Electronic Communications of the EASST
Volume 10 (2008)

Proceedings of the Seventh International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008)

Interaction nets: programming language design and implementation

Abubakar Hassan, Ian Mackie and Shinya Sato

16 pages

Guest Editors: Claudia Ermel, Reiko Heckel, Juan de Lara
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Interaction nets: programming language design and implementation

Abubakar Hassan, Ian Mackie and Shinya Sato

[1] Department of Informatics, University of Sussex, Falmer, Brighton BN1 9QJ, UK

[2] LIX, CNRS UMR 7161, École Polytechnique, 91128 Palaiseau Cedex, France

[3] Himeji Dokkyo University, Faculty of Econoinformatics, 7-2-1 Kamiohno, Himeji-shi, Hyogo 670-8524, Japan

Abstract: This paper presents a compiler for interaction nets, which, just like term rewriting systems, are user-definable rewrite systems which offer the ability to specify and program. In the same way that the λ-calculus is the foundation for functional programming, or horn clauses are the foundation for logic programming, we give in this paper an overview of a substantial software system that is currently under development to support interaction based computation, and in particular the compilation of interaction nets.

Keywords: Interaction Nets, programming languages

1 Introduction

The purpose of this article is to describe an on-going project to build a programming environment for interaction nets. We give an overview of the entire project, and then focus on the specific parts that are now complete: programming language design and implementation.

Interaction nets [7] are a graphical, visual, programming language. Programs are expressed as graphs, and computation is expressed as graph transformation. They are a general graph rewriting system, where both the nodes of the graph (the signature) and the rewrite rules are user-defined; in a very similar way to how a term rewriting system would be defined. One key factor that distinguishes interaction nets from other forms of graph rewriting is that the rules are highly constrained, and in such a way to obtain confluence of rewriting by construction.

From another perspective, interaction nets are a low level implementation language: we can define systems of interaction nets that are instruction sets for the target of compilation schemes of other programming languages (typically functional languages, based on the λ-calculus). They have proven themselves successful for application in computer science, most notably with the coding of the λ-calculus, where optimal reduction (specifically Lamping's algorithm [8]) has been achieved [6].

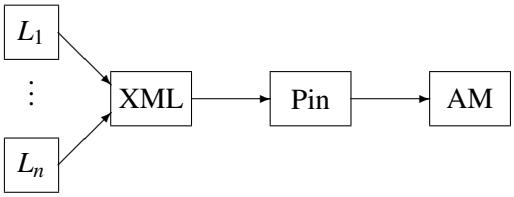
To facilitate the use of interaction nets, we need high quality, robust, implementations, and a programming language to support the programmer. Although we can already program in interaction nets (they are after all Turing complete) they lack the structure that we expect from modern programming languages. Thus the first contribution in this paper is a programming language for interaction nets, which lifts them from the "pure" world to allow them to be used in practice. To give some analogies as to what we are doing, consider the following where we give for a

particular formalism (or model), some examples of languages that have been created for it:

- λ-calculus: functional programming languages, such as Haskell [12], Standard ML [11], OCaml, etc.
- Horn clauses: logic programming languages, such as Prolog.
- Term rewriting systems: OBJ, Elan, Maude [4].
- π-calculus: PICT [13].

The first goal of this paper is to add interaction nets to this list by providing a corresponding programming language that we call Pin. In the list above, the programming language on the right is there to provide not only some syntactical sugar, but also to provide features that the theory does not offer. For instance, if we look in detail at the analogy with the λ-calculus and functional programming languages, functional languages allow the definition of functions such as: `twice f x = f(f x)`, which is a significant improvement over $\lambda fx.f(fx)$ as a programming language, as programs can be reused for instance. In addition languages provide a module system, data-types (built-in and a mechanism for user-defined data-types), input/output, etc.

The language Pin can be considered as an object language for (interaction net) tools such as INblobs [2]. We can write programs in this language, but consider that it shares the same advantages/disadvantages of assembly language. As a diagram, we can see how the whole programming system comes together:



Here, $L_1, \ldots, L_n$ are graphical tools that emit an XML [3] representation of the nets. We translate the XML file into the Pin language, then compile it to our abstract machine (AM) instruction set. When executed, the generated instructions will build the corresponding graph and execute it to normal form. The result of the computation is translated back into an XML file and visualised in the front end. We use XML as an intermediate language to keep Pin and the tools L_i somewhat independent.

The motivation for this line of compilation is that interaction nets have been implemented very efficiently (there are also parallel implementations [15]): by providing a compiler to interaction nets we potentially obtain parallel implementations of programming languages (even sequential ones) for free. However, this is not reported in the current paper: we are simply setting up the foundations and the framework for this development.

To summarise, the paper contains three main contributions: we define a programming language for interaction nets, we define an abstract machine for interaction nets, and we define a compiler for interaction nets.

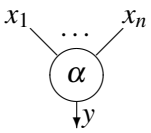
We have implemented this language, and we report on this at the end of the paper. In particular, we show that the new techniques developed for implementing interaction nets improve upon existing implementations. We also remark that the ideas used in this paper could be extended to work with other rewriting systems through translations to interaction nets. Finally, we remark that this is the first attempt in this area, there is now a need to develop tools and optimisation techniques for this paradigm.

Related work. There are several implementations of interaction nets in the literature: a graphical one [9] and a textual, parallel one [15]. These are interpreters for interaction nets and the goal of those works was to investigate interaction nets: our focus is on explicitly extending interaction nets to a rich programming language.

Structure. The rest of this paper is structured as follows. In the next section we recall some background information about interaction nets. In Section 3 we define a programming language for interaction nets. In Section 4 we define an abstract machine for interaction nets (IAM) and give the compilation schemes in Section 5. Section 6 gives details of the implementation. Finally, we conclude the paper in Section 8.

2 Background

An interaction net system is a set Σ of symbols, and a set $\mathscr{R}$ of interaction rules. Each symbol $\alpha \in \Sigma$ has an associated (fixed) *arity*. An occurrence of a symbol $\alpha \in \Sigma$ is called an *agent*. If the arity of α is n, then the agent has $n+1$ *ports*: a distinguished one called the *principal port* depicted by an arrow, and n *auxiliary ports* labelled $x_1, \ldots, x_n$ corresponding to the arity of the symbol. We represent an agent graphically in the following way:



If $n = 0$ then the agent has no auxiliary ports, but it will always have a principal port. We represent agents textually as: $\mathrm{y} \sim \alpha(\mathrm{x_1}, \ldots, \mathrm{x_n})$, and we omit the brackets if the arity of an agent is zero.

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

A pair of agents $(\alpha, \beta) \in \Sigma \times \Sigma$ connected together on their principal ports is called an *active pair*, which is the interaction net analogue of a redex. An interaction rule $((\alpha, \beta) \Longrightarrow N) \in \mathscr{R}$ replaces an occurrence of the active pair (α, β) by a net N. The rule has to satisfy a very strong condition: all the free ports are preserved during reduction, and moreover there is at most one rule for each pair of agents. The following diagram illustrates the idea, where N is any net built from Σ.

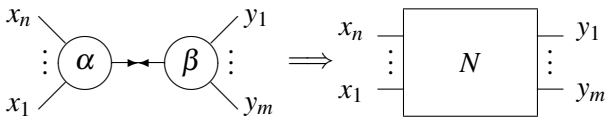


Figure 1 shows an example of a system of interaction nets, which is an append operation of two lists.

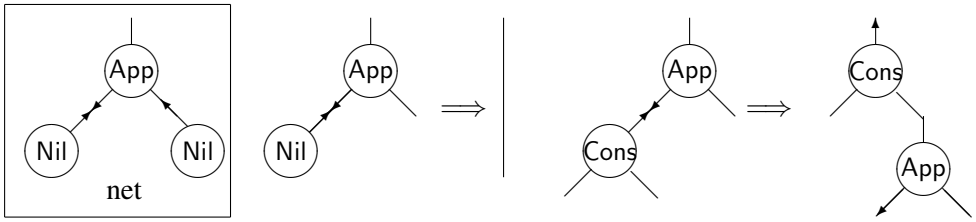
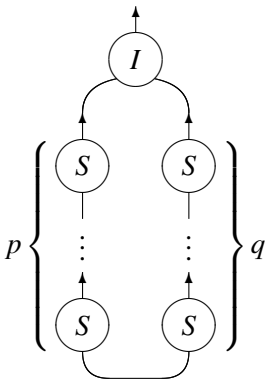


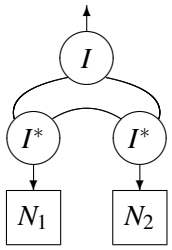
Figure 1: Example net and rules

An implementation of this rewriting process has to create the right-hand side of the net, and make all the connections (re-wirings). Although it may not be the most trivial computation step, it is a known, constant time operation. It is for this reason that interaction nets lend themselves to the study of cost models of computation. *All* aspects of a computation are captured by the rewriting rules—no external machinery such as copying a chunk of memory, or a garbage collector, are needed. Interaction nets are amongst the few formalisms which model computation where this is the case, and consequently they can serve as both a low level operational semantics and an object language for compilation, in addition to being well suited as a basis for a high-level programming language. We refer the reader to other papers on interaction nets for properties and additional theory.

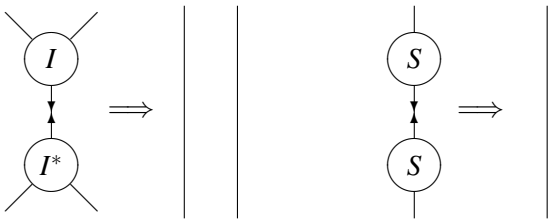
An example: encoding the integers. Many simple examples of systems of interaction nets can be found in the literature, for instance the encoding of arithmetic expressions, etc. However, interaction nets are well suited for alternative representations of data-structures and algorithms. Any integer n can be represented (non-uniquely) as a difference of two natural numbers: $p - q$. Using this idea, we can represent integers in the following way:



Here, the agent S (of arity 1) is interpreted as *successor*. A linear chain of S agents of length n is used to give a representation of a natural number n. The representation of an integer simply takes two chains of length p and q, and connects them together as shown, using the agent I (of arity 2). Although this representation is not unique, we can talk about *canonical forms* when $p = 0$ or $q = 0$ (or both), and there are interaction rules that can compute this. The integer zero is a net where $p = q$, in particular when $p = q = 0$: we can encode addition, subtraction and negation, for which we need several interaction rules. We just detail the encoding of addition. If N_1 and N_2 are the net representations of $z_1 = (p_1 - q_1)$ and $z_2 = (p_2 - q_2)$ respectively, then we can use the following configuration to encode the addition:



The interaction rules for the agents are:



To see how this works, there will be two interactions between the agents I (of nets N_1 and N_2) and I^* which concatenate the two chains. If z_1 and z_2 are both in canonical form and moreover both positive (or both negative) then the addition is completed with the two interactions. Otherwise there will be an additional $\min\{q_1, p_2\}$ interactions between two S agents to obtain a net in normal form.

3 Programming Language

Following [5], an interaction net system can be described as a configuration $c = (\Sigma, \Delta, \mathscr{R})$, where Σ is a set of symbols, Δ is a multiset of active pairs, and $\mathscr{R}$ is a set of rules. A language for interaction nets needs to capture each component of the configuration, and provide ways to structure and organise the components. Starting from a calculus for interaction nets we build a core language. A core language can be seen both as a programming language and as a target language where we can compile high-level constructs. Drawing an analogy with functional programming, we can write programs in the pure λ-calculus and can also use it as a target language to map high-level constructs. In this way, complex high-level languages can be obtained which by their definition automatically get a formal semantics based on the core language.

Nets are written as a comma separated list of agents, corresponding to a flattening of the graph. There are many different (equivalent) ways we could do this depending on the order we choose to enumerate the agents. Using the net on the left-most side of Figure 1 as an example, we can generate a representation as the following list:

$$\mathtt{x} \sim \mathtt{App}(\mathtt{r},\mathtt{a}),\ \mathtt{a} \sim \mathtt{Nil},\ \mathtt{x} \sim \mathtt{Nil}$$

This can be simplified by eliminating some names:

$$\mathtt{App}(\mathtt{r},\mathtt{Nil}) \sim \mathtt{Nil}$$

In this notation the general form of an active pair is $\alpha(\ldots) \sim \beta(\ldots)$. We assume that all variable names occur at most twice. If a name occurs once, then it corresponds to one of the free ports of the net (r is free in the above). If a name occurs twice, then this represents an edge between two ports. In this latter case, we say that a variable is *bound*. The limitation on at most two occurrences corresponds to the requirement that it is not possible to have two edges connected to the same port in the graphical setting.

We represent rules by writing $l \Longrightarrow r$, where l consists of two agents connected at their principal ports. Therefore rules can be written as $\alpha(..) \sim \beta(..) \Longrightarrow N$, and as such we replace the '$\sim$' by '><' so that we can distinguish an occurrence of a rule from an occurrence of an active pair. The two rules on the right of Figure 1 (append) can be represented as:

$$\begin{array}{lcl} \mathtt{App}(\mathtt{r},\mathtt{y}) \ >< \ \mathtt{Nil} & \Longrightarrow & \mathtt{r} \sim \mathtt{y} \\ \mathtt{App}(\mathtt{r},\mathtt{y}) \ >< \ \mathtt{Cons}(\mathtt{v},\mathtt{u}) & \Longrightarrow & \mathtt{r} \sim \mathtt{Cons}(\mathtt{v},\mathtt{z}), \mathtt{u} \sim \mathtt{App}(\mathtt{z},\mathtt{y}) \end{array}$$

The names of the bound variables in the two nets must be disjoint, and the free variables must coincide, which corresponds to the condition that the free variables must be preserved under reduction. Under this assumption, these two rules can be simplified to:

$$\begin{array}{lcl} \mathtt{App}(\mathtt{y},\mathtt{y}) \ >< \ \mathtt{Nil} & \Longrightarrow & \\ \mathtt{App}(\mathtt{Cons}(\mathtt{v},\mathtt{z}),\mathtt{y}) \ >< \ \mathtt{Cons}(\mathtt{v},\mathtt{App}(\mathtt{z},\mathtt{y})) & \Longrightarrow & \end{array}$$

In this notation we observe that the right-hand side of the rule can always be empty. In this case we will omit the '$\Longrightarrow$' symbol. This notation therefore allows sufficient flexibility so that we can either write nets in a compact way (without any net on the right of the '$\Longrightarrow$') or in a more intuitive way where the right-hand side of the net is written out in full to the right of the arrow. During compilation, all rules are transformed into this compact notation.

In Figure 2, we give the syntax for the core language discussed above.

Language extensions. The core language allows the representation of the three main components of an interaction net system: agents, rules and nets. The language is very attractive for theoretical investigations: it is minimal yet computationally complete. The price for this simplicity is programming comfort. Here, we give an overview of some practical extensions that enhance the usability of interaction nets. Details and examples of these extensions can be found in [1].

⟨*Pin*⟩ ::= { ⟨*ruleDef*⟩ | ⟨*netDef*⟩ }

⟨*ruleDef*⟩ ::= ⟨*agent*⟩ '><' ⟨*agent*⟩ ['=>' ⟨*ruleEquation*⟩ { ';' ⟨*ruleEquation*⟩ }]

⟨*ruleEquation*⟩ ::= ⟨*term*⟩ ∼ ⟨*term*⟩

⟨*netDef*⟩ ::= ⟨*equation*⟩ { ';' ⟨*equation*⟩}

⟨*equation*⟩ ::= ⟨*agent*⟩ ∼ ⟨*agent*⟩

⟨*term*⟩ ::= ⟨*agent*⟩ | ⟨*var*⟩

⟨*agent*⟩ ::= ⟨*agentName*⟩ [⟨*ports*⟩]

⟨*ports*⟩ ::= '(' ⟨*term*⟩ [',' ⟨*term*⟩] ')'

⟨*agentName*⟩ ::= ⟨*Name*⟩

⟨*var*⟩ ::= ⟨*Name*⟩

Figure 2: Syntax of the core language

Modular construction of nets. We allow nets to be named so that they can be built in a modular way. Named nets contain as their parameters a list of the net's free ports. Using this mechanism, we can build a new net by *instantiating* a named net with some argument nets. As an example, we can represent the example net in Figure 1 as:

```
Append x,r,a : App(r,a) ∼ x
Applist r : Append Nil,r,Nil
```

Agent Variables. Some rules have the same structure and only differ in the agent names used. Agent variables act as a place holder for agents in the rules. An instance of a rule with a set of agent variables A will create a new rule with elements of A replaced by the actual agents. In the example below, `Templ` has agent variables `A1,A2`. The instance of this rule creates `App(x,x) >< Nil` where the agent variables have been replaced with `App, Nil` appropriately.

```
Templ A1,A2 : A1(x,x) >< A2
Templ (App,Nil)
```

Further extensions. We have designed a module system and a set of built-in agents and rules that perform input/output and arithmetic operations—these are reported in [1].

4 The Abstract Machine

Here we describe the interaction net abstract machine (IAM) by giving the instructions operating on a machine state. We first set up some notation used in the description of the machine.

Definition 1 (Memory model) Let $Adr \subseteq \mathbb{N}$ be a set of memory locations. Agents of an interaction net system are $agent = name \times arity \times W$, where $arity \in \mathbb{N}$, $name$ is an identifier, and W models connectivity between $agent$ ports: $W = \{((l,p),(l',p')) \mid l,l' \in Adr, p,p' \in \mathbb{N}\}$. An element $((l_1,p_1),(l_2,p_2))$ in the set W is ordered if either $l_1 < l_2$ or $l_1 = l_2,\ p_1 < p_2$. If an agent stored at location l_k has it's auxiliary port p_i connected to the port p_j of some agent at location l_m, then $((l_k,p_i),(l_m,p_j)) \in W$.

We next define two functions to operate on the machine state:

Definition 2

- $\mathscr{L}_a : Adr \times \mathbb{N} \to Adr$ returns the location $l \in Adr$ pointed to by a given port of some $agent$: $\mathscr{L}_a(l_k,p_i) = l_m$ such that $((l_k,p_i),(l_m,p_j)) \in W$.
- $\mathscr{L}_p : Adr \times \mathbb{N} \to \mathbb{N}$ returns a port number that is connected to a port of some $agent$ node: $\mathscr{L}_p(l_k,p_i) = p_j$ such that $((l_k,p_i),(l_m,p_j)) \in W$.

We define the function $\Upsilon : name \times name \to Inst$ that given a pair of $names$ for an active pair (α,β) of a rule $(\alpha,\beta) \Longrightarrow N$, returns a sequence of IAM instructions that will build and rewire the net N. The mappings in Υ are constructed during the compilation of a rule.

Machine configuration components. The IAM machine consists of rules that transform configurations. A configuration is given by a 5-tuple $\langle C,\varsigma,F,A,H\rangle$ where C is a sequence of instructions, ς is a stack of frames. Each frame has an array of local variables and an operand stack. F is a set of pointers to the variable agents of the net, A is a stack of active pair agents, and H is the heap.

An IAM program C is a sequence of instructions that we summarise in Figure 3. We write '$-$' for the empty sequence and $i,v,p,ar \in \mathbb{N}$.

The component ς is a stack of frames. Each frame $f = (L,S)$ where L is a partial function with a finite domain of definition, mapping memory locations to their contents. If L is defined, then $L[i \mapsto l]$ means that $L(i) = l$. S is an operand stack produced by the grammar: $S := -\mid v : S$ where v is any value representing a constant or a memory location, and $-$ is the empty stack.

The component F is a mapping from identifiers to a pair of natural numbers defined by: $F(x) = (l,p)$. Intuitively, it is used to hold the interface of the net.

The heap $H : Adr \to agent$ returns an $agent$ node given some location $l \in Adr$. The special token $next$ is used to hold the next free location $l \in dom(H)$. Intuitively, H is a memory area filled with agent nodes. Whenever a new node is put into the heap, the unused area marked by $next$ is updated.

Figure 4 gives the IAM instructions as a set of transition rules. Each transition rule takes the form:

$$\Upsilon \vdash \langle C,\varsigma,F,A,H\rangle \Rightarrow \Upsilon \vdash \langle C',\varsigma',F',A',H'\rangle$$

which indicate how the components of the machine are transformed. We abbreviate $(L,S) : \varsigma$ to (L,S) in a configuration with only one frame in ς.

Initial and final states. The machine initialises the components C and Υ giving the initial configuration: $\Upsilon \vdash \langle C,-,[],-,[]\rangle$. The machine stops *successfully* when the instruction `halt`

is executed with the configuration $\Upsilon \vdash \langle -, -, F, -, H \rangle$ or prematurely if a pre-condition of an instruction is not satisfied. In this case, the final configuration is the one obtained after the last instruction that has been executed successfully.

The evaluation mechanism. The evaluation of the net is started by executing the `eval` instruction. This instruction is appended at the end of the code sequence for the start active pair or *initial expression*. Thus, before the evaluation of the net, there is at least one active pair in the machine's active pair stack A. The pair in the stack A is examined and the code sequence of the rule for the pair is appended to the code component C of the machine (see semantics of `eval` in Figure 4).

The code for a rule will load one of the active agents into the stack S using the instruction `loadActive`, then start to build and rewire the right hand side net of the rule to the auxiliary agents connected to the interacting agent in the stack. The instruction `pop` pops the active agent from the component A. Evaluation is terminated when A is empty and execution jumps to the instruction sequence after `eval`.

Instruction	*Description*
`enter`	push a new frame into the stack ς
`return`	remove the top frame from ς
`dup`	duplicate the top element on the stack S
`pop`	remove the top element on the active pair stack A
`load` i	push the element at index i of L onto the stack S.
`store` i	remove the top element of S and store it in L at index i.
`ldc` v	push the value v onto the stack S.
`fstore` x	store the top 2 elements at the top of S onto index x in F.
`fload` x	push the elements at index x of F onto the stack S.
`mkAgent` ar α	allocate (unused) memory for an agent node of arity ar and name α in the heap H.
`mkVar` x	allocate memory for a variable node of arity 2 and name x in the heap
`getConnection` p i	push the agent a and the port number of a that connects at the auxiliary port p of the agent stored in local variable i
`loadActive` α	push the active agent α from active pair stack
`connectPorts`	pop two agents and two port numbers and connects the ports of the agents. If both ports are 0 (active pair) push an agent to A
`eval`	evaluate the active pair on top of the active stack.
`halt`	stop execution.

Figure 3: Summary of IAM instructions

$$\Upsilon \vdash \langle \texttt{enter} : C, \varsigma, F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, ([], -) : \varsigma, F, A, H\rangle$$
$$\Upsilon \vdash \langle \texttt{return} : C, (L,S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, \varsigma, F, A, H\rangle$$
$$\Upsilon \vdash \langle \texttt{dup} : C, (L, v : S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L, v : v : S), F, A, H\rangle$$
$$\Upsilon \vdash \langle \texttt{pop} : C, (L,S), F, l : A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L,S), F, A, H\rangle$$
$$\Upsilon \vdash \langle \texttt{load}\, i : C, (L[i \mapsto v], S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L[i \mapsto v], v : S), F, A, H\rangle$$
$$\Upsilon \vdash \langle \texttt{store}\, i : C, (L, v : S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L[i \mapsto v], S), F, A, H\rangle$$
$$\Upsilon \vdash \langle \texttt{ldc}\, v : C, (L,S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L, v : S), F, A, H\rangle$$
$$\Upsilon \vdash \langle \texttt{fstore}\, x : C, (L, p : l : S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L, p : l : S), F[x \mapsto (l,p)], A, H\rangle$$
$$\Upsilon \vdash \langle \texttt{fload}\, x : C, (L,S), F[x \mapsto (l,p)], A, H[l_a \mapsto (n,a,\{((l_a,p_l),(l,p))\} \cup w)]\rangle \Rightarrow$$
$$\Upsilon \vdash \langle C, (L, p : l : S), F, A, H[l_a \mapsto (n,a,w)]\rangle$$
$$\Upsilon \vdash \langle \texttt{mkAgent}\ ar\ \alpha : C, (L,S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L, l : S), F, A, H[l \mapsto (\alpha, a, \emptyset)]\rangle$$
$$\text{where } l = next$$
$$\Upsilon \vdash \langle \texttt{mkVar}\ x\ : C, (L,S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L, 1 : l : S), F, A, H[l \mapsto (x, 2, \emptyset)]\rangle$$
$$\text{where } l = next$$
$$\Upsilon \vdash \langle \texttt{getConnection}\ p\ i : C, (L[i \mapsto l], S), F, A, H\rangle \Rightarrow$$
$$\Upsilon \vdash \langle C, (L[i \mapsto l], \mathscr{L}_p(l,p) : \mathscr{L}_a(l,p) : S), F, A, H\rangle$$
$$\Upsilon \vdash \langle \texttt{loadActive}\ \alpha : C, (L,S), F, l : A, H[l \mapsto (n,a,w)]\rangle \Rightarrow$$
$$if(n = \alpha)$$
$$\Upsilon \vdash \langle C, (L, l : S), F, \mathscr{L}_a(l,0) : A, H[l \mapsto (n,a,w)]\rangle$$
$$else$$
$$\Upsilon \vdash \langle C, (L, \mathscr{L}_a(l,0) : S), F, l : A, H[l \mapsto (n,a,w)]\rangle$$
$$\Upsilon \vdash \langle \texttt{connectPorts} : C, (L, p_1 : l_1 : p_2 : l_2 : S), F, A, H[l_1 \mapsto (n_1, a_1, w_1),$$
$$l_2 \mapsto (n_2, a_2, w_2)]\rangle \Rightarrow$$
$$if(p_1 + p_2 = 0)$$
$$\Upsilon \vdash \langle C, (L,S), F, l_2 : A,$$
$$H[l_1 \mapsto (n_1, a_1, w_1 \cup \{((l_1,0),(l_2,0))\}),$$
$$l_2 \mapsto (n_2, a_2, w_2 \cup \{((l_1,0),(l_2,0))\})]\rangle$$
$$else$$
$$\Upsilon \vdash \langle C, (L,S), F, A,$$
$$H[l_1 \mapsto (n_1, a_1, w_1 \cup \{((l_1,p_1),(l_2,p_2))\}),$$
$$l_2 \mapsto (n_2, a_2, w_2 \cup \{((l_1,p_1),(l_2,p_2))\})]\rangle$$
$$\Upsilon[(\alpha,\beta) \mapsto c] \vdash \langle \texttt{eval} : C, (L,S), F, l_1 : A, H[l_1 \mapsto (\alpha, a_1, w_1 \cup \{((l_1,0),(l_2,0))\}),$$
$$l_2 \mapsto (\beta, a_2, w_2 \cup \{((l_1,0),(l_2,0))\})]\rangle \Rightarrow$$
$$\Upsilon[(\alpha,\beta) \mapsto c] \vdash \langle c : \texttt{eval} : C, (L,S), F, l_1 : A, H[l_1 \mapsto (\alpha, a_1, w_1), l_2 \mapsto (\beta, a_2, w_2)]\rangle$$
$$\Upsilon \vdash \langle \texttt{eval} : C, (L,S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L,S), F, A, H\rangle$$
$$\Upsilon \vdash \langle \texttt{halt} : C, (L,S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle -, -, F, -, H\rangle$$

Figure 4: IAM instructions

5 Compilation

The compilation of Pin into IAM instructions is governed by the schemes: $\mathscr{C}_{pin}$ compiles a program, $\mathscr{C}_a$ compiles an agent, $\mathscr{C}_t$ compiles a term, $\mathscr{C}_n$ compiles a net and $\mathscr{C}_r$ compiles a rule. For correctness purposes, the list of functions generated during compilation must be unfolded in the following ordered way: if $\mathscr{C}_1;\ldots;\mathscr{C}_k$ are the schemes generated by $\mathscr{C}_{pin}$, then the next function to be unfolded is $\mathscr{C}_i$ such that all functions $\mathscr{C}_p$, $p < i$, have already been unfolded.

The compilation of a program generates the following code:

$$\mathscr{C}_{pin}[\![(\Sigma, \langle u_1 \sim v_1, \ldots u_n \sim v_n\rangle, \mathscr{R})]\!] = \\ \mathscr{C}_n[\![u_1 \sim v_1, \ldots, u_n \sim v_n]\!]; \texttt{eval}; \texttt{halt}; \mathscr{C}_r[\![r_1]\!]; \ldots \mathscr{C}_r[\![r_n]\!];$$

where $r_1, ..., r_n = \mathscr{R}$ are instances of rules. Σ is a set of symbols and each $u_i \sim v_i$ is an active pair. The compilation scheme $\mathscr{C}_n[\![u \sim v, \ldots, u_n \sim v_n]\!]$ compiles a sequence of active pairs. We use the scheme $\mathscr{C}_r[\![r_i]\!]$ to compile a rule $r_i \in \mathscr{R}$:

$$\mathscr{C}_r[\![r_i]\!] = Inst = \mathscr{C}_r[\![\alpha(t_1, \ldots, t_n) \ >< \ \beta(u_1, \ldots, u_n) => u_1 \sim s_1, \ldots, u_n \sim s_n]\!], \\ \Upsilon[(\alpha, \beta) \mapsto Inst, (\beta, \alpha) \mapsto Inst].$$

Compilation of a rule creates a mapping from active agent names to the instruction sequence *Inst* generated in the rule table Υ.

Figure 5 collects together the compilation schemes $\mathscr{C}_n$ and $\mathscr{C}_r$, that generate code for the input source text. The schemes use a set $\mathscr{N}$ of identifiers to hold all free variables of the net. Compiling a variable that already exists in $\mathscr{N}$ means that the variable is bound. The auxiliary function $ar(\alpha)$ returns the arity of the agent $\alpha \in \Sigma$.

We end this section with a concrete example of compilation, and give the code generated for the simple system given below:

```
Eps >< Eps
Eps ~ Eps
```

This system contains just one agent and one rule, and the net to be compiled is an instance of that rule. The output of the compiler is given below.

```
enter
mkAgent 0 Eps       enter
ldc 0               loadActive Eps
mkAgent 0 Eps       store 0
ldc 0               loadActive Eps
connectPorts        store 0
eval                pop
halt                return
```

The first and second columns are the code sequence for the active pair and rule respectively. The abstract machine loads the program into memory then sequentially executes the byte codes for the active pairs. The instruction `eval` calls the execution of the code block for the corresponding rule. The rule table Υ is examined to determine the start and end *address* of the code sequence for the rule. Below we give a snapshot of the execution trace.

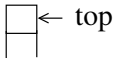
$$\mathscr{C}_t[\![x]\!] = \begin{cases} if\ (x \in \mathscr{N}) \\ \quad \texttt{fload}\ x; \\ \quad \mathscr{N} \setminus \{x\} \\ else \\ \quad \texttt{fstore}\ x; \\ \quad \texttt{mkVar}\ x; \\ \quad \mathscr{N} \cup \{x\} \end{cases} \qquad \mathscr{C}_e[\![t \sim s]\!] = \begin{cases} \mathscr{C}_t[\![t]\!]; \\ \mathscr{C}_t[\![s]\!]; \\ \texttt{connectPorts}; \end{cases}$$

$$\mathscr{C}_t[\![\alpha(t_1,\ldots,t_n)]\!] = \begin{cases} \texttt{mkAgent}\ ar(\alpha)\ \alpha; \\ \text{for } 1 \le i \le n \\ \quad \mathscr{C}_p\ [\![t_i]\!]\ i; \\ \texttt{ldc}\ 0; \end{cases} \quad \mathscr{C}_{rr}[\![t]\!]\ j\ i = \begin{cases} \texttt{getConnection}\ j\ i; \\ \mathscr{C}_t[\![t]\!]; \\ \texttt{connectPorts}; \end{cases}$$

$$\mathscr{C}_r[\![\alpha(t_1,\ldots,t_n) \\ >\!<\ \beta(v_1,\ldots,v_k) \\ \Rightarrow u_1 \sim s_1,\ldots, \\ u_m \sim s_m]\!] = \begin{cases} \texttt{enter}; \\ \mathscr{C}_{rt}[\![\alpha(t_1,\ldots,t_n)]\!]; \\ \mathscr{C}_{rt}[\![\beta(v_1,\ldots,v_k)]\!]; \\ \text{for } 1 \le i \le m \\ \quad \mathscr{C}_e[\![u_i \sim s_i]\!]; \\ \texttt{pop}; \\ \texttt{return}; \end{cases} \quad \mathscr{C}_n[\![u \sim v,\ldots,u_n \sim v_n]\!] = \begin{cases} \texttt{enter}; \\ \mathscr{C}_e[\![u \sim v]\!]; \\ \vdots \\ \mathscr{C}_e[\![u_n \sim v_n]\!]; \end{cases}$$

$$\mathscr{C}_{rt}[\![\alpha(t_1,\ldots,t_n)]\!] = \begin{cases} \texttt{loadActive}\ \alpha; \\ \texttt{store}\ 0; \\ \text{for } 1 \le i \le n \\ \quad \mathscr{C}_{rr}[\![t_i]\!]\ i\ 0; \end{cases} \qquad \mathscr{C}_p[\![t]\!]\ j = \begin{cases} \texttt{dup}; \\ \texttt{ldc}\ j; \\ C_t[\![t]\!]; \\ \texttt{connectPorts}; \end{cases}$$

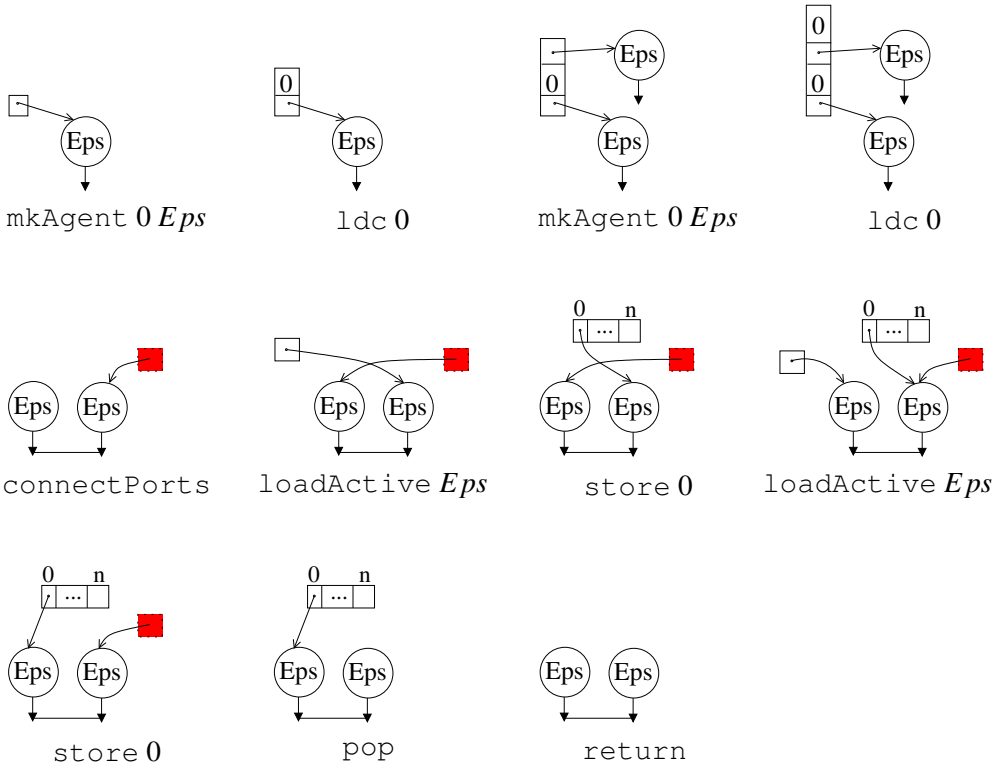
Figure 5: Compilation schemes

-  represents the active pair stack A of the machine.
- 

represents the stack S.
- 

represents the local variable array L.

The state after execution of each instruction is shown. Components that do not contain any value are omitted. Note that this net contains no interface, thus the interface list F does not appear in the execution trace. Refer to the transition rules in Figure 4 for details on how these configurations are obtained.



Observe that after the execution of `connectPorts`, a pointer to the newly created active pair is pushed into the stack A. Since the rule for this active pair contains an empty right hand side net, there is no re-wiring that is performed. After evaluation, the active pair stack becomes empty. After the last instruction `return` of the rule, remaining active pair agents in the heap are unreachable from any of the machine's components, and can be garbage collected or reused. We do not address the issues of heap garbage collection or agent reuse in this paper.

6 The implementation

Here we give a brief overview of the pragmatics of the language. We have implemented the compiler, and here we show example programs, the use of the system, and also some benchmark results comparing with other implementations of interaction nets.

The prototype implementation of the compiler and abstract machine can be downloaded from the project's web page. The compiler reads a source program and outputs an executable with the extension '.pin'. The pin file can then be executed by the abstract machine. Various examples and instructions on how to compile and execute a program are provided on the webpage.

The table below shows some benchmark results that we have obtained. We compare the execution time in seconds of our implementation (Pin) with Amine [14] - an interaction net interpreter, and SML [11] - a fully developed implementation. The last column gives the number of interactions performed by both Pin and Amine. The first two input programs are applications of Church numerals where $n = \lambda f.\lambda x.f^n x$ and $I = \lambda x.x$. The encodings of these terms into interaction nets

are given in [10]. The next programs compute the Ackermann function defined by:

$$A(m,n) = \begin{cases} n+1 \text{ if } m = 0 \\ A(m-1,1) \text{ if } n = 0 \text{ and } m > 0 \\ A(m-1,A(m,n-1)) \text{ if } m > 0 \text{ and } n > 0 \end{cases}$$

The following rules are the interaction net encoding of the Ackermann function:

```
Pred(Z) >< Z,                    Dup(Z,Z) >< Z,
Pred(x) >< S(x),                 Dup(S(a),S(b)) >< S(Dup(a,b)),
A(r,S(r)) >< Z,                  A1(Pred(A(S(Z),r)),r) >< Z,
A(A1(S(x),r),r) >< S(x),         A1(Dup(Pred(A(r1,r)),A(y,r1)),r) >< S(y),
```

and A(3,8) means computation of $\texttt{A}(\texttt{S}(\texttt{S}(\texttt{S}(\texttt{S}(\texttt{S}(\texttt{S}(\texttt{S}(\texttt{Z}))))))),\texttt{r}) \sim \texttt{S}(\texttt{S}(\texttt{S}(\texttt{Z})))$.

Input	Pin	Amine	SML	Interactions
322II	0.002	0.006	2.09	383
245II	0.016	0.088	1355	20211
A(3,8)	3	16	0.04	8360028
A(3,10)	51	265	0.95	134103148

We can see from the table that the ratio of the average number of interactions/sec of Pin to Amine is approximately 3 : 1. Interaction nets are by definition very good in sharing computation thus more efficient than SML in the first two programs. However, interaction nets do not perform well in computations that benefit from sharing data - interacting agents are consumed. Our short term goal is to extend interaction nets with data sharing mechanisms.

7 Acknowledgment

We thank our anonymous referees and all our colleagues at the workshop for their helpful comments.

8 Conclusions

In this paper we have given an overview of a programming language design and compilation for interaction nets. Experience with the compiler indicates that the system can be used for small programming activities, and we are investigating building a programming environment around this language, specifically containing tools for visualising interaction nets and editing and debugging tools.

Current research in this area is focused on richer programming constructs and higher level languages of interaction that do not burden the programmer with some of the linearity and pattern matching constrains. The compiler presented in this paper is a first attempt to compile interaction nets, and issues such as compiler optimisations are very much the subject of current research.

There are well-known translations of other rewriting formalisms into interaction nets: the compiler presented in this paper can consequently be used for these systems. Current work is investigating the usefulness of this approach.

Bibliography

[1] http://www.informatics.sussex.ac.uk/users/ah291/.

[2] J. B. Almeida, J. S. Pinto, and M. Vilaa. A tool for programming with interaction nets. In J. Visser and V. Winter, editors, *Proceedings of the Eighth International Workshop on Rule-Based Programming*. Elsevier, June 2007. to appear in Electronic Notes in Theoretical Computer Science.

[3] J. Berstel and L. Boasson. XML grammars. In *Mathematical Foundations of Computer Science*, pages 182–191, 2000.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *A Maude Tutorial*. SRI International, 2000.

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

[6] G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 15–26. ACM Press, Jan. 1992.

[7] Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, Jan. 1990.

[8] J. Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 16–30. ACM Press, Jan. 1990.

[9] S. Lippi. in^2 : A graphical interpreter for interaction nets. In S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 380–386. Springer, 2002.

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

[11] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[12] S. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[13] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report 476, Indiana, 1997.

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

[15] J. S. Pinto. Parallel evaluation of interaction nets with mpine. In A. Middeldorp, editor, *RTA*, volume 2051 of *Lecture Notes in Computer Science*, pages 353–356. Springer, 2001.