constructors and the other nodes as functions, this is entirely reasonable—why would we ever want a function's input port to interact with an input port of another function? To formalise this observation, we may assign a polarity to each port of a node (+ for output, − for input) and specify that two ports can only be connected when they have opposite polarity. Furthermore, we might want to deal with other data types, in which case we would not want to give an interaction rule for, say, Add with True. Thus we can think of the type of a port as a data type together with a polarity, and give types to ports as part of the definition of a node. The node Add, for example, has a principal port of type Nat^-, one auxiliary port of type Nat^- and another of type Nat^+. A well-typed net is one in which every connection is between two ports which have the same datatype and opposite polarities, and interaction rules are only given for pairs of nodes connected according to the type discipline. Once types are in place, it is reasonable to require that an interaction rule is given for *every* well-typed connection between principal ports. Interaction rules must also be well-typed, which means that the occurrences of a label on the left and right sides of a rule must be with ports of the same type. One consequence of this restriction is that a node cannot interact with itself.

If a net contains no reducible pairs of nodes, consider the path traced through the net by starting with some node, following its principal port to the next node, then following that node's principal port, and so on. There are two possibilities for such a path. It may end when a principal port is free, and in this case the net is ready for interactions which may become possible by the connection of another net to its free principal port. Or, the path may loop. The second case

looks something like



and represents a deadlock, as none of the nodes in the cycle can ever interact. Lafont has a refined type system which forbids the construction of deadlocked nets and ensures that they never arise as the result of interactions. We will not use his system, but will just require that the right hand side of an interaction rule is never deadlocked and occasionally specify deadlock-freedom of certain other nets.

In the rest of this paper, we will often talk about an interaction *system*, by which we mean a collection of nodes and interaction rules. An interaction system may be typed or untyped.

3 A Complete Interaction System

We now need to be precise about what we mean by completeness. An interaction system $\mathcal{S}$ is *complete* if given any collection $\mathcal{N}$ of typed nodes and interaction rules between them, the right hand side of each rule being a deadlock-free net over $\mathcal{S}$, then for each node N in $\mathcal{N}$ there is a net $[N]$ over $\mathcal{S}$ such that

- $[N]$ has one free principal port, and a free auxiliary port for each auxiliary port of N
- if $[M]$ and $[N]$ are connected by their principal ports, the resulting net reduces to the right hand side of the interaction rule for M with N.

Suppose we are defining a collection $\mathcal{N}$ of typed nodes. We may assume that there is a type α such that all of the principal ports of nodes in $\mathcal{N}$ have type α^+ or α^-. If this were not the case, we could simply partition $\mathcal{N}$ by principal port type, and carry out our construction in each partition separately. Now we divide $\mathcal{N}$ into two sets, $\mathcal{L}$ and $\mathcal{R}$, according to whether the principal port type is α^+ or α^-. It does not matter which set is which—all we need is the fact that interactions happen only between $\mathcal{L}$ and $\mathcal{R}$, not within $\mathcal{L}$ or $\mathcal{R}$. We can now forget about the types altogether; the only reason for requiring a typed set of definitions in the first place was to ensure that no node could interact with itself.

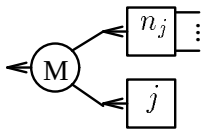
Let the nodes in $\mathcal{L}$ be $L_1, \ldots, L_m$, and those in $\mathcal{R}$ be $R_1, \ldots, R_n$. Write l_i for the number of auxiliary ports of L_i, and r_j for the number of auxiliary ports of R_j. Now assume that there is an interaction rule for every pair (L_i, R_j) of nodes. If this is not the case, arbitrary rules can be given for the missing pairs. Let the net on the right hand side of the rule for L_i and R_j be N_{ij}:

$$L_i \triangleright \triangleleft R_j \rightarrow N_{ij}$$

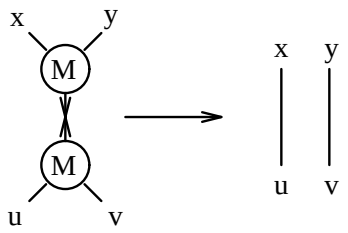
so that N_{ij} has $l_i + r_j$ free ports. Remember that N_{ij} is a net over $\mathcal{S}$, even though we do not yet know anything about which nodes are in $\mathcal{S}$.

Our aim now is to define $[L_i]$ and $[R_j]$, which are nets over $\mathcal{S}$, in such a way that when their principal ports are connected the resulting net reduces to a normal form N_{ij}. The idea of the construction is that $[L_i]$ contains $N_{i1}, \ldots, N_{in}$ and $[R_j]$ is able to select the correct one. Thus part of the structure of $[L_i]$ offers a choice of n possibilities, and there is a corresponding part of the structure of $[R_j]$ which is simply a coding of the natural number j. The rest of the structure of $[L_i]$ and $[R_j]$ is there to make the right connections between their auxiliary ports and the nets N_{ij} which $[L_i]$ contains.

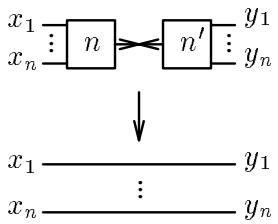
Assuming that we can define a net $\ulcorner p \urcorner$ (with a principal port and no auxiliary ports) to encode the natural number p, and a net $\vec{r}$ (with a principal port and r auxiliary ports) to group r ports together, the net $[R_j]$ is as shown.



The node M is the first node which will be required to be in $\mathcal{S}$. It interacts with itself as follows.

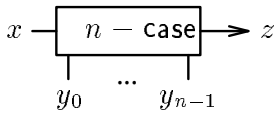


The net $\vec{p}$ is designed to behave as a p-ary M node; however, the net with which it can interact to produce a collection of wires is not $\vec{p}$ itself but a net which we shall call $\vec{p}^{-1}$. The nets $\vec{p}$ and $\vec{p}^{-1}$ will be defined so that



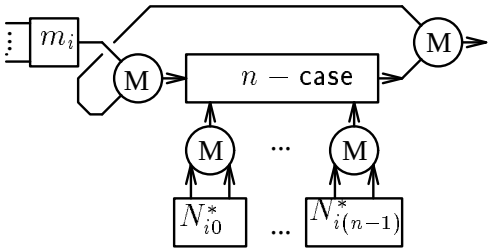
holds.

To define $[L_i]$ we need two further constructions. Firstly, let N^*_{ij} be the net obtained from N_{ij} by attaching a $\vec{l_i}$ net to the free ports corresponding to the auxiliary ports of L_i, and similarly attaching a $\vec{r_j}$ net to the free ports corresponding to the auxiliary ports of R_j. Thus N^*_{ij} has two principal ports. The net $[L_i]$ will contain the n nets $N^*_{i1} \ldots N^*_{in}$. Secondly, assume that we can construct a net $n - \mathsf{case}$ with one principal port and $n+1$ auxiliary ports:



which, if the net $\ulcorner p \urcorner$ $(0 \leq p < n)$ is attached to the principal port z, connects port x to port y_p and somehow deletes whatever is attached to the other y ports.

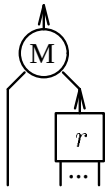
Using these constructions, the definition of $[L_i]$ is as shown.



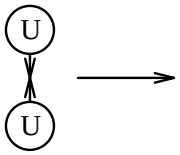
The effect of connecting $[L_i]$ and $[R_j]$ by their principal ports is to delete all of the $N_{..}$ apart from N_{ij}, and connect the auxiliary ports of $[L_i]$ and $[R_j]$ to the correct ports of N_{ij}. This can be verified from the assumptions we have made about the nets N^*_{ij}, $\ulcorner p \urcorner$, $\vec{p}$, $\vec{p}^{-1}$ and $n - \mathsf{case}$, and the interaction rule for the node M. All we need to do now is to define the nets which we have been using.

The nets $\vec{n}$ are defined by induction on n. $\vec{0}$ is a new node

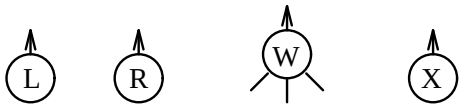
and, given $\vec{r}$, $r \vec{+} 1$ is defined by:



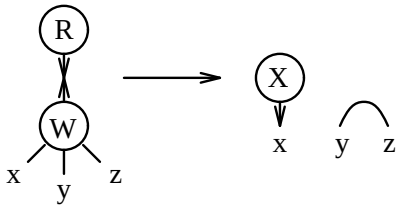
The nets $\vec{n}^{-1}$ are defined in the same way, but building the M node onto the other side of $\vec{r}^{-1}$ in the inductive step. The interaction rule for U with itself is



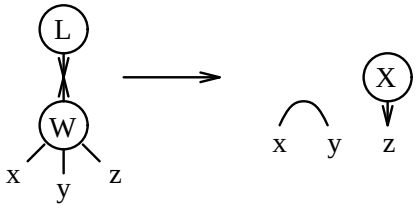
The nodes which allow construction of $n-$ `case` and choice between its options are



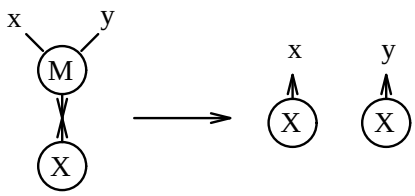
L stands for Left, R for Right, W for With (by analogy with Linear Logic), and X is an eraser. The interaction rules allowing L and R to choose different branches of W are

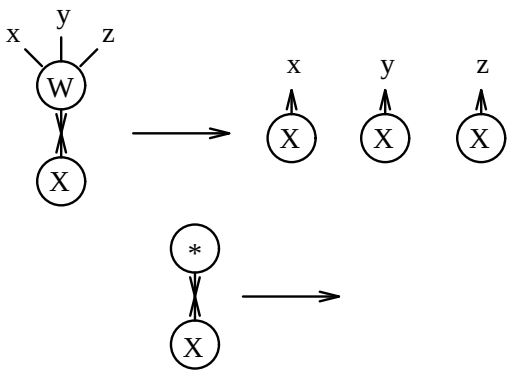


and

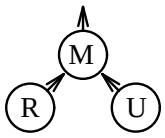


and X must erase absolutely everything ($*$ can be L, R, U or X):

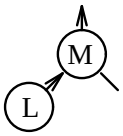




Natural numbers are coded in unary, as in Section 2, using

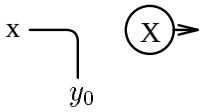


for Zero and

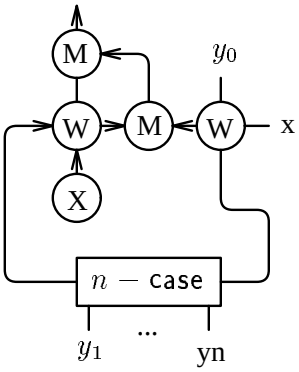


for Successor.

Now we can define $n-$ `case`. $1-$ `case` is defined by

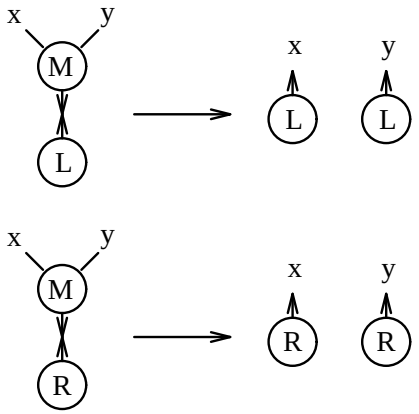


Since there is no choice, the number attached to the z port just has to be erased. Inductively, $(n+1)-$ `case` is defined by



The top-level M node interacts with the M node at the top of an encoded number, and splits into the first L or R and the rest of the number. The first

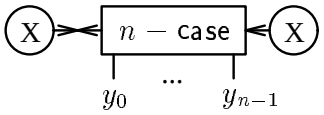
token is fed to the second M node, which plays the role of a copier by means of the rules



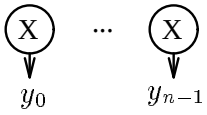
(note that it is only L and R nodes which get copied by M). The two copies of the first token are fed to W nodes. The left-most W node either erases the $r-$ case net and the r nets attached to its y ports, or feeds the rest of the number to it. The right-most W node chooses either the y_0 net or the output from $r-$ case.

To prove that $n-$ case works correctly, we need

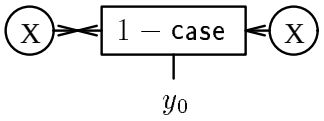
Lemma 1 The net



reduces to



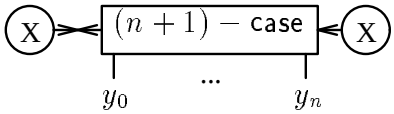
Proof: By induction. For $n = 1$ we have



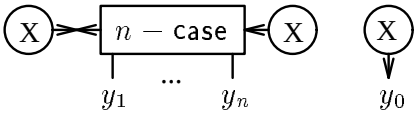
which is equivalent to

and this reduces in one step to

For $n+1$, we have



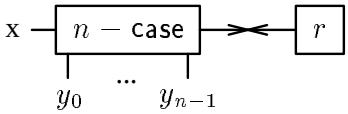
which reduces to



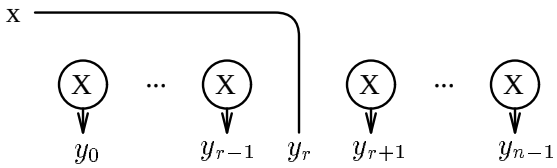
as the X nodes delete the nodes surrounding the occurrence of $n-$ case in the $(n+1)-$ case, and then by the induction hypothesis this reduces to the desired net. □

This allows us to prove

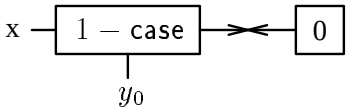
Proposition 2 If $r < n$ then the net



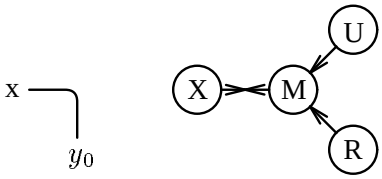
reduces to



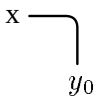
Proof: By induction. For $n = 1$ and $r = 0$ we have the net



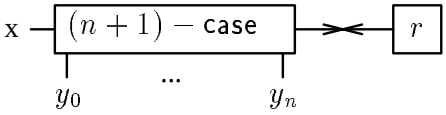
which is equivalent to



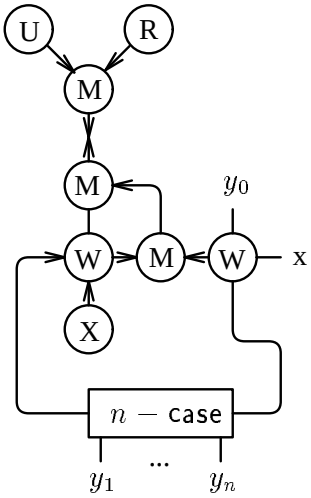
and this reduces to



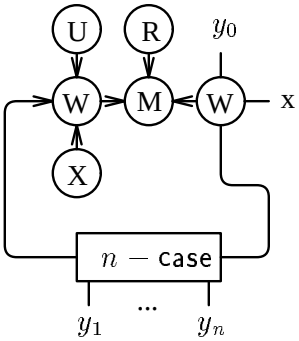
as required. For $n+1$ and $r > 0$ the net



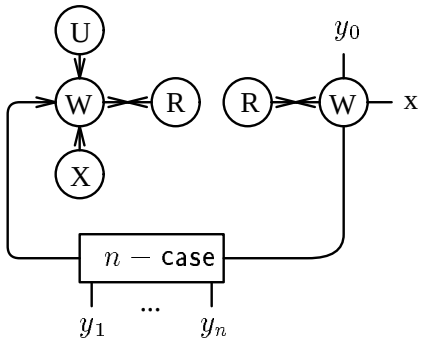
is equivalent to



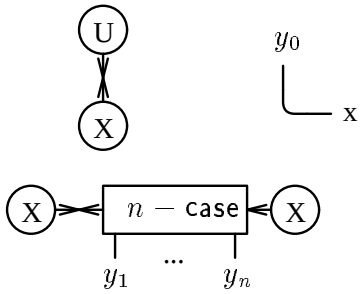
which reduces through



and

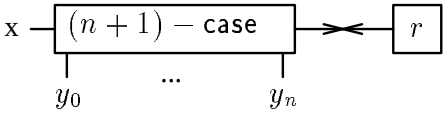


to

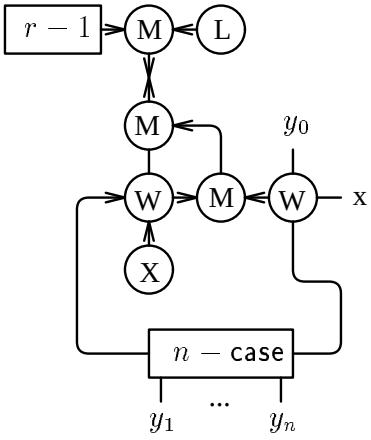


which by the induction hypothesis reduces to

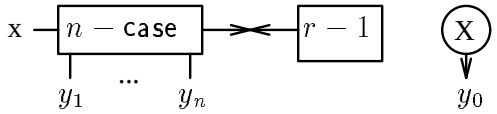
For $(n+1)$ and $r > 0$, the net



is equivalent to



and reduces to



from which point the induction hypothesis completes the proof. □

The constructions we have used will work for any set $\mathcal{S}$ of nodes which contains M, U, L, R, W and X with the interaction rules we have given. The only condition is that X must erase any node in $\mathcal{S}$—this is because the nets N_{ij} may contain any nodes, and we need to be able to erase them when necessary. Hence we have proved

Theorem 1 Any interaction system containing the nodes M, U, L, R, W and X is complete.

4 Compiling Interaction Nets

Exhibiting a complete interaction system is one thing, but we cannot yet claim to have a set of fundamental combinators for interaction nets. This is because an interaction nets program consists of a collection of interaction rules in which the right hand sides are nets over the agents being defined by the program. In other words, there is a set of mutually recursive definitions of agents. In order to represent an arbitrary set of interaction net agents in terms of fundamental nodes, we need to eliminate this recursion. In fact, we will replace recursion by copying.

To illustrate the idea, consider a recursive definition in a functional language:

$$fx = \ldots(fy)\ldots$$

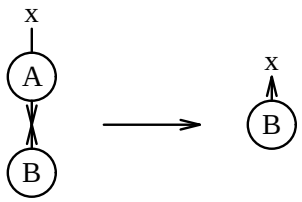
where the body of the function f contains a recursive call. We can define a new function F, which takes two arguments, by

$$Fgx = \ldots(ggy)\ldots.$$

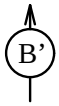
F has an extra functional argument g, which is used in the body as a replacement for the original recursive call of f. The key step is now that $FFx = \ldots(FFy)\ldots$, and so we can recover f as FF. Thus we have replaced recursion by copying of g (and self-application, but this will not be a problem as our complete interaction systems will always consist of untyped nodes). As a sideline, this technique can be used to rationally construct a fixpoint combinator, as follows. We want to define a λ-expression Y such that $Yf =$

$f(Yf)$. Defining instead Z by $Zgf = f(ggf)$ and setting $Y = ZZ$, we get $Y = (\lambda xy.y(xxy))(\lambda xy.y(xxy)$ which is indeed a fixpoint combinator.

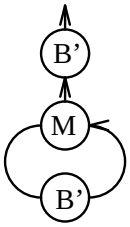
To apply this idea to interaction net definitions is fairly straightforward, although there are a couple of subtleties. Consider a very simple example of an interaction rule.



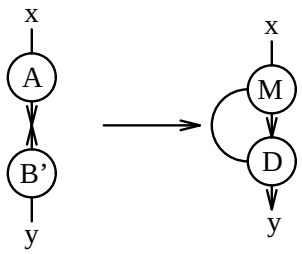
To remove the recursion, we use a new node



instead of B. The extra port of B' is analogous to the extra argument of the function F in the functional example; it will be used to hold another copy of B'. So we will replace the node B by this net.

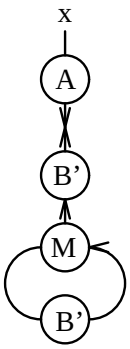


The interaction rule for A and B' must duplicate the stored copy of B', unpack one copy, and use the resulting ingredients to form the net which stands for B'.

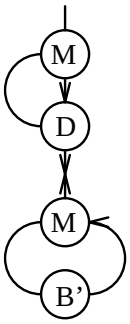


So, the net

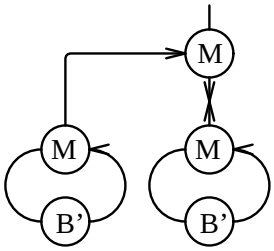
becomes



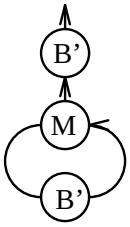
which reduces to



and then to

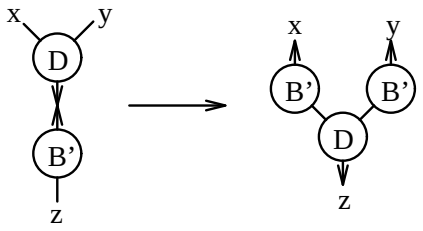


and finally to the original net representing B.

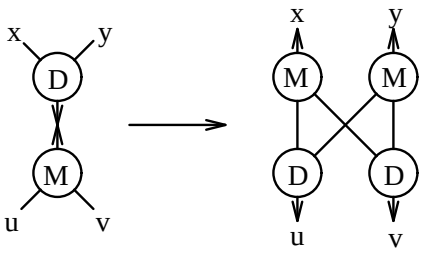


For this to work as claimed, the node D must be able to duplicate cyclic nets.

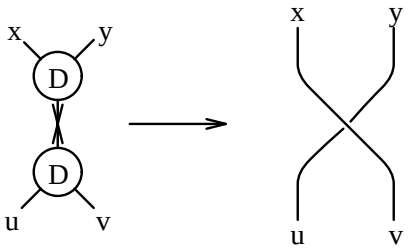
The interaction rules which allow this are the expected rules such as



and



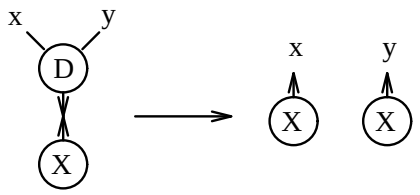
and also



which causes duplicators to annihilate each other when they meet. When a cyclic net is copied, D nodes spread throughout the net in all directions, copying as they go, until they meet and disappear.

We have now seen how to eliminate recursion from interaction net definitions. The same idea applies to more general situations than the very simple one which we have looked at. When both nodes on the left of an interaction rule appear on the right, they both need to be replaced in the same way that B was replaced by B'. If there is a collection of mutually recursive definitions, then all the nodes involved need to be replaced in this way, and the packaged copies need to be carried through all the interaction rules, even the ones which do not actually use them. It may also be necessary for a node to carry packaged copies of other nodes—it just depends on the form of the interaction rules. In terms of giving a general translation, the simplest approach is to divide the nodes into two classes according to the type of their principal ports, and then make *every* node in one class carry packaged copies of *all* the other nodes.

The idea of removing recursion from the original set of interaction rules was to produce a set of definitions in which the right hand side of every rule is a net over the nodes in the universal set. But we still have a slight problem—to remove recursion, we introduced an extra node, D, into the right hand sides of the rules. So what we need now is to add D to the universal set, taking care that the universal property is maintained. As stated previously, the constructions used in the proof of universality work for any set of nodes containing M, W, L, R, U and X, as long as X erases everything. So all we need is an interaction rule for X with D.

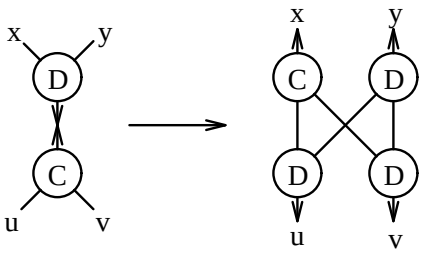


There is a pleasing symmetry between the rôles of X and D in this rule: X erases D and puts new erasers on the auxiliary ports of D, or alternatively, D copies X to form two erasers.

Unfortunately, this is still not enough. The problem now is that D only correctly copies nets which do not themselves contain D nodes. The reason for this is that when two D nodes meet, they do not copy each other but instead annihilate each other. This is essential for correct copying of nets which do not contain D nodes, but means that a D node inside a net does not get copied and, furthermore, interferes with the copying process by destroying one of the D nodes which have been sent in to do the copying. This wouldn't necessarily matter except that the ability to copy a net is crucial for the elimination of recursion. The way we get round this problem is to introduce what in linear logic would be called a *contract* node.

The way we think of C is as a node which stands for a copier, although it is inert in itself. To see how it is used, consider the example of the interaction rule between A and B. The node B is replaced by B' in order to eliminate the recursion. There is then a step of transltion into the basic combinators, in which the node B' becomes a net. It is this net which is then packaged up and which has to be copied. So the idea is that in this packaged net, C should be used instead of D. Then when the net is copied, one copy should have the C nodes replaced by D nodes again. Thus there will be one copy in which the inert contract nodes have been replaced by active duplicator nodes—this copy then does the work of the original interaction rule—and one copy which is still

inert and is attached as the packaged copy to another node. To achieve this, C and D must interact like this.



The result is a system of nodes, containing the nodes needed for completeness, and allowing nets to be copied in a restricted way which is nevertheless sufficient for the use we wish to make of it, namely eliminating recursion from definitions. Any collection of node definitions can be translated into this set, by first removing recursion and then applying the completeness result. This is the point at which we claim to have a means of compiling arbitrary interaction net definitions into a basic set of combinators.

Theorem 2 The nodes M, U, L, R, W, X, D and C form an interaction system into which any collection of typed interaction net definitions can be compiled.

5 Applications

The main potential application of this compilation procedure is to the implementation of interaction nets. While it is straightforward to implement interaction rules on a sequential system, the nature of a net clearly suggests that a parallel implementation should be possible. This is because interaction rules can be applied locally, and independently of the surrounding net. One scheme for a parallel implementation might be for a large number of processors to each store part of a net; each processor would then apply interaction rules to its own nodes. Clearly each processor has to know all the interaction rules, which is where our compilation scheme can be applied: if all nets can be represented in terms of a small set of fundamental nodes, then the set of interaction rules which each processor has to know about is quite small, and never has to be changed.

Unfortunately, this is not the complete answer to the parallel implementation question—there seem to be far more problems to do with the necessity of moving nodes from one processor to another. But at least it is a step in the right direction.

6 Related Work

There is a close connection between the fundamental set of nodes described in this paper, and the nodes used in interaction net implementations of linear logic [2]. Our M is a multiplicative node, either $\otimes$ or $\parr$ in linear logic. U is a multiplicative unit. W effectively implements a linear logic $\&$ construction, and the selectors play the rôles of insertions into the two branches of a $\oplus$ type. As in linear logic, any copying or erasing has to be done explicitly, although there is a difference in that we are allowed to copy or erase *any* node; there is no restriction corresponding to the use of ! types. On the other hand, we do have the contraction node C which has been used in interaction net implementations of linear logic and the λ-calculus [7].

Honda and Yoshida [3, 4] have studied a system of combinators for concurrency, which have certain similarities to our interaction net combinators. One important difference is that they consider situations in which communicating agents are placed in parallel without explicit connections.

After the present paper had been presented at the Theory and Formal Methods Workshop, it came to my attention that Lafont has studied exactly the same problem of finding combinators for interaction nets, and found a similar solution [6]. There are two respects in which his solution goes further than ours. The first is that he is able to deal with interaction rules in which a node reacts with itself. The second is that he only uses three combinators: the X, M and D of the present paper, and is able to encode all the necessary constructions in terms of them. Some of these encodings are straightforward: X and U become the same node, as they have the same rules, and the conditional construction using W, L and R can be replaced by a construction using M and X. What is really surprising is that he is also able to eliminate the contraction node C, by a clever encoding of the positions of D nodes within a net.

Acknowledgements

Ian Mackie reviewed this paper and made several useful comments. I have also enjoyed discussions with Yves Lafont, Kohei Honda and Nobuko Yoshida. I have been financially supported by an EPSRC Studentship and the EU Basic Research Project 9102 (COORDINATION).

References

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

[2] G. Gonthier, M. Abadi, and J.-J. Lévy. Linear logic without boxes. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 223–234. IEEE Computer Society Press, 1992.

[3] K. Honda and N. Yoshida. Combinatory representation of mobile processes. In *Proceedings, 21st ACM Symposium on Principles of Programming Languages*, 1994.

[4] K. Honda and N. Yoshida. Replication in concurrent combinators. In *Proceedings of the TACS Conference*, Lecture Notes in Computer Science. Springer-Verlag, 1994.

[5] Y. Lafont. Interaction nets. In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 95–108. ACM, ACM Press, January 1990.

[6] Y. Lafont. The paradigm of interaction, September 1994. Draft paper.

[7] I. Mackie. *The Geometry of Implementation.* PhD thesis, Department of Computing, Imperial College of Science Technology and Medicine, 1994.