7 Interaction-Based Models of Computation

In this chapter, we study interaction nets, a model of computation that can be seen as a representative of a class of models based on the notion of "computation as interaction". Interaction nets are a *graphical* model of computation devised by Yves Lafont in 1990 as a generalisation of the proof structures of linear logic. It can be seen as an abstract formalism, used to define algorithms and analyse their cost, or as a low-level language into which other programming languages can be compiled. This is fruitful because interaction nets can be implemented with reasonable efficiency.

An interaction net system is specified by a set of *agents* and a set of *interaction rules*. One can think of agents as logical symbols (connectives) and interaction rules as a specification of their meaning. There is also an analogy with electric circuits, where the agents are seen as gates and the edges as wires connecting the gates. Or we can simply think of the agents as computation entities, with interaction rules specifying their behaviour.

In the following sections, we give an overview of the interaction paradigm, give examples of uses of interaction nets to express algorithms, and also show how other computation models can be encoded in interaction nets.

7.1 The paradigm of interaction

Interaction net systems are specified by giving a set Σ of symbols used to build nets and a set $\mathcal{R}$ of rewrite rules, called *interaction rules*, that must satisfy the

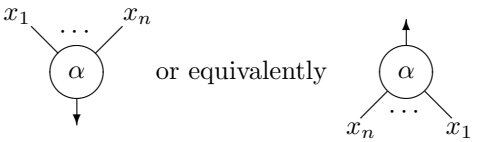
set of conditions given below. Each symbol $\alpha \in \Sigma$ has an associated (fixed) *arity*, a natural number. We assume that the function $\mathsf{ar} : \Sigma \to \mathsf{Nat}$ provides the arity of each symbol in Σ.

Definition 7.1 (Net)

A net N built on Σ is a graph (not necessarily connected) where nodes are labelled by symbols in Σ. A labelled node is called an *agent*, and an edge between two agents is called a *wire*, so nets are graphs built out of agents and wires.

The points of attachment of wires are called *ports*. If the arity of α is n, then a node labelled with α must have $n + 1$ ports: a distinguished one called the *principal port*, depicted by an arrow, and n *auxiliary ports* corresponding to the arity of the symbol.

We index ports clockwise from the principal port, and hence the orientation of an agent is not important. If $\mathsf{ar}(\alpha) = \mathsf{n}$, then an agent α is represented graphically in the following way:



Note that this agent has been rotated (not reflected) and the ports are indexed in the same way. If $\mathsf{ar}(\alpha) = 0$, then the agent has no auxiliary ports, but it will always have a principal port.

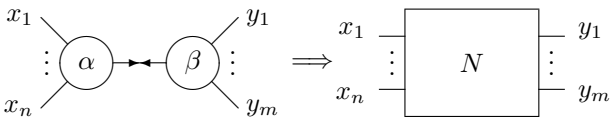
In an interaction net, edges connect agents together at the ports such that there is at most one edge at each port (edges may connect two ports of the same agent). The ports of an agent that are not connected to another agent are called *free ports*. There are two special instances of a net that we should point out. A net may contain only edges (no agents); this is called a *wiring*, and the free extremities of the edges are also called ports. In this case, if there are n edges, then there are $2n$ free ports in the net. If a net contains neither edges nor agents, then it is the *empty net*. The *interface* of a net is its set of free ports.

Definition 7.2 (Interaction rule)

A pair of agents $(\alpha, \beta) \in \Sigma \times \Sigma$ connected together on their principal ports is called an *active pair*; this is the interaction net analogue of a redex, and it will be denoted $\alpha \bowtie \beta$.

An interaction rule $\alpha \bowtie \beta \Longrightarrow N$ in $\mathcal{R}$ is composed of an active pair on the left-hand side and a net N on the right-hand side. Rules must satisfy two strong conditions:

1. In an interaction rule, the left- and right-hand sides have the same interface; that is, all the free ports are preserved. The following diagram illustrates the idea, where N is any net built from Σ.



   We remark that the net N may contain occurrences of the agents α and β. N can be just a wiring (but only if the number of free ports in the active pair is even), and if there are no free ports in the active pair, then the net N may be (but is not necessarily) the empty net.

2. In a set $\mathcal{R}$ of interaction rules, there is at most one rule for each unordered pair of agents (that is, only one rule for $\alpha \bowtie \beta$, which is the same as the rule for $\beta \bowtie \alpha$).

Interaction rules generate a reduction relation on nets, as shown below.

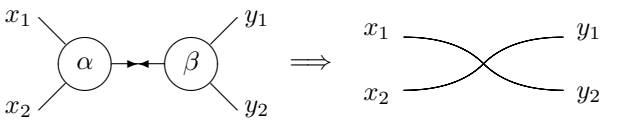
Definition 7.3

A reduction step using the rule $\alpha \bowtie \beta \Longrightarrow N$ replaces an occurrence of the active pair $\alpha \bowtie \beta$ by a net N. More precisely, we write $W \Longrightarrow W'$ if there is an active pair $\alpha \bowtie \beta$ in W and an interaction rule $\alpha \bowtie \beta \Longrightarrow N$ in $\mathcal{R}$ such that W' is the net obtained by replacing $\alpha \bowtie \beta$ in W with N (since N has the same interface as $\alpha \bowtie \beta$, there are no dangling edges after the replacement).

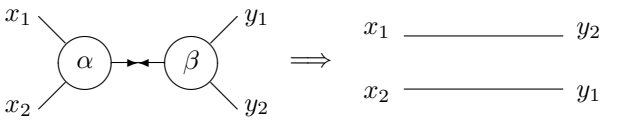
We write $\Longrightarrow$ for a single interaction step and $\Longrightarrow^*$ for the transitive reflexive closure of the relation $\Longrightarrow$. In other words, $N \Longrightarrow N'$ indicates that we can obtain N' from N by reducing one active pair, and $N \Longrightarrow^* N'$ indicates that there is a sequence of zero or more interaction steps that take us from N to N'.

We do not require a rule for each pair of agents, but if we create a net with an active pair for which there is no interaction rule, then this pair will not be reduced (it will be *blocked*).

It is important to note that the interface of the net is *ordered*. Adopting this convention, we can avoid labelling the free edges of a net. To give an example, we can write the rule



that connects x_1 with y_2 and x_2 with y_1 equivalently as the rule



but in the latter the labelling is essential (the difference being that we have changed the order of the free ports of the net). We will always make an effort, at the cost of making the rules look more complicated, to ensure that the order of the edges is always preserved when we write a rule to avoid having to label the edges (adopting the same convention for nets as we did for agents).

An interaction net is in *full normal form* (we will often just call it normal form) if there are no active pairs. The notation $N \Downarrow N'$ indicates that there exists a finite sequence of interactions $N \Longrightarrow^* N'$ such that N' is a net in normal form. We say that a net N is *normalisable* if $N \Downarrow N'$; N is strongly normalisable if all sequences of interactions starting from N are finite.

As a direct consequence of the definition of interaction nets, in particular of the constraints on the rewrite rules, reduction is (strongly) commutative in the following sense: If two different reductions are possible in a net N (that is, $N \Longrightarrow N_1$ and $N \Longrightarrow N_2$), then there exists a net M such that N_1 and N_2 both reduce in one step to M: $N_1 \Longrightarrow M$ and $N_2 \Longrightarrow M$. This property is stronger than confluence (it is sometimes called strong confluence or the diamond property); it implies confluence. Consequently, we have the following result.

Proposition 7.4

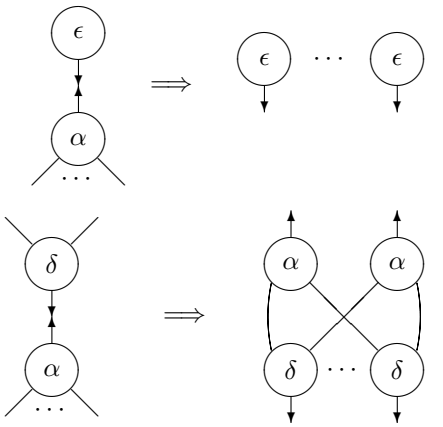
Let N be a net in an interaction system $(\Sigma, \mathcal{R})$. Then:

1. If $N \Downarrow N'$, then all reduction sequences starting from N are terminating (N is strongly normalisable).
2. Normal forms are unique: If $N \Downarrow N'$ and $N \Downarrow N''$, then $N' = N''$.

Below we give an example of the implementation of two familiar operations using interaction nets.

Example 7.5

The following interaction rules define two ubiquitous agents, namely the *erasing* agent (ϵ), which deletes everything it interacts with, and the *duplicator* (δ), which copies everything.



In the diagrams representing the rules, α denotes any agent. Indeed, there is one rule defining the interaction between ϵ and each agent α in Σ and also one rule for each pair $\delta \bowtie \alpha$.

According to the first rule above, the interaction between α and ϵ deletes the agent α and places erase agents on all the free edges of the agent. Note that if the arity of α is 0, then the right-hand side of the rule is the empty net; in this case the interaction marks the end of the erasing process. One particular case of this is when α is an ϵ agent itself. These rules provide the garbage collection mechanism for interaction nets.

In the second rule, we see that the α agent is copied, and the δ agents placed on the free edges can now continue copying the rest of the net.

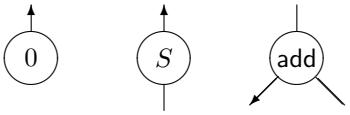
7.2 Numbers and arithmetic operations

Natural numbers can be represented using 0 and a successor function, as described in previous chapters. For example, the number 3 is represented by $S(S(S(0)))$. Consider the following specification of the standard addition operation:

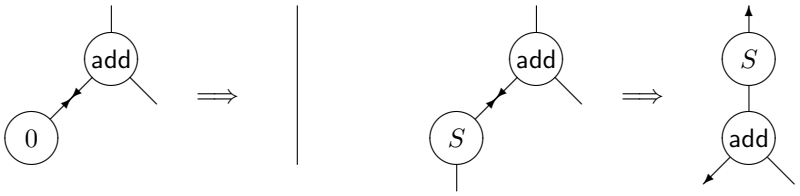
$$\begin{aligned} \mathsf{add}(0, y) &= y \\ \mathsf{add}(S(x), y) &= S(\mathsf{add}(x, y)) \end{aligned}$$

which indicates that adding 0 to any number y gives y as a result, and to add $x + 1$ to y, we need to compute $x + y$ and add 1.

To code this system into an interaction net program, we introduce three agents, corresponding to add, S, and 0. These are drawn as follows.

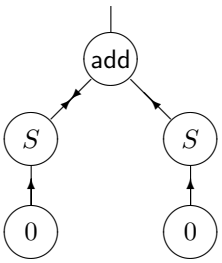


Next we must specify the rules of interaction. In this case, we can mirror the specification of addition given above. The two rules that we need are as follows:

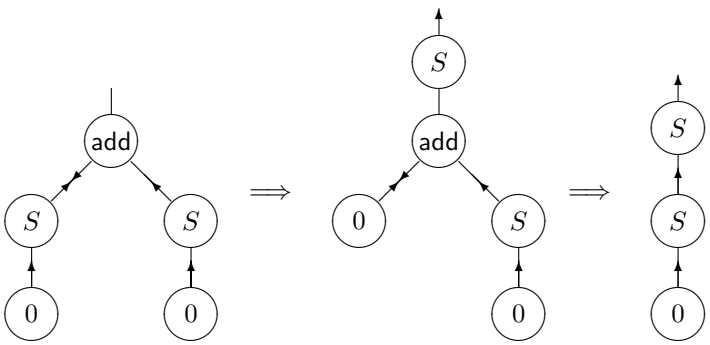


These rules trivially satisfy the requirements of preserving the interface for an interaction.

Consider the net corresponding to the term $\mathsf{add}(S(0), S(0))$:



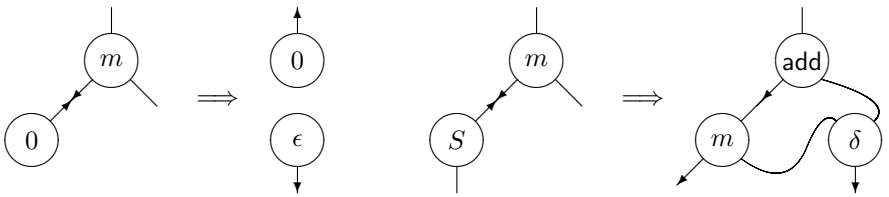
In this example, there is only one choice of reduction since at each step there is only one possible interaction that can take place. The complete sequence of reductions is shown below. The result is a net representing $S(S(0))$, as expected.



This example is rather too simple though to bring out the essential features of interaction nets. A more interesting example is the coding of the operation of multiplication, specified by

$$\begin{array}{lcl} \mathsf{mult}(0, y) & = & 0 \\ \mathsf{mult}(S(x), y) & = & \mathsf{add}(\mathsf{mult}(x, y), y) \end{array}$$

To give an algorithm to multiply numbers using interaction nets, we need to introduce a new agent, m, to represent the multiplication operator. The interaction rules for this agent are more involved than those for add due to the fact that multiplication is not a linear operation (as was the case with addition). To keep in line with the definition of an interaction rule, we must preserve the interface. To illustrate this, here are the two rules for multiplication:



To preserve the interface, we have used the erasing and duplicating agents (ϵ and δ), that were introduced in Example 7.5.

This example illustrates one of the most interesting aspects of interaction nets. It is impossible to duplicate active pairs, and thus sharing of computation is naturally captured. Indeed, to duplicate a net, δ must be able to interact with all the agents in the net, but if α and β are connected on their principal ports, they cannot interact with δ and therefore cannot be copied.

Below we give another example of an operation on numbers and its definition using interaction nets.

Example 7.6

Consider the function that computes the maximum of two natural numbers:

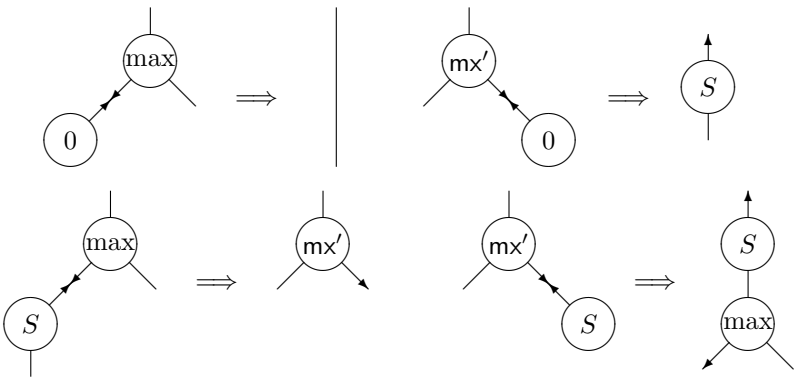
$$\begin{array}{lcl} \max(0, y) & = & y \\ \max(x, 0) & = & x \\ \max(S(x), S(y)) & = & S(\max(x, y)) \end{array}$$

The problem with this specification is that it is defined by cases on *both* of the arguments. If we follow the same ideas as in the previous examples, we would need two principal ports for the agent max, but this is not possible in interaction nets. However, we can transform the specification of max, introducing a new

function max′, to obtain an equivalent system where each operation is defined by cases on only one argument:

$$\begin{array}{lcl} \max(0, y) & = & y \\ \max(S(x), y) & = & \max'(x, y) \\ \max'(x, 0) & = & S(x) \\ \max'(x, S(y)) & = & S(\max(x, y)) \end{array}$$

The corresponding interaction rules are:



The definition of a system of interaction for computing the minimum of two numbers is left as an exercise (see Section 7.9).

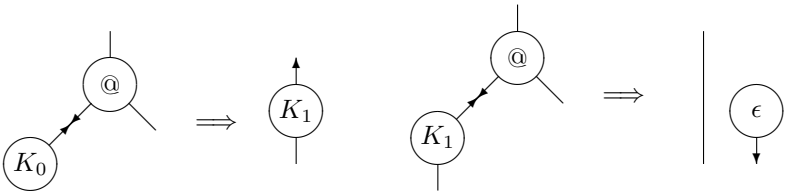
This example suggests a method of compiling functions on numbers into interaction nets. Indeed, it is possible to compile all functional programs into interaction nets. Interaction nets are in fact a universal programming language, as we will see in the next section.

7.3 Turing completeness

To show that a model of computation is Turing complete, we have to prove that any computable function can be represented. In the case of interaction nets, this can be shown for instance by giving an encoding of combinatory logic (CL). Combinatory logic was introduced in Chapter 3 (see Exercise 11) as a system of combinators with constants, S and K, and two reduction rules with the same power as the λ-calculus. Let us recall the reduction rules:

$$\begin{array}{lcl} K\ x\ y & \to & x \\ S\ x\ y\ z & \to & x\ z\ (y\ z) \end{array}$$

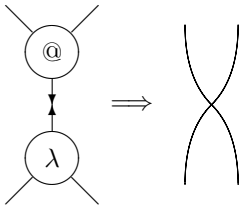
To represent CL as a system of interaction nets, we require an agent @ corresponding to application and several agents for the combinators. For example, the K combinator is encoded by introducing two agents, K_0 and K_1, and two interaction rules:



The combinator S can be defined in a similar way using three agents and three interaction rules; we leave it as an exercise.

Interaction nets have also been used to implement λ-calculus evaluators. Indeed, the first implementation of the optimal reduction strategy for the λ-calculus (that is, the strategy that makes the minimum number of β-reduction steps in order to normalise terms) used interaction nets. Interaction nets are also used in other (non-optimal, but in some cases more efficient) implementations of the λ-calculus.

Actually, if we restrict ourselves to linear λ-terms (see the definition of the linear λ-calculus in Exercise 10 of Chapter 3), we only need an application agent @ and an abstraction agent λ; variables can be encoded by wires. Then the β-reduction rule is simply encoded as follows:



For general λ-terms, we have to introduce copying agents and erasing agents, as well as auxiliary agents to keep track of the scope of abstractions.

7.4 More examples: Lists

We can represent lists in interaction nets in different ways. For instance, we can build a list by using a binary agent cons to link the first element of the list to the rest of the list. The empty list can be represented with an agent nil. This representation of lists mimics the traditional specification of the list data structure in functional languages using constructors cons and nil. For instance, in a

functional language, we could define list concatenation (the append function) as follows:

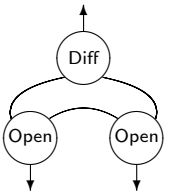
$$\begin{array}{lcl} \mathsf{append}(\mathsf{nil}, l) & = & l \\ \mathsf{append}(\mathsf{cons}(x, l), l') & = & \mathsf{cons}(x, \mathsf{append}(l, l')) \end{array}$$

Using this representation of lists, the time required to concatenate two lists is proportional to the length of the lists (more precisely, with the definition above, it is proportional to the length of the first list).

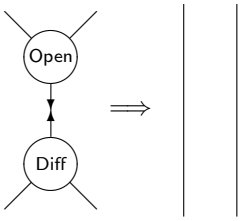
A trivial encoding of lists and the concatenation operator in interaction nets, following the specification above, uses three agents: cons, nil, and append. However, if we use graphs instead of trees to represent lists, then we can obtain a more efficient implementation. The idea, to speed up the append function, is to have direct access to the first and last elements of the lists. Since interaction nets are general graphs, not just trees, this can be achieved simply by representing a list as a linked structure, using an agent Diff to hold pointers to the first and last elements of the list (the name comes from difference lists) and an agent cons as usual to link the internal elements.

The empty list, nil, is then encoded by the net:

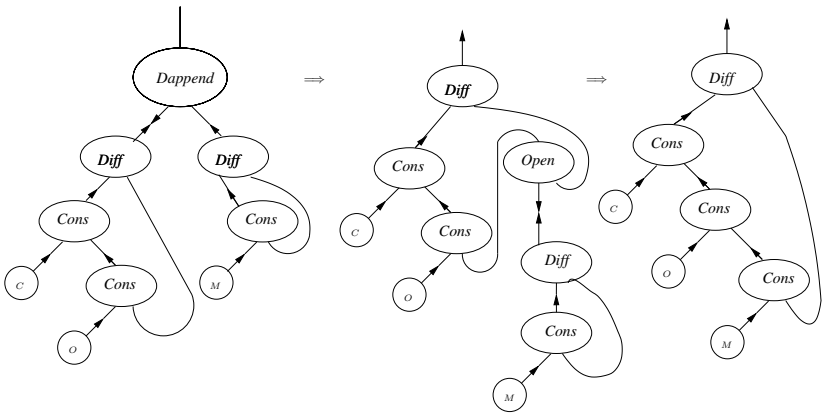
The operation of concatenation is implemented in constant time with the net



using an additional interaction rule that allows us to access the lists:



For example, we have the following reduction:



7.5 Combinators for interaction nets

The combinators S and K of combinatory logic provide a complete characterisation of computable functions; similarly, there is a universal set of combinators for interaction nets that uses three agents called δ, γ, and ϵ. In Figure 7.1, we give these three basic agents. The first two provide multiplexing operations (i.e., merging two wires into one), and the third is an erasing operation. All interaction nets can be built from these agents by simply wiring agents together.

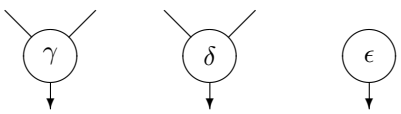


Figure 7.1 Interaction combinators: γ, δ, and ϵ.

In Figure 7.2, we give the six interaction rules for this system. It is clear that the ϵ agent behaves as an erasing operation in that it consumes everything it interacts with. The multiplexing agents either annihilate each other (if they are the same agent), giving a wiring, or they mutually copy each other (if they are different). Note that the right-hand side in the final rule is the empty net.

This system of combinators is *universal* in the sense that any other interaction net system can be encoded using these combinators. There are other universal systems of combinators for interaction nets.

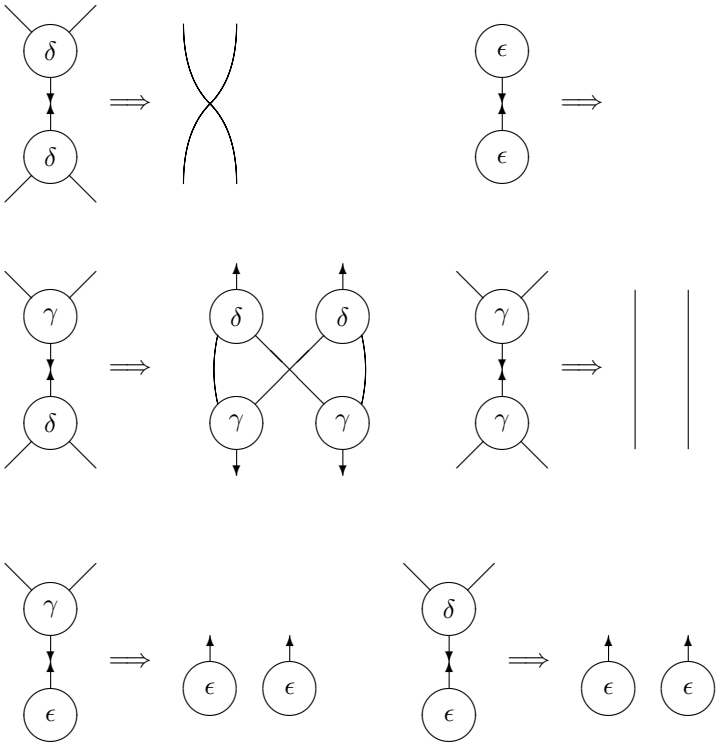


Figure 7.2 Interaction rules for the interaction combinators.

7.6 Textual languages and strategies for interaction nets

The graphical language of interaction nets is very natural, and diagrams are often easier to grasp than a textual description. However, a formal, textual account of interaction nets has many advantages: It simplifies the actual writing of programs (graphical editors are not always available), and static properties of nets, such as types, can be defined in a more concise way. Indeed, several textual notations for interaction nets have been devised. Below we describe three notations through an example before developing one of the notations into a full textual interaction calculus.

As a running example, consider the net given in Figure 7.3 and the interaction rule for β-reduction in the linear λ-calculus given in Section 7.3.

A natural textual notation for nets consists of listing all the agents, with their ports, using some convention. For instance, we could list the ports clock-

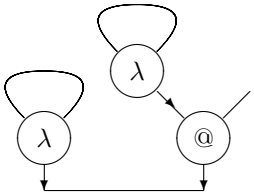


Figure 7.3 A λ-term represented as a net.

wise, starting from the principal port. Edges in the net can be represented by using the same port name. Using these conventions, the example net above is written

$$@(a,b,c), \lambda(a,d,d), \lambda(b,e,e)$$

since we have two λ agents and an application agent @. Note the repetition of name ports to define edges; for instance, in $\lambda(a,e,e)$, the repeated e indicates that there is a wire linking the two auxiliary ports of this λ agent.

The same notation can be used to represent interaction rules. For example, the interaction rule for linear β-reduction is written

$$@(a,b,c), \lambda(a,d,e) \Longrightarrow I(c,e), I(b,d)$$

where the symbol I is used to represent wirings (they are not attached to agents). Note that exactly the same ports are used on the left- and right-hand sides since interaction rules preserve the interface.

Another alternative is to use indices instead of names for ports, starting with the index 0 for the principal port. For the example net above, we use a set of agents:

$$\Sigma = \{@_1, \lambda_1, \lambda_2\}$$

The linear β-rule is written

$$(\lambda_i, @_j) \longrightarrow (\emptyset, \{\lambda_i.1 \equiv @_j.1, \lambda_i.2 \equiv @_j.2\})$$

and the example net is represented by

$$(\{@_1, \lambda_1, \lambda_2\}, \{@_1.0 \equiv \lambda_1.0, @_1.1 \equiv \lambda_2.0, \lambda_1.1 \equiv \lambda_1.2, \lambda_2.1 \equiv \lambda_2.2\})$$

A third alternative, which yields a more compact notation, is based on a representation of active pairs as equations. In this case, our example net is written

$$\lambda(a,a) = @(\lambda(b,b),c)$$

where the $=$ sign represents the connection between the principal port of the λ agent on the left-hand side and the principal port of the @ agent on the

right-hand side (i.e., the equation encodes an active pair formed by a λ agent and an @ agent). The left-hand side $\lambda(a, a)$ of the equation indicates that both auxiliary ports of this λ agent are connected and similarly for $\lambda(b, b)$.

We follow the same approach to represent rules, but this time we use the symbol $\bowtie$ instead of $=$. For example, the linear β-reduction rule is written

$$@(x, y) \bowtie \lambda(x, y)$$

This notation, being more concise, is more suitable for the implementation of interaction net systems. The textual calculus of interaction that we present below is based on these ideas.

7.6.1 A textual interaction calculus

In this section, we describe a textual calculus for interaction nets that gives a formal account of the reduction process.

Interaction nets are strongly confluent, but as in all reduction systems, there exist different notions of strategies and normal forms (for instance, irreducible nets, or weak normal forms associated with lazy reduction strategies). We will see that these can be precisely defined in the calculus. Such strategies have applications for encodings of the λ-calculus, where interaction nets have had the greatest impact, and where a notion of a strategy is required to avoid non-termination.

We begin by describing the syntax of the interaction calculus.

Agents: Let Σ be a set of symbols $\alpha, \beta, \ldots$, each with a given *arity* (formally, we assume that there is a function $\mathsf{ar}\colon \Sigma \to \mathsf{Nat}$ that defines the arity of each symbol). An occurrence of a symbol will be called an *agent*. The arity of a symbol corresponds precisely to its number of auxiliary ports.

Names: Let N be a set of names x, y, z, etc. N and Σ are assumed disjoint.

Terms: A term is built using agents in Σ and names in N. Terms are generated by the grammar

$$t ::= x \mid \alpha(t_1, \ldots, t_n)$$

where $x \in N$, $\alpha \in \Sigma$, and $\mathsf{ar}(\alpha) = n$, with the restriction that each name can appear at most twice in a term. If $n = 0$, then we omit the parentheses.

If a name occurs twice in a term, we say that it is *bound*; otherwise it is *free*. Since free names occur exactly once, we say that terms are *linear*.

We write $\vec{t}$ for a list of terms $t_1, \ldots, t_n$.

A term of the form $\alpha(\vec{t})$ can be seen as a tree with edges between the leaves if names are repeated; the principal port of α is at the root, and the terms $t_1, \ldots, t_n$ are the subtrees connected to the auxiliary ports of α. Note that all the principal ports have the same orientation, and therefore there are no active pairs in such a tree.

Equations: If t and u are terms, then the (unordered) pair $t = u$ is an *equation*. Δ, $\Theta, \ldots$ will be used to range over multisets of equations. Examples of equations include $x = \alpha(\vec{t})$, $x = y$, $\alpha(\vec{t}) = \beta(\vec{u})$. Equations allow us to represent nets with active pairs.

Rules: Rules are pairs of terms written as $\alpha(\vec{t}) \bowtie \beta(\vec{u})$, where $(\alpha, \beta) \in \Sigma \times \Sigma$ is the active pair of the rule (that is, the left-hand side of the graphical interaction rule), and $\vec{t}, \vec{u}$ are terms. All names occur exactly twice in a rule, and there is at most one rule for each pair of agents.

Definition 7.7 (Names in terms)

The set $\mathcal{N}(t)$ of names of a term t is defined in the following way, which extends to multisets of equations and rules in the obvious way.

$$\begin{array}{lcl} \mathcal{N}(x) & = & \{x\} \\ \mathcal{N}(\alpha(t_1, \ldots, t_n)) & = & \mathcal{N}(t_1) \cup \cdots \cup \mathcal{N}(t_n) \end{array}$$

Given a term, we can replace its free names by new names, provided the linearity restriction is preserved.

Definition 7.8 (Renaming)

The notation $t\{x \mapsto y\}$ denotes a renaming that replaces the free occurrence of x in t by a new name y. Note that since the name x occurs exactly once in the term, this operation can be implemented directly as an assignment, as is standard in the linear case. This notion extends to equations and multisets of equations in the obvious way.

More generally, we consider substitutions that replace free names in a term by other terms, always assuming that the linearity restriction is preserved.

Definition 7.9 (Substitution)

The notation $t\{x \mapsto u\}$ denotes a substitution that replaces the free occurrence of x by the term u in t. We only consider substitutions that preserve the linearity of the terms.

Note that renaming is a particular case of substitution. Substitutions have the following commutation property.

Proposition 7.10

Assume that $x \notin \mathcal{N}(v)$.

If $y \in \mathcal{N}(u)$, then $t\{x \mapsto u\}\{y \mapsto v\} = t\{x \mapsto u\{y \mapsto v\}\}$; otherwise $t\{x \mapsto u\}\{y \mapsto v\} = t\{y \mapsto v\}\{x \mapsto u\}$.

We now have all the machinery that we need to define nets in this calculus.

Definition 7.11 (Configurations)

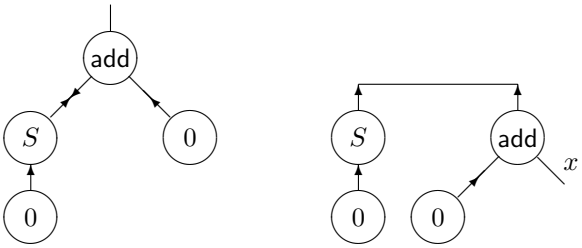
A *configuration* is a pair $c = (\mathcal{R}, \langle \vec{t} \mid \Delta \rangle)$, where $\mathcal{R}$ is a set of rules, $\vec{t}$ a sequence $t_1, \ldots, t_n$ of terms, and Δ a multiset of equations. Each name occurs at most twice in c. If a name occurs once in c, then it is *free*; otherwise it is *bound*. For simplicity, we sometimes omit $\mathcal{R}$ when the set of rules used is clear from the context. We use c, c' to range over configurations. We call $\vec{t}$ the *head* or *observable interface* of the configuration.

Intuitively, $\langle \vec{t} \mid \Delta \rangle$ represents a net that we evaluate using $\mathcal{R}$, and Δ represents the active pairs and the renamings of the net. It is a multiset (i.e., a set where elements may be repeated since we may have several occurrences of the same active pair). The roots of the terms in the head of the configuration and the free names correspond to ports in the interface of the net. We work modulo α-equivalence for bound names as usual. Configurations that differ only in the names of the bound variables are equivalent since they represent the same net.

There is an obvious (although not unique) translation between the graphical representation of interaction nets and the configurations that we are using. Briefly, to translate a net into a configuration, we first orient the net as a collection of trees with all principal ports facing in the same direction. Each pair of trees connected at their principal ports is translated as an equation, and any tree whose root is free or any free port of the net goes in the head of the configuration. We give below a simple example to explain this translation.

Example 7.12

The usual encoding of the addition of natural numbers (see Section 7.2) uses the agents $\Sigma = \{0, S, \mathsf{add}\}$, where $\mathsf{ar}(0) = 0$, $\mathsf{ar}(S) = 1$, $\mathsf{ar}(\mathsf{add}) = 2$. The diagrams below illustrate the net representing the addition $1 + 0$ in the "usual" orientation and also with all the principal ports facing up.



We then obtain the configuration $\langle x \mid S(0) = \mathsf{add}(x, 0)\rangle$, where the only port in the interface is x, which we put in the head of the configuration.

The reverse translation simply requires that we draw the trees for the terms, connect the common variables together, and connect the trees corresponding to the members of an equation together on their principal ports.

Definition 7.13 (Computation rules)

The operational behaviour of the system is given by the following set of computation rules:

Interaction: If $(\alpha(t'_1, \ldots, t'_n) \bowtie \beta(u'_1, \ldots, u'_m)) \in \mathcal{R}$, then

$$\langle \vec{t} \mid \alpha(t_1, \ldots, t_n) = \beta(u_1, \ldots, u_m), \Gamma\rangle \longrightarrow \langle \vec{t} \mid t_1 = t'_1, \ldots, t_n = t'_n, u_1 = u'_1, \ldots, u_m = u'_m, \Gamma\rangle$$

Indirection: If $x \in \mathcal{N}(u)$, then

$$\langle \vec{t} \mid x = t, u = v, \Gamma\rangle \longrightarrow \langle \vec{t} \mid u\{x \mapsto t\} = v, \Gamma\rangle$$

Collect: If $x \in \mathcal{N}(\vec{t})$, then

$$\langle \vec{t} \mid x = u, \Delta\rangle \longrightarrow \langle \vec{t}\{x \mapsto u\} \mid \Delta\rangle$$

Multiset: If $\Theta \rightleftharpoons^* \Theta', \langle \vec{t_1} \mid \Theta'\rangle \longrightarrow \langle \vec{t_2} \mid \Delta'\rangle, \Delta' \rightleftharpoons^* \Delta$, then

$$\langle \vec{t_1} \mid \Theta\rangle \longrightarrow \langle \vec{t_2} \mid \Delta\rangle$$

These rules generate a reduction relation $\longrightarrow$ on configurations. We denote by $\longrightarrow^*$ the reflexive and transitive closure of $\longrightarrow$.

The first rule, Interaction, is the main computation rule. When using this rule, we always apply an α-renaming to get a copy of the interaction rule with all variables fresh. Indirection and Collect are administrative rules that we use to obtain a more compact textual representation and to make explicit the active pairs that may be created after applying the Interaction rule. The symbol $\rightleftharpoons$ above denotes an equivalence relation that states the irrelevance of the order of equations in the multiset as well as the order of the members in an equation.

The calculus makes evident the real cost of implementing an interaction step, which involves generating an instance (i.e., a new copy) of the right-hand side of the rule, plus renamings (rewirings). Of course this also has to be done when working in the graphical framework, even though it is often seen as an atomic step.

Example 7.14 (Natural numbers)

We show two different encodings of natural numbers and addition using the interaction calculus. The first encoding is the standard one, and the second is a more efficient version that offers a constant time addition operation.

1. Let $\Sigma = \{0, S, \mathsf{add}\}$ with $\mathsf{ar}(0) = 0$, $\mathsf{ar}(S) = 1$, $\mathsf{ar}(\mathsf{add}) = 2$, and $\mathcal{R}$:

$$\begin{array}{lcl} \mathsf{add}(S(x), y) & \bowtie & S(\mathsf{add}(x, y)) \\ \mathsf{add}(x, x) & \bowtie & 0 \end{array}$$

   As shown in Example 7.12, the net for 1+0 is given by the configuration $(\mathcal{R}, \langle a \mid \mathsf{add}(a, 0) = S(0)\rangle)$. One possible sequence of reductions for this net is the following:

$$\begin{array}{l} \langle a \mid \mathsf{add}(a, 0) = S(0)\rangle \\ \quad \longrightarrow \langle a \mid a = S(x'), y' = 0, 0 = \mathsf{add}(x', y')\rangle \\ \quad \longrightarrow^* \langle S(x') \mid 0 = \mathsf{add}(x', 0)\rangle \\ \quad \longrightarrow \langle S(x') \mid x'' = x', x'' = 0\rangle \\ \quad \longrightarrow^* \langle S(0) \mid\rangle \end{array}$$

2. Let $\Sigma = \{S, N, N^*\}$, $\mathsf{ar}(\mathsf{S}) = 1$, $\mathsf{ar}(\mathsf{N}) = \mathsf{ar}(\mathsf{N}^*) = 2$. Numbers are represented as a list of S agents, where N is a constructor holding a link to the head and tail of the list. The number 0 is defined by the configuration $\langle N(x, x) \mid\rangle$, and in general n is represented by $\langle N(S^n(x), x) \mid\rangle$. The operation of addition can then be encoded by the configuration

$$\langle N(b, c), N^*(a, b), N^*(c, a)\rangle$$

   which simply appends two numbers. We only need one interaction rule

$$N(a, b) \bowtie N^*(b, a)$$

   which is clearly a constant time operation. To show how this works, we give an example of the addition of 1+1:

$$\begin{array}{l} \langle N(b, c) \mid N(S(x), x) = N^*(a, b), N(S(y), y) = N^*(c, a)\rangle \\ \quad \longrightarrow^* \langle N(b, c) \mid b = S(a), a = S(c)\rangle \\ \quad \longrightarrow^* \langle N(S(S(c)), c) \mid\rangle \end{array}$$

The interaction calculus is a Turing-complete model of computation, and therefore the halting problem (i.e., deciding whether a configuration produces an infinite reduction sequence) is undecidable in general. The following example shows that there are non-terminating configurations.

Example 7.15 (Non-termination)

Consider the net $\langle x, y \mid \alpha(x) = \beta(\alpha(y))\rangle$ and the rule $\alpha(a) \bowtie \beta(\beta(\alpha(a)))$. The following non-terminating reduction sequence is possible:

$$\begin{aligned} \langle x, y \mid \alpha(x) = \beta(\alpha(y))\rangle &\longrightarrow \langle x, y \mid x = a, \beta(\alpha(a)) = \alpha(y)\rangle \\ &\longrightarrow \langle a, y \mid \beta(\alpha(a)) = \alpha(y)\rangle \\ &\longrightarrow \cdots \end{aligned}$$

There is an obvious question to ask about this language with respect to the graphical formalism: Is it expressive enough to specify all interaction net systems? Under some assumptions, the answer is yes. There are in fact two restrictions. The first one is that there is no way of writing a rule with an active pair on the right-hand side. This is not a problem since it is possible to show that the class of interaction net systems where interaction rules are free of active pairs on the right-hand side has the same computation power as the class of rules that may include active pairs on the right-hand side. The second problem is the representation of interaction rules for active pairs without interface. In the calculus, an active pair without interface can only rewrite to the empty net. This is justified by the fact that disconnected nets can be ignored in this model of computation (only global computation rules can distinguish disconnected nets).

7.6.2 Properties of the calculus

This section is devoted to showing various properties of the reduction system defined by the rules Indirection, Interaction, Collect, and Multiset (see Definition 7.13). We have already mentioned these properties for the graphical formalism of interaction nets; they also hold for the calculus.

Proposition 7.16 (Confluence)

The relation $\longrightarrow$ is strongly confluent: If $c \longrightarrow d$ and $c \longrightarrow e$, for two different configurations d and e, then there is a configuration c' such that $d \longrightarrow c'$ and $e \longrightarrow c'$.

We write $c \Downarrow c'$ if and only if $c \longrightarrow^* c' \not\longrightarrow$. In other words, $c \Downarrow c'$ if c' is a normal form of c. As an immediate consequence of the previous property, we deduce that there is at most one normal form for each configuration: $c \Downarrow d$ and $c \Downarrow e$ implies $d = e$.

Although the calculus is non-terminating, as shown in Example 7.15, the restriction to the "administrative" rules Indirection and Collect is indeed terminating since applications of these rules reduce the number of equations in a configuration. Non-termination arises because of the Interaction rule (see Example 7.15), as expected.

7.6.3 Normal forms and strategies

Although we have stressed the fact that systems of interaction are strongly confluent, there are clearly many ways of obtaining the normal form (if one exists), and moreover there is scope for alternative kinds of normal forms, for instance those induced by weak reduction.

It is easy to characterise configurations that are fully reduced — we will call them full normal forms or simply normal forms. A configuration $(\mathcal{R}, \langle \vec{t} \mid \Delta \rangle)$ is in *full normal form* if Δ is empty or all the equations in Δ have the form $x = s$ with $x \in s$ or x free in $\langle \vec{t} \mid \Delta \rangle$.

We now define a weak notion of normal form, called the *interface normal form*, that is analogous to the notion of weak head normal form in the λ-calculus. This is useful in the implementation of the λ-calculus and functional programming languages to avoid non-terminating computations in disconnected nets.

Definition 7.17 (Interface normal form)

A configuration $(\mathcal{R}, \langle \vec{t} \mid \Delta \rangle)$ is in *interface normal form* (INF) if each t_i in $\vec{t}$ is of one of the following forms:

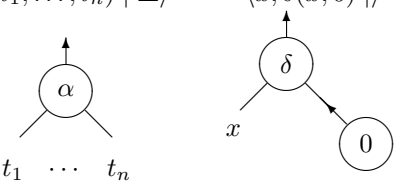
- $\alpha(\vec{s})$. For example, $\langle S(x) \mid x = Z \rangle$.
- x, where $x \in \mathcal{N}(t_j)$, $i \neq j$. This is called an *open path*. For example, $\langle x, x \mid \Delta \rangle$.
- x, where x occurs in a *cycle of principal ports* in Δ. For example, the configuration $\langle x \mid y = \alpha(\beta(y), x), \Delta \rangle$ has a cycle of principal ports (see the diagrams below).

Intuitively, an interaction net is in interface normal form when there are agents with principal ports on all of the observable interface, or, if there are

ports in the interface that are not principal, then they will never become principal by reduction (since they are in an open path or a cycle).

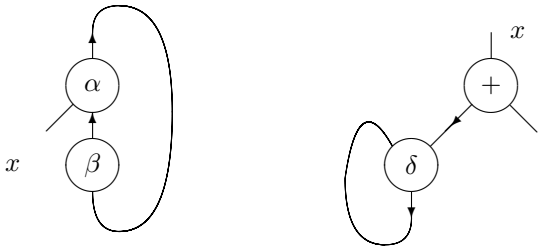
The following diagrams illustrate the notion of an interface normal form. The first diagram, a subnet in the configuration $\langle \alpha(t_1, \ldots, t_n) \mid \Delta \rangle$, has an agent α with a free principal port in the interface; the terms t_i connected to the auxiliary ports of α represent the rest of the net, and there may be active pairs in this net if Δ is not empty. The second net contains an open path (through the agent δ).

$\langle \alpha(t_1, \ldots, t_n) \mid \Delta \rangle$ $\qquad$ $\langle x, \delta(x, 0) \mid \rangle$



The following configurations are examples of nets with cycles of principal ports.

$\langle x \mid \alpha(\beta(y), x) = y \rangle$ $\qquad$ $\langle x \mid \delta(z, +(x, y)) = z, y = \ldots \rangle$



7.7 Extensions to model non-determinism

Interaction nets are a distributed model of computation in the sense that computations in a net can take place in parallel at any point in the net (no synchronisation is needed due to the strong confluence property of reductions in this model). However, interaction nets cannot model non-deterministic computations, which are a key ingredient of parallel programming languages.

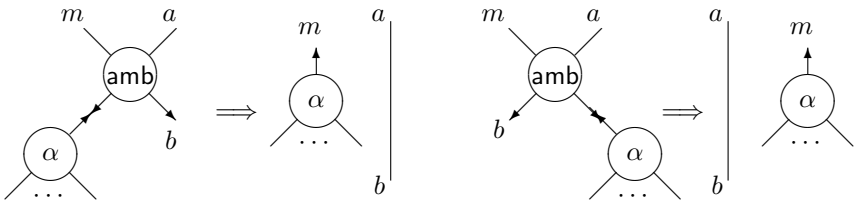
To obtain an abstract model of computation capable of expressing non-deterministic choice, several extensions of interaction nets have been proposed. For instance, we could extend interaction net systems by

1. permitting the definition of several interaction rules for the same pair of

agents, in which case one of the rules will then be chosen at random when the two agents interact;

2. permitting edges that connect more than two ports; or
3. generalising the notion of an agent in order to permit interactions at several ports (in other words, multiple principal ports are permitted in an agent).

The first alternative is simple but not powerful enough to model a general notion of non-determinism. The second and third alternatives are more powerful. In fact, in the third case, it is sufficient to extend the interaction net paradigm of computing with just one agent with two principal ports. This distinguished agent represents ambiguous choice and is usually called amb. It is defined by the following interaction rules.

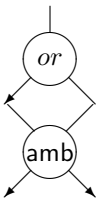


When an agent α has its principal port connected to a principal port of amb, an interaction can take place and the agent α arrives at the main output port of amb, which we called m in the diagram above. If in a net there are agents with principal ports connected to both principal ports of amb, the choice of the interaction rule to be applied is non-deterministic.

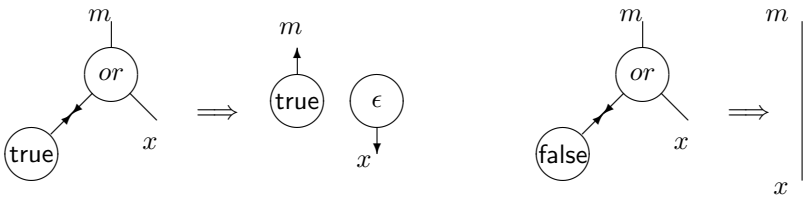
We illustrate the use of amb to program the *Parallel-or* function. This is an interesting Boolean operation that applies to two Boolean expressions and returns True if one of the arguments is True, independently of the computation taking place in the other argument. In other words, if both arguments have a Boolean value, this operation behaves exactly like an *or*, but even if one of the arguments does not return a value, as long as the other one is True, the Parallel-or function should return True. Since one of the arguments of Parallel-or may involve a partially defined Boolean function, the agent amb is crucial to detect the presence of a value True in one of the arguments. Below we specify this function using interaction nets extended with amb.

Example 7.18

The function *Parallel-or* must give a result True as soon as one of the arguments is True, even if the other one is undefined. Using an agent amb, we can easily encode *Parallel-or* with the net



where the agent *or* represents the Boolean function *or*, defined (in standard interaction nets) by two interaction rules:



The model of computation based on interaction nets extended with amb is strictly more powerful than the interaction net model in the sense that it allows us to define non-deterministic computations or non-sequential functions, such as Parallel-or.

In order to define parallel processes explicitly and facilitate the analysis of the behaviour of concurrent systems, in the next chapter we will present a formalism based on a notion of communication between processes.

7.8 Further reading

Yves Lafont's article [28] provides an introduction to interaction nets and many examples of their use. For more information on interaction combinators, and a proof of universality, we refer the reader to [29]. We refer to [42] for implementations of interaction nets. The compact textual notation for interaction nets described in this chapter was suggested by Lafont in his introductory article [28]; the calculus based on this notation is developed in [16]. We also refer the reader to [16] for more notions of normal forms and strategies of evaluation.

7.9 Exercises

1. Using interaction nets, define the following functions on numbers represented with 0 and S (successor):
   - is-zero, which produces a result True if the number is 0 and False otherwise;
   - min, which computes the minimum of two numbers;
   - factorial, which computes the factorial of a number.
2. Specify an interaction system that generates infinite computations (loops).
3. Complete the definition of the interaction system for combinatory logic given in Section 7.3. More precisely, define the agents and rules needed to define the S combinator (it can be defined with three agents and three rules).
4. a) Give an interaction system to compute the Boolean function *and*.
   b) Draw the interaction net representing the expression

   $$(True\ and\ False)\ and\ True$$

   How many reductions are needed to fully normalise this net?
   c) Modify the system so that the result is $True$ if and only if both arguments have the same value (i.e., both $True$ or both $False$).
5. Give a representation of lists in interaction nets, and use it to implement a function that interleaves the elements of two given lists. More precisely, define an interaction system that, given two lists l_1 and l_2, produces a new list containing the elements of l_1 interleaved with those of l_2. For instance, the result of interleaving [0, 2, 4] and [1, 3] is the list [0, 1, 2, 3, 4].
6. Textual rules defining addition were given in Example 7.14. Can you write the textual version of the rules for multiplication given in Section 7.2?
7. Explain why interaction nets are not suitable as a model for nondeterministic computations.
8. Define the function Parallel-and using the agent amb. Parallel-and is a binary Boolean operator returning the value False whenever one of the arguments is False and True when both are True.