Nordic Journal of Computing 2(1995), 407–443.

INTERACTION DIAGRAMS

JOACHIM PARROW
Department of Teleinformatics
Royal Institute of Technology
Electrum 204, S-164 40 Kista
Sweden
joachim@it.kth.se

Abstract. Interaction diagrams are graphic representations of concurrent processes with evolving access capabilities; in particular they illustrate the points of access and relations between them. The basic step of computation is the migration of an access point between processes. This paper explains interaction diagrams through a sequence of examples. Diagrams can be regarded as graphic counterparts of terms in the π-calculus and illuminate some interesting points on its construction.

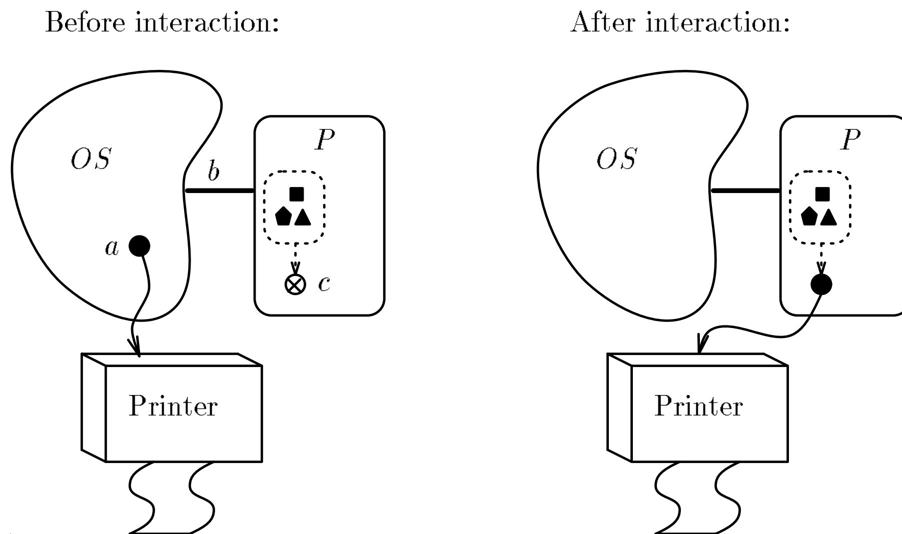
CR Classification: F.1.1, F.1.2, F.3.3

Key words: Concurrency, Graphic representation, π-calculus, Access migration.

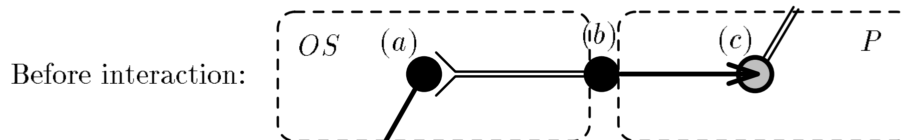
1. Introduction

This paper will explore *interaction diagrams* as graphic presentations of concurrent processes with emphasis on how interaction capabilities evolve; the main postulate is that a process may gain communication links by interacting with other processes. Such situations are common in operating systems and other concurrent systems with a dynamic distribution of shared resources. Consider the figure next page where a user process P needs to print a set of data. Since P lacks access to a printer its data is queued at c. In order to proceed P interacts, over the link b, with the operating system (OS) which controls a printer through a. As a result of this interaction P gains access to the printer and can transmit data to it directly. We see that a has two roles here: in the interaction between P and the operating system it is an *object* transported between processes; in an interaction with the printer it is a *port*, or a communication link, over which other objects can be transported.

Received August 8, 1994; revised July 21, 1995; accepted September 15, 1995.



An interaction diagram is a kind of graph where the access points and links (such as a, b and c above) are vertices called *locations*. These can function both as ports and objects. The edges between locations are called *arrows* and determine which objects can be sent or received along which ports. The following fragment of a diagram captures the essence of the example:



The diagram says that OS can transmit the object a, along the port b, to P which will store the received object in c. All of a, b and c are locations, but c is shaded differently to highlight the fact that it represents a placeholder for something to be received. There are two kinds of arrows: an "output" arrow meaning "send a on b", and an "input" arrow meaning "receive into c from b". These arrows combine to an *interaction* since they share the port b. There are two other arrows from a and c, here only indicated as short stumps, connected to other regions of the diagram: the arrow from a leads to the printer, and the arrow into c comes from the data to be printed. These arrows use a and c as ports.

When the interaction along b occurs it coalesces a and c, creating a location with the remaining arrows of both of them—the intuition is that a is injected into c, dragging remaining arrows with it.

There is now a new opportunity for interaction using the coalesced vertex as a port; in the example it will have an input (leading to the printer) and an output (coming from the data to be printed). Thus interaction diagrams, like Petri nets or data flowgraphs, represent concurrent activities providing for each other, but they additionally show how the links between processes migrate when processes execute.

A disclaimer is now in order to qualify my ambitions with the informal exposition in the present paper. Interaction diagrams will not be developed into an algebraic theory here, nor will they be put forth in a complete and immutable graphic language. It would no doubt be interesting and rewarding to do so—a formalisation is not unduly difficult and can be done in a few slightly different ways depending on the intended use—but the main purpose of this paper is to demonstrate examples and underlying ideas, using the graphic representation to facilitate understanding. To stress this intent I use the term "diagram" rather than "graph". Among the related algebraic theories there are already several versions of the π-calculus (see Milner *et al* [1992]), and I hope that this paper also works partly as a tutorial, after which the reader can understand the formalistic presentations with more ease. The calculi focus interest on *operators* such as parallel composition, while interaction diagrams provide insight in the nature of the objects manipulated by the operators. The relationship between interaction diagrams and π-calculi is thus comparable to that between transition diagrams and traditional process algebras: algebras define ways to construct objects hierarchically while diagrams provide intuition, form a basis for analysis algorithms, and can be developed into formal models.

In Sections 2 through 6 below the basics of interaction diagrams will be presented, and in Sections 7 through 10 these will be used to illustrate a sequence of examples, culminating with an encoding of the λ-calculus. In Section 11 related and further work are briefly discussed. A few remarks enclosed in square brackets contain comparisons with the π-calculus syntax and semantics, but the calculus will not be formally defined here, and an uninterested reader may skip them without loss of continuity. [*The example above is represented in the π-calculus as $\overline{b}a \mid b(c)$.*]

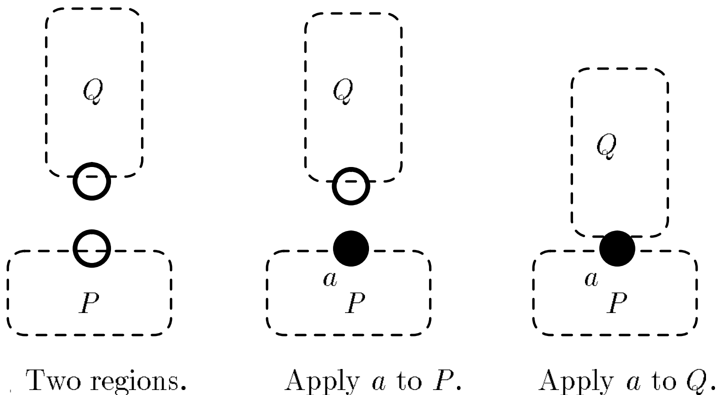
2. Locations and Arrows

Locations are the vertices in an interaction diagram and are depicted as circles; there will be a few different kinds of them as explained in the following. The basic form is a *free* location, represented as a filled circle with a name tag next to it:

Intuitively, this represents a global link or access point, known to the rest of the world by the name in its tag. An interaction diagram may have several free locations each bearing a unique name.

It will often be convenient to construct diagrams hierarchically, starting from small regions and combining them into larger diagrams. Sometimes a location may therefore represent a link to a not yet determined region, and it clarifies matters to give such locations a special status as *parameters*. A parameters does not carry a name tag (because we have not decided yet which name to give it) and is depicted as an unfilled circle:

In one sense these just act as placeholders, or "holes", which can be filled with other locations. Thus a diagram with a parameter potentially represents a class of diagrams, namely those which can be obtained by filling the hole. The "hole filling" operation is called *application*. To see how it works, consider the following example with two regions, each with one parameter:



Two regions. Apply a to P. Apply a to Q.

The effect of applying the (free) location a to both parameters is to give the previously disjoint regions a common location. This inherits all arrows connected to the two parameters, so P and Q can now interact using a as a port. Conversely, given a diagram we may "punch a hole" at one of the free locations to convert it to a parameter (this also removes the name tag). The hole punching operation is called *abstraction*, and a parameter is therefore sometimes called an *abstracted location*. A region may have several parameters. In the examples to follow we will consistently place the parameters on the perimeter of the region and distinguish them by their relative positions.

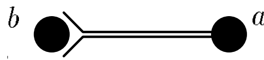
Application is one way to fill the holes of an abstraction, where we ourselves — the constructors of the diagrams — are the agents doing the filling. Another way is to let the filling agent be part of the diagram, i.e., a value for the parameter is received as input from another location. This is how a diagram can construct (or at least "rewire") itself. Graphically it is represented by drawing an *input* arrow, leading from a location (the *port* of the

input) to a parameter (the *target*); the parameter is thereby transformed to a *located parameter*, or *input*. This can then no longer be filled by application (nor located a second time)—it is so to speak "reserved" by the input arrow—but it does not yet represent a concrete location; this motivates a graphical representation as a circle neither unfilled nor completely filled:

and an input arrow leading to a parameter located at a, meaning "input from a", is

where the arrow intentionally goes into the circle to suggest that the circle will be filled through it. Input locations are those with an incoming input arrow so shading the circle is technically redundant, but the convention to reserve unfilled circles for parameters and filled ones for concrete locations helps to get an overview of complex diagrams. In the example the port is a free location, but any kind of location—parameters or even other inputs—can function as ports. Note that inputs have *exactly* one incoming input arrow, since a parameter can only be located once.

The "output" of a location onto a port is depicted by an *output* arrow



connecting the *source* b (representing the location to be transmitted) to the port a. The source and port can be any kind of location. A location can be the source of several outputs, and the port of several outputs and inputs; the only restriction when forming diagrams is that an input location has exactly one incoming input arrow.

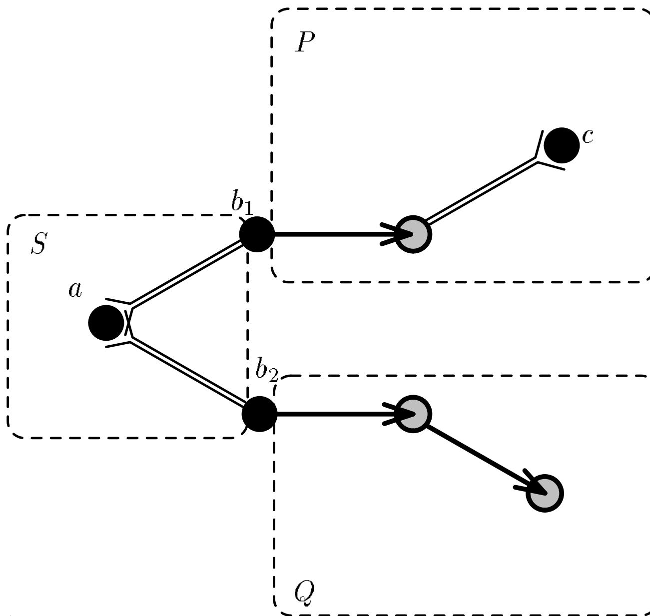
[*In the* π*-calculus free locations correspond to free names, input locations to names bound by the input prefix, and parameters to* λ*-abstracted names. Abstracting on* a *in* P *is written* λaP*, and applying* b *to* P *is written* Pb*. The arrows correspond to prefixes; input arrows to the input prefix* $a(x).P$ *where* a *is the port and* x *the target, and output arrows to the output prefix* $\overline{a}b.P$ *where* a *is the port and* b *the source. The prefixes additionally imply a temporal connection: the input must be received, or the output transmitted, before any interaction in* P *can occur.*

There are interaction diagrams without counterparts in the π*-calculus. An example is an abstraction located at itself (i.e., an input arrow leading back to the port). The precise expressive power of the diagrams and of various versions of the* π*-calculus with different schemes for expressing temporal orderings are interesting fields of further investigations.*]

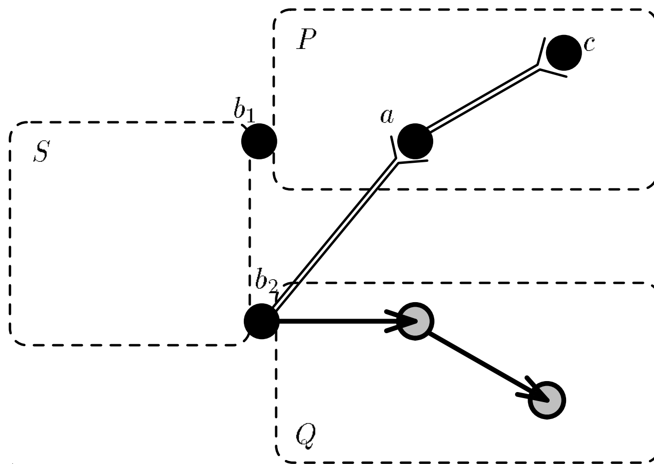
3. Reduction

A region of a diagram consisting of an output arrow and an input arrow on the same port, together with the source and target locations of these arrows, is called an *interaction*. The *reduction* of this interaction means that the arrows are removed and the source and target are coalesced, keeping any remaining arrows. The actual placement of the coalesced vertex is unimportant; although intuition suggests it should inherit the position of the target the remainder of the diagram may sometimes be more clearly drawn if it is placed elsewhere.

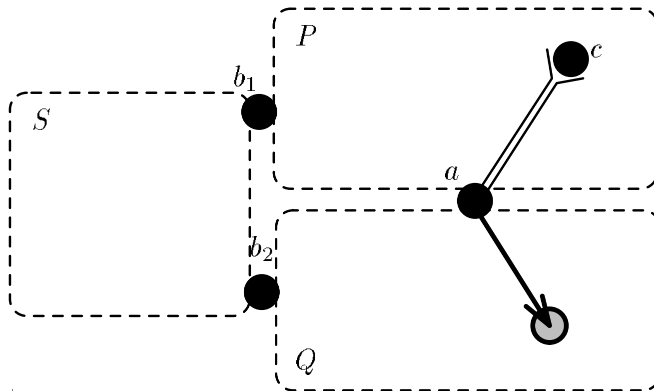
A canonical example of reduction was given in the introduction. For another simple example demonstrating that reductions may indeed increase communication capabilities, consider a situation where a process P wishes to send c to Q and that P and Q are not in direct contact so the communication cannot be performed immediately. Suppose further that they both communicate with a third process S on b_1 and b_2. For example, P and Q may be running under an operating system S, which can send both P and Q a shared locus of interaction a. The situation is depicted in the following diagram:



Here there are two interactions (along b_1 and b_2) and reducing the one along b_1 results in:



where P has received a and there is still an interaction along b_2. Reducing it gives:



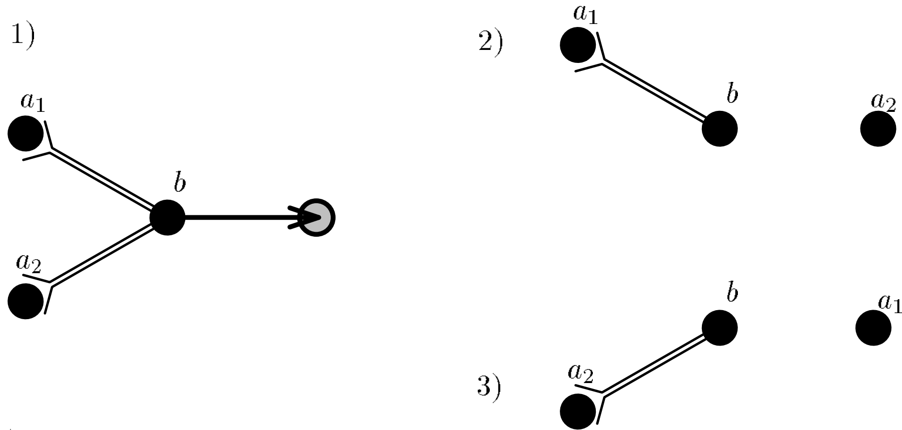
Now P and Q share the location a (and so are logically closer to each other), and c can be transmitted along it without further intervention by S. The locations b_1 and b_2 remain, but if they have no other arrows they will not be able to play any role in further interactions and can be removed.

In these diagrams we have outlined the processes (P, Q, S) with dashed boxes, but after the first interaction (along b_1) it is not obvious which process should be regarded as possessing a. It may be more intuitive to let it be shared between S and P if it is S (rather than P) that sends it to Q. This can be accomplished by placing a at a boundary of both S and P.

In general a location can be shared between many processes and have arrows in several remote regions; then the regions may become confusingly contorted. To remedy this we will sometimes use "improper" diagrams where

a free location is depicted at several places. This is merely a drawing convention and represents the "proper" diagram obtained by coalescing all places with the same name.

Reducing an interaction means that arrows are removed and this may imply that other interactions are preempted. For example in the diagram 1) below



there are two interactions sharing an input arrow, one involving a_1 and one involving a_2. Reducing either of them will remove the input arrow and thus preempt the other interaction, resulting in 2) or 3). In a way these interactions "compete" for transmission along b. Similarly two interactions may compete through a shared output arrow.

When the port or the source of an interaction is not a free location, it is not intuitively clear that the interaction should be reducible. If the source is an input it represents an object not yet received, and it is reasonable to delay the interaction until something is available for transmission. Similarly, if the port is an input it is reasonable to delay until the "vehicle" of interaction is available. This entails a principle of *provision*: An interaction can reduce only if it is fully provided for, i.e., neither the source nor the target is an input. So the following are irreducible interactions:

but they can become reducible if another reduction provides them with locations for the port or the source. It should be noted that proponents of so called "lazy" execution strategies would not necessarily adhere to the principle of provision, arguing that an interaction can reduce if its dependence on the yet unavailable object (as in the rightmost interaction above) is recorded.

When the port or the source is a parameter it does not represent a dependency *within* the diagram, rather it represents a diagram at an incomplete

stage of construction. A viable position is to altogether disallow reductions in diagrams with parameters, on the grounds that a diagram must be fully instantiated before we can talk of its computational behaviour. But since the dependency is here external to the diagram it will be harmless to admit such reductions, these can be thought of as *subjunctive* or hypothetical reductions in the sense that they tell us how the diagram would execute were we to instantiate it.

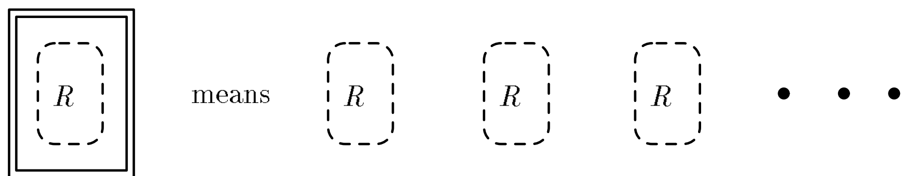
In summary, a reduction can make possible further reductions in two ways: first by the coalition of nodes which may create new interactions, and second by providing for previously irreducible interactions; it can also preempt other reductions by removing shared arrows.

[*In the* π*-calculus a reduction corresponds to a* τ*-transition arising from an input prefix and an output prefix in different parallel components. In analogy with term rewriting systems the pair of prefixes is sometimes called a "redex". In its original form the* π*-calculus enforces the principle of provision since in an input prefix* $a(x).P$ *no action in* P *can occur before the action implied by* $a(x)$*. Also,* λaP *is a kind of agent for which no transitions are defined so there are no subjunctive reductions. Current research, notably the "action structures" of Milner [1993], seems to indicate that these principles can be relaxed.*]

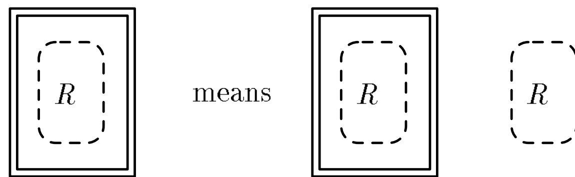
4. Replication and Recursion

The size of an interaction diagram (the number of nodes and arrows) is always reduced by a reduction. Consequently, only a finite number of reductions will ever be possible until it reaches an inert state. This severely limits the usefulness of the diagrams in areas like operating systems, where most behaviours are non-terminating. To overcome this limitation we need some regeneration mechanism. In this paper we will use *recursion*, where a diagram may recursively be part of itself, and *replication*, where a diagram can generate an unbounded number of copies of itself. Either of these can be defined in terms of the other, so if we are interested in a minimal formalism only one of them needs be taken as primitive.

Replication is drawn as a double box around a region of a diagram and stands for an infinite number of copies of that region. In pictures,



Replication thus is a finitary way to represent infinite diagrams. A reader uncomfortable with completed infinities may instead wish to think of the replication box as possessing the ability to spawn copies of its contents:

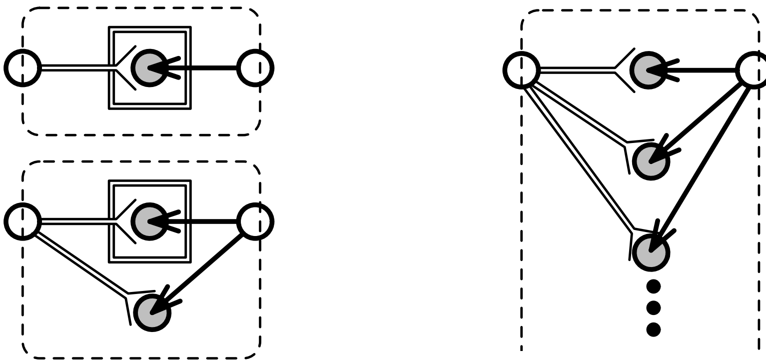


(This shows that replication can be defined through recursion.) We will avoid parameters within replication boxes since it would be strange to have a diagram with an infinite number of parameters (it would take an infinite amount of work to instantiate them). Otherwise the replication box may contain an arbitrary number of locations and arrows. Note that when free locations are replicated their names are not changed, so no new free locations are really created. They only get to occupy more places, resulting in an "improper" representation, and should be coalesced into one if we want a proper diagram. Thus a box does not affect free locations (although arrows connected to them are replicated).

Input and output arrows may cross the boundary of a replication box, such arrows must then also be replicated (since one end point is replicated). The only restriction is that an input arrow cannot lead out of a replication since the corresponding input location outside the box would then get several input arrows from different replicas.

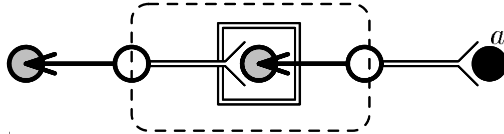
Reductions of diagrams with replication boxes work as before with the exception that they may not change the contents of any replication box. Any such change would correspond to an infinite series of changes, and thus require infinitely many execution steps. Instead we expand the box at need and conduct the reductions on the replicas. In this way the box can be viewed as a "template" or "process generator".

As an example, consider an unbounded and unordered buffer which repeatedly forwards values from one location to another; the following three representations are equivalent (but the third representation cannot be exhibited on a finite sheet of paper):

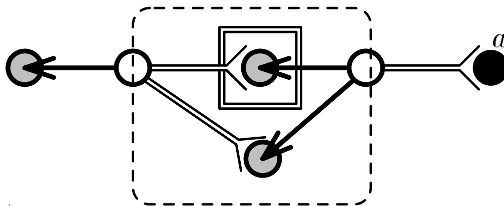


When something is output along the rightmost parameter it will be transported to a replica of the box, i.e., an input location, and from there be

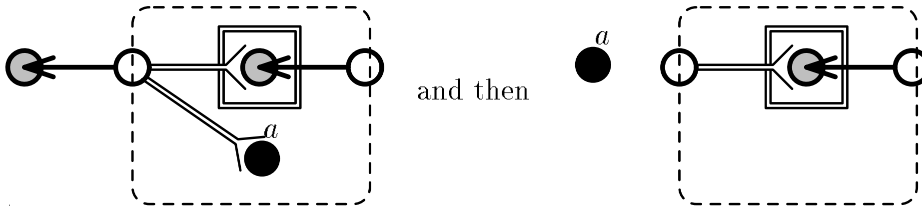
available for transmission along the leftmost parameter. As indicated by the third representation the buffer can store unboundedly many items, and does not preserve temporal ordering of arrival. When we use it as a region in a diagram:



none of the interactions may be reduced since they are part of the box. However, expanding the box we see that two interactions appear, of which one is irreducible by provision but one is reducible:

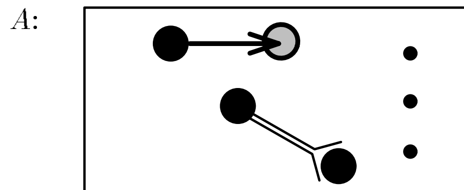


Reducing it we get

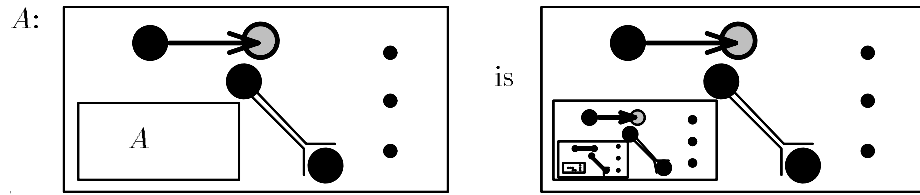


and we see that the buffer is back in its original state, after having transported a from right to left.

To use recursion with interaction diagrams we introduce a graphical notation for recursive definitions and invocations. Let diagram *labels* be a new syntactic class. A *definition* of a label A is drawn as “A: $\boxed{\cdots}$” where the box may contain an arbitrary diagram, for example:

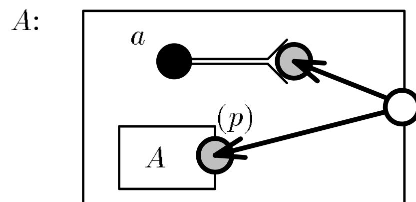


This binds the diagram in the box to the label A. An *invocation* of a label A is simply a box with the label written within it, $\boxed{A}$, this stands for a copy of the corresponding definition. Of course invocations may also be used within definitions, this is what gives the possibility for recursive definitions, as in:

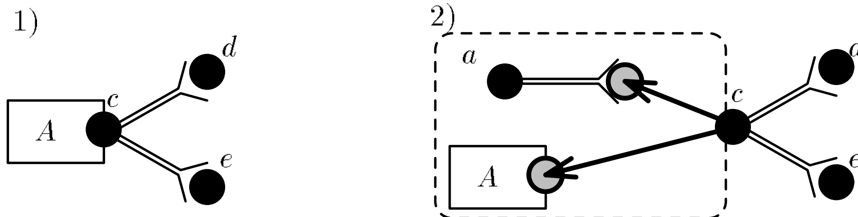


where the rightmost diagram can be fully represented on a finite sheet of paper only if we can print with unlimited resolution.

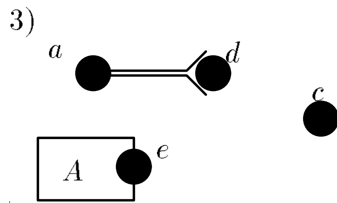
Reducing diagrams with invocations is unproblematic; an invocation can at any time be expanded by its definition. Thus a recursive definition functions as a replication but there is one difference: a replication can have arrows into and out from the box. To recover a possibility of interaction between an invocation and other regions we let a recursive definitions carry *formal parameters* displayed as unfilled circles at its perimeter. In an invocation the corresponding places must be instantiated by locations, the *actual parameters* of the invocation, and these may carry arrows to other locations. An example with one formal parameter is the following:



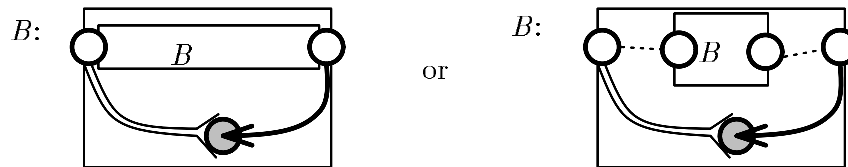
A has one parameter on which it will receive two things: an item to be forwarded on a and something sent to the actual parameter (p) of the recursive invocation. For example, see what happens when we invoke it with an actual parameter c on which we output d and e. The diagram is given in 1); expanding the invocation gives 2). There are now two ways to reduce along c, one outcome is 3) where the invocation has reappeared, now instantiated with e and with an extra "output d on a". The other possible outcome is like 3) but with d and e interchanged.



3)



In general a definition can have several formal parameters, these are identified by the position on the perimeter. An actual parameter can be any kind of location, in particular it can be a formal parameter of an enclosing definition. With this device recursion and replication are equipotent and the choice of which to use is determined by convenience and clarity. For example, the buffer can now be exhibited as:



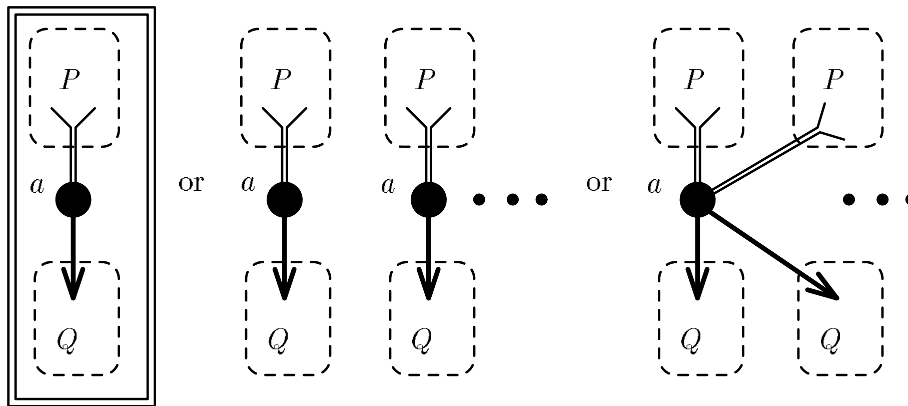
In the right hand version the dotted lines mean "same as" and allow an "improper" representation where each parameter occupies more than one place (just as a free location can occupy more than one place, by virtue of its global name); in some examples this drawing convention will improve the layout.

In summary we use four different kinds of boxes in diagrams: dashed boxes to indicate regions, single line boxes for recursive definitions, ditto for invocations, and double boxes for replications. Since definitions and regions both carry parameters we could replace all dashed boxes with single line boxes (thus letting each region constitute a definition), but we will continue to reserve definitions for the situations where recursive invocations are present.

[*In the π-calculus replication is written "!", for example the buffer would be written* $\lambda r \lambda l\, !r(d).\bar{l}d$. *The corresponding recursive formulation is* $B(r,l) \stackrel{\text{def}}{=} (r(d).\bar{l}d) \mid B(r,l)$. *For technical reasons no free names are admitted in recursive definitions, so all names must be formal parameters. It is usual practice to only admit* guarded *recursion; this means that recursive invocations must lie under a prefix. Similarly, guarded replication means that a prefix must begin the replicated agent. This practice means that although recursion and replication stand for infinite agents, only a finite number of interactions are available at any given time. We will see how guards can be represented in diagrams in Section 6.*]

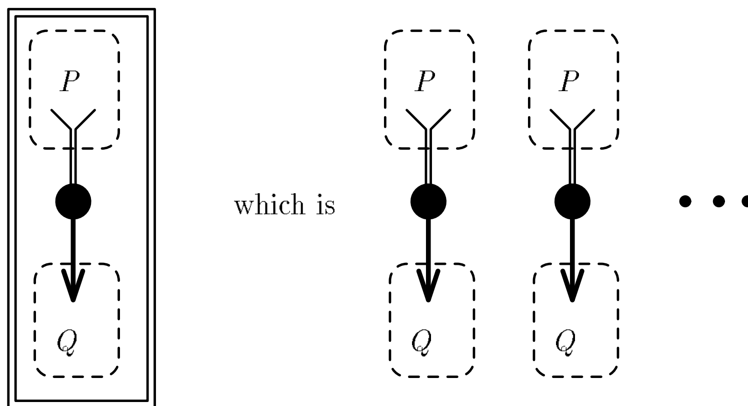
5. Private Locations

Free locations carry global names with a universal scope, and relying exclusively on them leads to a lack of modularity. This is particularly severely felt with the use of replication since all replicas use the same set of free locations. Consider for example a system consisting of a producer P and consumer Q communicating through a location a, and assume that we want several copies of this system. Using replication we get the following (the three representations are equivalent since a is a free location):



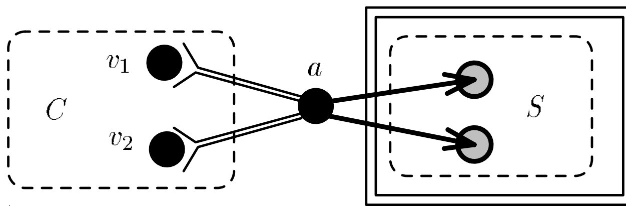
Now assume that each P instead wants to send several values along a to Q. Since all replicas use the same a, there is a possibility that the values from one P will be sent to different replicas of Q.

To let each P be certain that it always communicates with the same instance of Q we introduce a new kind of location: a *private* location, which functions and looks like a free location but does not bear a name tag, and cannot be abstracted or applied. When used inside replication boxes the private locations will be truly replicated, as seen when modifying the above example:

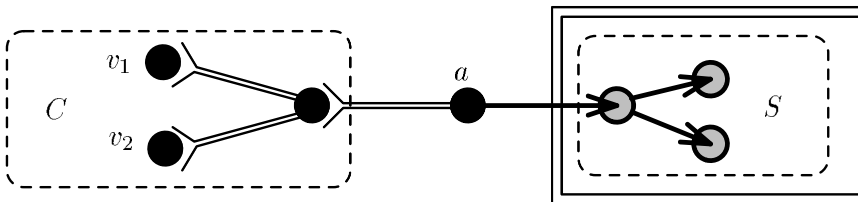


We now have four different kinds of locations: parameters, free locations, input locations, and private locations. One way to gain symmetry is to consider parameters to be the most basic form (or "ur-location") and think of three different ways to gain more specialised kinds: we can *apply* a free name to it to get a free location, or *locate* it on a port through an input arrow to get an input location, or *protect* it to get a private location. These kinds differ in how they relate to the rest of the diagram: a free location may be referred to through its name, an input location may be instantiated through its port, while a private location is guaranteed to be inaccessible from the outside. It can still function as a source in an interaction and thus coalesce with an input location, but no amount of reduction, abstraction and application can coalesce it with a free location, parameter, or another protected location.

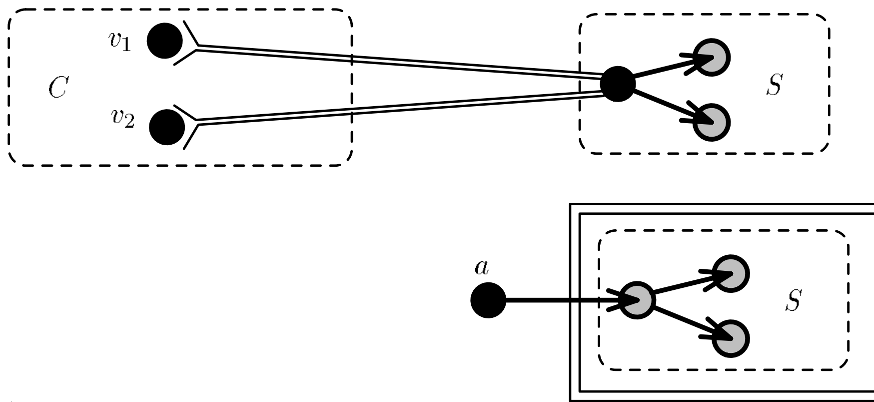
For another example consider a Server S, with an access point a, ready to receive *two* values for further processing from any Client. The Server is generic so many Clients can use it simultaneously, but the values from different Clients must not be mixed. The fact that the Server is generic can be represented by a replication box, but care must then be taken that values from the same Client do not end up in different replicas. Clearly the following picture is inadequate



since, when we expand the box, there are several interaction along a and these have targets in different replicas. Making a a private location will not improve the situation in this respect. Instead, a better way is the following:



Here C begins by transmitting, along a, a private location. After one reduction the replica of the Server can proceed to receive v_1 and v_2 along the private location (see the diagram below). This works even if there are other Clients concurrently served by S along a, since each Client begins by emitting its unique private location.

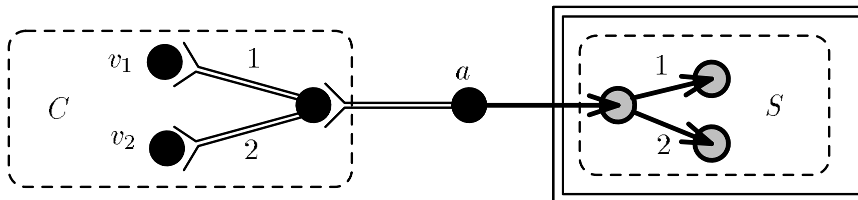


[The π-calculus operator for a private location is $(\boldsymbol{\nu} x)P$ or in early work $(x)P$, this introduces the name x as protected (or restricted) in P and is analogous to the CCS operator $\backslash x$. Formally, this is a binding operator just like input prefix. The Client–Server example can be written

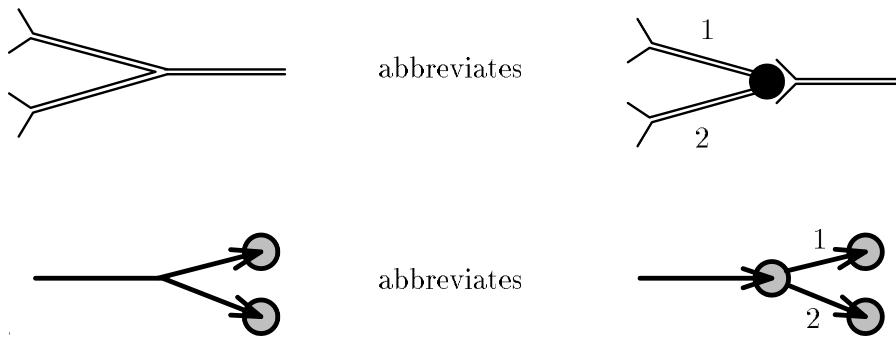
$$C = (\boldsymbol{\nu} l)(\overline{a}l \,|\, \overline{l}v_1 \,|\, \overline{l}v_2) \qquad S = !a(x).x(y).x(z)]$$

6. Polyadic Arrows and Guards

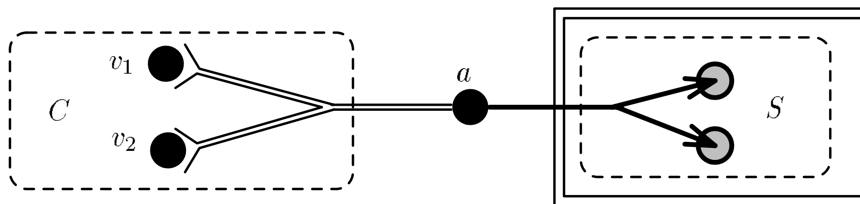
A snag with the Client–Server example in the previous section is if S is supposed to treat the two incoming values differently, since it is not determined which of v_1 and v_2 is transmitted first. Here there is a clear need for a mechanism to impose temporal dependencies between reductions, and we shall look at a few possibilities. The conceptually simplest way is to decorate the arrows with numbers which indicate in which order they should be reduced. Thus a numbered arrow may not be reduced if there is an arrow with a lower number in the diagram. The example becomes:



Here, after the reduction along a, first v_1 and then v_2 will be transmitted along the private location. This way to transmit a pair of values (or, in general, a tuple of values) by first establishing a private location as a vehicle of communication and then transmitting the values in order is sometimes called a "molecular" interaction. It is so useful that we have a special abbreviation for it by using multi-arrows:

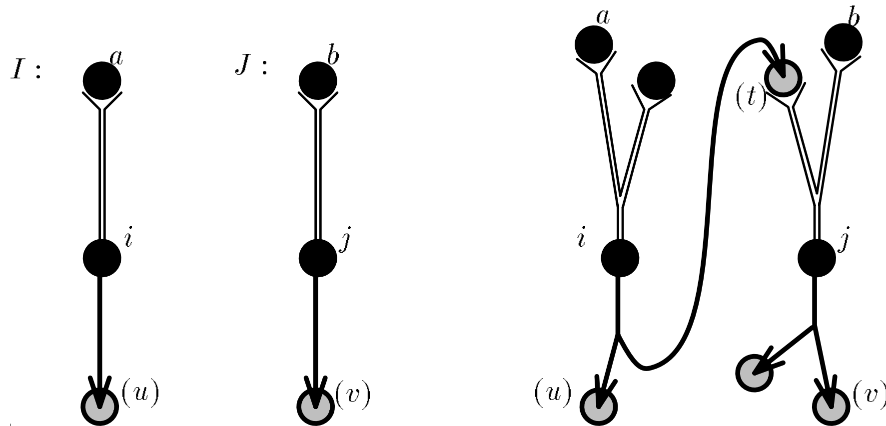


The notation easily generalises to any arity. A molecular interaction consists of an input arrow and an output arrow of the same arity on the same port, and its reduction coalesces all sources with the corresponding targets. With this abbreviation the Client–Server example becomes:



The numbering mechanism is quite powerful since through it we can enforce a temporal dependency between any arrows. But it may be unsatisfactory in that it is inherently "global": in order to determine whether a reduction can take place we must consider the whole diagram to ensure that no temporally preceding arrows are present. It is therefore worth looking for alternatives. One interesting such is to adopt the multi-arrows as *primitive* (rather than derived) constructs representing *polyadic* outputs and inputs, where several objects are transmitted simultaneously. A polyadic interaction is considered provisioned only when all its sources are provisioned. If we let the figure above represent a polyadic interaction then both v_1 and v_2 must arrive in the Client before the pair of them can be transmitted to the server.

With polyadic arrows we can easily encode arbitrary temporal sequencing between interactions as follows. Assume that the reduction of an interaction I must precede an interaction J. Increase the arities of both these interactions by one, and let the thus obtained extra target in I be the extra source in J (the other two extra locations are unimportant). Then J is irreducible by provisioning until I reduces. An example of this construction when I and J are unary:

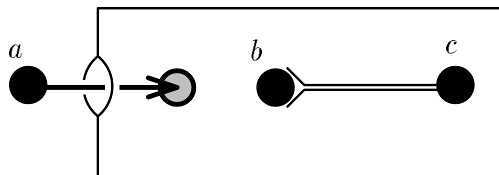


Here (t) is the new target of I and source of J; until it is provisioned by I it is impossible for J to reduce.

Conversely, polyadic arrows can be defined (as molecular arrows) using the numbering scheme. Therefore numbering and polyadicity are equally powerful but polyadic arrows have the advantage of being more "local": in order to determine if a reduction is possible it is only necessary to examine the involved arrows and locations.

When we adopt polyadic arrows we have to define the consequences of pathological situations where arrows of mismatching arities share a port, or ensure that such situations cannot arise. The latter alternative seems the most useful and gives rise to "sortings" of locations (where the sort of a location, roughly, is the tuple of the sorts of locations that can be passed along it). We will not pursue sortings further here, but assure the reader that mismatching arities will not surface in any example to come. In fact, for the rest of the paper it will not matter whether the multi-arrows are molecular (i.e., derived) or polyadic (primitive).

Another way to enforce temporal dependencies is to use *guards*. A guard is a region of a diagram with a distinguished arrow. No other arrow in the region is allowed to participate in a reduction until the distinguished arrow has reduced. In this sense that arrow blocks all reductions within the region. We draw a guard as a box with an opening through which the distinguished arrow passes, e.g,:



A reduction involving the arrow through the opening means that the box itself (but not the region inside it) disappears. The interior of the box is then free to reduce. So in the example above the output along c cannot happen until an input on a arrives. In general the region within a guard

may be arbitrarily large, and the distinguished arrow may be either an input or an output.

Clearly, through the use of guards we can achieve the effect of the numbering scheme in molecular actions by enclosing the numbered arrows in guards. Conversely, using polyadic actions we can ensure that all interactions within the guard are provisioned by a reduction involving the distinguished arrow, by increasing the arity of these interactions (see the construction at the end of Section 11). In conclusion there are several ways to achieve temporal dependencies, and for our purposes they are equally powerful. It will no doubt be an interesting enterprise to determine more exactly how they relate formally, but in the rest of this paper we will focus on examples related to the migration of locations. We will therefore primarily use polyadic arrows.

[*The Client–Server example with molecular interaction in the π-calculus is written*

$$C = \boldsymbol{\nu} l(\overline{a}l.\bar{l}v_1.\bar{l}v_2) \qquad S = !a(x).x(y).x(z)$$

The transmission of tuples of names as a primitive construct is studied at depth in the polyadic *versions of the calculus, first presented in Milner [1991]. There, the same example would be written:*

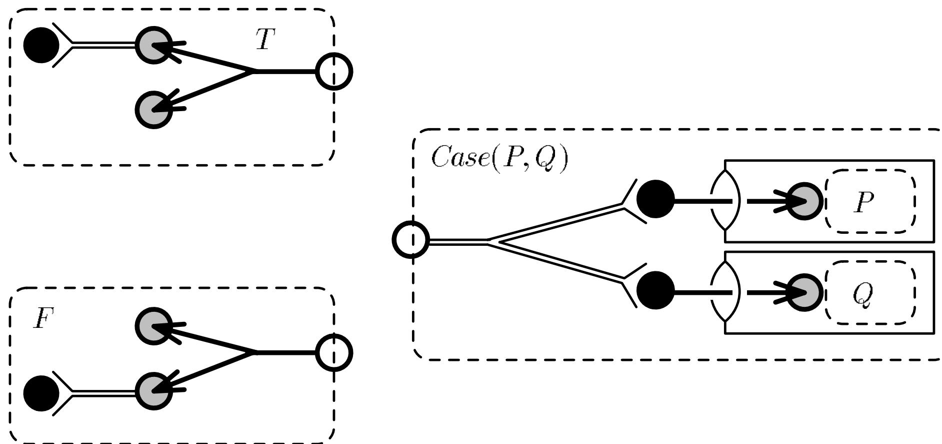
$$C = \overline{a}\langle v_1 v_2\rangle \qquad S = !a(yz)$$

Sorts and sortings have been studied by Gay [1993] and Pierce and Sangiorgi [1995]. Guards are ubiquitous in the calculus since in most dialects every prefix implies a temporal precedence between its action and its body. A notable exception is the asynchronous calculus of Honda and Tokoro [1991].]

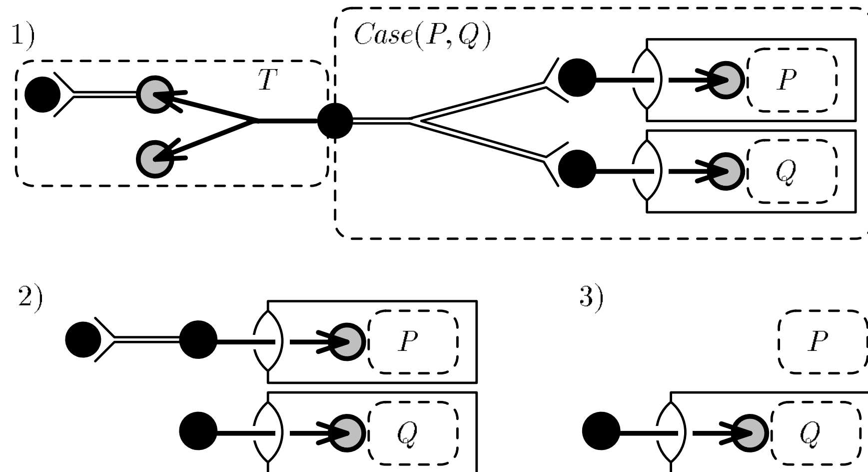
7. Finite Data Domains

In the previous section we alluded to a Client emitting "values" to a Server, but we did not go into the nature of the values—they were just schematically represented as free locations. For practical purposes we may want to extend the diagrams to contain data values such as booleans and lists, and functions over these. But it is interesting to see that data and functions can be encoded as diagrams without any extension to the basic machinery. The idea is to let each value and function be represented as a small diagram with some distinguished parameters, "handles", on which it interacts with the rest of the world. Transmitting a value, or even a function, then amounts to transmitting its handles.

We begin with values and computations over *finite* data domains, so assume a domain $\{1, \ldots, n\}$. A value k is coded as a small diagram with one parameter on which it will declare its identity in the following way: it first receives n locations (one for each value in the domain), and then outputs a private location on the kth of these, signifying that it represents value k. As an example we will here only consider the booleans but the principle generalises easily. The domain is then $\{T, F\}$ and the encodings are as follows:



Here is also shown a *Case* construct to demonstrate how the boolean values can have an effect on reductions. *Case* has one parameter which should be connected to its argument (a boolean value); if that value is T then $Case(P,Q)$ behaves as P, if the value is F then it behaves as Q. The example where $Case(P,Q)$ is connected by a shared private location to T is shown below.

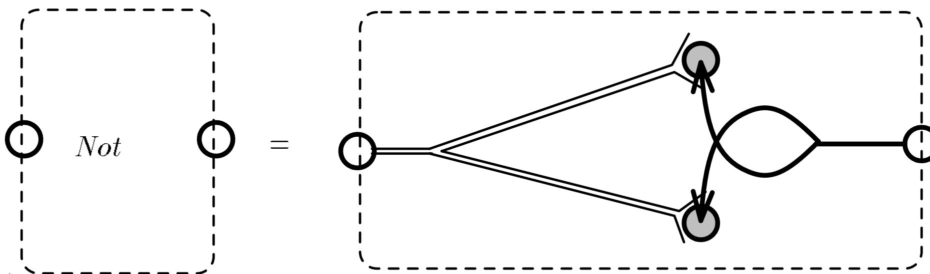


Case begins by sending two private locations to the argument. It then awaits an output on either of those. Here the argument is T, so as shown in 2) the upper input arrow will become part of an interaction. Reducing it gives 3) and means that P receives the private location from T, thereby reducing the guard. But Q is now in a situation where it can never receive anything on the input, so its guard will never be reduced and hence no reduction within Q will be possible. In situations where we are only interested in the computational behaviour of diagrams we will freely remove, or "garbage collect", such inert regions.

In this example we connected the regions *Case* and T by identifying (and protecting) the parameters. This entails a static view of how diagrams

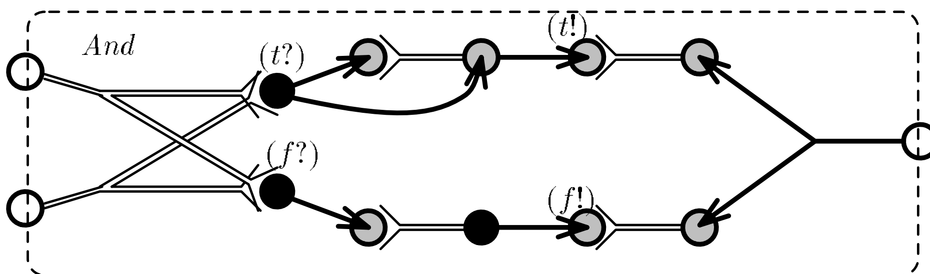
are built. But the parameters can be instantiated and then transmitted through other parts of a diagram, and this entails more dynamic view. The parameters are in this way "handles" to the values, and the values can be regarded as moving with their handles since they have no other connections with the rest of the diagram.

We can also define diagrams performing computations over the values. Consider for example the diagram *Not* below. It has two parameters: the left hand parameter is connected to its argument and the right hand parameter is the place where it will enact the result, i.e., act as the boolean complement of its argument.



The effect is achieved by commuting the order in the molecular (or polyadic) actions. The right hand (result) side will transmit to *Not* two locations, and expects *Not* to hit one of them to signal "True" or "False". So *Not* just forwards the locations to the argument in reversed order. Interestingly, this implies a "lazy" evaluation, since the argument will not be activated until *Not* receives an indication from the right hand side that a result is desired.

As a slightly more complex example consider the encoding of boolean conjunction below. It has two argument parameters on the left and one result parameter on the right. The idea for *And* is to act as T if both its arguments are T, otherwise act as F. In contrast to *Not* it is of an "eager" variety and will begin by activating the arguments before a result is called for. It is also "non-strict" in the sense that if one argument is F then *And* will act as F towards the result, even if the other argument delays it response.



And first sends two private locations, indicated by $(t?)$ and $(f?)$ in the diagram (note that these are just points of reference for the following explanation and not name tags) to both arguments, and then awaits their responses.

As soon as something is output on $(f?)$—from either of the arguments—the interaction in the lower chain will be provisioned and its reduction will place a (private) location in $(f!)$, thus the diagram will act as F towards the result parameter. The upper interaction will not be fully provisioned until *two* locations are received on $(t?)$, then it will be reducible and place a location at $(t!)$.

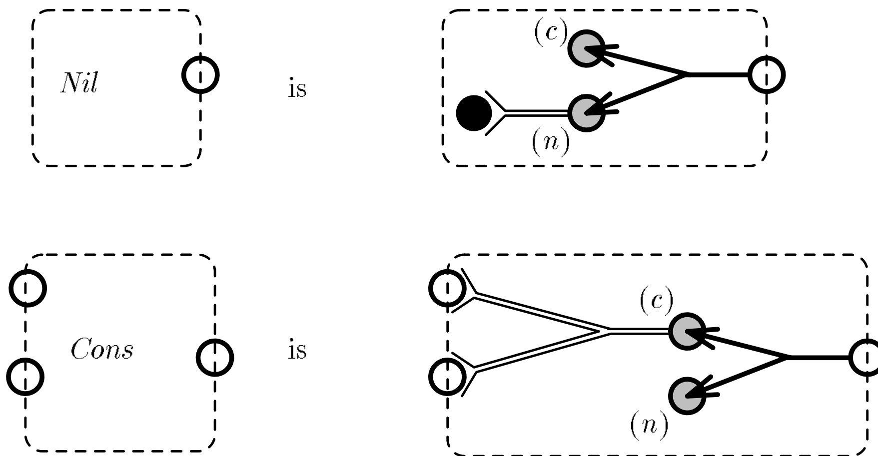
The alert reader will have noticed that the encoding can be optimised by locating $(f!)$ directly at $(f?)$, i.e., drawing the input arrow from $(f?)$ to $(f!)$, thus saving two locations (but destroying the symmetry).

Combining *And* and *Not* we can obtain arbitrary boolean functions. The interested reader is invited to construct more examples, for example strict or lazy versions of other functions like equality. It is an indication of the expressive power that not only can arbitrary functions be defined but also arbitrary evaluation strategies. The parameters, or handles, of the functions can of course be transmitted between different regions of a diagram just as other locations. Thus we can describe dynamic systems where not only values but also functions on values are mobile.

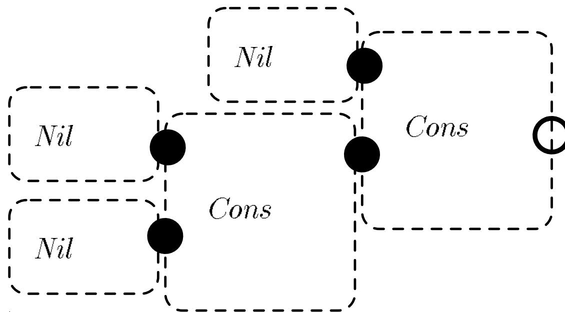
8. Infinite Data Domains

The scheme from the previous section is obviously restricted to finite domains, since the interaction with a value begins by sending it a representation of the entire domain in the form of one location for each possible value. If we want to consider infinite data domains a value can instead present itself piece by piece—each piece of the value is one out of finitely many, and the pieces can be interlinked through arrows or shared locations.

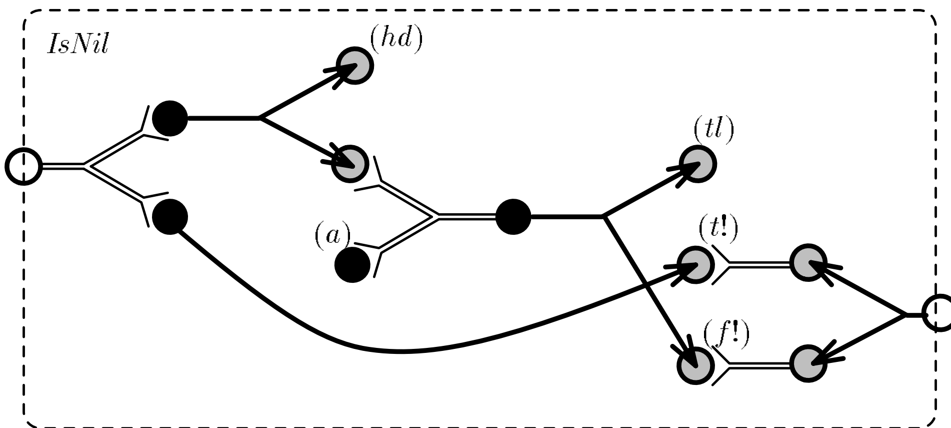
As an example we here concentrate on a simple variety of list structures. They are built out of just two constructors: the binary *Cons* and the nullary *Nil*. In analogy with the booleans, a list therefore receives two locations, call them (c) and (n), on its parameter. If it is *Nil* it will hit (n) with a private location, if it is $Cons(X, Y)$ it will hit (c) with the handles of the encodings of X and Y, its head and tail.



Note that in this encoding *Nil* is the same as *False*, but *Cons* is different from *True* since it has three parameters. The two leftmost are to be instantiated with the locations corresponding to the parameters of its head and tail. Thus a list is a diagram with one parameter, on which it will gradually reveal its identity. For example, the list *Cons*(*Nil*, *Cons*(*Nil*, *Nil*)) is



So just as booleans, each list can be represented by its parameter, and this can be transmitted in interactions. We can now encode functions on lists as diagrams. As an example, the function *IsNil* has one argument and one result parameter; it will act towards its result as T if the list is *Nil* and as F if it is distinct from *Nil*, i.e., begins with a *Cons* cell.



The diagram will begin by sending two private locations to the argument. If the argument is *Nil* it will respond on the second location, firing the lower (curved) input arrow which will provision $(t!)$ and enable the diagram to act as T. If the argument begins by *Cons* it will send the head and tail locations through the first location. This provisions a binary interaction sending a private location (a) to $(f!)$, enabling the diagram to act as F. Note that the head and tail locations from *Cons* end up in (hd) and (tl) respectively; these are not further used by *IsNil*.

It is tempting to optimise by sending the head (hd) directly to $(f!)$, but this would break or convention for the booleans: a boolean is supposed to respond by emitting an *anonymous* private location, without any connecting

arrows. For example, our encoding of *And* in the previous section uses the anonymity: one of the received locations will be used as a port, and it is then important that this has no other arrows—if so there may arise other interactions whose reductions preempt the required interactions in *And*. Now the location ending up in (hd) is the handle of a list and may have outgoing input arrows. Therefore, care must be taken to fill $(f!)$ with the anonymous private location (a).

There are of course several alternatives to this encoding. One is to use an explicit temporal precedence (or guarding) between the reception of the head and the provisioning of $(f!)$ instead of the binary interaction. Another is to redefine the encodings of the booleans so that the anonymity of the private location is not critical.

While these examples show that interaction diagrams have a high expressive power, they also reveal that they suffer from the weakness of any low-level programming formalism: if we want to use high-level concepts such as lists, we must define the encodings exactly. The choice of encoding depends on which functions we want to compute efficiently. It is then a comfort that diagrams can be constructed hierarchically in a nice way. As an example we will show how *ListEq* for determining structural equality of lists is constructed from some other functions (Fig. 1 on next page).

The diagram takes on an appearance of a recursive data flow network; *ListEq* is a recursive definition with three parameters, two on the left for the lists to be compared, and one on the right where it will enact the result. It begins by testing both lists for emptiness through a modified *IsNil* which emits the head and tail of a nonempty argument on two additional parameters on the upper side. This modification of *IsNil* is obtained by connecting each of the locations (hd) and (tl) through an output arrow to a new parameter.

ListEq calls itself recursively twice to compare these heads and tails. The results of the comparisons are collected by a few boolean functions. It is important that the boolean functions are non-strict, for example if both lists are empty the rightmost *Or* in F will perceive T at one argument but nothing at the other.

In F we use a region *BDup* to duplicate a boolean value. This is necessary since in our encoding a boolean value is destroyed when it is tested — in the next section we shall see an alternative encoding where the values are more robust. The definition of *BDup* is straightforward through the *Case* construct: just let its two branches each contain *two* copies of T and F respectively.

The regions are mainly interconnected by shared private locations. In cases where it is awkward to place these on the appropriate boundaries we again resort to dotted lines meaning "the same as". Note that *IsNil* is connected to the recursive invocations of *ListEq* via input arrows rather than shared locations; this is because *IsNil* transmits the head and tail *through* its parameters rather than enacting them *on* the parameters.

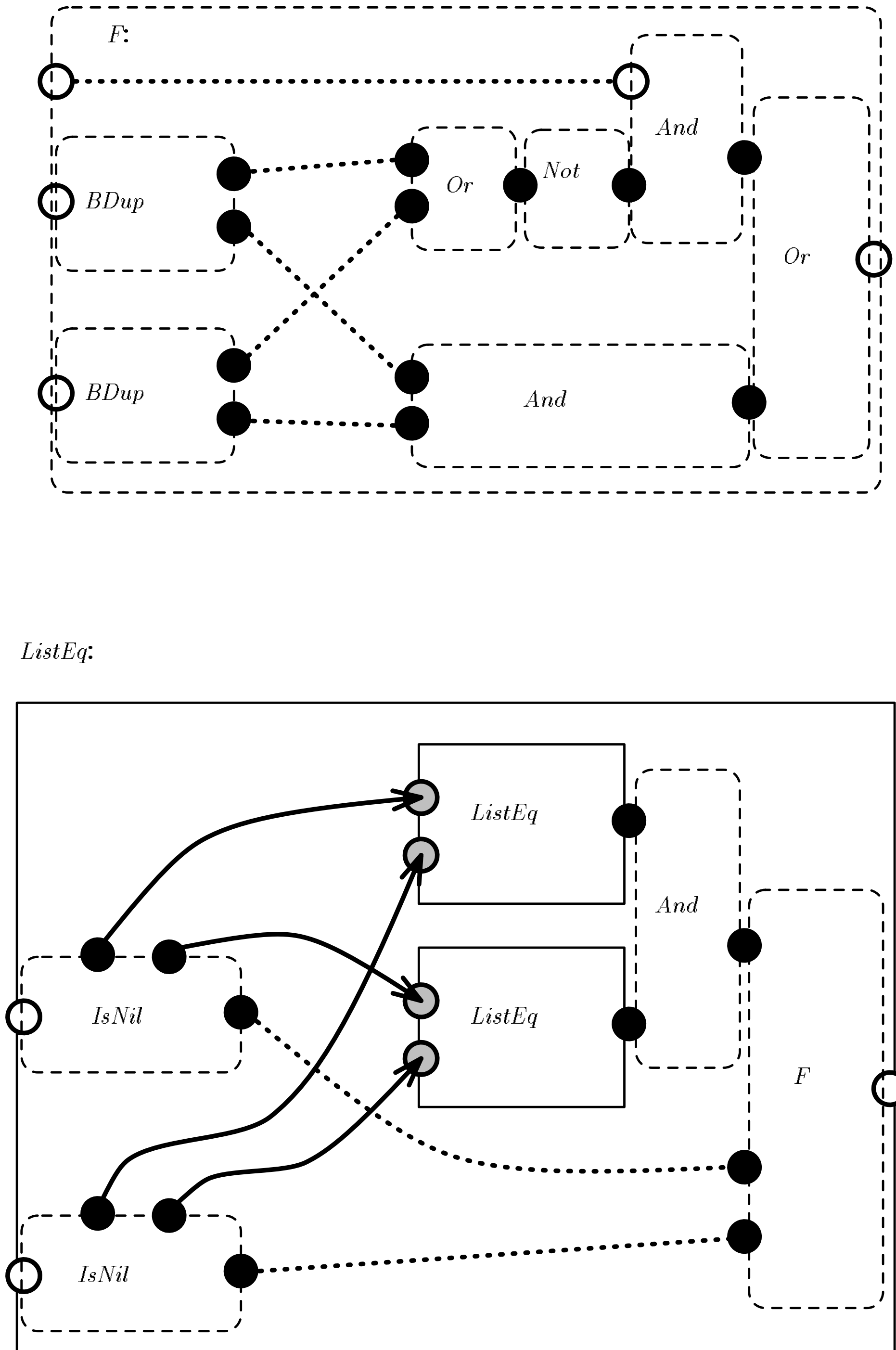
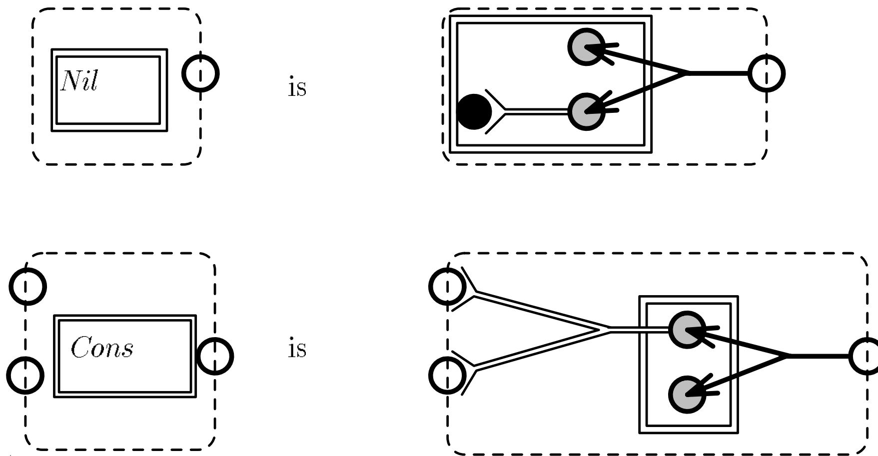


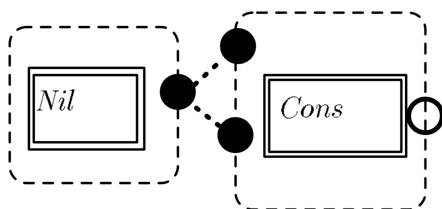
Fig. 1: List equality

9. Shared Data

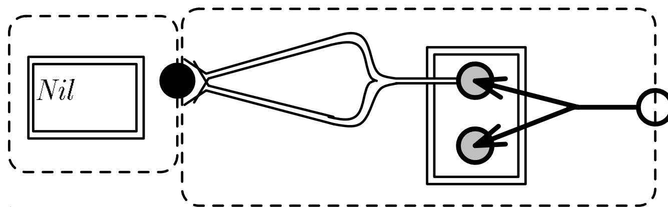
In the encodings so far the data values have been "ephemeral" in the sense that examining them once—as for example done by one of the functions—destroys them. This is unproblematic if we want to represent purely functional computing. In concurrent computing, however, we may want parallel activities sharing the data. In particular the data values must then tolerate more than one examination. Rather than repeatedly introducing duplicator functions such as *BDup* of the preceding section we can use a more "persistent" encoding where the values replicate as soon as they are examined. Returning to the example of lists, such an encoding is:



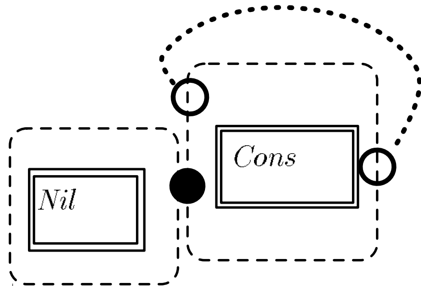
When a cell is queried for its contents it receives, as usual, two locations which are sent to a *replica* of the box. The cell is then ready to accept more queries. Thus sharing is a meaningful concept: a cell can be part of several lists. For example the list *Cons*(*Nil*, *Nil*) can be given as



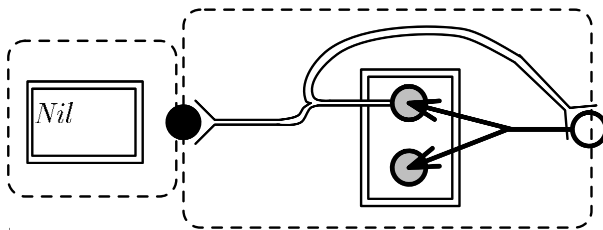
or equivalently, by expanding *Cons*:



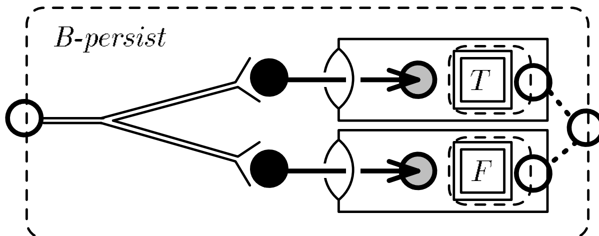
Without the replication in *Nil* this would be incorrect since one cell could then not enact both head and tail. In the same way arbitrary shared structures can be represented, even circular ones, like the list $L = Cons(L, Nil)$:



or equivalently



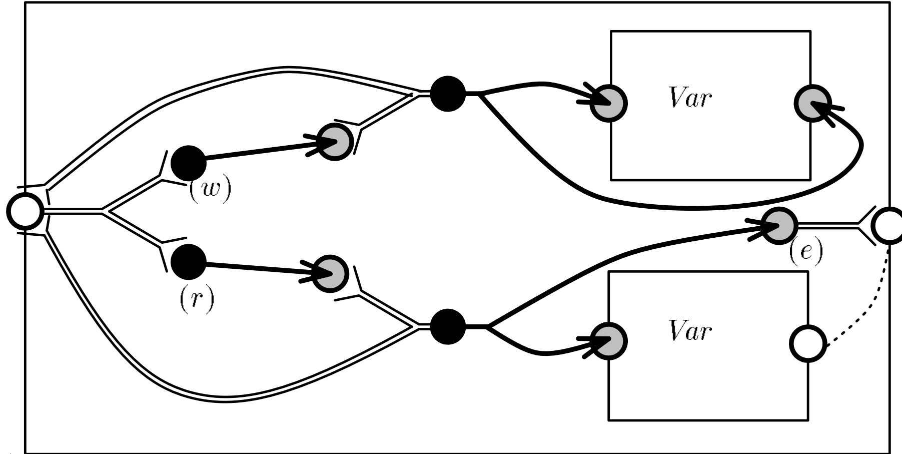
The functions on Booleans and lists in the preceding sections require modification in order to return persistent encodings. This can be achieved by diagrams which convert between the encodings, such as the following:



B-persist expects a boolean value to its left and will enact the persistent encoding of it to the right.

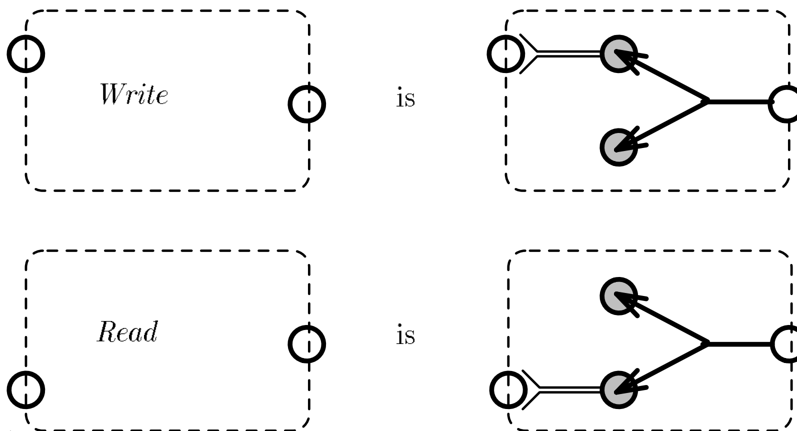
The most pronounced effect of persistent values is in scenarios where values can be updated. None of the encodings until now have any such ability but it can easily be implemented by giving them additional parameters over which they accept "updating" commands. As a generic example of an updatable entity we consider an untyped variable *Var* from an imperative programming language. It has two parameters; the leftmost is the access point of the variable where users interact to perform retrieval and updating, and the rightmost connects to the value stored by the variable (we assume nothing about this value, except that this parameter represents its "handle").

Var:

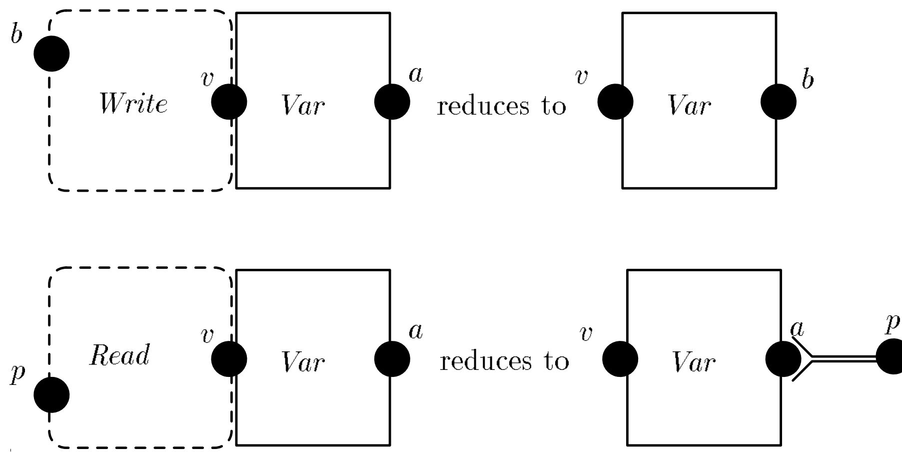


An operation on the variable begins by the transmission, on its access parameter, of two private locations (w) and (r). The intention is that whoever uses the variable can employ them to signal "write" and "read" respectively. The user does a write by sending the new value along (w). The upper half of the diagram corresponds to the write operation, which sends this new value into a parameter of the recursive invocation. The user does a read by sending, along (r), a location on which the current value should be emitted. The lower half corresponds to the read operation, which sends this location to (e).

After a read or write, the diagram recurs. Observe that the left hand parameter (the access point of the variable) is unchanged by a read or write, and is transmitted to the actual left hand parameter of the recursive invocation. For the disciplined use of a variable we can define small *Write* and *Read* diagrams:



Through a sequence of reductions we then obtain that



Here a and b are handles to values, such as booleans or lists. There are no reductions along a or b, these values are only transmitted by the variable and not examined. Consequently the variable will work for any data type.

It may appear tedious to have to describe intuitively clear objects such as lists and variables in this detail, but the point is that any data type, or any variation of variable (for example in logic programming the operations on variables include unification) can be defined. We can combine variables with lists to get lists of variables, and we can get a variable whose value is another variable (by connecting its value parameter with the access parameter of the other variable), even a variable whose value is itself (by identifying the two parameters of the variable). In short, the variables can be seen as a special data type where the access parameters are the handles. Variables can be transmitted between different regions (cf the "call by reference" parameter passing mechanism in imperative programming), and their scope can be restricted by protecting their handles (this makes it impossible for anything outside the region to access the variable). Many of the fundamental paradigms of imperative and object-oriented programming seem possible to represent in this way.

[*The reader may have noticed the absence of* π*-calculus comparisons in the last sections. This is because the encodings given here differ from encodings in the* π*-calculus, at least as presented until now. The reason is that in the* π*-calculus the temporal precedence implied by prefix can make some of the encodings shorter (some of the polyadic interactions which only serve to enforce temporal precedence can be removed). But the basic principles are the same. A* π*-calculus semantics of an object-oriented language is given by Walker [1995]. It is interesting to see that we can achieve the same effect without explicit temporal sequencing.*]

10. The λ-calculus

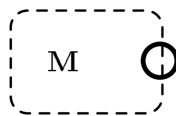
As a final example we shall consider β-reduction in the λ-calculus. This is a calculus where terms represent higher-order functions (that is, they take functions as parameters), and it is expressively complete in the sense that all computable higher-order functions can be represented in it.

To avoid confusion with the π-calculus we will represent terms of the λ-calculus in boldface and let $\mathbf{M}, \mathbf{N}$ range over λ-terms. They are built from a set of variables ranged over by $\mathbf{x}$, $\mathbf{y}$, …and just two combinators: abstraction over a variable in a term, $\boldsymbol{\lambda}\mathbf{x}\mathbf{M}$ which binds $\mathbf{x}$ in $\mathbf{M}$, and application of a term to another term, $\mathbf{MN}$. The basic computational step is the β-reduction:

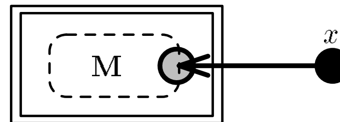
$$(\boldsymbol{\lambda}\mathbf{x}\mathbf{M})\mathbf{N} \longrightarrow_\beta \mathbf{M}[\mathbf{x} := \mathbf{N}]$$

where $\mathbf{M}[\mathbf{x} := \mathbf{N}]$ is the term obtained by replacing every free occurrence of $\mathbf{x}$ in $\mathbf{M}$ with $\mathbf{N}$. At first it may seem a trivial task to encode this into diagrams since abstraction and application are already present there. But in the λ-calculus these mechanisms are significantly more powerful; they imply the replacement of a variable with a whole *term*, whereas in the diagrams we just replace a parameter with a *location*.

The coding trick, in analogy with the previous sections, is to give each term a handle in the form of a parameter. β-reductions are mimicked by reducing interactions, but instead of transporting an entire argument, these interactions transport a private location that *represents* the argument through an input arrow to its handle:



A λ-term $\mathbf{M}$

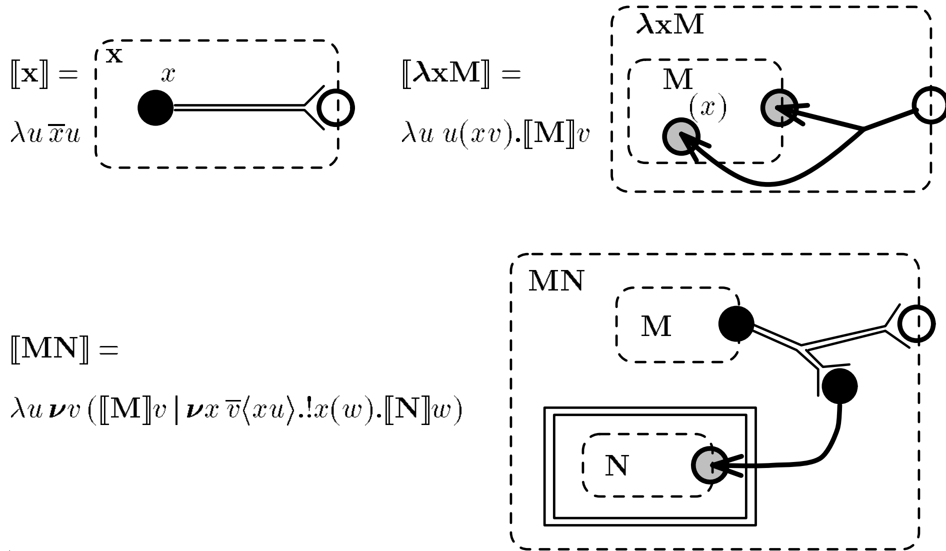


$\mathbf{M}$ "represented by" x

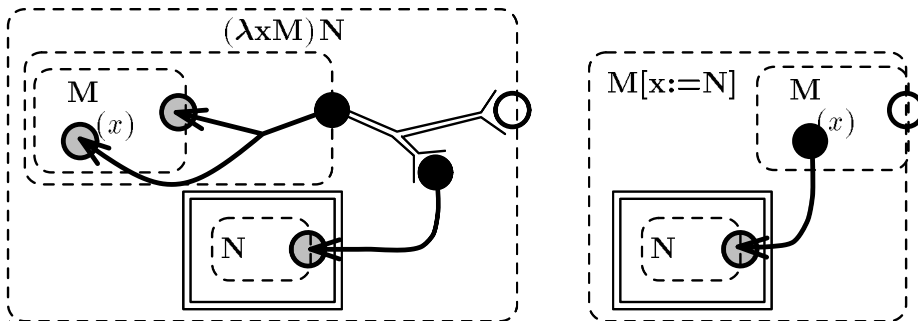
When this x is received as an object in an interaction it can be used to invoke $\mathbf{M}$, by sending (representatives of) arguments along x. Note that $\mathbf{M}$ is replicated so it can be invoked in this way several times—this is necessary since the argument in a λ-calculus application may occur several times in the reduct, as in $(\boldsymbol{\lambda}\mathbf{x}\,\mathbf{xx})\mathbf{M} \longrightarrow_\beta \mathbf{MM}$

The complete encoding is given below. To each λ-variable $\mathbf{x}$ we correlate a distinct location, this will be a free location x where $\mathbf{x}$ is free, i.e., not bound by an enclosing $\boldsymbol{\lambda}\mathbf{x}$, and an input location where $\mathbf{x}$ is bound. The encoding of the λ-term $\mathbf{x}$ will just output its parameter on x; this will activate something represented by x. The encoding of $\boldsymbol{\lambda}\mathbf{x}\mathbf{M}$ will receive two things on its parameter: first something which will make the correlate (x) of $\mathbf{x}$ represent an argument, and second the parameter for further interaction with its environment. (If $\mathbf{M}$ has no free $\mathbf{x}$ then (x) is a new location.) Conversely, the encoding of $\mathbf{MN}$ will send to $\mathbf{M}$ (through its parameter) two

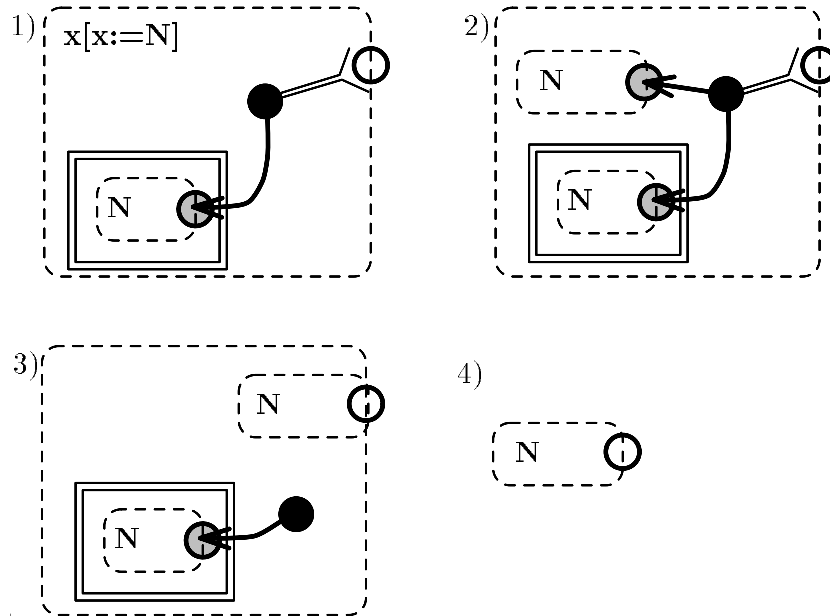
things: first a private location representing **N**, and second the parameter for further interaction.



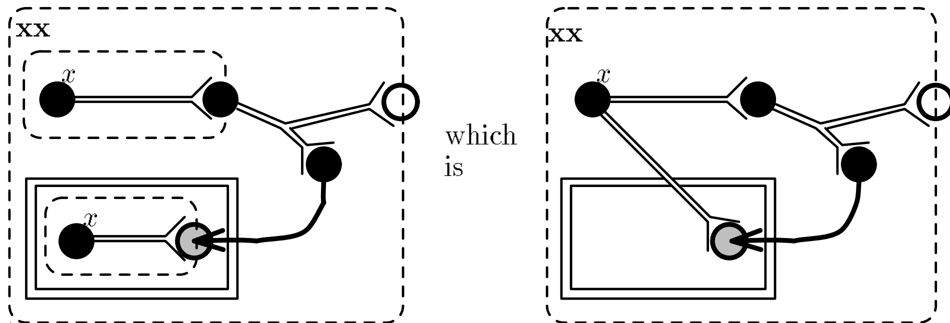
The encoding in the polyadic π-calculus of **M** is shown as ⟦**M**⟧. As a simple example consider the encoding of (**λxM**)**N**, and how it reduces to **M**[**x** := **N**]:



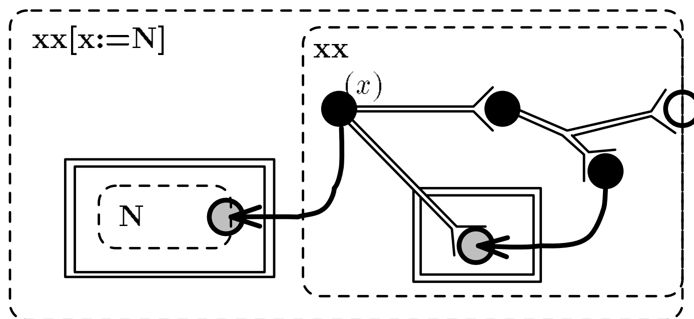
Thus, where **M** used to have **x** it now has a location representing **N**. To continue the example, assume **M** = **x**, that is, we consider the term (**λxx**)**N**. Inserting the encoding of **x** for **M** yields 1) below. Spawning a replica gives 2), reducing the interaction gives 3), and garbage collecting the replication box (which is disconnected from its environment) gives 4) which is just the encoding of **N**.



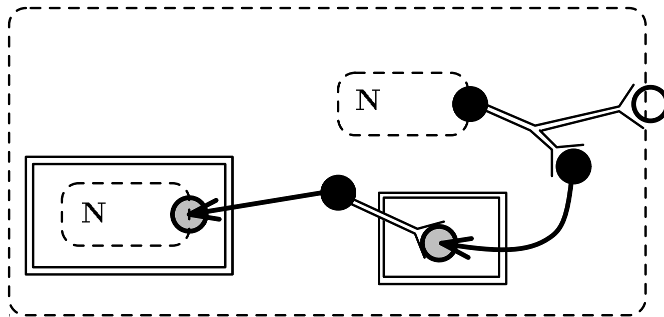
In this example **N** was only invoked once, so the replication box was not really necessary. For a more advanced example, let **M** be **xx** (i.e., **x** applied to itself). Its encoding is obtained by two instances of **x** and an application. This results in an improper diagram where x occurs twice:



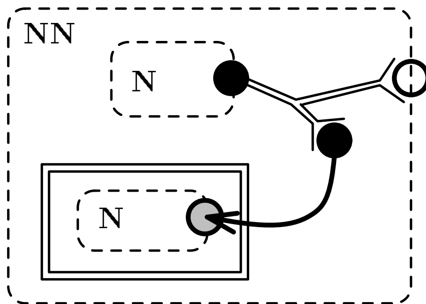
Remembering what **M[x:=N]** is we plug in **xx** for **M** in that diagram:



Here we spawn a replica of the leftmost box and reduce an interaction:



It is interesting to compare with the encoding of **NN**:



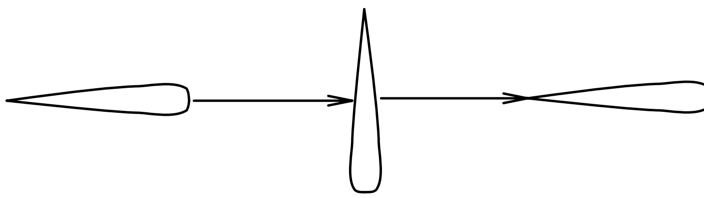
obviously the diagrams are not isomorphic but they will behave similarly, the only difference is the route into the replication box for **N**, which is one step shorter in **NN**.

[*The encoding of the* λ*-calculus into the* π*-calculus was first studied in depth by Milner [1992] and subsequently by Sangiorgi [1993], and the correctness criteria and proofs are nontrivial. In fact, as given here it correctly represents only the* lazy *reduction strategy of the* λ*-calculus, where a* β*-reduction can only take place for the first* λ*-abstraction in a term. To precisely capture the "laziness" in the diagrams we would need an additional temporal requirement that no reduction can occur in a replica of a box until its handle has been provisioned (otherwise, for example* $(\boldsymbol{\lambda}\mathbf{xM})((\boldsymbol{\lambda}\mathbf{yN})\mathbf{L})$ *would have an inner reduction). This effect can be achieved by guarding. Milner also studies the "call-by-value" reduction strategy, where a* β*-reduction from* $(\boldsymbol{\lambda}\mathbf{xM})\mathbf{N}$ *can only occur if* **N** *is* a value, *i.e., an abstraction or a variable. The encoding is only slightly longer and involves no new principles; it is an excellent exercise to look it up and represent it in diagrams!*]

11. Conclusion

We have seen how interaction diagrams depict processes where the linking structure evolves as the processes execute. In this way the diagrams go beyond existing graphical formalisms. For example, in transition systems the vertices represent states and edges are possible transitions between states, but there is no notion of "process" or concurrent activity. In flow graphs the vertices are processes and the edges are communication links between them, but there is no representation of how a graph can evolve. Petri nets (see, e.g., Reisig [1985]) capture concurrent activities and their evolutions, but the linking structure remains while the processes evolve. Reducing an interaction in a diagram roughly corresponds to firing a transition in a Petri net, but the counterparts of ports (Petri net transitions) and transmitted objects (tokens) are of different kinds and cannot coalesce. The fact that these graphs only represent some aspects of concurrency is also a strength since it admits a clearer focus on particular points of interest; they have had considerable success in clarifying important constructions, both in practice by depicting particular concurrent systems and in theory development by making the underlying intuitions visible. There are even several graphic languages, for example G-LOTOS (Bolognesi *et al* [1994]), SDL (CCITT [1988]), and Statecharts (Harel [1987]), to mention just a few, where ideas from transition systems and Petri nets are developed to the point where they can successfully deal with large-scale applications through automated tools. But no such language deals directly with the migration of access points. It remains to be seen if the ideas on interaction diagrams presented here can supplement them in a fruitful way and how they can be supported by tools.

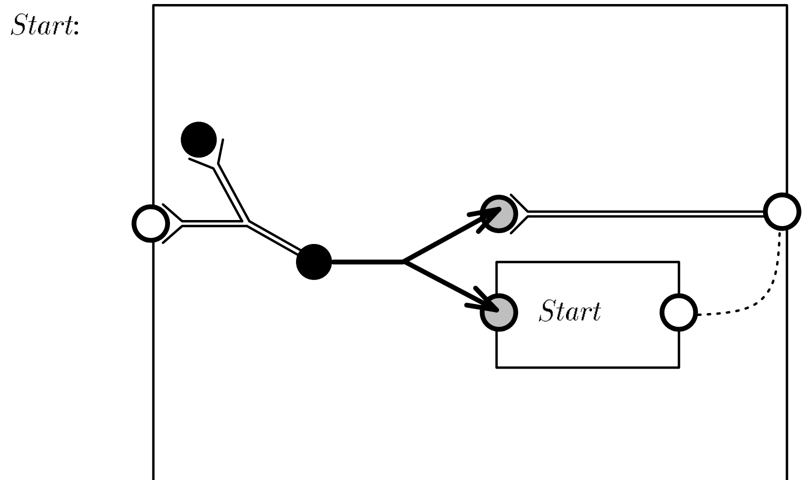
Milner's recent "π-nets" (Milner [1994]), although developed independently as a formal model of an action structure, share many characteristics of interaction diagrams, notably the idea that a reduction coalesces two nodes. There is a superficial difference in graphic representation: an interaction (or *redex*) in a π-net is depicted:



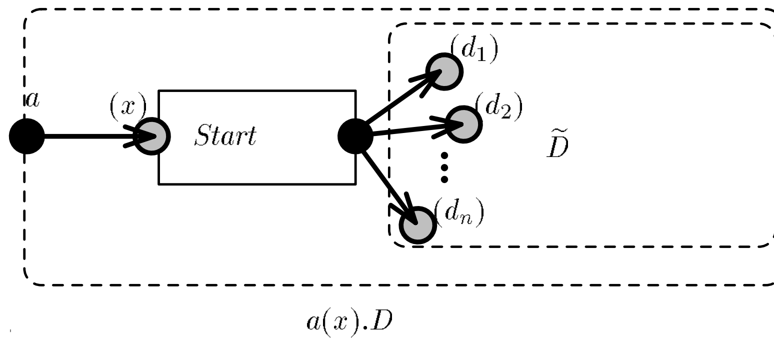
As can be seen there is just one torpedo-shaped kind of node and one kind of arrow. The function of "input" or "output" is represented by the asymmetry of nodes: an arrow from the blunt end to the waist is an output, and from the waist to the tail is an input. Thus the nodes above are, from left to right, a source, a port, and a target. But there are also deeper differences motivated by the properties of the action structure, for example the principle of provision does not hold (instead there is an operator **box** expressing temporal dependencies explicitly).

Although interaction diagrams arose through my wish to visualise the π-calculus the present paper is largely independent of its syntactic construction. A primary aim of the π-calculus, as of any algebra, is to formalise interesting operators on the objects; these operators are important because they let us build systems incrementally from smaller parts. It is a promising avenue of further research to explore operators on interaction diagrams in this way. This paper has only alluded to a few: conjoining regions (a form of parallel composition), coalescing locations (through abstraction and application), and protecting parameters to gain private locations. Presumably many π-calculus operators can also be represented without too much difficulty, but there may be exceptions (like unguarded nondeterministic choice) and the diagrams may suggest further interesting operators. If the diagrams are to function as a formal model of an algebra then all operators must have counterparts in the diagrams. This can easily be achieved by giving operators graphic representations which identify their operands, e.g., by enclosing them in boxes, but doing so indiscriminately will compromise clarity with a plethora of boxes.

To take just one example of a π-calculus operator, consider input prefix $a(x).P$. Its meaning is to first input something for x on a and then behave as P, in particular it implies that P cannot begin execution until the input has arrived. This is stronger than our principle of provision, which would allow P to execute as long as it does not use x. If we take polyadic interaction as primitive we can encode the prefix as follows: given a diagram D, build $\widetilde{D}$ by incrementing the arity of all its arrows by one. Let $d_1, \ldots, d_n$ be the new source locations required for these arrows (the new target locations are unimportant). Let *Start* be a diagram with two parameters; as soon as it receives anything from the left it repeatedly outputs on the right:



Then the prefix operator $a(x)$ can be represented as:



$a(x).D$

since until something arrives from a no transition in $\widetilde{D}$ will be fully provisioned. Thus we can use any instance of prefix as a derived construction in the diagrams, and in some cases it may help to introduce it graphically as guards. There are many similar constructions possible, and presumably many ways to develop an algebra of diagrams.

There are some curious symmetries in diagrams which may merit further investigation. For example, given that we admit subjunctive reductions (involving parameters) the only difference between a free location and a parameter is the way in which we refer to it: a free location is referred to by its name while a parameter is referred to by its relative position. Since a free location can be converted to a parameter and vice versa by abstraction and application, a more basic type of diagram can use only one of these kinds of location. Another example is the symmetry between parameters and input locations: while an input will receive something from the port of its input arrow, the parameter will receive something from outside the diagram—an input is "located" at a port while a parameter is "located" at a diagram builder. But the principle of provision breaks the symmetry. Therefore we can hope to gain a more symmetric formalism by dropping the principle; the price for this appears to be that temporal dependencies cannot be expressed by polyadic interactions. In the same way there is a symmetry between output arrows and the application operation.

A precise measure of expressiveness for various versions of diagrams or related algebras is not yet established, and questions on decidability of computational properties are largely open. In view of the λ-calculus encoding certainly all interesting properties should be undecidable, but it may be that restricting the use of replication and recursion gives a class of diagrams with nontrivial expressive power and interesting decidable properties. Similar ideas and problems have been considered in the context of the π-calculus; through interaction diagrams we may hope to attack them from another angle.

Acknowledgements

This work has been partly supported by Esprit Basic Research Project 6454 CONFER and the Human Capital and Mobility Project EXPRESS.

References

BOLOGNESI, T. AND NAJM, E. AND TILANUS, P. 1994. G-LOTOS: A graphical language for concurrent systems. *Computer Networks and ISDN Systems* 26, 1101–1127.

CCITT 1988. Recommendation Z.100: Specification and Description Language SDL. Blue Book, Volume X.1.

GAY, S. 1993. A sort inference algorithm for the polyadic π-calculus. In *Proceedings of 20th ACM Symposium on Principles of Programming Languages*, ACM Press 1993.

HAREL, D. 1987. Statecharts: A visual formalism for complex system. *Science of Computer Programming* 8(3), 231–274.

HONDA, K. AND TOKORO, M. 1991. An object calculus for asynchronous communication. In P. America (Ed) *European Conference on Object-Oriented Programming*, Springer Verlag LNCS 788, 133–147.

MILNER, R. 1991. The polyadic π-calculus: A tutorial. Research Report ECS-LFCS-91-80, Department of Computer Science, University of Edinburgh 1991. Also in L. Bauer, W. Brauer and H. Schwichtenberg (Eds) *Logic and Algebra of Specification*, Springer Verlag 1993.

MILNER, R. 1992. Functions as processes. *J. of Mathem. Structures in Computer Science* 2(2), 119–141.

MILNER, R. 1993. Action structures for the π-calculus. Research Report ECS-LFCS-93-264, Department of Computer Science, University of Edinburgh 1993.

MILNER, R. 1994. Pi-nets: a graphical form of pi-calculus. In D. Sanella (Ed) *Proceedings of European Symposium on Programming*, Springer Verlag LNCS 788, 26–42.

MILNER, R. AND PARROW, J. AND WALKER, D. 1992. A calculus of mobile processes, Part I and II. *Information and Computation* 100, 1–77.

PIERCE, B. AND SANGIORGI, D. 1995. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, to appear. (A summary is in IEEE Symposium on Logic in Computer Science 1993.)

REISIG, W. 1985. *Petri Nets.* EATCS Monographs on Theoretical Computer Science, Vol. 4, Springer Verlag.

SANGIORGI, D. 1993. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, ECS-LFCS-93-266, Department of Computer Science, University of Edinburgh.

WALKER, D. 1995. Objects in the π-calculus. *Information and Computation*, 116(2), 253–271.