Combining Interaction nets with Externally Defined Programs

Maribel Fernández
DI-LIENS (UMR 8548)
École Normale Supérieure
45 rue d'Ulm
75005 Paris
France
maribel@di.ens.fr

Ian Mackie
CNRS (UMR 7650), LIX
École Polytechnique
Route de Saclay
91128 Palaiseau Cedex
France
mackie@lix.polytechnique.fr

Jorge Sousa Pinto
Departamento de Informática
Universidade do Minho
Campus de Gualtar
4710-057 Braga
Portugal
jsp@di.uminho.pt

Abstract

Many calculi, for instance the λ-calculus and term rewriting systems, have benefitted from extensions, especially to include data structures and operations which are more naturally defined in another language. A simple example of this is PCF where the λ-calculus is extended to include natural numbers and some basic functions over this type, which avoids having to use inefficient encodings of numbers. In this paper we present a generalization of interaction nets along these lines. We begin by adding a fixed set of constants and predefined functions, before presenting the main contribution of the paper which is a system of interaction nets combined with an external language where functions and richer data types can be defined.

Keywords: Interaction nets, Rewriting Systems, Combinations.

1 Introduction

The last few years have seen a number of developments in the use of interaction nets, both as a programming and implementation language. There are now many techniques for reasoning about interaction nets [1, 7], implementations [11], and also encodings of other rewriting systems such as linear logic [9, 5], λ-calculus [8, 4], term rewriting systems [2], etc.

One of the main aspects of interaction nets is that they are very simple, and moreover they capture all of the low-level computational details. For instance, the encoding of the λ-calculus requires that the substitution propagation, including duplication and erasing, is fully explained. However, one of the inconveniences of interaction nets is that usual data types (and simple functions over these types) must be encoded in some way which are often far from reasonable from an efficiency point of view. This negative aspect of course is not unique to interaction nets: in the λ-calculus and term rewriting systems one must code the natural numbers in some way (for instance, Church numerals in the λ-calculus, and from zero (0) and successor (S) in term rewriting systems). To overcome this, one can consider adding δ rules (externally defined functions in some programming language) or combining systems.

The purpose of this paper is to make a study of various ways in which interaction nets can be extended along these lines. Specifically, we consider in turn the following extensions to interaction nets:

- Adding fixed data types and predefined functions. This system is analogous to extending the λ-calculus to PCF, where numbers and booleans, together with predefined functions (succ, pred, iszero, and a conditional) are added.
- Allowing user-defined (external) programs. Here the user has control over the external language to define data types and programs over these types. Such an extension is very liberal: all of the computation can of course be done in the external program. However, it is

mixtures of the systems that we expect would provide a useful programming language. We remark that this extension also allows side effects (if the external language allows it).

We introduce a general framework, which we call GIN($\mathcal{L}$), that allows to combine interaction nets and an arbitrary programming language $\mathcal{L}$, and as a particular instance of this system we show several examples of programs in the combination of interaction nets with functional languages.

In order to study the properties of GIN($\mathcal{L}$) we define a textual calculus of interaction, in the spirit of [3], and use it to give an operational semantics to this system. We show that some properties of interaction nets may be lost in the extension, but if $\mathcal{L}$ is a confluent language then so is GIN($\mathcal{L}$).

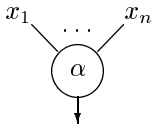
One of the main applications that we see for this work is the use of interaction nets for the implementation of realistic programming languages. Specifically, we would like to take advantage of efficient arithmetical operations in some external language, and also benefit from features such as side effects (including printing, file handling, etc.) and non-determinism, which currently have to be excluded.

Overview The rest of this paper is structured as follows: In the following section we recall interaction nets, and some general results that we shall need in the rest of the paper. Section 3 gives the first contribution which gives an extension analogous to PCF for the λ-calculus. Section 4 introduces generalized interaction nets, with user defined data types and functions. In Section 5 we look at the operational semantics of this language. In Section 6 we show some properties of the system. Finally, we conclude the paper in Section 7.

2 Background

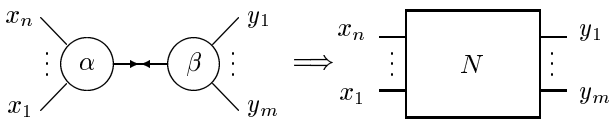
2.1 Interaction Nets

An interaction net system [6] is specified by giving a set Σ of symbols, and a set $\mathcal{R}$ of interaction rules. Each symbol $\alpha \in \Sigma$ has an associated (fixed) *arity*. An occurrence of a symbol $\alpha \in \Sigma$ will be called an *agent*. If the arity of α is n, then the agent has $n+1$ *ports*: a distinguished one called the *principal port* depicted by an arrow, and n *auxiliary ports* labeled $x_1, \ldots, x_n$ corresponding to the arity of the symbol. Such an agent is depicted in the following way:



A net N built on Σ is a graph (not necessarily connected) with agents at the vertices. The edges of the graph connect agents together at the ports such that there is only one edge at every port (edges may connect two ports of the same agent). The ports of an agent that are not connected to another agent are called the free ports of the net, and define its *interface*. There are two special instances of a net: a wiring (no agents), and the empty net.

A pair of agents $(\alpha, \beta) \in \Sigma \times \Sigma$ connected together on their principal ports is called an *active pair*; the interaction net analog of a redex. An interaction rule $((\alpha, \beta) \Longrightarrow N) \in \mathcal{R}$ replaces an occurrence of the active pair (α, β) by a net N. Rules must satisfy two conditions: the interfaces of the left-hand side and right-hand side are equal (this implies that all the free ports are preserved during reduction), and there is at most one rule for each pair of agents. The following diagram illustrates the idea, where N is any net built from Σ.



If a net does not contain any active pairs then we say that it is in normal form. We use the notation $\Longrightarrow$ for one-step reduction and $\Longrightarrow^*$ for its transitive closure. Additionally, we write $N \Downarrow N'$ if there is a sequence of interaction steps $N \Longrightarrow^* N'$, such that N' is a net in normal form. The strong constraints on the definition of an interaction rule imply that reduction is strongly commutative (the one-step diamond property holds), and thus confluence is easily obtained. Consequently, any normalizing interaction net is strongly normalizing.

We recall interaction nets by means of an example: Lafont's interaction combinators [7], which are a fixed system of interaction nets which consists of just three agents and six interaction rules. Lafont proved that this extremely simple system of rewriting is *universal*—any other system of interaction nets can be encoded (we also note that interaction nets are Turing complete: they can simulate a Turing machine.) This important result in interaction nets is analogous to the functional completeness of **S** and **K** in Combinatory Logic. We refer the reader to [7] for the proof that this system is indeed universal, and also for examples of nets. Below we give the three interaction combinators γ (a constructor), δ (a duplicator) and ϵ (an eraser), and the six interaction rules for this system.

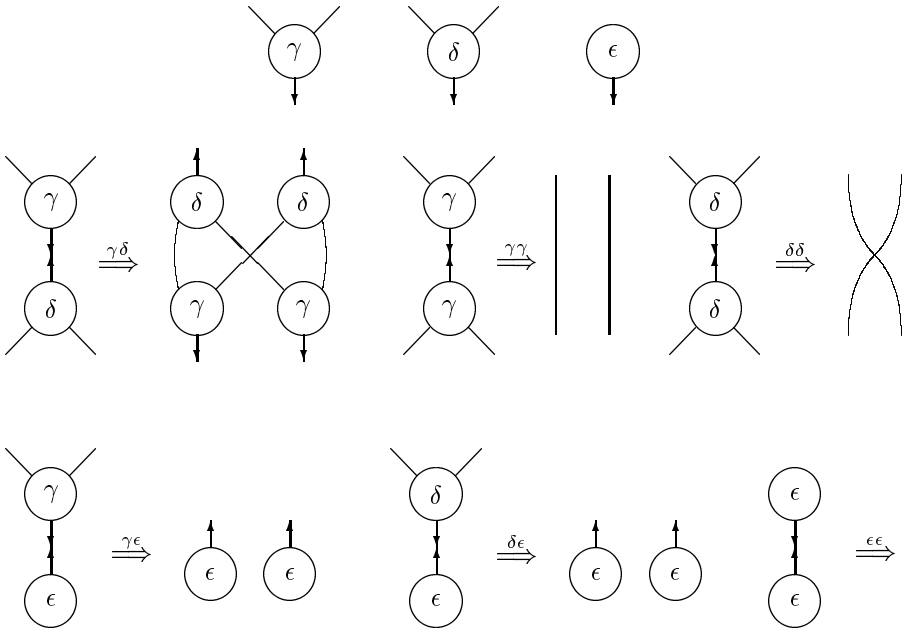


Figure 1: Interaction Combinators

2.2 A Calculus for Interaction Nets

We now recall the calculus for interaction nets [3]. This textual calculus gives a formal account of the rewriting of interaction nets, allowing to reason about this process. In particular, it allows to study strategies for reduction and notions of canonical forms (which would be harder to do in the graphical setting). The calculus has also been used as the basis for an abstract machine for interaction net reduction, see [11].

The first ingredient of the calculus is a language in which to write nets and rules. Terms of our language are either variables taken from a set $\mathcal{V}$ (ranged over by $x, y, z, \ldots$), or of the form $\alpha(t_1, \ldots, t_n)$, where $\alpha \in \Sigma$ with arity n, and $t_1, \ldots, t_n$ are terms. We will write the list of terms $t_1, \ldots, t_n$ as $\mathbf{t}$, thus $\alpha(\mathbf{t})$ is a term.

We will use $\Delta, \Theta, \ldots$ to range over *multisets of equations* of the form $t = u$, with t, u terms. The set $\mathcal{N}(\cdot)$ of variables occurring in a term, equation, or multiset is defined in the obvious way, as is the *substitution* of the variable x by the term u in term t (denoted by $t[u/x]$).

To see how interaction nets can be described by equations, it is enough to remark that any net can be orientated as a set of trees. At the bottom we have wires connecting together pairs of roots of these trees – corresponding to active pairs or equations between terms – and at the top we have wires connecting together pairs of auxiliary ports – variables occurring twice in the net.

Rules can also be described in this language, as was done by Lafont [6]. It suffices to see that a rule can be seen as an interaction net with exactly one active pair (thus one equation) and empty interface, by connecting together corresponding free ports in its left and right-hand sides. We will write these as $t \bowtie u$, to distinguish them from ordinary equations. We give as an example the rules for the combinators system of Figure 1:

$$\begin{array}{rcl} \delta(\gamma(a,b),\gamma(c,d)) & \bowtie & \gamma(\delta(a,c),\delta(b,d)) \\ \epsilon & \bowtie & \gamma(\epsilon,\epsilon) \end{array} \qquad \begin{array}{rcl} \gamma(a,b) & \bowtie & \gamma(b,a) \\ \epsilon & \bowtie & \delta(\epsilon,\epsilon) \end{array} \qquad \begin{array}{rcl} \delta(a,b) & \bowtie & \delta(a,b) \\ \epsilon & \bowtie & \epsilon \end{array}$$

We define configurations as nets to be reduced using a given set of interaction rules:

Definition 2.1 (Configuration) *A* configuration *is a pair* $(\mathcal{R}, \langle \mathbf{t} \mid \cdot \rangle)$ *where* $\mathcal{R}$ *is a set of interaction rules,* $\mathbf{t}$ *a sequence of terms called the* interface *of the net, and* Δ *a multiset of equations, such that each variable occurs* exactly twice *in* $\langle \mathbf{t} \mid \Delta \rangle$.

We will now give a set of rules for rewriting a net $\langle \mathbf{t} \mid \Delta \rangle$ in a configuration $(\mathcal{R}, \langle \mathbf{t} \mid \cdot \rangle)$.

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

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

Indirection: $x \in \mathcal{N}(u) \;\Rightarrow\; \langle \mathbf{t} \mid x = t, u = v, \Gamma \rangle \longrightarrow \langle \mathbf{t} \mid u[t/x] = v, \Gamma \rangle$.

Collect: $x \in \mathcal{N}(\mathbf{t}) \;\Rightarrow\; \langle \mathbf{t} \mid x = u, \Delta \rangle \longrightarrow \langle \mathbf{t}[u/x] \mid \Delta \rangle$.

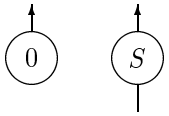
Multiset: $\Theta \rightleftharpoons^* \Theta', \langle \mathbf{t_1} \mid \Theta' \rangle \longrightarrow \langle \mathbf{t_2} \mid \Delta' \rangle, \Delta' \rightleftharpoons^* \Delta \;\Rightarrow\; \langle \mathbf{t_1} \mid \Theta \rangle \longrightarrow \langle \mathbf{t_2} \mid \Delta \rangle$.

In the above, the $\rightleftharpoons$ is an equivalence that just states the irrelevance of the order of equations in the multiset, as well as the order of the members in an equation. Two nets which are the same up to renaming of variables are called α-convertible, and in the first rule above we always use α-conversion to get a copy of the interaction rule with all variables fresh.

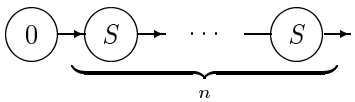
These rules constitute an operational semantics for interaction nets. We note that normal forms either have an empty equation multiset, or this multiset contains only cycles (deadlocks). The above reduction $\longrightarrow$ is strongly confluent, and normal forms are thus unique.

3 Interaction Combinators with Constants

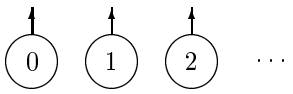
In interaction nets there are several ways that one can encode the natural numbers. The simplest system is built from the agents 0 and S:



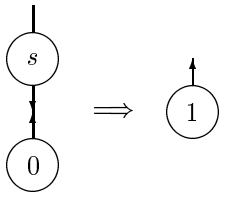
The number n is then built from a sequence:



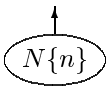
This finite system built from the constructors 0 and S allows natural numbers to be represented in interaction nets, and the usual functions over this data type can easily be defined. An alternative would be to have an infinite system, consisting of one agent for each natural number:



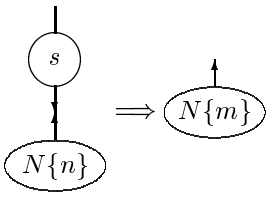
Any function that we want to write over the naturals would then require that we give an infinite number of rules, for instance the add-one function would have to be defined with rules like:



where we need one rule for each agent $0, 1, 2, \ldots$ However, from a more practical perspective, an alternative representation would be to have just one agent for natural numbers which *contains* a number:



where the notation $N\{n\}$ indicates that N is the name of the agent, and n is the value contained within N. With this representation it is possible to write the rules for the successor function in the following style:



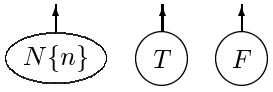
where $m = n + 1$. Thus there is just one interaction rule, and the contents of the agent N on the right-hand side of rule is a function of n, which is defined outside interaction nets.

In this section we present the first contribution in the paper, which is a generalization of interaction nets which has built-in natural numbers and booleans, and several functions over these data types, inspired by the discussion above. We give this extension together with the interaction combinators (see Figure 1), thus this extension can be seen as analogous to the extension of the λ-calculus with integer and boolean constants to obtain PCF [12].

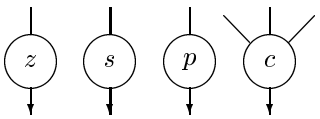
We begin with the definition of the most basic system, which we call ICCF (interaction combinators for computable functions).

Definition 3.1 (ICCF) *This interaction system is defined by adding to the interaction combinators γ, δ, ϵ:*

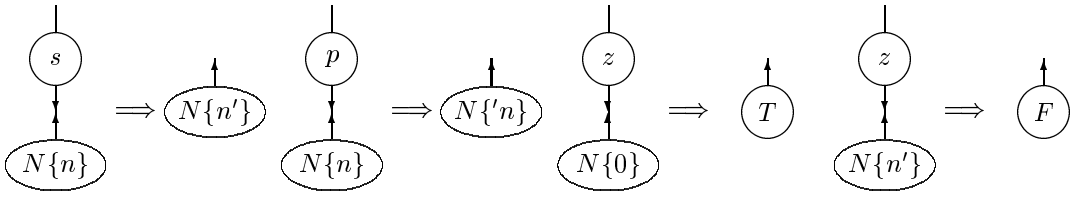
1. *constants: a generic agent $N\{n\}$ denoting the natural numbers, and two agents T, F, denoting the boolean values True and False:*



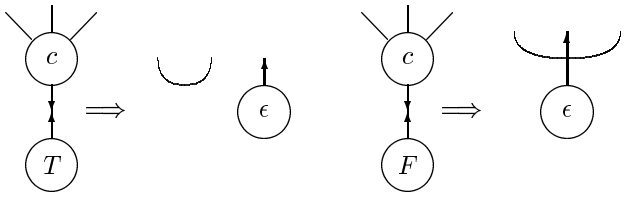
2. *the unary agents s, p and z, which correspond to the successor, predecessor, and test for zero respectively, and an agent c denoting the conditional:*



We add to the interaction rules defining the combinators, rules for these new agents:



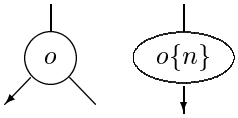
where n' is an abbreviation for $n+1$, and $'n$ is an abbreviation for $n-1$ (where $0-1=0$). Finally, the rules for the conditional are the following:



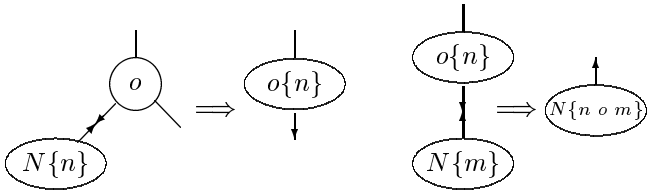
We remark that there are no rules for certain pairs of agents, since we will assume that only "well-typed" interactions will take place (there will never be an interaction between two N agents, for instance). Also note that we allow pattern matching on the built-in types for the interaction rules given above.

3.1 General Arithmetic

Next we extend ICCF with several agents which will allow for basic arithmetic operations to be performed: addition, subtraction, multiplication and division, and also comparison operations such as tests for equality and less than. We introduce the following agents which represent respectively the operator and the partially applied operator:



for $o \in \{+, -, *, \div, =, <\}$. The interaction rules for these agents are given by the following general form:



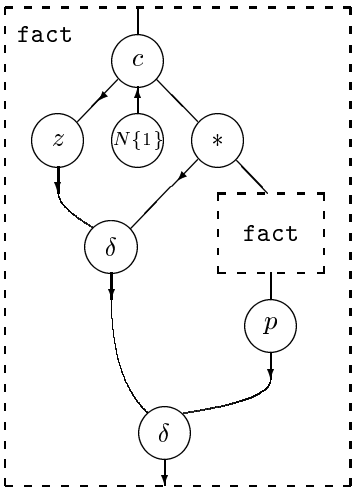
for $o \in \{+, -, *, \div\}$. The rules for $o \in \{=, <\}$ follow the same pattern, where either the agent T or F occurs on the right-hand side of the second rewrite rule.

3.2 Recursion

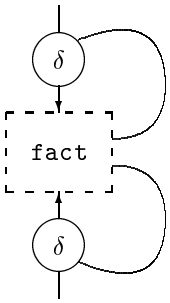
We introduce the method of encoding recursion in interaction nets by a simple example. Consider the factorial function:

```
fact n = cond(zero(n),1,n*fact(pred(n)))
```

which we can draw as the following net:



where the inner dashed box (`fact`) is a copy of the outer dashed box, and so on (note that the number of free edges of the inner box is the same as the outer box). The next step is to replace this recursion by duplication: the inner box simply takes a copy of the outer box, which suggests the following encoding:



where the two additional free edges of the `fact` net are those created by removing the inner dashed box above. This completes the essential part of encoding recursion, where in general there will be n free edges created by removing the inner box, and n δ agents must be used to connect each one to the corresponding edge in the outer box. Also, if there are several recursive calls then additional copies can be made in the same way.

However, there remains one additional problem to resolve, which is that δ agents can only duplicate a net which is free from δ agents (in the example above, there are two δ agents in the net to be duplicated). Nevertheless, there are now well-known techniques which can encode a net so that it is free from δ agents. We refer the reader to [7] for details of how this can be done, which completes the technique required to encode recursion in interaction nets.

3.3 Completeness

A natural question to ask at this point is whether we have extended interaction nets in some way, or just provided some useful abbreviations. The following result shows that the extensions defined are nothing other than convenient representations of usual data.

Theorem 3.2 *ICCF can be encoded in the system of interaction combinators.*

Proof: We observe that the extensions can be encoded as λ-terms, and then transformed into interaction combinators by the results of [10]. □

Many simple examples of using the additional agents can then be obtained from PCF programs. In the next section we generalize this idea to work with user-defined data structures in interaction systems.

4 Generalized Interaction Nets

We can see the agent $N\{n\}$ representing natural numbers in ICCF as a unique agent N which carries a constant n so different occurrences of the agent may represent different numbers. This idea can be generalized to other data types, such as pairs, lists, trees; or even further, to include functions and nets. More generally, we can see this as a generalization that consists of allowing agents to carry a program in an external language. For instance, we can associate to each agent a data structure and/or a set of functions defined in a functional programming language. Alternatively, we could think of agents as a name plus an object (for example a natural number object), where the object has a field containing a value (for example a number) and some methods that provide basic operations (for example, Successor, Predecessor, Addition, Multiplication). Some agents may have no field and just some operations (for instance an agent Factorial that interacts with a natural number to become a natural number, the result of the factorial operation). Of course the external language needs not be functional nor object oriented. We could equally define a combined framework of interaction nets and an imperative language, say C, where agents carry C programs defining data structures and operations.

We first define this extension of interaction nets parametric on an external language $\mathcal{L}$.

Definition 4.1 (GIN($\mathcal{L}$)) *A program is defined by:*

1. *An* external-language definition $\mathcal{L}$*, which determines the contents of the agents in the interaction system. In this part one selects a language (functional, imperative, object-oriented, etc.) and defines the data structures that are going to be used in the interaction system.*

2. *A set* Σ *of* agents*, each with a* name*, a fixed* arity *corresponding to the number of auxiliary ports as usual, and, optionally, a* program content *written in* $\mathcal{L}$*.*

3. *A set of* Interaction Rules *which define the possible interactions between agents. Interaction rules satisfy the usual constraints: the left-hand side is a pair of agents connected by their principal ports, the interface is preserved, and there is at most one rule for each pair of agents. Note that this implies that we cannot interact directly with the contents of the agents. We will accept an exception to this: allow pattern-matching only on integers, for instance, on the value of* n *if we use an agent* $N\{n\}$ *to represent numbers, as in ICCF (see the previous section).*

4. *A net to be evaluated, which is a graph built using the agents in* Σ *as usual.*

In order to give concrete examples, we will use GIN(ML), that is, we use interaction nets combined with a functional language of the ML family.

Example 4.2 *1.* Arithmetic: Natural numbers $N\{n\}$, Addition, Factorial. *We can define different encodings of natural numbers in ML, for instance we can use the predefined integer type:*

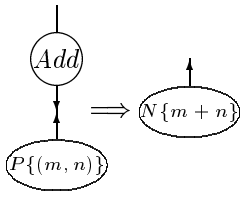
```
type nat = int
```

or a user-defined integer type such as:

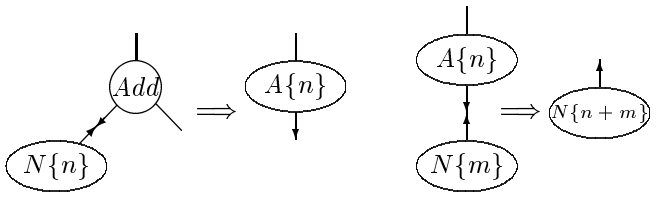
```
datatype nat = Z | S of nat
```

where `Z` *and* `S` *are constructors representing 0 and successor, respectively.*

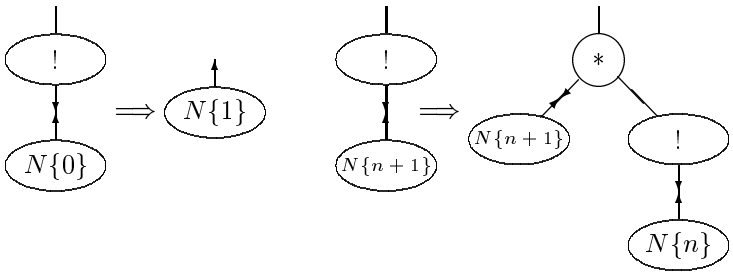
Assuming the first encoding is used, we can define addition as an agent that interacts with a pair of numbers $P\{(m,n)\}$ (here we use the pair structure) giving a natural number $N\{m+n\}$, as a result. We show the interaction rule for addition; note that in the expression $m+n$ we are using the predefined addition operation of ML:



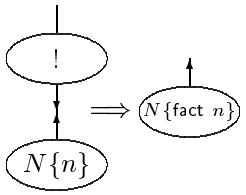
We can also give an alternative definition of addition, that takes an argument at a time, where the program content of the agent A is a natural number:



We can recursively define an agent ! representing factorial using multiplication, which shows a combination of performing computation with the system of interaction nets, and also the external program.

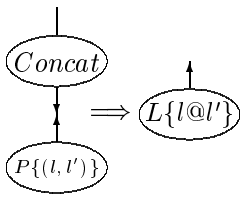


The alternative would be given by the following rule:

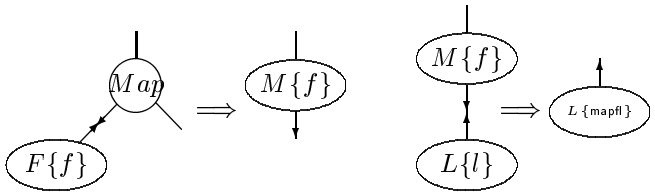



```
where fact n = if n = 0 then 1 else n*(fact (n-1))
```

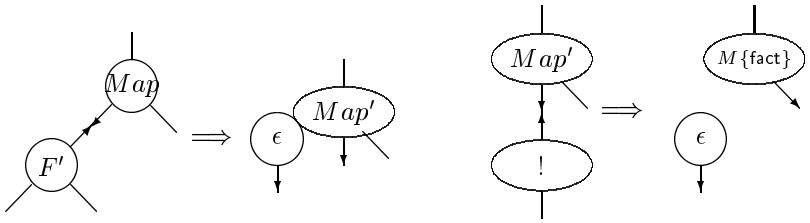
2. Lists and concatenation. *We use agents $L\{l\}$ to represent lists. Again we have the choice to use the predefined lists of ML in the program content of the agent L, or to define a new list type. Using predefined lists, we can program the operation of concatenation as follows:*



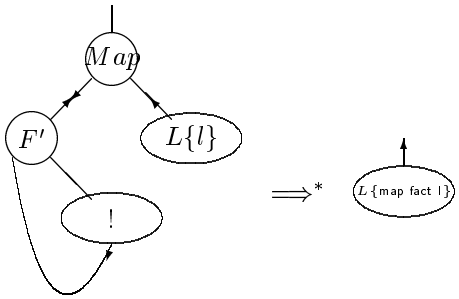
where we use the (polymorphic) agent P to carry a pair of lists, and the result is an agent L carrying a list obtained by appending l and l' (we use the predefined ML operation @ to append two lists). We can easily define Map since we have functions in the external language: we use an agent $F\{f\}$ where f is an ML definition of a function (of course, there is an alternative, which is to represent functions with a user-defined data type).



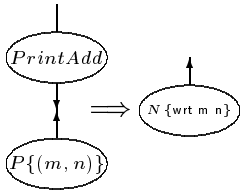
Remark that in order to map the function factorial to a list of integers, where factorial is represented by an agent ! as above, we need specific interaction rules (to pack the agent ! as a function using an agent F', and recover its program content):



As expected, we have the following reduction:



3. Printing: *Pure interaction nets do not allow side-effects; we can use this generalization to model the standard input/output features of programming languages. For example, we can add two numbers and print the result.*




```
where wrt m n = print-int (m+n); m+n
```

4. Non-determinism: *Using the facilities of the external language (for instance a random number generator) we can simulate non-determinism in interaction nets. Example: We can simulate an agent* Amb *by a net consisting of an agent* Cond *with the principal port connected to an agent* Test *whose associated program content is the test* Random-Int$(200) > 100$ *(that is, the ML function* Random-Int*(200) will generate a random integer between 0 and 200, and the conditional agent will branch non-deterministically, according to the result of the test).*

Since GIN(ML) is a combined framework, and both interaction nets and ML are computationally complete, one has of course the choice to write a program using only pure interaction nets, only the language ML, or a more or less fair combination. This choice is left to the user of the language.

Remark 4.3 *We can see a program in GIN($\mathcal{L}$) as composed of a specification in interaction nets (for instance, factorial interacts with a number n to give $n!$) plus an implementation in $\mathcal{L}$.*

We can take this approach to the limit: GIN(IN) uses interaction nets as external language. We can see this as a modular use of interaction nets, where the *external* language provides the interaction net implementation of the agents defined in the *internal* system. Another way of seeing it is to think of agents as *boxes* containing interaction nets.

Example 4.4 *For addition, we can replace the agent $N\{n\}$ with a net consisting of a list of n successor agents, and a constructor N holding a pointer to the head and tail of the list. Using the notation of the textual calculus for interaction nets: if $n = 0$ we obtain the net $N(x, x)$, otherwise $N(S^n(x), x)$. The operation of addition can then be encoded by the net*

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

which simply appends the two lists representing the numbers. There is only one interaction rule that we need:

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

which is a constant time operation.

However, in GIN(IN) the computation in the external and the internal system is kept separate, that is, if in an interaction rule we use an agent implemented externally, then the external evaluation is done and the result is transmitted to the internal system. We could of course allow computations to mix completely, and internalize the boxes. In this way, we could allow an external program to return a net (adding some primitives to build nets). We thus obtain a Higher-Order Interaction Net System, which is the subject of future work.

From an implementation point of view, interaction nets have been successfully used to implement functional languages (specifically the λ-calculus). One of the advantages of interaction nets is that each step is atomic so one can compute the cost of evaluating a program simply by counting the number of interactions needed to obtain a result. Of course, in GIN($\mathcal{L}$) this is no longer the case: interaction steps have variable cost, depending on the external code to be executed. From this point of view, there are several interesting restrictions of GIN($\mathcal{L}$): we may require that

1. $\mathcal{L}$ consists only of machine operations (hardware arithmetic for example)
2. $\mathcal{L}$ consists only of constant time operations
3. $\mathcal{L}$ implements only linear functions (for example, take $\mathcal{L}$ = linear λ-calculus)
4. $\mathcal{L}$ implements only polynomial algorithms

5. the code associated to an agent can be translated to interaction nets. This option rules out some of the applications mentioned in Example 4.2, since we know that, for instance, the non-deterministic agent Amb cannot be defined in standard interaction nets. The advantage is that in this case we can explain the interaction rules of GIN($\mathcal{L}$) as abbreviations, and their associated cost corresponds to the number of interaction steps required to build the right-hand side. For example: for numbers it is easy to define a decoder from $N\{n\}$ to the usual representation using an agent 0 and an agent S, and we can use an agent Add for addition with the usual rules (interacting with 0 and S), or there is also the choice of decoding $N\{n\}$ as a difference list and using a net for addition (see Example 4.4). We can also encode lists and even a Map agent that applies a function to the elements of the list: for this we need an implementation of the λ-calculus in interaction nets, YALE for instance, and we need a compilation algorithm from λ-terms into nets.

5 Operational Semantics

The calculus defined in [3] (see Section 2) provides a framework where it is easy to formalize notions of strategies of reductions and normal forms. In order to give a formal operational semantics for GIN($\mathcal{L}$) we will define a textual calculus which generalizes the calculus for interaction nets. We begin by introducing the syntactic categories:

Agents: Let Σ be a set of symbols, ranged over by $\alpha, \beta, \ldots$, each with a given *arity* $\mathsf{ar} : \Sigma \to Nat$. An occurrence of a symbol will be called an *agent*, and will have an associated program content p, written in $\mathcal{L}$. The arity of a symbol corresponds precisely to the number of auxiliary ports.

Names: Let $\mathcal{V}$ be a set of names, ranged over by x, y, z, etc. $\mathcal{V}$ and Σ are assumed disjoint.

Terms: A term is built on Σ and $\mathcal{V}$ by the grammar:

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

where $x \in \mathcal{V}$, $\alpha \in \Sigma$, $p \in \mathcal{L}$, $\mathsf{ar}(\alpha) = n$ and $t_1, \ldots, t_n$ are terms, with the restriction that each name can appear at most twice. 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 $\alpha\{p\}(\vec{t})$ can be seen as a tree with the principal port of α at the root, and where the terms $t_1, \ldots, t_n$ are the subtrees connected to the auxiliary ports of α.

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. Equations allow us to represent active pairs: $\alpha\{p\}(\vec{t}) = \beta\{q\}(\vec{u})$ represents a net with an edge connecting the principal ports of α and β.

Rules: Rules are pairs of terms with an optional $\mathcal{L}$ program, written as $\alpha\{p\}(\vec{t}) \bowtie \beta\{q\}(\vec{u})$ where P, where $(\alpha, \beta) \in \Sigma^2$ is the active pair of the rule and P contains the local definitions that are needed in order to evaluate the programs associated to the agents in $\vec{t}, \vec{u}$. In general, the programs p and q will be variables of the external language or natural numbers (which can be used in $\vec{t}, \vec{u}$), since there is no pattern matching on the program contents of the agents α, β, except in the case of numbers. All names occur exactly twice in a rule. There is at most one rule for each pair of agents except in the case of agents containing integers, where there might be several rules but with disjoint patterns (as in Example 4.2 in the definition of ! using $*$).

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.

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

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

Definition 5.1 (Substitution) *The notation $t[u/x]$ denotes a term with 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. Renaming is a particular case of substitution where u is a name.*

Lemma 5.2 *Assume that $x \notin \mathcal{N}(v)$. If $y \in \mathcal{N}(u)$ then $t[u/x][v/y] = t[u[v/y]/x]$, otherwise $t[u/x][v/y] = t[v/y][u/x]$.*

We now have all the machinery that we need to define interaction nets in GIN($\mathcal{L}$).

Definition 5.3 (Configurations) *A* configuration *is a triple: $c = (\mathcal{P}, \mathcal{R}, \langle \mathbf{t} \mid \Delta \rangle)$, where $\mathcal{P}$ is a program written in $\mathcal{L}$ (typically, $\mathcal{P}$ contains all the data structures and auxiliary definitions that are needed to use the program contents of the agents), $\mathcal{R}$ is a set of rules, $\mathbf{t}$ a multiset $\{t_1, \ldots, t_n\}$ of terms, and Δ a multiset of equations. Each variable 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 $\mathcal{P}$ and $\mathcal{R}$ when there is no ambiguity. We use c, c' to range over configurations. We call $\mathbf{t}$ the* head *and Δ the* body *of a configuration.*

Intuitively, $\langle \mathbf{t} \mid \Delta \rangle$ represents a net that we evaluate using $\mathcal{R}$; Δ gives the set of active pairs and the renamings of the net. 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, but also for free names. Configurations that differ only on the names of the free variables are equivalent, since they represent the same net.

Example 5.4 *The degenerate case of the empty net is represented by $\langle \mid \rangle$.*
We show now the representation of the nets and rules of GIN(ML) given in Example 4.2.

1. Arithmetic: Natural numbers $N\{n\}$, Addition, Factorial. *Let P be the ML program*

   ```
   type nat == int
   ```

   The rule defining the interaction between the addition agent and a pair of natural numbers is written as:

   $$Add(N\{m+n\}) \bowtie P\{(m,n)\}$$

   The alternative definition, taking one argument at a time, is given by:

   $$Add(x, A\{n\}(x)) \bowtie N\{n\}$$
   $$A\{n\}(N\{m+n\}) \bowtie N\{m\}$$

   The definition of factorial is given by:

   $$!(N\{\mathsf{fact}\ n\}) \bowtie N\{n\}$$

   ```
   where fact n = if n = 0 then 1 else n*(fact (n-1))
   ```

2. Lists and concatenation. *The rule defining the interaction between the agent Concat and a pair of lists is written as:*

   $$Concat(L\{l@l'\}) \bowtie P\{(l,l')\}$$

   where we use the predefined ML operation @ to build the net in the right-hand side (an agent L containing the append of the two lists in the left).

   The following is the definition of Map:

   $$Map(x, M\{f\}(x)) \bowtie F\{f\}$$
   $$M\{f\}(L\{\mathsf{map}\ f\ l\}) \bowtie L\{l\}$$

If we want to apply the factorial function defined above to all the elements of a list, then we use the packing agent F' with the rules:

$$Map(z, l) \bowtie F'(Map'(z, l), \epsilon)$$
$$Map'(z, M\{\mathsf{fact}\}(z)) \bowtie !(\epsilon)$$

3. Printing: *the rule that adds two numbers and prints the result is written as:*

$$PrintAdd(N\{A\ m\ n\}) \bowtie P\{(m, n)\}$$

```
where A m n = print-int (m+n); m+n
```

The dynamics of the system is defined by the following system of computation rules:

Definition 5.5 (Computation Rules) *We have four computation rules.*

Indirection: $x \in \mathcal{N}(u) \ \Rightarrow\ \langle \mathbf{t} \mid x = t, u = v, \Gamma\rangle \longrightarrow \langle \mathbf{t} \mid u[t/x] = v, \Gamma\rangle$.

Interaction: $(\alpha\{p'\}(t'_1, \ldots, t'_n) \bowtie \beta\{q'\}(u'_1, \ldots, u'_m)$ where $P) \in \mathcal{R}\ \Rightarrow$

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

where p, q are instances of p', q', $\widehat{t'_i}$ (resp. $\widehat{u'_i}$) is a fresh (renamed) instance of t'_i (resp. $\widehat{u'_i}$) where the substitutions $[p/p'][q/q']$ have been done and all the programs occurring in the agents in $\vec{t}$, $\vec{u}$ have been evaluated.

Collect: $x \in \mathcal{N}(\mathbf{t})\ \Rightarrow\ \langle \mathbf{t} \mid x = u, \Delta\rangle \longrightarrow \langle \mathbf{t}[u/x] \mid \Delta\rangle$.

Multiset: $\Theta \rightleftharpoons^* \Theta', \langle \mathbf{t_1} \mid \Theta'\rangle \longrightarrow \langle \mathbf{t_2} \mid \Delta'\rangle, \Delta' \rightleftharpoons^* \Delta\ \Rightarrow\ \langle \mathbf{t_1} \mid \Theta\rangle \longrightarrow \langle \mathbf{t_2} \mid \Delta\rangle$,
where $\rightleftharpoons$ is an equivalence that just states the irrelevance of the order of equations in the multiset, as well as the order of the members in an equation.

The calculus puts in evidence the real cost of implementing an interaction step, which involves generating an instance of the right-hand side of the rule, i.e. a new copy where the programs in $\mathcal{L}$ that define the agents of the right-hand side have been evaluated, plus renamings (rewirings) to paste the new net in the interface of the active pair.

Remark 5.6 1. *We can define a* lazy *version of the computation rules, where the programs of the external language are evaluated only when they are needed, for instance for pattern matching or when there are no more interactions to be done.*

2. *Using the calculus we can formally define different notions of value (which are not necessarily normal forms with respect to the rewrite system), as done in [3] for pure interaction nets. For instance, we can define interface normal forms (analogous to weak head normal forms in functional languages) and an evaluation strategy to compute this kind of value.*

We now show a more elaborated example, with an alternative definition of lists in the combined framework.

Example 5.7 *We define lists of numbers using a unary agent Cons which holds a number. An algorithm such as sort can be programmed using interaction nets except for the operations on numbers which are more efficiently done in the external language. We use an auxiliary agent Insert to put a number in the correct place in a sorted list.*

$$Insert\{n\}(Cons\{\mathsf{min}\ m\ n\}(x)) \bowtie Cons\{m\}(Insert\{\mathsf{max}\ m\ n\}(x)) \quad \text{where}$$
$$\mathsf{min\ m\ n} = \mathsf{if\ m} \leq \mathsf{n\ then\ m\ else\ n}$$
$$\text{and } \mathsf{max\ m\ n} = \mathsf{if\ m} \geq \mathsf{n\ then\ m\ else\ n}$$
$$Insert\{n\}(Cons\{n\}(Nil)) \bowtie Nil$$
$$Sort(Nil) \bowtie Nil$$
$$Sort(y) \bowtie Cons\{n\}(Sort(Insert\{n\}(y))$$

Using these interaction rules

$$\langle x \mid Sort(x) == Cons\{3\}(Cons\{2\}(Cons\{1\}(Nil)))\rangle \longrightarrow^* \langle Cons\{1\}(Cons\{2\}(Cons\{3\}(Nil))) \mid \rangle$$

as expected.

6 Properties

The constraints imposed in the definition of interaction rule imply that interaction nets enjoy several useful properties:

1. Interactions are binary: only two agents participate in a rewrite step.
2. Interactions are local: the applicability of a rule does not depend on any global conditions on the net, and it only affects the part of the net where the redex is located. As a consequence, a reduction step is easier to implement than in general term or graph rewriting systems.
3. Interaction nets are strongly confluent. This means that the final result does not depend on the order of application of the rules, so that we are free to implement reduction in any order, even in parallel. Moreover, there is only one reduction sequence for a given net, modulo permutation of steps, since there are no superpositions between the rules.
4. If there is a normalizing reduction for a net, then all its reductions are normalizing and lead to the same result (that is, weak normalization is equivalent to strong normalization in interaction nets).

A natural question to ask about the generalized framework GIN($\mathcal{L}$) is whether it enjoys the same properties of interaction nets. For the first two properties in the list above, i.e. locality and binary interactions, the answer is positive. However, the application of an interaction rule in GIN($\mathcal{L}$) is not as simple, since it now involves the evaluation of external programs in order to obtain the right-hand side net associated to the rule. As a consequence, properties like the unicity of results are more difficult to obtain in the generalized framework. This property depends on the language $\mathcal{L}$ that we are considering: it is clear that if $\mathcal{L}$ has non-deterministic primitives then the strong confluence property will be lost.

Theorem 6.1 *If $\mathcal{L}$ is confluent then GIN($\mathcal{L}$) is confluent.*

Proof: If $\mathcal{L}$ is confluent then GIN($\mathcal{L}$) has the diamond property (strong confluence): If for a configuration c we have two reductions $c \longrightarrow c'$ and $c \longrightarrow c''$ (where $c' \neq c''$) then the steps are non-overlapping since each agent has a unique principal port and there is at most one interaction rule that can be applied (recall that there is at most one rule for each pair of agents except when pattern matching is allowed in which case patterns must be disjoint). Therefore these reductions are independent and can be done in any order. □

Corollary 6.2 (Unicity of Results) *If $\mathcal{L}$ is confluent then GIN($\mathcal{L}$) is deterministic.*

With respect to termination, it is clear that the fact that $\mathcal{L}$ is terminating (that is, the execution of any program terminates) is not sufficient to obtain termination of GIN($\mathcal{L}$). However, it guarantees that all the interaction steps are well-defined (note that the termination of the evaluation of the programs used in the right-hand side of an interaction rule is a necessary condition for the definition of the interation step).

7 Conclusions and Further Work

In this paper we have made an initial investigation into generalizing interaction nets to allow programs in some external language to be combined with interaction net evaluation. Natural examples of such a system is the use of interaction nets to implement programming languages where we can use the interaction net framework to represent part of the computation, and the external language to capture the computational content which is less natural in the interaction net framework (for example the implementation of PCF).

Various directions of research open up from this initial investigation, which appear to offer a very rich programming paradigm. Some of the current themes being developed include:

- Types: extending the interaction system with a type system, and allow the declaration of polymorphic rules.
- Relation with Process Calculi: several process calculi can be seen as some kind of generalization of interaction nets. With our extension, we can of course model such languages, and the relationship with other work in this area is to be studied.
- Shared global structures (State): the external language can provide the machinery to model state, and thus different interactions can access this information to provide many additional features to interaction nets.
- Higher order interaction nets: this is the most exciting aspect of our current work, which we have only hinted at in the present paper. There appears to be several possible notions of "higher-order" interaction net that can be defined.

References

[1] M. Fernández and I. Mackie. Coinductive techniques for operational equivalence of interaction nets. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS'98)*, pages 321–332. IEEE Computer Society Press, June 1998.

[2] M. Fernández and I. Mackie. Interaction nets and term rewriting systems. *Theoretical Computer Science*, 190(1):3–39, January 1998.

[3] M. Fernández and I. Mackie. A calculus for interaction nets. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, number 1702 in Lecture Notes in Computer Science, pages 170–187. Springer-Verlag, September 1999.

[4] G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proceedings of ACM Symposium Principles of Programming Languages*, pages 15–26, 1992.

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

[6] Y. Lafont. Interaction nets. In *Proceedings, 17th ACM Symposium on Principles of Programming Languages*, pages 95–108, 1990.

[7] Y. Lafont. Interaction combinators. *Information and Computation*, 137(1):69–101, 1997.

[8] I. Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, 1998.

[9] I. Mackie. Interaction nets for linear logic. *Theoretical Computer Science*, 247(1):83–140, September 2000.

[10] I. Mackie and J. S. Pinto. Encoding linear logic with interaction combinators. *Information and Computation, To appear*, 2001.

[11] J. S. Pinto. Sequential and concurrent abstract machines for interaction nets. In J. Tiuryn, editor, *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, number 1784 in Lecture Notes in Computer Science, pages 267–282. Springer-Verlag, 2000.

[12] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.