

# An Implementation Model for Interaction Nets

Ian Mackie

Shinya Sato

To study implementations and optimisations of interaction net systems we propose a calculus to allow us to reason about nets, a concrete data-structure that is in close correspondence with the calculus, and a low-level language to create and manipulate this data structure. These work together so that we can describe the compilation process for interaction nets, reason about the behaviours of the implementation, and study the efficiency and properties.

## 1 Introduction

Interaction nets offer a visual aspect to rewriting. Analogous to term rewriting systems, a specific system is defined from a user-defined set of nodes (cf. terms) and a user-defined set of rewrite rules. Nets are graphs built from the set of nodes and rules are graph transformations. Interaction nets are therefore a specific, in fact constrained, form of graph rewriting.

Interaction nets have been used as a programming language, an intermediate language, and as a target language for the compilation of other programming languages. In all these application areas, prototype implementations have been built to support the work, but they are not frequently documented. The purpose of this paper is to take another look at the implementation of interaction nets. Specifically, we are interested in documenting the implementation process, and in particular showing a compilation of nets to a low-level language that we define. We aim to build on the past experience and knowledge obtained from building other implementations, and make a new contribution to this investigation. Specifically, we define:

- a calculus that can represent and express results about interaction nets.
- a data-structure that is in close correspondence to the calculus, together with a low-level language (set of instructions) to manipulate this data-structure. When talking about data structures, we have in mind something written in a language like C, and the low-level language directly manipulating the pointers of this data structure.
- a compilation of the calculus to this low-level language.

In addition to defining the above, a main contribution for this work is the compilation of interaction rules: the ability to implement rules efficiently will impact greatly on an implementation. The low-level language is specifically close enough to machine code that we essentially get atomic operations so that we can understand the cost of an interaction. From a practical perspective, we get reliable, efficient implementations of interaction nets from this work. In this short paper we just present a simplified version of this model.

**Related work.** A number of evaluators for interaction nets have been developed, starting from Lafont [4]. In 2002, a graphical interpreter  $\text{in}^2$  was proposed by Lippi [5] and it showed an aspect of interaction nets as a visual programming tool. Some evaluators have been proposed towards efficient computation, called INETS [2] and *amineLight* [3] in 2009 and 2010 respectively. In  $\text{in}^2$  and INETS a net is represented as a double linked structure, whereas *amineLight* uses single links for net connections except for connections between names. In this paper our evaluator requires single links only.

**Structure.** The rest of this paper is organised as follows. The next section gives some background for the work. Section 3 describes the calculus. Section 4 we give the data-structure that corresponds to the calculus together with a low-level language to manipulate this data-structure. We include also some notes about the compilation of nets into the low-level language. We conclude in Section 5.

## 2 Background and Motivation

In the graphical rewriting system of interaction nets [4], we have a set  $\Sigma$  of *symbols*, which are names of the nodes in our diagrams. Each symbol has an *arity*  $ar$  that determines the number of *auxiliary ports* that the node has. If  $ar(\alpha) = n$  for  $\alpha \in \Sigma$ , then  $\alpha$  has  $n + 1$  *ports*:  $n$  auxiliary ports and a distinguished one called the *principal port*. Nodes are drawn variably as circles, triangles or squares as shown in Figure 1. A *net* built on  $\Sigma$  is an undirected graph with nodes at the vertices. The edges of the net connect nodes together at the ports such that there is only one edge at every port.

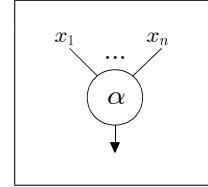


Figure 1: Agents

A port which is not connected is called a *free port*.

Two nodes  $(\alpha, \beta) \in \Sigma \times \Sigma$  connected via their principal ports form an *active pair*, which is the interaction nets analogue of a redex. A rule  $((\alpha, \beta) \Rightarrow N)$  replaces the pair  $(\alpha, \beta)$  by the net  $N$ . All the free ports are preserved during reduction, and there is at most one rule for each pair of agents.

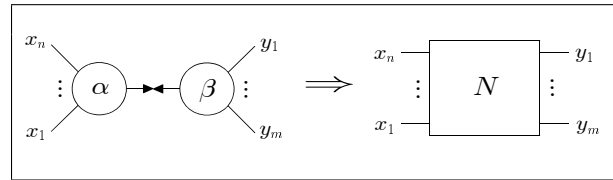


Figure 2: Interaction rules

The diagram in Figure 2 illustrates the idea, where  $N$  is any net built from  $\Sigma$ .

The most powerful property of this graph rewriting system is that it is one-step confluent: the order of rewriting is not important, and all sequences of rewrites are of the same length (in fact they are permutations). This has practical consequences: the diagrammatic transformations can be applied in any order, or even in parallel, to give the correct answer.

There are several possibilities for a data structure to represent interaction nets. One choice is some representation of a graph that can be single or double linked. Since agents have exactly one principal port we can use this to our advantage and represent nets as a collection of trees, and thus use a single linked structure. Here we focus on one way: we represent all nets as trees by drawing them in a specific orientation. This gives a very simple data structure, even if it is not the most efficient way to do things. Computationally the cost of evaluating a net is mostly due to the cost of the rewrite step: building the right-hand side of the rule; connecting the new net to the old one; freeing up the left-hand side of the rule (agents  $\alpha$  and  $\beta$  in the diagram in Figure 2).

The goal of this paper is to build an implementation model, together with a calculus, that explains this process and allows the study of the cost of computation. Specifically, in the full paper we will elaborate on the compilation of rules where we can perform the above steps in the fewest instructions.

## 3 Calculus

Although it is possible to reason about the graphical representation of nets, it is convenient to have a textual calculus for compact representation. There are several proposals in the literature, and here we present a refined and extended version of [1] that is appropriate for our use. We begin by defining a number of syntactic categories.

**Agents:** Let  $\Sigma$  be a set of symbols, ranged over by  $\alpha, \beta, \dots$ , each with a given *arity*  $ar : \Sigma \rightarrow \mathbb{N}$ . An occurrence of a symbol will be called an *agent*. The arity of a symbol corresponds precisely to the number of auxiliary ports.

**Names:** Let  $N$  be a set of names, ranged over by  $x, y, z$ , etc.  $N$  and  $\Sigma$  are assumed disjoint. Names correspond to wires in the graph system.

**Terms:** A term is built on  $\Sigma$  and  $N$  by the grammar:  $t ::= x \mid \alpha(t_1, \dots, t_n) \mid \$t$ , where  $x \in N$ ,  $\alpha \in \Sigma$ ,  $\text{ar}(\alpha) = n$  and  $t_1, \dots, t_n$  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 (i.e. occurs once) it is *free*. Since free names occur exactly once, we say that terms are *linear*. We write  $s, t, u$  to range over terms, and  $\vec{s}, \vec{t}, \vec{u}$  to range over sequences of terms. A term of the form  $\alpha(t_1, \dots, t_n)$  can be seen as a tree with the principal port of  $\alpha$  at the root, and where the terms  $t_1, \dots, t_n$  are the subtrees connected to the auxiliary ports of  $\alpha$ . The term  $\$t$  represents an indirection node which is created by reduction, and is not normally part of an initial term. Intuitively,  $\$t$  corresponds to a wire bounded with  $t$ .

**Equations:** If  $t$  and  $u$  are terms, then the (ordered) pair  $t = u$  is an *equation*.  $\Delta, \Theta, \dots$  will be used to range over sequences of equations.

**Rules:** Rules are pairs of terms written:  $\alpha(x_1, \dots, x_n) = \beta(y_1, \dots, y_m) \rightarrow \Theta$ , where  $(\alpha, \beta) \in \Sigma \times \Sigma$  is the active pair of the rule, and  $\Theta$  is the right-hand side of the rule. All names occur exactly twice in a rule, and there is at most one rule for each pair of agents. The free names  $x_1, \dots, x_n, y_1, \dots, y_m$  in  $\Theta$  are called *parameters*. All other names are bound names.

**Configurations:** A *configuration* is a pair:  $c = (\mathcal{R}, \langle \vec{t} \mid \Delta \rangle)$ , where  $\mathcal{R}$  is a set of rules,  $\vec{t}$  a sequence  $t_1, \dots, t_n$  of terms, and  $\Delta$  a sequence of equations. Each variable occurs at most twice in  $c$ . A name that occurs exactly once is *free*, and a name that occurs twice is *bound*. The rules set  $\mathcal{R}$  contains at most one rule between any pair of agents, and it is closed under symmetry, thus if  $\alpha(\vec{x}) = \beta(\vec{y}) \in \mathcal{R}$  then  $\beta(\vec{y}) = \alpha(\vec{x}) \in \mathcal{R}$ . For simplicity we sometimes omit  $\mathcal{R}$  when there is no ambiguity. We use  $c, c'$  to range over configurations. We call  $\vec{t}$  the *head* and  $\Delta$  the *body* of a configuration.

**Definition 1 (Names in terms)** The set  $\mathcal{N}(t)$  of names of a term  $t$  is defined in the following way, which extends to sequences of terms, equations, sequences of equations, and rules in the obvious way.

$$\mathcal{N}(x) = \{x\}, \quad \mathcal{N}(\alpha(t_1, \dots, t_n)) = \mathcal{N}(t_1) \cup \dots \cup \mathcal{N}(t_n), \quad \mathcal{N}(\$t) = \mathcal{N}(t)$$

We can replace its free names by new names, provided the linearity restriction is preserved.

**Definition 2 (Substitution)** The notation  $t[u/x]$  denotes a substitution that replaces the free occurrence of  $x$  by the term  $u$  in  $t$ . We only consider substitutions that preserve the linearity of the terms. Remark that since the name  $x$  occurs exactly once in the term, this operation can be implemented directly as an assignment, as is standard in the linear case. This notion extends to sequence of terms, equations, sequences of equations, and configurations in the obvious way.

**Definition 3 (Instance of a rule)** If  $r$  is a rule  $\alpha(x_1, \dots, x_n) = \beta(y_1, \dots, y_m) \rightarrow \Theta$ , then  $\hat{\Theta}$  denotes a new generic instance of  $r$ , that is, a copy of  $\Theta$  where we introduce a new set of bound names so that those new names do not overlap with others that already exist (names cannot occur more than twice), but leave the free names (parameters) unchanged. Example: if  $\Theta$  is  $\alpha(x, x) = \beta(a)$ , then  $\hat{\Theta}$  is  $\alpha(y, y) = \beta(a)$ , where  $y$  is a fresh name.

The configuration  $\langle t_1, \dots, t_n \mid \Delta \rangle$  represents a net that we evaluate using  $\mathcal{R}$ ;  $\Delta$  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.

**Example 4** The configuration  $\langle r \mid \text{Add}(r, Z) = Z \rangle$  represents a net with one active pair and an interface.

|                                                                                                                                                                                                                                                                                                                                     |                                                                                                                                         |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| <b>Var 1:</b> $\langle \vec{u} \mid x = t, \Delta \rangle \longrightarrow \langle \vec{u} \mid \Delta \rangle [t/x]$ where $t \neq \$s$                                                                                                                                                                                             | <b>Var 2:</b> $\langle \vec{u} \mid t = x, \Delta \rangle \longrightarrow \langle \vec{u} \mid \Delta \rangle [t/x]$ where $t \neq \$s$ |
| <b>Interaction:</b> $\alpha(x_1, \dots, x_n) = \beta(y_1, \dots, y_m) \longrightarrow \Theta \in \mathcal{R}$ , then<br>$\langle \vec{u} \mid \alpha(t_1, \dots, t_n) = \beta(s_1, \dots, s_m), \Delta \rangle \longrightarrow \langle \vec{u} \mid \hat{\Theta}[t_1/x_1, \dots, t_n/x_n, s_1/y_1, \dots, s_m/y_m], \Delta \rangle$ |                                                                                                                                         |
| <b>Indirection 1:</b> $\langle \vec{u} \mid \$t = s, \Delta \rangle \longrightarrow \langle \vec{u} \mid t = s, \Delta \rangle$ .                                                                                                                                                                                                   | <b>Indirection 2:</b> $\langle \vec{u} \mid t = \$s, \Delta \rangle \longrightarrow \langle \vec{u} \mid t = s, \Delta \rangle$ .       |
| <b>Indirection 3:</b> $\langle \vec{u}_1, \$u, \vec{u}_2 \mid \Delta \rangle \longrightarrow \langle \vec{u}_1, u, \vec{u}_2 \mid \Delta \rangle$ .                                                                                                                                                                                 |                                                                                                                                         |

Figure 3: Computation rules

**Definition 5 (Computation Rules)** *The operational behaviour of the system is given by the set of computation rules as shown in Figure 3.*

These rules will correspond directly to the graphical data structure and operations given in the next section. Indirection is introduced so that the data-structure manipulations can be kept simple. However, there is then the overhead of dealing with indirection nodes. We remark also that normal forms can have indirections in them, but only inside constructors. Computationally the interaction rule is the most expensive to implement: the other rules will turn out to be implemented with a small number of instructions or will be equivalences in the data structure. In particular, the renaming part of the hat operation is zero cost, as no renaming is needed in an implementation.

## 4 Data structures and language

In this section we fix a data-structure and a low-level language to implement nets, specifically to follow the definition of the calculus in the previous section. There are many choices that we could make, but we just aim to give a very simple solution (we discuss alternatives in the full paper).

The concrete representation of a configuration can be summarised by a diagram in Figure 4, where  $I$  represents the interface,  $EQ$  a stack of equations, and  $\Gamma$  heaps of graph elements for a net.

We have the following instructions that work with the equation stack: `push(x, y)` pushes the pair  $x, y$  on the stack  $EQ$ , and `stackFree()` erases (pops) the top of the stack.

For the net  $\Gamma$ , there are three kinds of graph element: agents, variables and indirection nodes. These will be represented as allocated memory on a heap to hold the kind of the element and pointers to other elements connected on the auxiliary ports. We keep all cell sizes fixed corresponding to the maximum arity of the system being encoded. Graphically, the elements are represented as shown in Figure 5, where we fix the arity to 3 and we show an agent  $\alpha$  of arity 3,  $\beta$  of arity 2 and the variable and indirection node. Agent names are represented as numbers also. Names of variables are not important once the net is compiled.

Agents and indirection nodes can point to other graph elements; variables represent leaves of the structure and point no further. All these elements can be pointed to from other elements, in addition to the interface  $I$  or an element of the active pair stack  $EQ$ . This single linked structure is sufficient to represent nets and keeps the data manipulation simple and efficient. The instructions to create agents, variables and indirection nodes are: `x=mkAgent(alpha)`, `y=mkVar()`, `z=mkInd()`. These instructions allocate a block of memory to store the name of the element and a number of pointers. We also have a simple

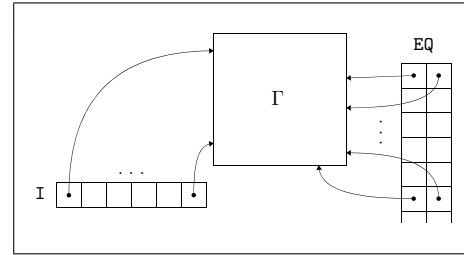


Figure 4: Configuration

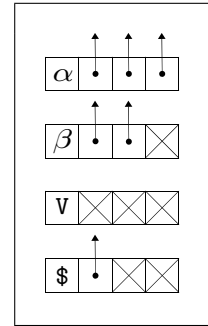


Figure 5: Graph elements

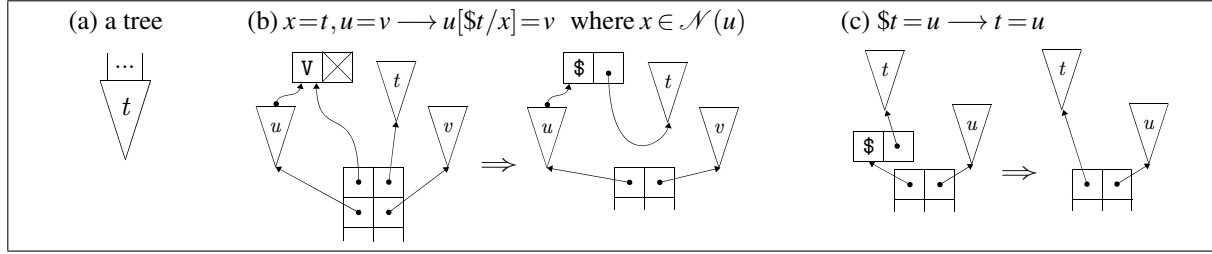


Figure 7: depicted agents as trees and computation rules

assignment language so that the structure can be modified. For example,  $x[1]=y$  sets the first pointer of  $x$  to  $y$ . Figure 6 shows an instruction list to create an equation  $Add(r, Z) = Z$  in Example 4.

When agents are connected together, they are trees as depicted in Figure 7 (a), where the free names are at the top of the tree: We can also show each of the computation rules represented as transformations on the data structure. Figure 7 (b) and (c) are instances of the VAR and the INDIRECTION rules to illustrate the ideas.

**Compilation.** Using this machinery we are now able to give a direct compilation of the calculus of nets in to the low-level language. This resulting code will build the data structures outlined above, and can then be evaluated. A configuration:  $\langle \vec{u} \mid \Delta \rangle$  will be compiled into the following structure: Code to set up the basic run-time structure, i.e. interface and equation stacks, code for each of the rules, and code to build the initial net, initialising the interface and the equation stack.

The only data structures that we need are the interface, the active pair stack and the initial net. The code that we generate will create, manipulate and destroy the net elements. In the full paper we provide the definitions for each of these with examples and benchmark results comparing other data representations.

## 5 Conclusion

In this short paper we have demonstrated that it is possible to design a calculus for interaction nets that has a direct relationship with the internal representation (data structure). As a consequence, we can use the calculus to reason about the rewriting process, and also to study the cost of reduction. This leads to an investigation into optimising rules which will be given in detail in the full paper together with benchmark results comparing different representations.

## References

- [1] Maribel Fernández and Ian Mackie. A calculus for interaction nets. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *Lecture Notes in Computer Science*, pages 170–187. Springer-Verlag, September 1999.
- [2] Abubakar Hassan, Ian Mackie, and Shinya Sato. Compilation of interaction nets. *Electr. Notes Theor. Comput. Sci.*, 253(4):73–90, 2009.
- [3] Abubakar Hassan, Ian Mackie, and Shinya Sato. A lightweight abstract machine for interaction nets. *ECE-ASST*, 29, 2010.
- [4] Yves Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, 1990.
- [5] Sylvain Lippi.  $\text{in}^2$ : A graphical interpreter for interaction nets. In Sophie Tison, editor, *RTA*, volume 2378 of *Lecture Notes in Computer Science*, pages 380–386. Springer, 2002.

```

a = mkAgent(Add)
z1 = mkAgent(Z)
r = mkVar()
a[1] = r
a[2] = z1
z2 = mkAgent(Z)
push(a, z2)

```

Figure 6: Instructions for  $Add(r, Z) = Z$