Sequential and Concurrent Abstract Machines for Interaction Nets

Jorge Sousa Pinto[⋆]

Laboratoire d'Informatique (LIX - CNRS UMR 7650)
École Polytechnique
91128 Palaiseau Cedex, France
`pinto@lix.polytechnique.fr`

Abstract. This paper is a formal study of how to implement interaction nets, filling an important gap in the work on this graphical rewriting formalism, very promising for the implementation of languages based on the λ-calculus. We propose the first abstract machine for interaction net reduction, based on a decomposition of interaction rules into more atomic steps, which tackles all the implementation details hidden in the graphical presentation. As a natural extension of this, we then give a concurrent shared-memory abstract machine, and show how to implement it, resulting in the first parallel implementation of interaction nets.

1 Introduction

Interaction Nets (INs) are an extension of Proof-Nets for the multiplicative fragment of Linear Logic [5], proposed by Yves Lafont [8, 9] as a simple and inherently parallel graphical formalism for programming. By way of a number of translations of the λ-calculus, interaction nets have proved to be a useful new paradigm for implementing functional languages, specifically when controlling the sharing of terms is a priority. The research effort has however been directed more at these translations than at the implementation of interaction net reduction – in particular, no parallel implementations exist. In this paper we study the sequential and concurrent implementation of interaction nets, by means of abstract machines in which interaction steps are decomposed into simple machine operations.

The interest of interaction nets for functional programming is twofold: on one hand they allow to control the amount of shared reductions performed. In particular, they have been used for the implementation of Optimal Reduction (as formalized in [12], and brought to practice in successive studies [10, 6, 1]), but also other efficient strategies for the λ-calculus have been proposed [13].

On the other hand, we have their potential (which has not yet been fully explored) to be implemented *in parallel*: unlike general graph-rewriting systems,

[⋆] On leave from Universidade do Minho. Research partially supported by PRAXIS XXI grant BD/11261/97.

interaction nets possess locality and confluence properties which allow for all the active pairs in a given net to be reduced simultaneously, without interference.

Implementations of INs have to this point been quite ad-hoc, since the formalism, being a graphical representation, has always been assumed to be trivial, and no formal studies of its implementation have been undertaken. However, there is a considerable gap between this presentation and a running implementation, in the sense that each basic interaction rewriting step may require many 'rewiring' steps, implemented by non-trivial sequences of machine operations.

Additionally, a decomposition of interaction steps into sequences of basic, close-to-machine operations is essential if parallel implementations are to be studied. We may then investigate whether the benefits are limited to the potential parallelism contained in the nets (simultaneous redexes) or if there are other opportunities for parallelizing, at the level of small-grain machine operations.

Abstract machines for the λ-calculus such as the SECD machine [11] or Krivine's machine [3] have been proposed as implementation devices encompassing (and decomposing) both the β-reduction relation and the variable substitution mechanism. A mechanism along these lines does not however exist for interaction nets. In this paper we introduce such an abstract machine, providing a suitable decomposition of interaction rewriting steps into fine grain operations.

From this machine we then obtain a concurrent (multi-threaded, shared memory) version as a simple generalization, where basic machine tasks are distributed among threads. This may be implemented on any platform offering support for multi-threaded computation, although some care is required to guarantee correctness. The result is the first parallel reducer for interaction nets.

Structure of the Paper. We start by briefly reviewing interaction nets and some details of their implementation. In Sect.3 we introduce notation and then define machine configurations as appropriate tuples of data-structures, and show how to obtain a configuration from a net, before giving the definition of the sequential abstract machine. Section 4 is devoted to the study of correctness of this machine. We then present the concurrent machine in Sect.5. In Sect.6 we mention some implementation aspects, notably with respect to the parallel implementation of the concurrent abstract machine, and finally conclude in Sect.7.

2 Background and Motivation

An interaction net is an undirected graph built from a set of cells or agents, each of which contains a *principal port*, and a number (possibly zero) of *auxiliary ports*. Edges in this graph connect any two ports, but no more than one edge may be connected to the same port. A *free port* in the net has a hanging edge connected to it, i.e, an edge which is not connected to anything at the other extremity. The *observable interface* of a net is the set of its free ports arranged in a sequence. Computation (in the form of graph-rewriting) takes place only at special edges of the net, those connecting two cells by their principal ports. Such a pair of cells is called an *active pair*, and it may be rewritten using an

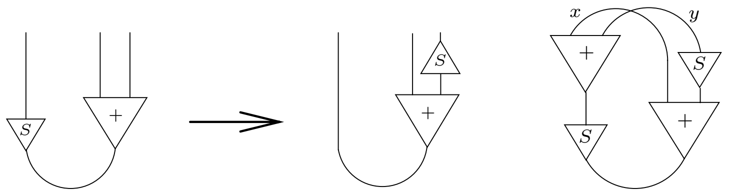

Fig. 1. Example interaction rule and its representation using Lafont's notation

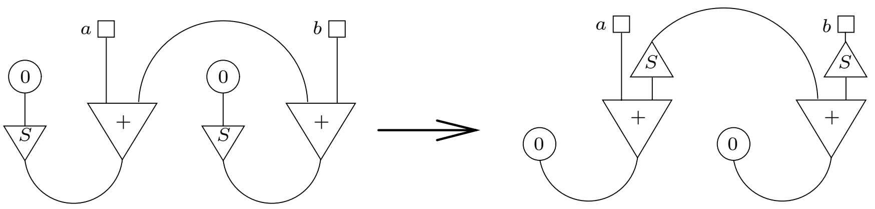

Fig. 2. Example interaction net

appropriate *interaction rule.* An interaction system specifies a set of agents (from which to build nets) and rules (with which to rewrite them).

An Example. The left-hand side of Fig.1 represents an interaction rule in a system containing agents (0, S, and $+$) and rules for natural numbers arithmetic. The cells are the following: 0 is a single-port agent, representing the constant 0. S is a constructor with two ports, the principal port representing the successor of the number in the auxiliary port. Finally, the $+$ agent has two auxiliary ports, one of which is for the sum of the numbers in the principal port and in the other auxiliary port. We define addition inductively on its first argument, so this must be associated to a principal port, where interaction is possible.

The rule is to be applied as a graph-rewriting rule: in a net in which a sub-net matching the left side of the rule occurs, this sub-net may be substituted by the net on the right. The *interface preservation* property ensures that there is a wire in the right-hand side net to connect to every wire left hanging when the sub-net is removed from the initial net. Figure 2 is an example of how this rule can be applied. We distinguish observable values (a and b) graphically using small squares. In the figure, the two active pairs are reduced in parallel, using the rule in Fig.1. The reader will have no difficulty in writing the interaction rule (between the 0 and $+$ agents) that would be needed for reduction to proceed.

The motivations for parallelism can be easily understood. Interaction always happens *locally,* when two cells are connected via their principal ports. The rewritable elements are always pairs of cells, so any cell can be involved in at most one such element; no critical pairs exist. Any two active pairs in a net can be rewritten in arbitrary order; strong local confluence holds, resulting in the fact that the sequences of rewriting steps used in the normalization of a net are all permutations of the same set of individual rewrites.

Implementation Issues. This very simple example raises important questions:

1. What data-structures are used to represent interaction nets, and how can the graphically trivial notion of rewiring be formalized and implemented?
2. How is the observable interface updated (this is a particular case of rewiring)?
3. Each application of an interaction rule involves producing a copy of its right-hand side. How is this accounted for?
4. How are active pairs identified? For instance, in the right-hand side of Fig.2, 0 and $+$ are connected by their principal ports, but $+$ and S, even though they are connected, do not form a new active pair.
5. What are the space and time resources required by each operation?
6. If the implementation is parallel, what model of concurrency is used?
7. What are the control mechanisms used for granting correct access to shared resources by the different parallel processing elements?

An abstract machine, by decomposing interaction into atomic operations, should provide answers to these questions, and should be directly implementable.

A Language for Interaction Nets. The language we use to describe nets was originally given by Lafont [8] and developed in [4]. Agents are written as algebraic constructors with arity equal to the number of auxiliary ports. Active pairs are then equations, equalities between terms with variables. A wire linking two leaves (two auxiliary ports) in two such terms (trees) is represented by two occurrences of the same variable. Variables are allowed as members in equations, to allow for modular descriptions. Each variable occurs *exactly twice* in the net.

A net may then be described as a pair $\langle \mathbf{t} \mid \Delta \rangle$, with $\mathbf{t}$ a sequence of terms (its observable interface) and Δ a multiset of equations. With respect to rules, each one may be represented succinctly as a net with one active pair (and empty observable interface), by wiring together each free port occurring in the left-hand side of the rule and the corresponding port in its right-hand side (see Fig. 1).

Semantics. We briefly review a calculus for interaction nets [4], inspired by the Chemical Abstract Machine [2]. Let $\rightleftharpoons$ be the smallest equivalence satisfying the structural rules $\Delta, t = u, \Theta \rightleftharpoons \Delta, u = t, \Theta$ and $\Delta, t = u, v = w, \Theta \rightleftharpoons \Delta, v = w, t = u, \Theta$. $\mathcal{N}$ is the function giving the set of variables occurring in a term. We give a set of conditional reduction rules for interaction nets. We assume no variable occurs simultaneously in a rule and in the net.

Interaction: $(\alpha(t'_1, \ldots, t'_n), \beta(u'_1, \ldots, u'_m))$ is an interaction rule $\Rightarrow$
$\langle \mathbf{t} \mid \alpha(t_1 \ldots t_n) = \beta(u_1 \ldots u_m), \Gamma \rangle \longrightarrow$
$\langle \mathbf{t} \mid t_1 = t'_1, \ldots, t_n = t'_n, u_1 = u'_1, \ldots, u_m = u'_m, \Gamma \rangle$.
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: $\Theta \rightleftharpoons^* \Theta', \langle \mathbf{t_1} \mid \Theta' \rangle \longrightarrow \langle \mathbf{t_2} \mid \Delta' \rangle, \Delta' \rightleftharpoons^* \Delta \;\Rightarrow\; \langle \mathbf{t_1} \mid \Theta \rangle \longrightarrow \langle \mathbf{t_2} \mid \Delta \rangle$.

Properties. Some properties of this system, proved in [4], are *strong local confluence* (diamond property) and *uniqueness of normal forms.* A standard result of (abstract) rewrite systems also yields, as a consequence of strong confluence:

Lemma 1. *Any normalizing interaction net is* strongly *normalizing.*

Remarks. Each variable occurs exactly once in the term (or list) where it is substituted, and no two rules may perform substitution of the same variable.

The above semantics defines a notion of canonical forms for interaction nets: $P \Downarrow Q$ iff $P \longrightarrow^* Q$ and $Q \not\longrightarrow$. These canonical forms correspond to irreducible nets $\langle \mathbf{t} \mid \varepsilon \rangle$ or $\langle \mathbf{t} \mid \mathcal{LC} \rangle$, with $\mathcal{LC}$ a list of cycles of the form $x = t$, with $x \in \mathcal{N}(t)$.

Structural Equivalence of Interaction Nets. Two interaction nets are *α-convertible*, written $\mathcal{A} \approx_\alpha \mathcal{B}$, if they are the same up to renaming of variables. We define *structural equivalence* ($\equiv$) as satisfying $\mathcal{A} \equiv \langle \mathbf{t} \mid \Delta \rangle$ whenever $\mathcal{A} \approx_\alpha \langle \mathbf{t} \mid \Theta \rangle$ and $\Theta \rightleftharpoons \Delta$. This is clearly an equivalence relation that is preserved by reduction.

3 A Sequential Abstract Machine for Interaction Nets

Interaction Systems. An interaction system is a tuple $\langle \Sigma, \mathcal{R}, \mathcal{V} \rangle$, where Σ is a set of *agents* and $\mathcal{R}$ is a set of *interaction rules.* Greek letters $\alpha, \beta, \ldots$ range over agents. n^α is the *arity* of agent α. $\mathcal{V}$ is the set of variables in the system, ranged over by $x, y \ldots$, and which allows one to define the set *Terms* for this system, which are either variables or *agent terms* of the form $\alpha(t_1, \ldots, t_{n^\alpha})$, with $t_1, \ldots, t_{n^\alpha} \in$ *Terms*. We will sometimes write this as $\alpha(\boldsymbol{t})$.

The function $\mathcal{N}$: *Terms* $\rightarrow \mathcal{P}(\mathcal{V})$ returns the set of variables in a term (it can be trivially extended to pairs, sequences, and sequences of pairs of terms). The $\widehat{\ }$ operator is applied to a rule to produce a copy of it in which all variable names are fresh (w.r.t. a certain context – a machine configuration) and unique.

We then define $\mathcal{T}$ as the subset of *Terms* in which each variable occurs at most once. This linearity condition may be generalized to lists of terms and lists of pairs of terms, allowing substitution to be defined trivially as assignment, since no erasing or copying of the substituted term will happen: $t[u/x] = t[x := u]$.

Each rule $r \in \mathcal{R}$ in the system is a tuple $r = (t_1, t_2, \phi_r)$ where $t_1, t_2 \in \mathcal{T}$ but $t_1, t_2 \notin \mathcal{V}$, $\mathcal{N}(t_1) \cap \mathcal{N}(t_2) = \emptyset$, and $\phi_r : \mathcal{N}(t_1) \cup \mathcal{N}(t_2) \rightarrow \mathcal{N}(t_1) \cup \mathcal{N}(t_2)$ is a fixpoint-free involutive permutation (or involution) on variables, i.e, $y = \phi_r(x)$ implies $x = \phi_r(y)$, and $\phi_r(x) \neq x$. We shall denote by $\phi_r[x \leftrightarrow y]$ the permutation which maps x to y, y to x, and any other variable z to $\phi_r(z)$. We require that no two interaction rules exist in $\mathcal{R}$ for the same pair of agents, and also that $\mathcal{R}$ is closed under symmetry so that the order of t_1 and t_2 is irrelevant. To write rules as given before in this framework, it suffices to give different names to the two occurrences of each variable, and store the linking information in ϕ_r.

We finally define a set $\mathcal{T}_a$ of *annotated terms.* These are either variables, or terms of the form $\{X\}.t$ with $t \in \mathcal{T}$, and X a list of variables, possibly containing the symbol $\square$. We shall see in Sect.3 how these annotations will be used.

Configurations. A machine configuration is a tuple $\langle \Gamma \mid S \mid \phi_N \mid V \mid C \rangle$, where no variable occurs repeatedly in two different components of the tuple:

– $\Gamma : \mathcal{V} \to \mathcal{T}_a$ is a *Heap*, a mapping such that $x \in dom(\Gamma)$ implies $x \notin \{\mathcal{N}(\Gamma(y)) \mid y \in dom(\Gamma)\}$, and $x, y \in dom(\Gamma)$ implies $\mathcal{N}(\Gamma(x)) \cap \mathcal{N}(\Gamma(y)) = \emptyset$. We use the notation $\Gamma[x \mapsto \{N\}.t]$ to refer to the heap which maps x to $\{N\}.t$ and any other variable y to $\Gamma(y)$;
– $S \in (\mathcal{T}_a \times \mathcal{T}_a)^*$ is a sequence of *Pairs* of terms, representing equations;
– $\phi_N : N \to N$ is a fixpoint-free *Involution* on variables, with $N = \mathcal{N}(S) \cup \mathcal{N}(V) \cup \mathcal{N}(C) \cup dom(\Gamma) \cup \{\mathcal{N}(\Gamma(x)) \mid x \in dom(\Gamma)\}$;
– $V \in (\mathcal{T}_a \cup \{\Box\})^*$ is the *Observable Interface* of the net (a sequence of terms);
– $C \in ((\mathcal{V} \times \mathcal{T}_a) \cup \{\Box\})^*$ is a sequence of *Cycles.*

Interaction Functions. We will use the usual $[\ldots]$ notation for lists, ε for the empty list, : for *cons*, @ for *append.* For an interaction system $\mathcal{S}$, we define:

$$\begin{aligned}\mathcal{I}_{\mathcal{S}}(\alpha(\boldsymbol{t}), \beta(\boldsymbol{u})) &= [(\nu(t_1), \nu(tt_1)), \ldots (\nu(t_{n^\alpha}), \nu(tt_{n^\alpha})), \\ &\qquad (\nu(u_1), \nu(uu_1)), \ldots (\nu(u_{n^\beta}), \nu(uu_{n^\beta}))]; \\ \Phi_{\mathcal{S}}(\alpha(\boldsymbol{t}), \beta(\boldsymbol{u})) &= \phi\phi_r;\end{aligned}$$

where $r = (\alpha(\boldsymbol{t'}), \beta(\boldsymbol{u'}), \phi_r) \in \mathcal{R}$, $\widehat{r} = (\alpha(tt_1, \ldots tt_{n^\alpha}), \beta(uu_1, \ldots uu_{n^\beta}), \phi\phi_r)$ is a fresh copy of r, and ν annotates (agent) terms with a sequence of the variables occurring in them: $\nu(x) = x$, $\nu(\alpha(\boldsymbol{t})) = \{an(\alpha(\boldsymbol{t})) : \Box : \varepsilon\}\,.\alpha(\boldsymbol{t})$, with *an* defined by $an(x) = [x] \mid an\,(\alpha(t_1, \ldots t_{n^\alpha})) = an(t_1)$ @ $\cdots$ @ $an(t_{n^\alpha})$. We will denote by $\circ$ an auxiliary function on lists for removing the first occurrence of the $\Box$ mark.

Loading the Abstract Machine. In order to obtain initial configurations $\Sigma[\mathbf{t} \mid \Delta]$ corresponding to an interaction net $\langle \mathbf{t} \mid \Delta \rangle$ we first need to obtain a net $\langle \mathbf{t'} \mid \Delta' \rangle$ by splitting variables and linking split pairs in an involution ϕ_N. Then $\Sigma[\mathbf{t} \mid \Delta]$ is any configuration $\langle\, \emptyset \mid \nu_{lp}(seq(\Delta')) \mid \phi_N \mid \nu_l(\mathbf{t'}) : \Box : \varepsilon \mid \Box : \varepsilon\, \rangle$, where $seq(\Delta')$ is a list obtained by arbitrarily ordering the set Δ', and ν_l, ν_{lp} generalize the previously defined ν for lists of terms and lists of pairs of terms, respectively.

The basic idea. We describe the machine succinctly: equation pairs are stored in the list S. Each execution step pops a pair, and an appropriate rule is selected (by pattern-matching) to handle that pair.

If it is an active pair $(\alpha(\boldsymbol{t}), \beta(\boldsymbol{u}))$, an interaction will be performed, invoking the interaction functions and pushing the newly generated pairs $\mathcal{I}_{\mathcal{S}}(\alpha(\boldsymbol{t}), \beta(\boldsymbol{u}))$ onto the stack, and including in the involution the pairs given by $\Phi_{\mathcal{S}}(\alpha(\boldsymbol{t}), \beta(\boldsymbol{u}))$.

If it is a pair of variables (x, y) such that $\phi_N(x) = z$ and $\phi_N(y) = w$, we remove from ϕ_N the pairs $x \leftrightarrow z$ and $y \leftrightarrow w$, and add to it the pair $z \leftrightarrow w$.

For a pair $(x, \alpha(\boldsymbol{t}))$, we simply create a new entry $x \mapsto \alpha(\boldsymbol{t})$ in the heap.

The machine stops when S is empty. For now we consider that the sequence of pairs is accessed using a LIFO strategy, thus as a stack. However it is not important how we implement the multiset of equations as a linear data-structure; by opting for a stack we simply increase the locality of the machine.

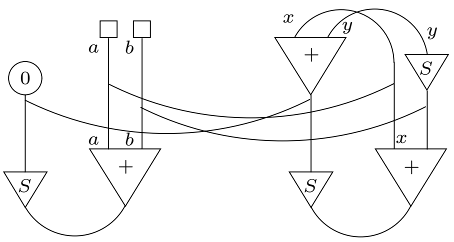

Fig. 3. Example of the rewiring involved in the application of an interaction rule

Rewiring is handled by the two last cases. Using proof-net terminology, a cut between two axioms gives origin to a single axiom, whereas a cut between an axiom and any other net results in this last net being stored in the heap. For this reason, final configurations are totally contained in the heap, since the information progressively stored in it has not been used anywhere. We thus need a set of *post-processing* rules that will traverse the observable interface and update every variable x_i in it with the value stored in the heap for $\phi_N(x_i)$. We shall not worry about post-processing here.

An Example. Let us see how this machine deals with a simplified version of the example in Sect.2. First observe, on the right in Fig.1, the interaction rule for the pair $(+, S)$ redrawn to reflect our notation for rules, giving $r_1 = (S(+(x_1, y_1)), +(x_2, S(y_2)), \{x_1 \leftrightarrow x_2, y_1 \leftrightarrow y_2\})$.

We show in Fig.3 a net consisting of a single active pair, together with a copy of the rule mentioned. The rewiring that needs to be done is shown in the picture, and consists in wiring together all the corresponding arguments of each agent in the active pair in the net and in the rule (and removing the active pair from both). Formally, we start with an initial configuration

$$\Sigma_0 = \langle \emptyset \mid [(S(0), +(a_2, b_2))] \mid \{a_1 \leftrightarrow a_2, b_1 \leftrightarrow b_2\} \mid [a_1, b_1, \square] \mid [\square] \rangle,$$

The first step of the abstract machine produces a new configuration by popping the pair from the stack and invoking the interaction functions:

$$\begin{aligned} \mathcal{I}_{\mathcal{S}}(S(0), +(a_2, b_2)) &= [(b_2, S(y_2)), (a_2, x_2), (0, +(x_1, y_1))], \\ \Phi_{\mathcal{S}}(S(0), +(a_2, b_2)) &= \{x_1 \leftrightarrow x_2, y_1 \leftrightarrow y_2\}, \end{aligned}$$

resulting in the configuration

$$\begin{aligned} \Sigma_1 = \langle \emptyset \mid & [(b_2, S(y_2)), (a_2, x_2), (0, +(x_1, y_1))] \\ & \mid \{a_1 \leftrightarrow a_2, b_1 \leftrightarrow b_2, x_1 \leftrightarrow x_2, y_1 \leftrightarrow y_2\} \mid [a_1, b_1, \square] \rangle. \end{aligned}$$

The pair at the top is now made of a variable and an agent term, which will now be associated to the variable, in the heap.

$$\begin{aligned} \Sigma_2 = \langle \{b_2 \mapsto S(y_2)\} \mid & [(a_2, x_2), (0, +(x_1, y_1))] \\ & \mid \{a_1 \leftrightarrow a_2, b_1 \leftrightarrow b_2, x_1 \leftrightarrow x_2, y_1 \leftrightarrow y_2\} \mid [a_1, b_1, \square] \rangle. \end{aligned}$$

The next pair to be popped has two variables, so we update the involution:

$$\Sigma_3 = \langle \{b_2 \mapsto S(y_2)\} \mid [(0, +(x_1, y_1))] \mid \{a_1 \leftrightarrow x_1, b_1 \leftrightarrow b_2, y_1 \leftrightarrow y_2\} \mid [a_1, b_1, \Box] \rangle.$$

We shall not proceed with the reduction. Instead we show the result of updating the Observable Interface of Σ_3 with the values stored in the heap:

$$\Sigma_3' = \langle \emptyset \mid [(0, +(x_1, y_1))] \mid \{a_1 \leftrightarrow x_1, y_1 \leftrightarrow y_2\} \mid [\Box, a_1, S(y_2)] \rangle.$$

A Machine with Substitutions. The above machine has some drawbacks that we will eliminate by refining its definition. The first problem has to do with cycles: in fact, not every pair $(x, \alpha(\boldsymbol{t}))$ should be moved to the heap, since it may happen that $\phi_N(x) \in \mathcal{N}(\boldsymbol{t})$. This is an example of a vicious circle, which may be generated during reduction. Our machine would store $x \mapsto \alpha(\ldots, \phi_N(x), \ldots)$ in the heap. Now since $\phi_N(x)$ occurs only in the term associated with x, its value may never be substituted in the interface, thus this part of the net will be forever lost in the heap. Configurations where this happens are called *degenerate.*

We must then include a special structure (C) in configurations, to store these pairs, and the machine must specify how the test $\phi_N(x) \in \mathcal{N}(\boldsymbol{t})$ is performed.

A different kind of degenerate configuration exists, in which active pairs, rather than cycles, are irrecoverably contained in the heap. Suppose for instance the stack looks again like $(x, \alpha(\boldsymbol{t})) : (y, \beta(\boldsymbol{u})) : \cdots$, but we now have $\phi_N(x) = y$. After a first step of operation we get a heap $\Gamma[x \mapsto \alpha(\boldsymbol{t})]$ and stack $(y, \beta(\boldsymbol{u})) : \cdots$. A second machine step produces a heap $\Gamma[x \mapsto \alpha(\boldsymbol{t}), y \mapsto \beta(\boldsymbol{u})]$, and the active pair is lost forever in it, since no substitution of x or y can be made.

It is easy to show other examples where degenerate configurations are created, either with lost cycles or lost active pairs. We will solve this problem by performing substitution of variables stored in the heap *during* the operation of the machine. For every non-active pair at the top of the stack, we will exhaustively substitute variables before actually popping it.

Looking again at the previous example, instead of performing the second step given above, we would substitute the variable y with the term $\Gamma[\phi_N(y)]$, to get the stack $(\alpha(\boldsymbol{t}), \beta(\boldsymbol{u})) : \cdots$, where the active pair has been recovered.

Implementing Substitutions on the Top of the Stack. In order to allow for the exhaustive substitution of variables on the top of the stack to be handled efficiently, we add to each term an *annotation list* containing all the variables in it. These are included in the configurations when the machine is loaded, and kept up-to-date (at a low cost) by every machine rule. Instead of traversing a whole term (a tree) looking for variables with values in the heap, we simply traverse *circularly* this annotation structure. The $\Box$ mark is initially placed at the end of every such annotation list so it can be detected when it has been fully traversed.

The Abstract Machine. We will now reformulate our configurations by including explicitly a *processing element* or *thread.* Configurations will be of the form $\langle \Gamma \mid S \mid \phi_N \mid V \mid C \mid t \rangle$, with t a *Thread*, built from the following signature:

$$\begin{array}{rlrl} \mathsf{process}: & \mathcal{T}_a \times \mathcal{T}_a \to Thread & \mathsf{enlist}: & (\mathcal{T}_a \times \mathcal{T}_a)^* \to Thread \\ \mathsf{delist}: & Thread & \mathsf{cycle}: & \mathcal{V} \times \mathcal{T}_a \to Thread \end{array}$$

Each operator corresponds to a state of the machine, in which it performs one specific action. When it is $\mathsf{process}(t,u)$, it processes the pair (t,u) using the rules previously outlined; when it is delist it will take a pair from the Pairs stack to be processed; when it is $\mathsf{enlist}(l)$ it will add to the stack all the pairs in l; finally, when it is $\mathsf{cycle}(x,t)$ it will add the cycle (x,t) to the Cycles structure. The abstract machine is loaded with $t = \mathsf{delist}$, and stops with $S = \varepsilon$ and $t = \mathsf{delist}$.

We give in Table 1 the abstract machine rules, where the families of rules I to III correspond to pair-processing as sketched before (including variable substitution), and Family IV manages the life-cycle of the thread.

As an example of the flexibility of this presentation of the machine, consider what needs to be changed if we want to access the Pairs structure as a Queue: rule T.2 simply has to give instead $S@[(t,u)]$ in its right-hand side.

A final remark: observe that all the machine rules perform simple tasks such as assignments and rotating lists one position, except for rule I, which depends of course on the size of the right-hand side of the interaction rule applied.

Properties. It is immediate to verify the determinism of this set of rules. Most of the proofs of properties stated here are left for the long version of this paper.

Definition 1 (Correct and Complete Annotations). *We say a configuration has* correct annotations *if for every annotated term* $\{A\}.\alpha(\boldsymbol{t})$ *occurring in it we have* $\mathbf{set}(\overset{\circ}{A}) \subseteq \mathcal{N}(\boldsymbol{t})$. *It has* complete annotations *if* $\mathbf{set}(\overset{\circ}{A}) \supseteq \mathcal{N}(\boldsymbol{t})$.

Proposition 1. *Let* $\langle \mathsf{t} \mid \Delta \rangle$ *be an interaction net. If* $\Sigma[\mathsf{t} \mid \Delta] \longrightarrow^* \Sigma'$, *then* Σ' *has correct and complete annotations.*

Proof. Straightforward induction on the reductions. $\Sigma[\mathsf{t} \mid \Delta]$ has correct and complete annotations, and every machine rule preserves the property. □

Definition 2 (Degeneracy). *A configuration is* degenerate *if the heap contains an active pair* $\{x \mapsto \alpha(\boldsymbol{t}), y \mapsto \beta(\boldsymbol{u})\}$ *with* $\phi_N(x) = y$, *or a cycle* $\{x_i \mapsto t_i\}, i = 1 \ldots N$, *with* $\phi_N(x_i) \in \mathcal{N}(t_{i+1})$ *for* $i = 1 \ldots N-1$, *and* $\phi_N(x_N) \in \mathcal{N}(t_1)$.

Proposition 2. *If* $\Sigma[\mathsf{t} \mid \Delta] \longrightarrow^* \Sigma'$, $\langle \mathsf{t} \mid \Delta \rangle$ *is an IN, then* Σ' *is not degenerate.*

We remark that if $\Sigma \longrightarrow \Sigma'$, Σ' may be degenerate even if Σ is not.

4 Correctness of the Sequential Abstract Machine

Our first goal will now be to define the interpretation of a machine configuration as an interaction net. We will then introduce some notation and give several lemmas needed for the correctness proof.

Table 1. Abstract machine rules

I	Permut	ϕ_N	$\phi_N \cup \Phi_{\mathcal{S}}\,(\alpha(\boldsymbol{t}), \beta(\boldsymbol{u}))$
	Thread	process $(\{l_1\}.\alpha(\boldsymbol{t}),\{l_2\}.\beta(\boldsymbol{u}))$	enlist $(\mathcal{I}_{\mathcal{S}}\,(\alpha(\boldsymbol{t}), \beta(\boldsymbol{u})))$

II.1	Permut	$\phi_N[x \leftrightarrow y]$	$\phi_N[x \leftrightarrow y]$
	Thread	process(x, y)	cycle(x, y)
II.2	Heap	$\Gamma\,[z \mapsto \{X\}.\sigma(\boldsymbol{v})]$	Γ
	Permut	$\phi_N[x \leftrightarrow z]$	ϕ_N
	Thread	process(x, y)	process $(\{X\}.\sigma(\boldsymbol{v}), y)$
II.3	Heap	$\Gamma\,[z \mapsto \bot, w \mapsto \{Y\}.\tau(\boldsymbol{u})]$	Γ
	Permut	$\phi_N[x \leftrightarrow z, y \leftrightarrow w]$	$\phi_N[x \leftrightarrow z]$
	Thread	process(x, y)	process $(x, \{Y\}.\tau(\boldsymbol{u}))$
II.4	Heap	$\Gamma\,[z \mapsto \bot, w \mapsto \bot]$	Γ
	Permut	$\phi_N[x \leftrightarrow z, y \leftrightarrow w]$	$\phi_N[z \leftrightarrow w]$
$y \neq z, x \neq w$	Thread	process(x, y)	delist

III.0	Thread	process $(\{Y\}.\alpha(\boldsymbol{t}), x)$	process $(x, \{Y\}.\alpha(\boldsymbol{t}))$
III.1	Heap	$\Gamma\,[x \mapsto \{X\}.\beta(\boldsymbol{u})]$	Γ
	Permut	$\phi_N[x \leftrightarrow y]$	ϕ_N
	Thread	process $(z, \{y : T\}.\alpha(\boldsymbol{t}))$	process$(z,\{(\overset{\circ}{X})\,@T\}.\alpha(\boldsymbol{t})[y := \beta(\boldsymbol{u})])$
III.2	Heap	$\Gamma\,[x \mapsto \bot]$	Γ
	Permut	$\phi_N[x \leftrightarrow y]$	$\phi_N[x \leftrightarrow y]$
$x \neq z$	Thread	process $(z, \{y : T\}.\alpha(\boldsymbol{t}))$	process $(z, \{T@[y]\}.\alpha(\boldsymbol{t}))$
III.3	Permut	$\phi_N[x \leftrightarrow y]$	$\phi_N[x \leftrightarrow y]$
	Thread	process $(x, \{y : T\}.\alpha(\boldsymbol{t}))$	cycle $(x, \{y : T\}.\alpha(\boldsymbol{t}))$
III.4	Heap	$\Gamma\,[x \mapsto \{X\}.\beta(\boldsymbol{u})]$	Γ
	Permut	$\phi_N[x \leftrightarrow z]$	ϕ_N
	Thread	process $(z, \{\Box : T\}.\alpha(\boldsymbol{t}))$	process $(\{X\}.\beta(\boldsymbol{u}), \{\Box : T\}.\alpha(\boldsymbol{t}))$
III.5	Heap	$\Gamma[x \mapsto \bot]$	$\Gamma\,[z \mapsto \{T@[\Box]\}.\alpha(\boldsymbol{t})]$
	Permut	$\phi_N[x \leftrightarrow z]$	$\phi_N[x \leftrightarrow z]$
	Thread	process $(z, \{\Box : T\}.\alpha(\boldsymbol{t}))$	delist

T.1	Pairs	$(t, u) : S$	S
	Thread	delist	process(t, u)
T.2	Pairs	S	$(t, u) : S$
	Thread	enlist$((t, u) : T)$	enlist(T)
T.3	Thread	enlist(ε)	delist
T.4	Cycles	C	$(t, u) : C$
	Thread	cycle(t, u)	delist

Definition 3 (Updating). *Let $\Sigma = \langle \Gamma \mid S \mid \phi_N \mid V \mid C \mid t\rangle$. The* update *of Σ is the configuration $\mathbf{U}[\Sigma]$ that results from recursively substituting (updating annotations) every variable x occurring in any component S, V, C, t of Σ with the value stored for the variable $\phi_N(x)$ in the heap Γ, removing this entry from the heap and the pair $x \leftrightarrow \phi_N(x)$ from the permutation.*

It is straightforward to expand the post-processing rules with others for updating the stack and the thread. Together they implement the above notion of updating.

Proposition 3. *If Σ is non-degenerate then $\mathbf{U}[\Sigma]$ has an empty heap.*

Definition 4 (Collapsing). *From an interaction net $N = \langle \mathbf{t} \mid \Delta\rangle$ and an involution ϕ we obtain the* collapse *of N by ϕ, a net denoted by $\mathbf{Clp}[N, \phi]$, by substituting in $\mathbf{t}$ and Δ every pair of variables x, y such that $\phi(x) = y$ with a fresh variable, which we will by convention call k_{xy}.*

Auxiliary Functions. We need a family of auxiliary functions defined as follows: $\lfloor\rfloor_0$ takes a term and removes its annotation, $\lfloor x\rfloor_0 = x \mid \lfloor\{N\}.\alpha(\boldsymbol{t})\rfloor_0 = \alpha(\boldsymbol{t})$; $\lfloor\rfloor_1$ takes a list of terms and removes the $\Box$ mark as well as all the annotations, $\lfloor\varepsilon\rfloor_1 = \varepsilon \mid \lfloor\Box : t\rfloor_1 = \lfloor t\rfloor_1 \mid \lfloor h : t\rfloor_1 = \lfloor h\rfloor_0 : \lfloor t\rfloor_1$; $\lfloor\rfloor_2$ takes a list of pairs of terms and returns the corresponding multiset of equations, $\lfloor\varepsilon\rfloor_2 = \emptyset \mid \lfloor\Box : t\rfloor_2 = \lfloor t\rfloor_2 \mid \lfloor(h_1, h_2) : t\rfloor_2 = \{\lfloor h_1\rfloor_0 = \lfloor h_2\rfloor_0\} \cup \lfloor t\rfloor_2$. Finally, $\lfloor\rfloor_t$ takes a thread and returns a set containing the pair being processed by the thread, or the empty set if the thread does not contain any pair, $\lfloor\mathsf{process}(t_1, t_2)\rfloor_t = \{\lfloor t_1\rfloor_0 = \lfloor t_2\rfloor_0\} \mid \lfloor\mathsf{delist}\rfloor_t = \emptyset \mid \lfloor\mathsf{enlist}(l)\rfloor_t = \lfloor l\rfloor_2 \mid \lfloor\mathsf{cycle}(x, t)\rfloor_t = \{x = \lfloor t_2\rfloor_0\}$. We shall use the notation $\lfloor\rfloor$ for any of $\lfloor\rfloor_0$, $\lfloor\rfloor_1$, $\lfloor\rfloor_2$, $\lfloor\rfloor_t$, whenever the distinction is clear from context. We also need an auxiliary function $r_\Box$ to rotate a list until $\Box$ is at the end, easily defined as: $r_\Box(\Box : t) = t@(\Box : \varepsilon) \mid r_\Box(h : t) = r_\Box(t@(h : \varepsilon))$.

Definition 5 (Interpretation). *The* interpretation *of a machine configuration Σ is an interaction net $[\![\Sigma]\!]$ obtained as follows:*

1. *We compute $\mathbf{U}[\Sigma] = \langle \Gamma^{U[\Sigma]} \mid S^{U[\Sigma]} \mid \phi_N^{U[\Sigma]} \mid V^{U[\Sigma]} \mid C^{U[\Sigma]} \mid t^{U[\Sigma]}\rangle$.*
2. *We then build the net $N^\Sigma = \left\langle \lfloor r_\Box(V^{U[\Sigma]})\rfloor \mid \lfloor S^{U[\Sigma]}\rfloor \cup \lfloor C^{U[\Sigma]}\rfloor \cup \lfloor t^{U[\Sigma]}\rfloor \right\rangle$.*
3. *$[\![\Sigma]\!] = \mathbf{Clp}[N^\Sigma, \phi_N^{U[\Sigma]}]$ concludes our construction.*

Notice that the interpretation of a configuration is *unique*. It is immediate to see that all the configurations $\Sigma[\mathbf{t} \mid \Delta]$ have the same interpretation $\langle \mathbf{t} \mid \Delta\rangle$.

The interpretation of a degenerate configuration disregards all the active pairs and cycles contained in the heap of a degenerate configuration. This is appropriate in the sense that this is information that cannot be read back.

Lemma 2. *If Σ is an irreducible configuration, then $[\![\Sigma]\!]$ is in normal form.*

Proof. Immediate. Δ^Σ may be empty (if C^Σ is), or contain cycles. $\Box$

Notation. In what follows we do not distinguish notationally between the reduction $\longrightarrow$ (or its transitive $\longrightarrow^*$ or refexive $\longrightarrow^\equiv$ closures) of machine configurations and of interaction nets, since the distinction can be made from context. The same is true for structural equivalence ($\equiv$).

Lemma 3. *Let Σ be a configuration with correct annotations, $\Sigma \longrightarrow \Sigma'$ using any rule except I, II.4, and III.5, and Σ, Σ' non-degenerate. Then $[\![\Sigma]\!] \equiv [\![\Sigma']\!]$.*

Lemma 4. *Let $\Sigma \longrightarrow \Sigma'$, with Σ, Σ' non-degenerate and correctly-annotated. Then $[\![\Sigma]\!] \longrightarrow^\equiv [\![\Sigma']\!]$.*

Lemma 5. *The set of rules II.1 to II.3, III.0 to III.4, and T.1 to T.4 is normalizing, with the same normal forms as the complete set of rules (i.e, they have an empty S component and* delist *thread), together with all the configurations to which one of the rules I, II.4, or III.5 can be applied.*

Proposition 4 (Correctness). *Let $N = \langle \mathsf{t} \mid \Delta \rangle$ be an interaction net. Then*

$$N \Downarrow \overline{N} \quad \textit{iff} \quad \Sigma[\mathsf{t} \mid \Delta] \longrightarrow^* \overline{\Sigma},$$

with $\overline{\Sigma}$ an irreducible machine configuration, and $[\![\overline{\Sigma}]\!] \equiv \overline{N}$.

Observe that a result such as $[\![\Sigma]\!] \longrightarrow [\![\Sigma']\!] \Rightarrow \Sigma \longrightarrow^* \Sigma'$ does *not* hold, since the semantics is non-deterministic and besides, it allows for variables in the observable interface to be updated at any time.

5 A Concurrent Abstract Machine

The machine we have been considering is inherently sequential: it is always deterministically decided which rule is applied at each stage. However, the way the machine has been formulated makes it trivial to obtain a concurrent machine from it, simply by including more processing threads within a configuration.

Definition 6 (Multi-thread Configuration). *An n*-thread configuration *is:*

$$\langle \Gamma \mid S \mid \phi_N \mid V \mid C \mid [t_1, \dots t_n] \rangle$$

where $[t_1, \dots t_n]$ is a list of threads and all other components are as before.

Definition 7 (Concurrent reduction). $\xrightarrow{ct}$ *is the smallest relation verifying*

$$\frac{\langle \Gamma \mid S \mid \phi_N \mid V \mid C \mid t_i \rangle \longrightarrow \langle \widehat{\Gamma} \mid \widehat{S} \mid \widehat{\phi_N} \mid \widehat{V} \mid \widehat{C} \mid \widehat{t_i} \rangle}{\langle \Gamma \mid S \mid \phi_N \mid V \mid C \mid [t_1 \dots t_i \dots t_n] \rangle \xrightarrow{ct} \langle \widehat{\Gamma} \mid \widehat{S} \mid \widehat{\phi_N} \mid \widehat{V} \mid \widehat{C} \mid [t_1 \dots \widehat{t_i} \dots t_n] \rangle}$$

This definition is intuitive: a pool of threads work on the shared data-structures, each thread proceeding individually and deterministically, according to the sequential machine rules. Each $\xrightarrow{ct}$ *step* is a step of one of the individual threads.

Non-determinism is introduced because it may be the case that several threads are willing to operate. For instance, when several threads are in the delist state and the list is not empty, one of them will fetch the top pair and change state to $\mathsf{process}$, while the others will keep trying to access the list.

The resulting machine almost falls into the standard producers-consumers synchronization model of shared-memory computation, with the synchronization problem solved by implementing a shared queue of tasks (pairs to be processed). The diference is that every thread is a consumer and *may be a producer*, of tasks.

Properties. The definition of the reduction of a multi-thread configuration as individual reductions of any of its threads makes all the inductively proved properties of the sequential machine also valid for the concurrent one. Lemmas 2, 3, 4, and 5, and Propositions 1 and 3, all hold, with the interpretation of a multi-thread configuration modified to include all the pairs being processed by threads, and irreducible configurations having an empty Pairs list and all threads delist.

Unfortunately, Prop.2 is no longer true, which destroys our correctness result. This is due to a race condition when traversing an annotation list to perform substitutions, which allows the generation of irrecoverable cycles in the heap.

As an example of such a situation, consider the following 2-thread configuration (we omit the V and C components), for a net containing a 2-cell cycle which we show may get lost in the heap. Consider that $w \in \mathcal{N}(t)$ and $z \in \mathcal{N}(u)$.

$$\begin{array}{ll}
 & \langle \emptyset \mid (x, [w : \Box].t) : (y, [z : \Box].u) : S \mid \phi_N[x \leftrightarrow z, y \leftrightarrow w] \mid [\mathsf{delist}, \mathsf{delist}] \rangle. \\
\xrightarrow{ct}{}^{*}_{T.1} & \langle \emptyset \mid S \mid \phi_N[x \leftrightarrow z, y \leftrightarrow w] \mid [\mathsf{process}(x, [w : \Box].t), \mathsf{process}(y, [z : \Box].u)] \rangle. \\
\xrightarrow{ct}_{III.2} & \langle \emptyset \mid S \mid \phi_N[x \leftrightarrow z, y \leftrightarrow w] \mid [\mathsf{process}(x, [w : \Box].t), \mathsf{process}(y, [\Box : z].u)] \rangle. \\
\xrightarrow{ct}_{III.2} & \langle \emptyset \mid S \mid \phi_N[x \leftrightarrow z, y \leftrightarrow w] \mid [\mathsf{process}(x, [\Box : w].t), \mathsf{process}(y, [\Box : z].u)] \rangle. \\
\xrightarrow{ct}_{III.5} & \langle \{x \mapsto [w : \Box].t\} \mid S \mid \phi_N[x \leftrightarrow z, y \leftrightarrow w] \mid [\mathsf{delist}, \mathsf{process}(y, [\Box : z].u)] \rangle. \\
\xrightarrow{ct}_{III.5} & \langle \{x \mapsto [w : \Box].t, y \mapsto [z : \Box].u\} \mid S \mid \phi_N[x \leftrightarrow z, y \leftrightarrow w] \mid [\mathsf{delist}, \mathsf{delist}] \rangle.
\end{array}$$

Recovering Correctness. First, we remark that this form of degenerate configurations is not very harmful, since one could simply forbid nets containing cycles. However, a simple way exists to recover lost cycles, by keeping in the configurations an additional component: a list of variables stored in the heap, kept up-to-date by the rules. After post-processing, and according to Prop.3 (remember post-processing rules implement updating of configurations with empty lists of Pairs), this list will be empty if the configuration is not degenerate. If it is not empty, then the machine may be restarted with any pair (variable, term) still stored in the heap. Once a pair in a cycle is recovered in this way, regular machine operation will proceed to store the cycle in the appropriate structure.

6 Implementing the Abstract Machine

Sequential Implementation. The abstract machine may be programmed sequentially in a straightforward way. It suffices to choose appropriate data-structures for configurations, and to map machine operations into these structures. The 'motor' that runs the machine is simply a loop that pops a pair of terms from the stack and decides which machine operation to apply. We remark this decision is not complex, as the number of operations might lead one to think: the form of the pair being processed restricts the pattern-matching to a family of rules.

Optimizations. During operation of the machine the size of Pairs tends to grow considerably. One possible optimization with respect to keeping the list reasonably small is to give priority to the execution of rules that remove pairs from it. This is a straightforward modification: it suffices to add a second list of pairs to the configurations (for active pairs), and to substitute rules T.1 to T3. by a new set of rules that manages the two lists according to the desired priority scheme.

Concurrent Implementation. A well-suited technology exists for implementing our concurrent abstract machine: POSIX Threads, which are lightweight processes running in the address space of the same UNIX process, individually scheduled. This technology has the advantage of producing implementations that may be run (without modifications in the code) in machines with any number of processors. If conveniently supported by the kernel of the operating system, threads will run in true parallelism, assigned to different processors.

The way to implement our abstract machine using this technology is to launch as many threads (running the same code, as specified by the machine rules) as there are processors in the machine (this can be done automatically at run-time).

Now it is time to ask whether the correspondence between the resulting implementation and the abstract machine is perfect. In a uniprocessor machine this is indeed the case: a single thread is executed at any time, and this is what the machine captures, even though in practice each machine operation is decomposed into a sequence of program instructions, which means that time-slicing between threads may occur in the middle of the execution of an operation.

In the case of multiprocessors, however, in order to obtain a correct implementation, one has to stipulate that *a parallel reduction step is any sequence of concurrent reduction steps in which every thread is involved at most once.*

To illustrate this point, consider the situation of two threads in the delist state. The outcome if both execute rule T.1 simultaneously is not predictable. The correct behaviour is that two different configurations may result, corresponding to the two orders in which the threads may take pairs from the list.

Now the parallel reducer must keep to this behaviour, protecting the access to the Pairs and Cycles data-structures by means of some synchronization mechanism: the first thread to gain access to a shared structure will make the other thread(s) *block* until the first one liberates the structure. *Locks* [7] are used to implement a linearizable queue, and this is thus not a wait-free implementation.

We leave a detailed treatment of this matter for the long version of this paper; there is another situation requiring locking, which concerns the case of two adjacent pairs of variables (two axiom cuts) being processed simultaneously.

Other situations are critical with respect to race conditions in true parallelism. Take for example the two threads $\mathsf{process}(x, \{\square : T\}.t)$ and $\mathsf{process}(y, \{\square : T'\}.u)$, with $\phi_N(x) = y$. Our abstract machine prevents this from generating a lost active pair in the heap: once one of the threads executes rule III.5, the other will be unable to execute it (executing III.4 instead). But in true parallelism the threads may execute III.5 simultaneously, giving a degenerate configuration. We remark that the problem now is not in the access to the data structures.

One solution to this problem not requiring additional synchronization is the mechanism explained in Sect.5 to recover entries in the heap back to Pairs. This optimistic solution relies on the low probability of the race situations to occur.

7 Conclusions

We have presented an abstract machine which provides answers to the questions raised in Sect.2. Specifically, the machine proposes concrete data-structures, together with an algorithm for implementing interaction net reduction. We have also specified how rules are represented and applied. Identification of active pairs is automatic, and question 2 is answered by the post-processing operations. With respect to 5, all the machine rules perform simple operations in constant time.

We have implemented this machine following Sect.6, and obtained a robust reducer that performs well. The *granularity* of the machine operations is quite small, while still keeping the operations sufficiently atomic: our tests show that each interaction is decomposed in between 7 and 12 machine operations.

The concurrent machine additionally provides answers to questions 6 and 7, resulting in the first parallel implementation of interaction nets, with the significant advantage of running on commonly available workstations. We remark that this parallel implementation is not limited by the number of active pairs in particular nets, since all rewiring operations are performed in parallel. Our current research involves studying the performance of this implementation.

References

[1] Andrea Asperti, Cecilia Giovannetti, and Andrea Naletto. The bologna optimal higher-order machine. *Journal of Functional Programming*, 6(6):763–810, November 1996.

[2] Gérard Berry and Gérard Boudol. The Chemical Abstract Machine. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 81–94, 1990.

[3] P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82(2):389–402, May 1991.

[4] Maribel Fernndez and Ian Mackie. A calculus for interaction nets. In Gopalan Nadathur, editor, *Principles and Practice of Declarative Programming'99 Conference*

Proceedings, number 1702 in Lecture Notes in Computer Science. Springer-Verlag, September/October 1999.

[5] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[6] Georges Gonthier, Martn Abadi, and Jean-Jacques Lvy. The geometry of optimal lambda reduction. In *Proceedings of ACM Symposium Principles of Programming Languages*, pages 15–26, January 1992.

[7] P. Jayanti. Wait-free computing. In *Distributed Algorithms, 9th International Workshop, WDAG '95*, volume 972 of *Lecture Notes in Computer Science*, pages 19–50, 1995.

[8] Yves Lafont. Interaction nets. In *Seventeenth Annual Symposium on Principles of Programming Languages*, pages 95–108, San Francisco, California, 1990. ACM Press.

[9] Yves Lafont. From proof nets to interaction nets. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, pages 225–247. Cambridge University Press, 1995. Proceedings of the Workshop on Linear Logic, Ithaca, New York, June 1993.

[10] John Lamping. An algorithm for optimal lambda-calculus reductions. In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 16–30. ACM, ACM Press, January 1990.

[11] Peter Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4), 1963.

[12] Jean-Jacques Lévy. *Réductions Correctes et Optimales dans le Lambda Calcul.* Thèse de Doctorat d'Etat, University of Paris VII, 1978.

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