An Implementation Model for Interaction Nets

Abubakar Hassan
Theory and Practice of Software Ltd
London, UK

Ian Mackie
LIX, Ecole Polytechnique
91128 Palaiseau Cedex, France

Shinya Sato
University of Sussex
Brighton, UK

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 [5] 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 then graphs built from the set of nodes and rules are graph transformations. Interaction nets are therefore a specific, in fact very 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 often not documented. The purpose of this paper is to take a new 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. 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 with a low-level language that corresponds exactly to the calculus; and a compilation of the calculus to this low-level language.

In addition to defining the above, an important aspect of 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 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 paper we provide the foundations for this work, and point out a number of directions that are currently being investigated.

A number of evaluators have been developed for interaction nets, and one of the first abstract machines was given by Sousa Pinto [9]. From another direction, a graphical interpreter in^2 was proposed by Lippi [7] and it showed an aspect of interaction nets as a visual programming tool. Some evaluators have been proposed towards efficient computation, called INET [2] and amineLight [3]. Our approach is to build the simplest implementation model for interaction nets that we believe can be the most useful as well as providing the basis for more efficient (including parallel) implementations in the future. We shall give some evidence to support this claim in the current paper.

The next section recalls some background material, and in Section 3 we describe the new calculus. In Section 4 we give the data-structure that corresponds to the calculus together with a low-level language. We include also some notes about the compilation of nets into the low-level language. In Section 5 we evaluate the work, and give some directions for future work. We conclude the paper in Section 6.

A. Middeldorp and F. van Raamsdonk (Eds.): 8th International
Workshop on Computing with Terms and Graphs (TERMGRAPH 2014)
EPTCS 183, 2015, pp. 66–80, doi:10.4204/EPTCS.183.5

2 Background

In the graphical rewriting system of interaction nets [5], we have a set Σ of *symbols*, which are names of the nodes. 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 α has $n+1$ *ports: n* auxiliary ports and a distinguished one called the *principal port*. Nodes are drawn as shown in Figure1. A *net* built on Σ 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. 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 (α,β) by the net N. All the free ports are preserved during reduction, and there is at most one rule for each pair of agents. The diagram in Figure 2 illustrates the idea, where N is any net built from Σ. The most powerful property of this graph rewriting system is that it is one-step confluent—all reduction sequences are permutation equivalent.

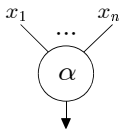


Figure 1: Agents

There are many possible data-structures that can be used to represent interaction nets, for example single or double linked graphs. Agents have exactly one principal port, so we can use this to our advantage and represent nets as a collection of trees, and use a simple, single linked structure. 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 α and β in the diagram in Figure 2).

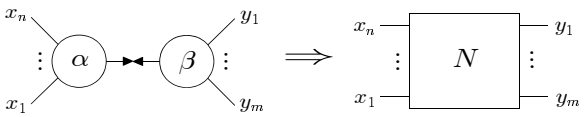


Figure 2: Interaction rules

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, we provide a framework where we can focus on the compilation of rules where we can perform the above steps in the fewest instructions.

3 Calculus

It is possible to reason about the graphical representation of nets, but it is convenient to have a textual calculus for compact representation. There are several calculi in the literature, and here we review the *Lightweight calculus* [3], which is a refined version of [1].

Agents: Let Σ be a set of symbols, ranged over by $\alpha,\beta,\ldots$, each with a given *arity* $\mathsf{ar} : \Sigma \to \mathbb{N}$. An occurrence of a symbol is called an *agent*, and the arity is the number of auxiliary ports.

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

Terms: A term is built on Σ and $\mathscr{N}$ by the grammar: $t ::= x \mid \alpha(t_1,\ldots,t_n)$, where $x \in \mathscr{N}$, $\alpha \in \Sigma$, $\mathsf{ar}(\alpha) = n$ 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*. 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 $\alpha(t_1,\ldots,t_n)$ can be seen as a tree with the principal port of α at the root, and the terms $t_1,\ldots,t_n$ are the subtrees connected to the auxiliary ports of α. The term $\$t$ represents an indirection node which is created by reduction, and is not normally part of an initial term.

Equations: If t, u are terms, then the unordered pair $t = u$ is an *equation*. Δ ranges over multisets of equations.

Rules: Rules are pairs of terms written: $\alpha(x_1,\ldots,x_n) = \beta(y_1,\ldots,y_m) \Rightarrow \Delta$, where $(\alpha,\beta) \in \Sigma \times \Sigma$ is the active pair, and Δ 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. We call the free names $x_1,\ldots,x_n,y_1,\ldots,y_m$ in Δ *parameters* and write it as $\Delta(x_1,\ldots,x_n,y_1,\ldots,y_m)$. All other names are bound.

Configurations: A *configuration* is a pair $(\mathscr{R}, \langle \vec{t} \mid \Delta \rangle)$, where $\mathscr{R}$ is a set of rules, $\vec{t}$ a sequence of terms, and Δ a multiset of equations. Each variable occurs at most twice in a configuration, and we extend the nomenclature of free and bound names from terms. The rules set $\mathscr{R}$ contains at most one rule between any pair of agents, and it is closed under symmetry, thus if $\alpha(\vec{x}) = \beta(\vec{y}) \in \mathscr{R}$ then $\beta(\vec{y}) = \alpha(\vec{x}) \in \mathscr{R}$. We use C, C' to range over configurations. We call $\vec{t}$ the *head* and Δ the *body* of a configuration.

Definition 1 (Names in terms) *The set $\mathsf{Name}(t)$ 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.*

$$\mathsf{Name}(x) = \{x\}, \quad \mathsf{Name}(\alpha(t_1,\ldots,t_n)) = \mathsf{Name}(t_1) \cup \cdots \cup \mathsf{Name}(t_n), \quad \mathsf{Name}(\$t) = \mathsf{Name}(t).$$

The notation $t[u/x]$ denotes a substitution that replaces the free occurrence of x by the term u in t. This extends to equations and configurations in the obvious way.

Definition 2 (Instance of a rule) *If r is a rule $\alpha(x_1,\ldots,x_n) = \beta(y_1,\ldots,y_m) \Rightarrow \Delta$, then $\widehat{\Delta}$ denotes a new generic* instance *of r, that is, a copy of Δ where we introduce a new set of bound names so that those new names do not overlap with others already exists, but leave the free names (parameters) unchanged. Example: if Δ is $\alpha(x,x) = \beta(a)$, then $\widehat{\Delta}$ is $\alpha(y,y) = \beta(a)$, where y is a fresh name.*

The configuration $(\mathscr{R}, \langle \vec{t} \mid \Delta \rangle)$ represents a net that we evaluate using $\mathscr{R}$; Δ gives the set of active pairs and the renamings of the net. We write $\langle \vec{t} \mid \Delta \rangle$ without $\mathscr{R}$ when there is no ambiguity. 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. The computation rules are defined below, and we use $\to$ instead of $\to_{\mathrm{com}}, \to_{\mathrm{sub}}, \to_{\mathrm{col}}, \to_{\mathrm{int}}$ when there is no ambiguity.

Communication: $\langle\ \vec{u} \mid \Delta, x = t, x = s\ \rangle \to_{\mathrm{com}} \langle\ \vec{u} \mid \Delta, t = s\ \rangle$.

Substitution: $\langle\ \vec{u} \mid \Delta, \beta(\vec{t}) = u, x = s\ \rangle \to_{\mathrm{sub}} \langle\ \vec{u} \mid \Delta, \beta(\vec{t})[s/x] = u\ \rangle$ where $\beta \in \Sigma$ and x occurs in $\vec{t}$.

Collect: $\langle\ \vec{u} \mid \Delta, x = s\ \rangle \to_{\mathrm{col}} \langle\ \vec{u}[s/x] \mid \Delta\ \rangle$ where x occurs in $\vec{u}$.

Interaction: $\langle\ \vec{u} \mid \Delta, \alpha(t_1,\ldots,t_n) = \beta(s_1,\ldots,s_m)\ \rangle \to_{\mathrm{int}} \langle\ \vec{u} \mid \Delta, \widehat{\Delta_r}[t_1/x_1,\ldots,t_n/x_n, s_1/y_1,\ldots,s_m/y_m]\ \rangle$
where $\alpha(x_1,\ldots,x_n) = \beta(y_1,\ldots,y_n) \Rightarrow \Delta_r \in \mathscr{R}$.

Example 3 *Rules in Figure 3 can be represented as follows:*

$$\mathtt{Add}(x_1,x_2) = \mathtt{S}(y) \Rightarrow \mathtt{Add}(x_1,w) = y, x_2 = \mathtt{S}(w), \qquad \mathtt{Add}(x_1,x_2) = \mathtt{Z} \Rightarrow x_1 = x_2.$$

The net in Figure 3 is represented as $\langle r \mid \mathtt{Add}(\mathtt{Z},r) = \mathtt{S}(\mathtt{Z}) \rangle$, and it is performed:

$$\begin{aligned} \langle\ r \mid \mathtt{Add}(\mathtt{Z},r) = \mathtt{S}(\mathtt{Z})\ \rangle \quad &\to_{\mathrm{int}} \quad \langle\ r \mid \mathtt{Add}(Z,w') = \mathtt{Z}, r = \mathtt{S}(w')\ \rangle \to_{\mathrm{col}} \langle\ \mathtt{S}(w') \mid \mathtt{Add}(\mathtt{Z},w') = \mathtt{Z}\ \rangle \\ &\to_{\mathrm{int}} \quad \langle\ \mathtt{S}(w') \mid \mathtt{Z} = w'\ \rangle \to_{\mathrm{col}} \langle\ \mathtt{S}(\mathtt{Z}) \mid\ \rangle. \end{aligned}$$

We define $C_1 \Downarrow C_2$ by $C_1 \to^* C_2$ where C_2 is in normal form. The following theorem shows that all Interaction rules can be performed without using Substitution and Collect [3]:

Theorem 3.1 *If $C_1 \Downarrow C_2$, then there is a configuration C such that $C_1 \to^* C \to^*_{\mathrm{sub}} \to^*_{\mathrm{col}} C_2$ and C_1 is reduced to C by applying only* Communication *rule and* Interaction *rule.* □

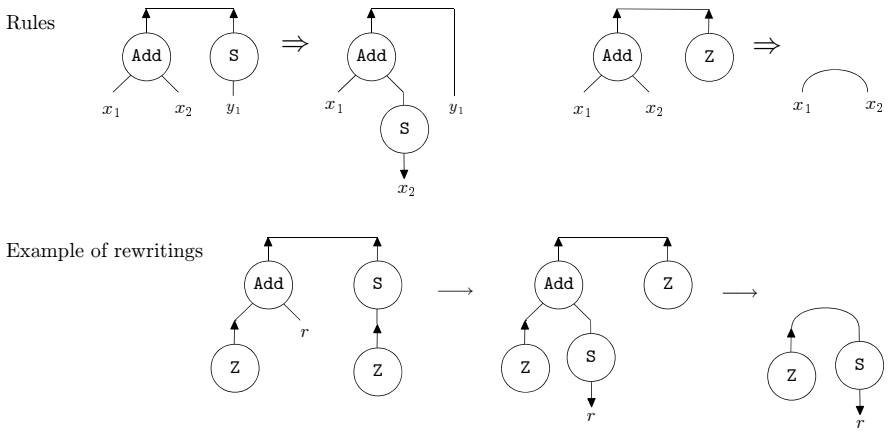


Figure 3: An example of rules and rewritings of interaction nets

We define $C_1 \Downarrow_{\mathrm{ic}} C_2$ by $C_1 \to^* C_2$ where C_2 is a $\{\to_{\mathrm{int}}, \to_{\mathrm{com}}\}-$normal form. Because all critical pairs that are generated by $\to_{\mathrm{int}}$ and $\to_{\mathrm{com}}$ are confluent, the determinacy property holds [3]:

Theorem 3.2 (Determinacy) *Let $C_1 \Downarrow C_2$. When there are configurations C', C'' such that $C_1 \Downarrow_{\mathrm{ic}} C'$ and $C_1 \Downarrow_{\mathrm{ic}} C''$, then C' is equivalent to C''.* □

3.1 Simpler calculus

In Lightweight calculus, equations are defined as unordered pairs and configurations use multisets of equations. Here, in order to facilitate the correspondence between the calculus and an implementation model that has a code stack and an environment such as SECD-machine [6], we introduce another refined calculus of Lightweight one, called *Simpler calculus*, by changing the definition of equations into ordered pairs and in configurations multisets of equations into sequences of ones.

Terms: A term is built on Σ and $\mathscr{N}$ by the grammar: $t ::= x \mid \alpha(t_1,\ldots,t_n) \mid \$t$. Intuitively, $\$t$ corresponds to a variable bounded with t (or a state such that an environment captures t).

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

Rules: Rules are pairs of terms written: $\alpha(x_1,\ldots,x_n) = \beta(y_1,\ldots,y_m) \Rightarrow \Theta$.

Configurations: A *configuration* is a pair $(\mathscr{R}, (\vec{t} \mid \Theta))$, where Θ is a sequence of equations. The rules set $\mathscr{R}$ contains at most one rule between any pair of agents, and it is closed under symmetry. We use S, S' to range over configurations.

Definition 4 (Computation Rules) *The operational behaviour of the system is given by the following:*

Var1: $(\vec{u} \mid \Theta, x = t) \longrightarrow (\vec{u} \mid \Theta)[\$t/x]$ *where* $t \neq \$s$. ***Var2:*** $(\vec{u} \mid \Theta, t = x) \longrightarrow (\vec{u} \mid \Theta)[\$t/x]$ *where* $t \neq \$s$.

Indirection1: $(\vec{u} \mid \Theta, \$t = s) \longrightarrow (\vec{u} \mid \Theta, t = s)$. ***Indirection2:*** $(\vec{u} \mid \Theta, t = \$s) \longrightarrow (\vec{u} \mid \Theta, t = s)$.

Interaction: $\alpha(x_1,\ldots,x_n) = \beta(y_1,\ldots,y_m) \Rightarrow \Theta_r \in \mathscr{R}$, *then*
$(\vec{u} \mid \Theta, \alpha(t_1,\ldots,t_n) = \beta(s_1,\ldots,s_m)) \longrightarrow (\vec{u} \mid \Theta, \widehat{\Theta_r}[t_1/x_1,\ldots,t_n/x_n, s_1/y_1,\ldots,s_m/y_m])$.

To remove indirection terms, we introduce an operation remInd (it is extended to sequences of terms and configurations in the obvious way):

$\mathsf{remInd}(x) \stackrel{\text{def}}{=} x, \quad \mathsf{remInd}(\$t) \stackrel{\text{def}}{=} \mathsf{remInd}(t), \quad \mathsf{remInd}(\alpha(t_1,\ldots,t_n)) \stackrel{\text{def}}{=} \alpha(\mathsf{remInd}(t_1),\ldots,\mathsf{remInd}(t_n))$.

Example 5 *We show the computation of the configuration* $(\, r \mid \mathtt{Add}(r,\mathtt{Z}) = \mathtt{S}(\mathtt{Z})\,)$ *in Figure 3:*

$(\, r \mid \mathtt{Add}(\mathtt{Z},r) = \mathtt{S}(\mathtt{Z})\,) \longrightarrow (\, r \mid \mathtt{Add}(\mathtt{Z},x) = \mathtt{Z}, r = \mathtt{S}(x)\,) \longrightarrow (\, \$\mathtt{S}(x) \mid \mathtt{Add}(\mathtt{Z},x) = \mathtt{Z}\,)$
$\longrightarrow (\, \$\mathtt{S}(x) \mid \mathtt{Z} = x\,) \longrightarrow (\, \$\mathtt{S}(\$\mathtt{Z}) \mid\,)$.

$\mathit{remInd}(\, \$\mathtt{S}(\$\mathtt{Z}) \mid\,) = (\, \mathtt{S}(\mathtt{Z}) \mid\,)$.

These rules 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 an overhead of dealing with indirection nodes. Computationally the interaction rule is the most expensive: the other rules will turn out to be implemented with a small number of instructions or will be equivalences in the data-structure.

3.2 Expressive power

We compare the expressive power of Simpler and Lightweight calculi. We define a translation of configurations from Simpler calculus into the Lightweight one as: $\mathsf{ToLight}((\vec{t} \mid \Theta)) \stackrel{\text{def}}{=} \langle \mathsf{remInd}(\vec{t}) \mid \mathsf{remInd}(\Theta) \rangle$.

Lemma 6 *Let S_1 and S_2 be configurations such that $S_1 \longrightarrow S_2$. When it is by Indirection1 or Indirection2 rules, $\mathit{ToLight}(S_1) = \mathit{ToLight}(S_2)$. Otherwise, $\mathit{ToLight}(S_1) \to \mathit{ToLight}(S_2)$.*

Proof. In the case of Var1: $S_1 = (\, \vec{u} \mid \Theta, x = t\,) \longrightarrow (\, \vec{u} \mid \Theta)[\$t/x] = S_2$. When we assume $\mathsf{ToLight}(S_1) = \langle \vec{u'} \mid \Theta', x = t' \rangle$, then $\mathsf{ToLight}(S_2) = \langle \vec{u'} \mid \Theta' \rangle [t'/x]$, and thus $\mathsf{ToLight}(S_1) \to_{\text{com}} \mathsf{ToLight}(S_2)$ or $\mathsf{ToLight}(S_1) \to_{\text{sub}} \mathsf{ToLight}(S_2)$. □

Every equation is reduced by a rule in Simpler calculus, the following holds:

Lemma 7 *When $S_1 \Downarrow (\, \vec{u} \mid \Theta)$, then Θ is empty.* □

We define a translation of configurations ToSimple from Lightweight calculus into Simpler ones:

$\mathsf{ToSimple}(\langle \vec{t} \mid \Delta \rangle) \stackrel{\text{def}}{=} (\, \vec{t} \mid \Theta\,)$ where Θ is a sequence that is the result of fixing an order of the multiset Δ.

Theorem 3.3 *Let C be a configuration in Lightweight calculus. When there is a configuration S in Simpler one such that $\mathit{ToSimple}(C) \Downarrow S$, then $C \Downarrow \mathit{ToLight}(S)$.*

Proof. Assume $S = (\, \vec{u} \mid \Theta)$, and then Θ is empty by Lemma 7. Since $\mathsf{ToLight}(\, \vec{u} \mid\,)$ is a normal form, $C \Downarrow \mathsf{ToLight}(\, \vec{u} \mid\,)$ by Lemma 6. □

Theorem 3.4 *Let C_1 and C_2 be configurations in Lightweight calculus such that $C_1 \Downarrow_{\text{ic}} C_2$. Then there is a configuration S in Simpler one such that $\mathit{ToSimple}(C_1) \Downarrow S$ and $C_2 \Downarrow \mathit{ToLight}(S)$.*

Proof. If $\mathsf{ToSimple}(C_1)$ has no normal form, corresponding to an infinite reduction sequence from $\mathsf{ToSimple}(C_1)$ we can construct an infinite reduction sequence starting from C_1 by Lemma 6 since each reduction produces at most one equation such as $\$t = s$ or $t = \$s$. This contradicts the assumption of this theorem. There is, thus, a configuration S such that $\mathsf{ToSimple}(C_1) \Downarrow S$. By Theorem 3.4 $C_1 \Downarrow \mathsf{ToLight}(S)$, and thus there is a configuration C_3 such that $C_1 \Downarrow_{\text{ic}} C_3$ and $C_3 \Downarrow \mathsf{ToLight}(S)$ by Theorem 3.1. By the assumption $C_1 \Downarrow_{\text{ic}} C_2$ and the determinacy (Theorem 3.2), $C_3 = C_2$. □

We define a configuration of our abstract machine state by the following 3-tuple $(\, \mathrm{E} \mid \vec{t} \mid \Theta)$, where

- E is an environment, which is a subset of $\mathscr{N} \times \mathscr{T}$ ($\mathscr{N}$ is a set of names, $\mathscr{T}$ is the set of terms),

- $\vec{t}$ is an interface, which is a sequence of terms,
- Θ is a sequence of equations to operate.

In contrast to the SECD machine [6], the stack S, the environment E and the control C in the machine correspond to the term sequence $\vec{t}$, the map E, and the equation sequence Θ in this abstract machine respectively. There is no element corresponding to the dump D in SECD machine because, during an execution of a rule, other rules are not called. To manage the environment, we define the following.

Definition 8 (Operations for pairs) *Let* P *be a set of pairs.*

- *We define a map* P *as a set of pairs:* $\mathrm{P}(n) \stackrel{\text{def}}{=} m$ *if* $(n,m) \in \mathrm{P}$, $\bot$ *otherwise.*
- *We use the following notations to operate maps:*
  - $\mathrm{P}(n) := \bot$ *as a set* $(\mathrm{P} - \{(n,m)\})$ *for any m,*
  - $\mathrm{P}(n) := m$ *as a set* $(\mathrm{P}[n] := \bot) \cup \{(n,m)\}$.

We give the semantics of the machine as a set of the following transitional rules of the form $(\mathrm{E} \mid \vec{u} \mid \Theta) \Longrightarrow (\mathrm{E}' \mid \vec{u} \mid \Theta')$ by applying in the order from A to C2:

$$
\begin{array}{rlcll}
\mathrm{A}: & (\mathrm{E} \mid \vec{u} \mid \Theta, \alpha(\vec{t}) = \beta(\vec{s})) & \Longrightarrow & (\mathrm{E} \mid \vec{u} \mid \Theta, \Theta_1) & \text{where } (\mid \alpha(\vec{t}) = \beta(\vec{s})) \longrightarrow (\mid \Theta_1) \\
\mathrm{B1}: & (\mathrm{E}(x) = \bot \mid \vec{u} \mid \Theta, x = t) & \Longrightarrow & (\mathrm{E}(x) := t \mid \vec{u} \mid \Theta) & \\
\mathrm{B2}: & (\mathrm{E}(x) = \bot \mid \vec{u} \mid \Theta, t = x) & \Longrightarrow & (\mathrm{E}(x) := t \mid \vec{u} \mid \Theta) & \\
\mathrm{C1}: & (\mathrm{E}(x) = s \mid \vec{u} \mid \Theta, x = t) & \Longrightarrow & (\mathrm{E}(x) := \bot \mid \vec{u} \mid \Theta, s = t) & \\
\mathrm{C2}: & (\mathrm{E}(x) = s \mid \vec{u} \mid \Theta, t = x) & \Longrightarrow & (\mathrm{E}(x) := \bot \mid \vec{u} \mid \Theta, t = s) &
\end{array}
$$

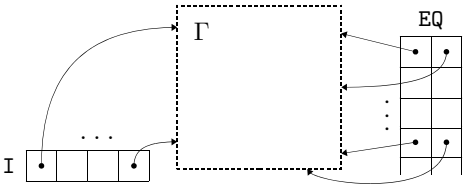
Intuitively, the rule 'A' corresponds Interaction rule, 'B1' and 'B2' correspond Var1 and Var2, and 'C1' and 'C2' correspond Indirection1 and Indirection2. To force captured terms in the environment to be replaced when the execution is finished, we define Update operation:

$$
\begin{array}{lcll}
\mathsf{Update}(\mathrm{E} \cup \{(x,s)\} \mid \vec{u} \mid \Theta) & = & \left\{ \begin{array}{ll} \mathsf{Update}(\mathrm{E}[s/x] \mid \vec{u}[s/x] \mid \Theta[s/x]) & \text{(when } x \text{ occurs in E}, \vec{u} \text{ or } \Theta) \\ \mathsf{Update}(\mathrm{E} \mid \vec{u} \mid \Theta, x = s) & \text{(otherwise)} \end{array} \right. \\
\mathsf{Update}(\emptyset \mid \vec{u} \mid \Theta) & = & (\vec{u} \mid \Theta) &
\end{array}
$$

Example 9 *A configuration* $(r \mid \mathtt{Add}(\mathtt{Z}, r) = \mathtt{S}(\mathtt{Z}))$ *which represents the net in Figure 3 is performed:*
$(\emptyset \mid r \mid \mathtt{Add}(\mathtt{Z}, r) = \mathtt{S}(\mathtt{Z})) \Longrightarrow (\emptyset \mid r \mid \mathtt{Add}(\mathtt{Z}, x) = \mathtt{Z}, r = \mathtt{S}(x)) \Longrightarrow (\{(r, \mathtt{S}(x))\} \mid r \mid \mathtt{Add}(\mathtt{Z}, x) = \mathtt{Z})$
$(\{(r, \mathtt{S}(x))\} \mid r \mid \mathtt{Z} = x) \Longrightarrow (\{(r, \mathtt{S}(x)), (x, \mathtt{Z})\} \mid r \mid -)$.
$\mathit{Update}(\{(r, \mathtt{S}(x)), (x, \mathtt{Z})\} \mid r \mid -) = \mathit{Update}(\{(r, \mathtt{S}(\mathtt{Z}))\} \mid r \mid -) = \mathit{Update}(\emptyset \mid \mathtt{S}(\mathtt{Z}) \mid -) = (\mathtt{S}(\mathtt{Z}) \mid)$.

4 Data-structures and language

Here we give a low-level language, called LL0, which defines a set of instructions to build and reduce a net to normal form. The concrete representation of a configuration can be summarised by the diagram shown on the right, where Γ represents the net, EQ a stack of equations, and I an interface.



For a net, we need two kinds of graph element: agents and names nodes. Each of these is allocated memory in the heap. An element, such as an agent, may contain pointers to other elements (representing auxiliary ports). An agent can be coded in C as follows:

```
typedef struct Agent {
  int id; struct Agent *port[];
} Agent;
```

Instruction	Description
`#agent` $\alpha_1 : p_1, \cdots, \alpha_n : p_n$	Declare $\alpha_1, \cdots, \alpha_n$ as symbols of agents whose arity are $p_1, \cdots, p_n$.
`I=mkInterface(`n`)`	Create a fixed n-size interface and assign its pointer to the variable `I`.
x`=mkAgent(`id`)`	Allocate (unused) memory for an agent node whose id is id and assign it to the variable x.
x`=mkName()`	Allocate (unused) memory for a name node, and assign it to the variable x.
`free(`x`)`	Dispose of just an assigned allocation x of a graph element (not recursively).
x`[`p`]=`y	Assign a graph element y to a port $p > 0$ of an agent node x.
x`[0]=`α	Change the id of an agent node x into α.
`push(`x`,`y`)`	Create an equation of two graph element x, y in the stack of equations.
`stackFree()`	Dispose of the top element of the equation stack.

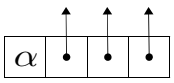
Figure 4: Instructions of LL0

In this `Agent` structure, each symbol $\alpha_1, \ldots, \alpha_n$ for agents is distinguished by a unique `id`. The length of `port` corresponds to the number of auxiliary ports of an agent. The stack of equations `EQ` is initially empty. Intuitively, an element of this stack can be written using the following code fragment in C:

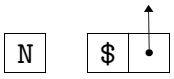
```
typedef struct Equation {
  Agent *a1; Agent *a2;
} Equation;
```

The interface `I` is a node arrays of fixed size n as the size of the observable interface of a net can be pre-determined (and it is preserved during execution). By using LL0, we encode Simpler calculus.

Building nets. Graph elements, the stack of equations `EQ` and the interface `I` are managed by instructions as shown in Figure 4. The port numbers start from 1, and by using the instruction x`[`p`]=`y, we can assign a graph node y into a port $p > 0$ of a graph node x. We also use the port 0 to refer to the id of an element. For instance, x`[0]=`α changes the id of an agent node x into α.



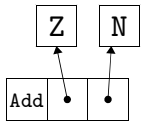
Here, we build terms in Simpler calculus. To assign an arity to an agent, we use the following declaration: `#agent` $\alpha_1 : p_1, \cdots, \alpha_n : p_n$, where p_i is the arity for an agent symbol α_i such that $ar(\alpha_i) = p_i$. After this declaration, a symbol α_i can be represented by a unique number and an agent's arity p_i can be referred to by $\mathsf{arity}(\alpha_i) = p_i$. We draw an agent node α of arity 3 as shown the right above figure.



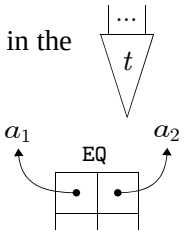
Name nodes are graph elements whose `id` is denoted by the symbol `N` and the arity is 0. We also use indirection nodes by `$` and the arity is 1. We assume that `N` and `$` are selected from a set that does not overlap with the set of agent symbols. To allocate an agent node whose id is id to a variable x, we use the following instruction: x`=mkAgent(`id`)`. A name node is allocated by x`=mkName()`. An assigned allocation a of a graph element is disposed of (not recursively, just one node) by using `free(`a`)`.

A connection between a principal port and an auxiliary port is encoded by an assignment. In this language, to assign a pointer of an existing graph element b to a port p of another graph element a, we use the following instruction: a`[`p`]=`b. We note that the index of these ports start from 1. For instance, a term `Add(Z,r)` is encoded as shown below, together with the graphical representation.

```
1. #agent Z:0, Add:2
2. aAdd=mkAgent(Add)
3. aZ=mkAgent(Z)
4. aAdd[1]=aZ
5. r=mkName()
6. aAdd[2]=r
```



Generally, when agent nodes are connected together, they are trees that we represent in the following way, where the free ports are at the top of the tree.



The stack of equations EQ is initially created empty. An equation node can point to two graph elements. To create an equation of two graph elements a_1, a_2 in the stack EQ, we use the instruction: `push(`a_1`,`a_2`)`. To pop an equation from the top of the stack EQ, we use the instruction: `stackFree()`. We represent a connection between principal ports by creating an equation between the two agent nodes into the stack.

Interfaces are created with the instruction: `I=mkInterface(`n`)`. Elements in I can be accessed using the usual array notation `I[1]`$,\cdots,$`I[n]` and can point to one graph element. As an example, a configuration $(r \mid \texttt{Add}(\texttt{Z},r) = \texttt{S}(w), \texttt{Add}(\texttt{Z},w) = \texttt{S}(\texttt{Z}))$ is encoded using the instructions below. Figure 5 gives the corresponding data-structure. For a connection between two auxiliary ports, we assign one name node to two ports.

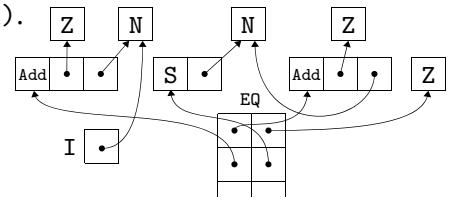


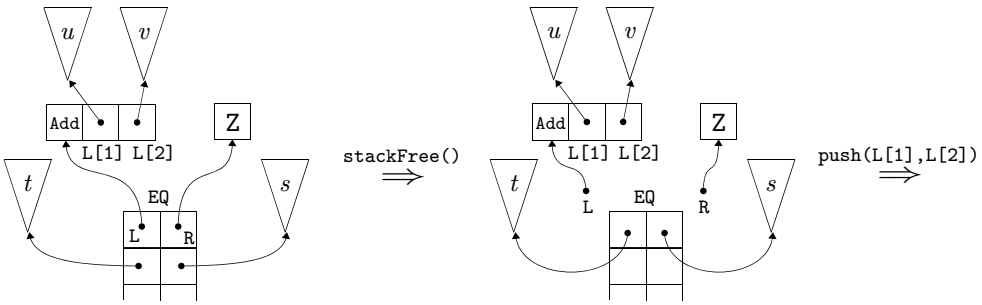
Figure 5: Representation of $(r \mid \texttt{Add}(\texttt{Z},r) = \texttt{S}(w), \texttt{Add}(\texttt{Z},w) = \texttt{S}(\texttt{Z}))$

```
 1. #agent Z:0,S:1,Add:2
 2. /*interface*/
 3. I=mkInterface(1)
 4. /*Add(Z,r)*/
 5. aAdd=mkAgent(Add)
 6. aZ=mkAgent(Z)
 7. aAdd[1]=aZ
 8. r=mkName()
 9. aAdd[2]=r
10. /*S(w)*/
11. bS=mkAgent(S)
12. w=mkName()
13. bS[1]=w
14. /*Add(Z,r)=S(w)*/
15. push(aAdd,bS)
16. /*Add(Z,w)*/
17. aAdd=mkAgent(Add)
18. aZ=mkAgent(Z)
19. aAdd[1]=aZ
20. aAdd[2]=w
21. /*S(Z)*/
22. bS=mkAgent(S)
23. bZ=mkAgent(Z)
24. bS[1]=bZ
25. /*Add(Z,w)=S(Z)*/
26. push(aAdd,bS)
27. /*interface*/
28. I[1]=r
```

Defining interaction rules. We introduce *rule procedures* to perform interaction rules. For an interaction rule between $\alpha(\vec{x})$ and $\beta(\vec{y})$ we define a rule procedure using the syntax: `rule` α β $\{\ldots\}$ and we write instructions between the brackets $\{$ and $\}$ (rule block). In execution, the procedures provide special variables L,R that are pointers to the left and the right-hand side agents of the active pair equation. Variables used in the instructions are only visible within the rule procedure. Generally, these rule procedures are represented as transformations on the data-structure. For instance, the rule between Add and Z given by $\texttt{Add}(x_1,x_2) = \texttt{Z} \Rightarrow x_1 = x_2$ is represented using the following procedure:

```
1. rule Add Z {
2.   stackFree()
3.   push(L[1],L[2])
4.   free(L)
5.   free(R)
6. }
```

The following illustrates transformations that will be applied by the rule procedure given above.



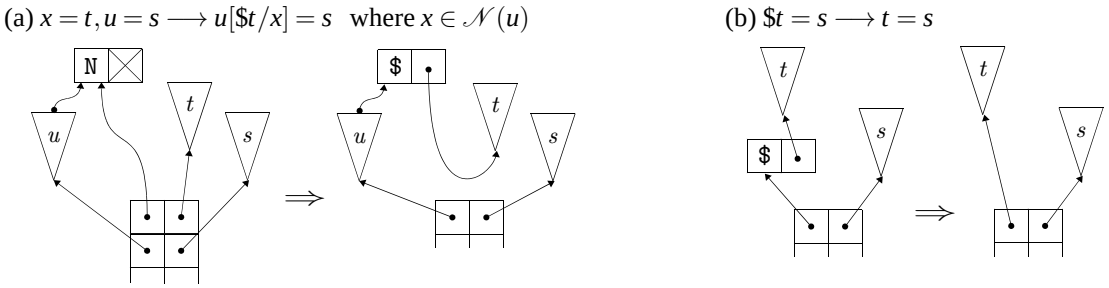
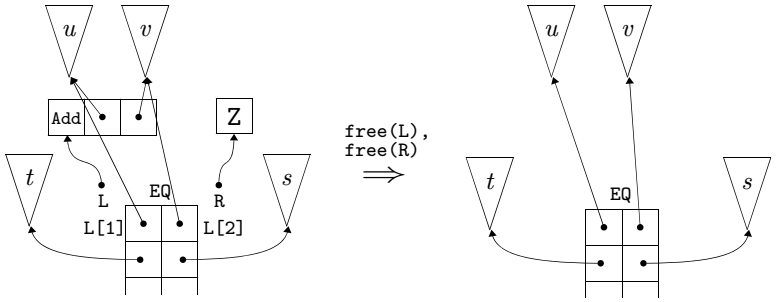


Figure 6: Computation rules for name and indirection nodes



In order to manage equations in the Simpler calculus, we need mechanisms that perform rules Var1, Indirection1 and so on. Figure 6 (a) and (b) are instances of Var1 and Indirection1 rules to illustrate this.

Compilation of Simpler calculus into LL0. Here we introduce a translation of Simpler calculus into LL0. We use a set of pairs and operations for the pairs defined in Definition 8. We also use a notation for strings. We write " and " as a pair of delimiters to represent a string explicitly. We use the notation $\{x\}$ in a string as the result of replacing the occurrence $\{x\}$ with its actual value. For instance, if $x =$ "`abc`" and $y = 89$ then "`1`$\{x\}$`2`$\{y\}$" = "`1abc289`". We use $+$ as an infix binary operation to concatenate strings.

Definition 10 (Compilation of terms and nets) *Here we defined our compilation schemes that will generate LL0 for a given interaction net system.*

- *We use a subset* N *of* $\mathcal{N} \times$ Str *(*$\mathcal{N}$ *is a set of names) so that a name* $x \in \mathcal{N}$ *can correspond to a string of a variable name in a code sequence and those correspondences can be looked up from compilation functions. We also use two operations* $\mathrm{N}(x) := \bot$ *and* $\mathrm{N}(x) :=$ str *for* str $\in$ Str *as defined in Definition 8. We define a function makeN to make such a set* N *and a code sequence for those names by a given name set* $\{x_1, \ldots, x_n\}$*. The function freshStr() returns a fresh name.*

$$
\begin{array}{lcl}
\mathit{makeN}(\{x_1,\ldots,x_n\}) & \stackrel{\text{def}}{=} & \mathit{makeN'}(\{x_1,\ldots,x_n\},\emptyset); \\
\mathit{makeN'}(\{x_1,\ldots,x_n\},\mathrm{N}) & \stackrel{\text{def}}{=} & \mathit{let}\quad \mathrm{N}_0 = \emptyset; \\
 & & a_1 = \mathit{freshStr()};\quad c_1 = \text{“}\{a_1\}\texttt{=mkName()}\text{”};\quad \mathrm{N}_1 = (\mathrm{N}_0(x_1) := a_1);\cdots \\
 & & a_n = \mathit{freshStr()};\quad c_n = \text{“}\{a_n\}\texttt{=mkName()}\text{”};\quad \mathrm{N}_n = (\mathrm{N}_{n-1}(x_n) := a_n); \\
 & & \mathit{in}\quad (c_1 + \cdots + c_n, \mathrm{N}_n)\quad \mathit{end};
\end{array}
$$

- *We define a translation* $\mathit{Compile}_s$ *from a symbol set* Σ *into a code string as follows:*

$$
\begin{array}{lcl}
\mathit{Compile}_s(\emptyset) & \stackrel{\text{def}}{=} & \text{“”} \\
\mid \mathit{Compile}_s(\{\alpha_1,\ldots,\alpha_n\}) & \stackrel{\text{def}}{=} & \texttt{“\#agent ”} + \text{“}\{\alpha_1\}:\{ar(\alpha_1)\}\text{”} + \cdots + \text{“,}\{\alpha_n\}:\{ar(\alpha_n)\}\text{”};
\end{array}
$$

- *A translation $Compile_t$ from a term into a code string is defined as follows:*

$$Compile_t(x,\mathrm{N}) \stackrel{\text{def}}{=} (\text{“”},\mathrm{N}(x))$$

$$\begin{array}{l}\mid Compile_t(\alpha(t_1,\ldots,t_n),\mathrm{N}) \stackrel{\text{def}}{=} \quad \mathit{let}\quad a = \mathit{freshStr()};\quad c = \text{“}\{a\}\texttt{=mkAgent(}\{\alpha\}\texttt{)”};\\ \qquad\qquad (c_1,a_1) = Compile_t(t_1,\mathrm{N});\quad c_1 = c_1 + \text{“}\{a\}\texttt{[1]=}\{a_1\}\text{”};\quad \cdots\\ \qquad\qquad (c_n,a_n) = Compile_t(t_n,\mathrm{N});\quad c_n = c_n + \text{“}\{a\}\texttt{[}n\texttt{]=}\{a_n\}\text{”};\\ \qquad\quad \mathit{in}\quad (c + c_1 + \cdots + c_n, a)\quad \mathit{end};\end{array}$$

- *A translation $Compile_i$ from an interface $u_1,\ldots,u_n$ into a code sequence is defined as follows:*

$$Compile_i(-,\mathrm{N}) \stackrel{\text{def}}{=} \text{“”}$$

$$\begin{array}{l}\mid Compile_i(u_1,\ldots,u_n,\mathrm{N}) \stackrel{\text{def}}{=} \quad \mathit{let}\quad (c_1,a_1) = Compile_t(u_1,\mathrm{N});\quad c_1 = c_1 + \text{“}\texttt{I[1]=}\{a_1\}\text{”};\quad \cdots\\ \qquad\qquad (c_n,a_n) = Compile_t(u_n,\mathrm{N});\quad c_n = c_n + \text{“}\texttt{I[}n\texttt{]=}\{a_n\}\text{”};\\ \qquad\quad \mathit{in}\quad \text{“}\texttt{I=mkInterface[}n\texttt{]”} + c_1 + \cdots + c_n\quad \mathit{end};\end{array}$$

- *A translation $Compile_e$ from an equation into a code string is defined as follows:*

$$\begin{array}{l}Compile_e(t = s,\mathrm{N}) \stackrel{\text{def}}{=} \mathit{let}\quad (c_1,a_1) = Compile_t(t,\mathrm{N});\quad (c_2,a_2) = Compile_t(s,\mathrm{N});\\ \qquad\qquad \mathit{in}\quad c_1 + c_2 + \text{“}\texttt{push(}\{a_1\}\texttt{,}\{a_2\}\texttt{)”}\quad \mathit{end};\end{array}$$

- *A translation $Compile_{es}$ from an equation sequence into a code string is defined as follows:*

$$Compile_{es}(e_1,\ldots,e_n,\mathrm{N}) \stackrel{\text{def}}{=} Compile_e(e_1,\mathrm{N}) + \cdots + Compile_e(e_n,\mathrm{N});$$

- *We define a translation $Compile_c$ from a configuration $(\vec{u} \mid \Theta)$ with a symbol set Σ into a code string c as follows:*

$$\begin{array}{l}Compile_c(\Sigma,(\vec{u} \mid \Theta)) \stackrel{\text{def}}{=} \mathit{let}\quad c_0 = Compile_s(\Sigma);\quad (c_1,\mathrm{N}) = makeN(Name(\vec{u} \mid \Theta));\\ \qquad\qquad c_2 = Compile_{es}(\Theta,\mathrm{N});\quad c_3 = Compile_i(\vec{u},\mathrm{N});\\ \qquad\quad \mathit{in}\quad c_0 + c_1 + c_2 + c_3\quad \mathit{end};\end{array}$$

- *We write just Compile when there is no ambiguity.*

Example 11 *Let us take a configuration $(r \mid \texttt{Add(Z,}r\texttt{)} = \texttt{S(Z)})$ with a symbol set $\{\texttt{Z},\texttt{S},\texttt{Add}\}$ as an example. The compilation $Compile_c(\{\texttt{Z},\texttt{S},\texttt{Add}\},(r \mid \texttt{Add(Z,}r\texttt{)} = \texttt{S(Z)}))$ generates the following instructions:*

```
1. #agent Z:0,S:1,Add:2
2. r=mkName()
3. a1=mkAgent(Add)
4. a2=mkAgent(Z)
5. a1[1]=a2
6. a1[2]=r
7. b1=mkAgent(S)
8. b2=mkAgent(Z)
9. b1[1]=b2
10. push(a1,b1)
11. I=mkInterface[1]
12. I[1]=r
```

Definition 12 (Compilation of rules) *We define a translation $Compile_r$ from a rule into a sequence of code strings as follows:*

$$\begin{array}{l}Compile_r(\alpha(\vec{x}) = \beta(\vec{y}) \Rightarrow \Theta) \stackrel{\text{def}}{=}\\ \quad \mathit{let}\\ \qquad \mathrm{N}_l = Compile_{rn}(\vec{x},\mathrm{L},\emptyset);\quad \mathrm{N}_r = Compile_{rn}(\vec{y},\mathrm{R},\mathrm{N}_l);\\ \qquad (c_1,\mathrm{N}) = makeN'(Name(\Theta) - \{\vec{x},\vec{y}\},\mathrm{N}_r);\\ \qquad c_2 = Compile_{es}(\Theta,\mathrm{N});\\ \quad \mathit{in}\\ \qquad \text{“}\texttt{rule }\{\alpha\}\ \{\beta\}\ \{\text{”}\\ \qquad +\text{“}\texttt{stackFree()}\text{”}\\ \qquad + c_1 + c_2\\ \qquad +\text{“}\texttt{free(L)}\text{”} + \text{“}\texttt{free(R)}\text{”} + \text{“}\}\text{”}\\ \quad \mathit{end};\end{array}$$

$$\begin{array}{l}Compile_{rn}((x_1,\ldots,x_n),LR,\mathrm{N}) \stackrel{\text{def}}{=}\\ \quad \mathit{let}\\ \qquad \mathrm{N}_0 = \mathrm{N};\\ \qquad \mathrm{N}_1 = (\mathrm{N}_0(x_1) := \{LR\}\texttt{[1]});\\ \qquad\quad \vdots\\ \qquad \mathrm{N}_n = (\mathrm{N}_{n-1}(x_n) := \{LR\}\texttt{[}n\texttt{]});\\ \quad \mathit{in}\\ \qquad \mathrm{N}_n\\ \quad \mathit{end};\end{array}$$

Example 13 *The results of $Compile_r(\texttt{Add}(x_1,x_2) = \texttt{Z} \Rightarrow x_1 = x_2)$ and $Compile_r(\texttt{Add}(x_1,x_2) = \texttt{S}(y) \Rightarrow \texttt{Add}(x_1,w) = y, x_2 = \texttt{S}(w))$ are as follows:*

```
1. rule Add Z {
2.   stackFree()
3.   push(L[1],L[2])
4.   free(L)
5.   free(R)
6. }
```

```
1. rule Add S {
2.   stackFree()
3.   w=mkName()
4.   a1=mkAgent(Add)
5.   a1[1]=L[1]
6.   a1[2]=w
7.   push(a1,R[1])
8.   b1=mkAgent(S)
9.   b1[1]=w
10.  push(L[2],b1)
11.  free(L)
12.  free(R)
13. }
```

Back-end of the compilation. Here we show how these translated codes are evaluated on the standardised implementation model in the C language, showing the correspondence of codes in LL0 with the C language.

- `#agent` $\alpha_1 : p_1, \ldots, \alpha_n : p_n$. For each sort of agent, we assign a unique number that is greater than 1. We also assign 0 to the id for name nodes. The declaration for agent symbols corresponds as follows:

  ```
  #define ID_NAME 0
  #define ID_α1 1
    ...
  #define ID_αn n
  #define MAX_AGENTID n
  ```

  In addition, to manage symbols and arities, we define two arrays `Symbols` and `Arities` as follows:

  ```
  char Symbols[MAX_AGENTID+1] = {"", "α1",...,"αn"};
  int Arities[MAX_AGENTID+1] = {1,p1,...,pn};
  ```

- `I=mkInterface(`*n*`)`. This makes a global *n*-size array for the interface and corresponds to:

  ```
  #define SIZE_INTERFACE n
  Agent *I[SIZE_INTERFACE];
  ```

- *x*`=mkAgent(`*id*`)`. This makes a variable *x* whose type is `Agent` and assigns an agent node whose `id` is *id*. This instruction corresponds to: `Agent *`*x*`=mkAgent(`*id*`);`

- *x*`=mkName()`. This makes a variable *x* whose type is `Agent` and assigns an agent node whose `id` is `ID_NAME`. Then it assigns `NULL` to `port[0]` of the *x* in order to be distinguished from indirection nodes: `Agent *`*x*`=mkAgent(ID_NAME);` *x*`->port[0]=NULL;`

- `free(`*x*`)`. This disposes of a graph node assigned to *x* (not recursively, just an assigned node): `freeAgent(`*x*`);`

- *x*`[`*p*`]=`*y*. This assigns a graph element *y* to a port *p* of an agent node *x*. The port *p* in LL0 corresponds to the port $p-1$ in the standardised implementation method, and thus this instruction corresponds to the following code: *x*`[`$p-1$`]=`*y*`;`

- *x*`[0]=`α. This changes the id of an agent *x* into α. This corresponds to the following code: *x*`->id=`α`;`

- `push(`*x*`,`*y*`)`. This pushes two agents onto the equation stack. This corresponds to the following code: `pushActive(`*x*`,`*y*`);`

- `stackFree()`. This disposes of the top element of the equation stack. In the translation result, it occurs in rule procedures. In this implementation, the function `popActive` manages the index of the equation stack, and thus no code is required.

Next we manage the translated LL0 instructions for rule procedures. A rule procedure in LL0 such as "`rule Alpha Beta`" is encoded as a function that is named as `Alpha_Beta`, takes two pointers `*a1` and `*a2` to two elements of the equation, and creates nets according to interaction rules. The special variables `L` and `R` in the rule procedures are denoted as `*a1` and `*a2`, and thus `L[1],L[2],...,R[1],R[2],...` are expressed as: `a1->port[0],a1->port[1],...,a2->port[0],a2->port[1],...`. As an example the rule procedures for `Add` and `Z` and for `Add` and `S` are encoded as follows:

```
1. void Add_Z(Agent *a1, Agent *a2) {
2.   pushActive(a1->port[0],a1->port[1]);
3.   freeAgent(a1);
4.   freeAgent(a2);
5. }
```

```
1. void Add_S(Agent *a1, Agent *a2) {
2.   Agent *aS=mkAgent(ID_S);
3.   Agent *aAdd=mkAgent(ID_Add);
4.   Agent *w=mkName();
5.   aAdd->port[0]=a1->port[0];
6.   aAdd->port[1]=w;
7.   pushActive(aAdd, a2->port[0]);
8.   aS->port[0]=w;
9.   pushActive(a1->port[1], aS);
10.  freeAgent(a1);
11.  freeAgent(a2);
12. }
```

To manage these functions, we define a rule table `R`, which stores pointers to those functions. Here, for simplicity, we use the following simple matrix:

```
typedef void (*RuleFun)(Agent *a1, Agent *a2);
RuleFun R[MAX_AGENTID+1][MAX_AGENTID+1];
```

For instance, the above function is stored as: `R[ID_Add][ID_Z]=&Add_Z;`. The run-time function `eval` is written as follows:

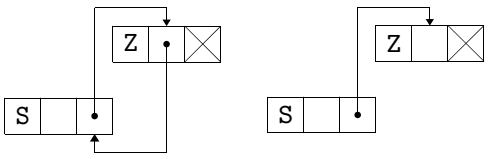
```
1. void eval() {
2.  Agent *a1, *a2;
3.  while (popActive(&a1, &a2)) {
4.   if (a2->id != ID_NAME) {
5.    if (a1->id != ID_NAME) { //Interact
6.     R[a1->id][a2->id](a1, a2);
7.    } else if (a1->port[0] != NULL) {
8.     Agent *a1p0=a1->port[0]; //Ind1
9.     freeAgent(a1);
10.    pushActive(a1p0, a2);
11.   } else a1->port[0]=a2; //Var1
12.  } else if (a2->port[0] != NULL) {
13.    Agent *a2p0=a2->port[0]; //Ind2
14.    freeAgent(a2);
15.    pushActive(a1, a2p0);
16.  } else a2->port[0]=a1; //Var2
17. }
18. }
```

5 Discussion

To examine how data structures in INET, in^2, amineLight (the fastest evaluator) and our implementation affect execution speeds, we implemented a number of evaluators using the different encoding methods. We fix the number of ports as `MAX_PORT` that is obtained during compilation, and we pre-populate the heap with these nodes. The fixed-size node representation has the disadvantage of using more space than needed, but the advantage of being able to manage and reuse nodes in a simpler way [4]. INET and in^2 are based on the graph calculus of interaction nets. Agent nodes are represented as C structures:

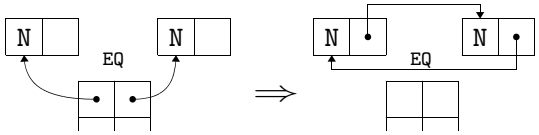
```
1. typedef struct Agent {
2.   int id; struct Port *port[MAX_PORT];
3. } Agent;
4. typedef struct Port {
5.   Agent *agent; int portNum;
6. } Port;
```

(a) INET (b) in^2



In contrast with our method, the principal ports are assigned to `port[0]`, and connections between auxiliary ports are encoded as mutual links between ports of agent nodes; in INET every connection is linked mutually as shown in the above (a), while in^2 uses such mutual connections only for the connection between auxiliary ports as shown in (b). Although in^2 has been proposed before INET, in^2 can be regarded as a refined version of INET. We call the method to use mutual links for the connection between auxiliary ports *undirected encoding*.

(c) Representation of $x = y$ in amineLight



amineLight is based on the Lightweight calculus, and uses names to represent connections between auxiliary ports, and every term is encoded by single links at the start of the execution. Our method, called *directed encoding*, uses single links for the connection between auxiliary ports. In the case of amineLight, an equation such as $x = y$ becomes represented by mutual links during execution as shown in the figure (c), while in our method, the equation preserves the directed encoding, thus a single link (Figure 6). In undirected encoding method names do not occur, and in directed encoding method substitution for each name is performed by removing the indirected connection via the name locally. Thus, the implementation needs no environments for substitutions. We do not garbage collect, taking account of an optimisation mentioned subsequently in this section.

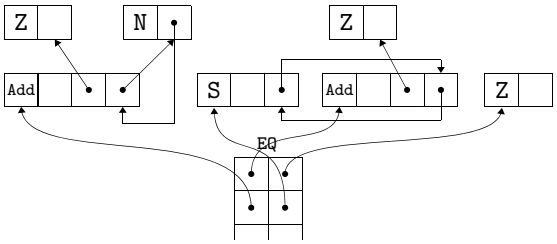
The table below shows execution times in seconds for computing Fibonacci (F_n), Ackermann (A) and Church numerals [8]. We see from the table that our execution times are almost similar to those of amineLight and thus in terms of the cost, the undirected encoding method of in^2 is the best.

	Undirected(INET)	Undirected(in^2)	Directed(Light)	Directed(Simpler)
F_{32}	1.58	1.37	1.52	1.49
F_{33}	2.62	2.29	2.52	2.49
F_{34}	4.37	3.80	4.21	4.15
$A(3,10)$	1.77	1.42	1.59	1.58
$A(3,11)$	7.12	5.73	6.44	6.39
$A(3,12)$	29.47	24.01	26.39	26.14
`2 7 6 I I`	0.73	0.71	1.26	1.28
`2 7 7 I I`	2.12	2.13	3.58	3.68

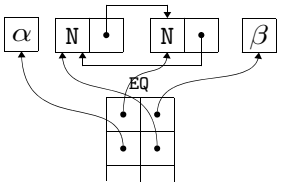
In comparison to amineLight, our implementation computes Fibonacci numbers and Ackermann function a little faster. On the other hand, amineLight performs better in the Application of Church numerals. The reason is that Application of Church numerals demand a lot of computation for names, especially for equations such as $x = y$, yet these operations require extra computational steps in our implementation. To illustrate this point, let us look at the computation of the following sequence of equations: $\alpha = x, y = \beta, x = y$. The lightweight abstract machine in amineLight reduces it to $\beta = \alpha$ in two steps: $\langle \mid \alpha = x, y = \beta, x = y \rangle \rightarrow_{\mathrm{com}} \langle \mid \alpha = y, y = \beta \rangle \rightarrow_{\mathrm{com}} \langle \mid \alpha = \beta \rangle$, whereas our encoding method takes four steps: $(\mid \alpha = x, y = \beta, x = y) \longrightarrow (\mid \alpha = \$y, y = \beta) \longrightarrow (\mid \alpha = y, y = \beta) \longrightarrow (\mid \alpha = \$\beta) \longrightarrow (\mid \alpha = \beta)$. This is because Lightweight calculus manages both sides of an equation, while Simpler one manages only a single side. To illustrate further, the table below shows ratios of name operations (denoted as "N") to interaction operations (as "I").

	I	Light		Simpler	
		N	N/I	N	N/I
F_{32}	74636718	51008017	0.68	65106325	0.87
F_{33}	123315177	82532797	0.67	105344341	0.85
F_{34}	203654818	133540964	0.66	170450820	0.84
$A(3,10)$	134103148	134094952	1.00	134094952	1.00
$A(3,11)$	536641652	536625264	1.00	536625264	1.00
$A(3,12)$	2147025020	2146992248	1.00	2146992248	1.00
`2 7 6 I I`	15676873	43111255	2.75	64538288	4.12
`2 7 7 I I`	46118916	126826871	2.75	190190039	4.12

With respect to the computation of Application of Church numerals, the ratio increases to 4.12 compared to only 2.75 in the amineLight encoding method. Even though the cost of each operation for those names is quite small, as shown in the computation of Fibonacci number that is faster where the ratio increases by 0.18, the much accumulation of the cost induces the less efficiency. We anticipate that it is possible to reduce the cost of operations for names by enhancing the data-structures.



The advantage of the directed encoding method is locality of the rewriting in parallel execution. An active pair must be reduced with the interface preserved, and thus reduction of two active pairs that are connected via an auxiliary port(s) of an interacting agent need to be managed differently because each rewrite will update the same set of auxiliary ports. As an example take the graph shown in the right side figure, which is a graph encoded in in^2 of the net in Figure 5. The two active pairs are connected to each other via the auxiliary port of the interacting agent `Add` and `Z`, and this connection information must be preserved when the active pairs are reduced at once. This checking process could be spread into other parts of the net globally. In the case of the directed encoding method, the connection is preserved by a name as shown in Figure 5, and thus reduction of the two active pairs are performed in parallel as long as critical sections are used to manage names.



The mutual links affect the locality and thus we have proposed the new method of encoding so that a connection between names can be represented by a single link. With respect to the encoding method in amineLight, though it is the directed one, the connections between names are represented as mutual links (Figure (c)) and we need to check for the lock and this can also spread globally. Take the right side figure as an example. This shows a graph after the first step computation of $\alpha = x, y = \beta, x = y$. The two elements of the stack should not be performed at once because each rewriting affects another, so the checking process is also required.

In addition, our model is simpler than the model of amineLight in terms of dealing with equations, thus only a single side of an equation is managed. This derives less critical sections that are caused only by the computational rules Var1 (Figure 6 (a)) and Var2, since name nodes can be pointed-to by two active pairs (that is, auxiliary ports of an active pair are connected). Moreover, those are performed by connecting the ports of names to other principal ports of unlocked agent nodes, therefore these can be locked with an atomic operation such as Compare-and-swapping as follows:

```
1. void eval() {
2.  Agent *a1,*a2;
3.  while (popActive(&a1,&a2)) {
4. loop:
                    ⋮
12.    } else if (!(__sync_bool_compa
   re_and_swap(&(a1->port[0]),NULL,a2)))
13.     goto loop; //retry
```

We finish this section by outlining an important optimisation, but leave the implementation details for future work. Once a net is compiled into an instruction list of LL0, operations such as producing, disposing and connecting ports of agents is done at the level of execution of those instructions. We illustrate the optimisation by considering the rule between `Add` and `S`: $\texttt{Add}(x_1,x_2) = \texttt{S}(y) \Rightarrow \texttt{Add}(x_1,w) = y, x_2 = \texttt{S}(w)$. The compilation result of this rule illustrated in Example 13. In the right-hand side of this rule, the active pair agents `Add` and `S` also occur. Thus, instead of producing new agents, it is possible to reuse the active pair agents. By introducing `StackL` and `StackR` to refer the top elements of the stack *EQ*, it is possible to obtain an alternative sequence of instructions where number of instructions, especially for heap allocations, decreases and thus faster execution is expected:

```
1. rule Add S {          4.   tmpR=StackR          7.   push(x2,tmpR)
2.   w=mkName()          5.   StackR=tmpR[1]       8. }
3.   x2=StackL[2]        6.   tmpR[1]=w
```

6 Conclusion

In this paper we have designed a simple data-structure for representing interaction nets, and designed a corresponding calculus that has a direct relationship with the structure. As a consequence, we can use the calculus to reason about the rewriting process, and also to study the cost of reduction. This led to an investigation into optimising rules which we outlined in Section 5. We believe that this model can provide the basis for further development implementation technology for interaction nets.

References

[1] M. Fernández & I. Mackie (1999): *A Calculus for Interaction Nets*. In G. Nadathur, editor: *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99), Lecture Notes in Computer Science* 1702, Springer-Verlag, pp. 170–187, doi:10.1007/10704567_10.

[2] A. Hassan, I. Mackie & S. Sato (2009): *Compilation of Interaction Nets*. *Electr. Notes Theor. Comput. Sci.* 253(4), pp. 73–90, doi:10.1016/j.entcs.2009.10.018.

[3] Abubakar Hassan, Ian Mackie & Shinya Sato (2010): *A lightweight abstract machine for interaction nets*. *ECEASST* 29. Available at http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/416.

[4] S.L. Peyton Jones (1987): *The Implementation of Functional Programming Languages*. Prentice-Hall International.

[5] Y. Lafont (1990): *Interaction Nets*. In: *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, ACM Press, pp. 95–108, doi:10.1145/96709.96718.

[6] P. J. Landin (1964): *The mechanical evaluation of expressions*. *Computer Journal* 6, pp. 308–320, doi:10.1093/comjnl/6.4.308.

[7] S. Lippi (2002): *in^2 : A Graphical Interpreter for Interaction Nets*. In Sophie Tison, editor: *RTA, Lecture Notes in Computer Science* 2378, Springer, pp. 380–386, doi:10.1007/3-540-45610-4_29.

[8] I. Mackie (1998): *YALE: Yet Another Lambda Evaluator Based on Interaction Nets*. In: *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, ACM Press, pp. 117–128, doi:10.1145/291251.289434.

[9] J. Sousa Pinto (2000): *Sequential and Concurrent Abstract Machines for Interaction Nets*. In Jerzy Tiuryn, editor: *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS), Lecture Notes in Computer Science* 1784, Springer-Verlag, pp. 267–282, doi:10.1007/3-540-46432-8_18.