Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2009) 73–90
www.elsevier.com/locate/entcs

Compilation of Interaction Nets

Abubakar Hassan Ian Mackie Shinya Sato

Department of Informatics, University of Sussex, Falmer, Brighton, BN1 9QJ, U.K.

LIX, CNRS UMR 7161, École Polytechnique, 91128 Palaiseau Cedex, France

Faculty of Econoinformatics, Himeji Dokkyo University, 5-7-1 Kamiohno, Himeji-shi, Hyogo 670-8524, Japan

Abstract

This paper is about a new implementation technique for interaction nets—a visual programming language based on graph rewriting. We compile interaction nets to C, which offers a robust and efficient implementation, in addition to portability. In the presentation of this work we extend the interaction net programming paradigm to introduce a number of features which make it a practical programming language.

Keywords: interaction nets, compilation

1 Introduction

Interaction nets [6] are a graph rewriting system where programs are represented as graphs and computation is based on graph rewriting. They enjoy good properties such as strong confluence, Turing completeness and locality of reduction. For these reasons, optimal [3,7] and efficient [9] λ-calculus evaluators based on interaction nets have evolved. Indeed, interaction nets have proved to be very fruitful in the study of the dynamics of computation. However, they are currently only useful for theoretical investigations.

In this paper we take a step towards developing a practical programming language for interaction nets. In the same way that functional languages are based on the λ-calculus, logic languages are based on Horn clauses, or the pict [10] language is based on the π-calculus, here we present a language based on this graph rewriting system and give a compilation into C.

There are several implementations of interaction nets [12,8,4,5], but they all suffer at least from one or more drawbacks: execution speed, lack of modern language constructs such as built-in types, input/output etc. The main goal of this paper is to address these issues so that we can shift the use of interaction nets from theoretical investigations to a practical programming paradigm. Firstly, we develop a textual

doi:10.1016/j.entcs.2009.10.018

syntax for interaction nets with higher level constructs that provide programming comfort. We then show how this language can be compiled down to native codes via the C programming language [13]. We can use C as a machine independent low-level language that is well suited as a portable target language for the implementation of programming languages. Over the years, C compilers have gone through many improvements to generate optimised machine code. By compiling to C, we also benefit in the improvements of C code generation. In addition, we gain instant portability because C is implemented on a variety of platforms. Many languages [15,1,14] have benefited from this line of compilation.

To summarise, the main contributions of this paper are as follows:

- We extend the definition of interaction nets to allow: built-in data types and conditional rewrite rules; states and state transformers.
- We define a compiler from interaction nets to native codes via the C language.

The extensions will break some of the main theoretical properties of interaction nets, but our computations stay deterministic since we fix a particular strategy. Interaction nets are one-step confluent, which means that all reduction sequences to normal form are the same length: by picking one at random we are not affecting the efficiency of the system.

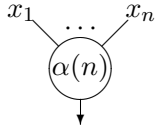
In our previous work [4] we defined a textual language for interaction nets (Pin) and described how it is transformed into an intermediate language. We then defined an abstract machine that executes Pin instructions. This paper is concerned with: developing a richer language for interaction nets, extending the language to cater for the introduced source language constructs and compiling into C code.

In the next section we give some background material on interaction nets. We discuss our source language in Section 3. In Sections 4 and 5 we define the compilation schemes from our source language to C. In Section 6 we give the implementation details before concluding the paper in Section 7.

2 Interaction nets

Here we review the basic notions of interaction nets. We refer the reader to [6] for a more detailed presentation. Interaction nets are specified by the following data:

- A set Σ of *symbols*. Elements of Σ serve as *agent* (node) labels. Each symbol has an associated arity ar that determines the number of its *auxiliary ports*. 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*. Each agent may have attributes. In this paper, we will restrict attributes to just base types: integers and booleans, and we write the attribute in brackets after the name.



We can represent this agent textually as $x_0 \sim \alpha(n)[x_1, \ldots, x_n]$, where x_0 is the

principal port.

- A *net* built on Σ is an undirected graph 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. A port which is not connected is called a *free port*. A set of free ports is called an *interface*.
- Two agents $(\alpha, \beta) \in \Sigma \times \Sigma$ connected via their principal ports form an *active pair* (analogous to a redex). An interaction rule $((\alpha, \beta) \Longrightarrow N) \in \mathcal{R}$ 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 following diagram illustrates the idea, where N is any net built from Σ.

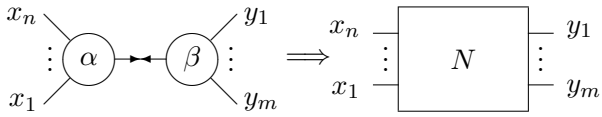


Figure 1 gives a simple example of an interaction net system that encodes the addition operation. We represent numbers using agents `S` and `Z`, corresponding to the usual constructors. Figure 2 gives an example reduction sequence that shows how a net representing $1 + 1$ is reduced to 2.

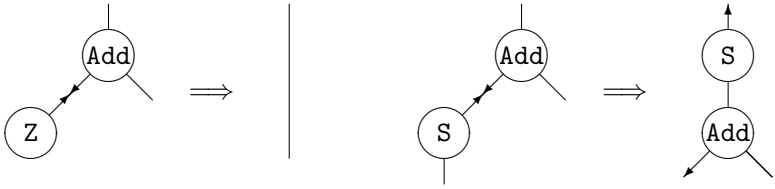


Fig. 1. Rules for addition

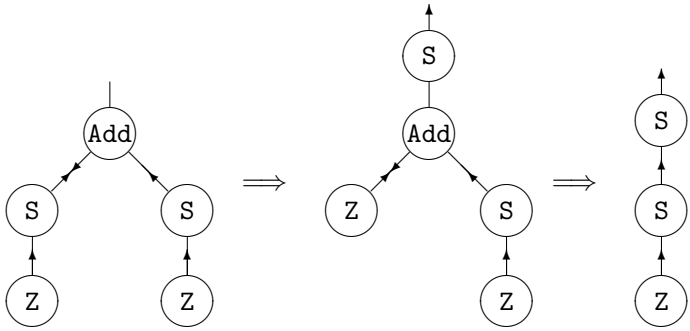


Fig. 2. Example reduction sequence

3 The Source Language - INETS

Following [2], an interaction net system can be described as a configuration $c = (\Sigma, \Delta, \mathcal{R})$, where Σ is a set of symbols, Δ is a multiset of active pairs, and $\mathcal{R}$ 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.

We write nets textually as a comma separated list of agents. This just corresponds to a flattening of the net, and there are many different (equivalent) ways to do this depending on the order the agents are enumerated. As an example, we write the initial net in Figure 2 as:

```
a∼Add[x,y], a∼S[b], b∼Z, y∼S[v], v∼Z.
```

This can be simplified by replacing equals for equals:

```
S[Z]∼Add[x,S[Z]].
```

In this notation the general form of an active pair is $\alpha[\ldots] \sim \beta[\ldots]$. All variable names occur at most twice. If a name occurs once, then it corresponds to a free port of the net (x is free in the above). If a name occurs twice, then it represents an edge between two ports, in which case we say that the variable is *bound*.

We represent rules by writing $l \Longrightarrow r$, where l is an active pair on the left of the rule, and r is the resulting net. In particular, we note that l will always consist of two agents connected at their principal ports. We also note that all rules can be written in a form $\alpha(..)[..] \sim \beta(..)[..] \Longrightarrow N$, and as such we replace the '∼' by '><' so that we can distinguish an occurrence of a rule from an occurrence of an active pair. For example, the rules for the addition operation in Figure 1 can be represented using the syntax:

```
Add[x,y] >< Z =>  x∼y
Add[x,y] >< S[a] =>  x∼S[b], a∼Add[b,y]
```

or in a more compact way:

```
Add[x,y] ><
 Z =>  x∼y
 S[a] =>  x∼S[b], a∼Add[b,y]
```

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.

The introduction of agents with values provide us with an efficient representation of data types in interaction nets. For example, we can represent numbers in a way that is directly supported by hardware rather than using `S` and `Z` agents. We also introduce a set of deterministic operations on the built-in data types: booleans, integers and characters. The example rule below shows how we can encode the addition operation in a way that is directly supported by hardware.

```
Fact[result] >< Num(int x) =>
    if(x < 0)
      result∼Error;
    else  if(x == 0)
       result∼Num(1);
    else
      Fact[Mult(x)[result]]∼Num(x-1) ;
Mult(int x)[res] >< Num(int y) =>
      res∼Num(x*y);
main(){
  Fact[result]∼Num(6);
}
```

Fig. 3. Example program: factorial

```
Add(int x)[res] ><  Num(int y) => res∼Num(x+y);
```

3.1 Conditional rewrite rules

We allow rules to contain multiple right-hand side (rhs) nets. If either (or both) of the interacting agents are holding a value, then we can use these values to give different rhs of the rule by specifying a condition. The conditions must be all disjoint (there cannot be two rhs nets that can be applied). During reduction, we evaluate the conditional expression to determine which net will be applied to an active pair. Proposition 3.1 shows that these conditional rewrite rules do not affect the Strong Confluence property of interaction nets. Figure 3 gives an example program that computes the factorial of a number. The rhs net for the rule between agents `Fact` and `Num` will depend on the value x.

3.2 State transformers

We allow our programs to be decorated with *states* and *state transformers*. Formally, a $State : Adr \hookrightarrow Val$ is a partial function from (memory) locations Adr to (storable) values Val. A state transformer $ST \in \{agn, new, read\} : State \rightarrow (Val \times State)$ is a function which given a state produces a value and a new state. The transformer agn updates the value at a memory location, new allocates a new

memory location and *read* reads the value at a given location:

$$agn(x,y)\ \sigma = \begin{cases} (\bot,\bot), & \text{if } y \notin dom(\sigma) \\ (v, \sigma[x \mapsto v]), & \text{otherwise} \\ \text{where } \sigma(y) = v \end{cases}$$

$$new\ x\ \sigma = \begin{cases} (\bot,\bot), & \text{if } \sigma = \bot \\ (\bot, \sigma[x \mapsto \bot]), & \text{otherwise} \\ \text{where } \ x \notin dom(\sigma) \end{cases}$$

$$read\ x\ \sigma = \begin{cases} (\bot,\bot), & \text{if } x \notin dom(\sigma) \\ (v, \sigma) & \text{otherwise} \\ \text{where } \sigma(x) = v \end{cases}$$

We represent these transformers in our source language using:

$$stm \in ST ::= x \texttt{ = } y \mid T\ x \mid x \mid stm\texttt{;}stm$$

where x=y represents $agn(x,y)$, $T \in \{\texttt{int},\texttt{bool},\texttt{char}\}, T\ x$ is a declaration of a variable and represents the function *new* x. An occurrence of a variable represents the *read* operation and $stm\texttt{;}stm$ represents a sequence of state transformers. State transformers defined in a rule will be executed either before or after reducing the active pair. The structure of a rule is given by:

$$\alpha(..)[..]\ \texttt{><}\ \beta(..)[..]\ \Longrightarrow\ stm_1\ \ N\ \ stm_2$$

The sequence of statements stm_1 will be executed before the application of the rule α, β and stm_2 will be executed after the rewrite. The example in Figure 3 can be written with state decorations, as we show in Figure 4. In this example a global variable `counter` is declared that counts the number of interactions. When each rule is applied, the counter is incremented. The net named `main` is the entry point to the program. In the main net, we initialise the global variable `counter` followed by an active pair definition and a print statement.

Generally speaking, the use of states breaks the Strong Confluence property. Below we give an example program where the result depends on the evaluation strategy.

```
Boolean b;
   P >< Q[r] => if(b) {r~N1; b = false;} else r~N2;
   P >< R[r] => if(b) {r~N3; b = false;} else r~N4;

  main(){
       b = true;
       P~Q[r1],P~R[r2]; // reduces to r1~N1,r2~N4
    // P~R[r1],P~Q[r2];    reduces to r1~N3,r2~N1
  }
```

```
int counter;
Fact[result] >< Num(int x) =>
     counter = counter + 1;
     if(x < 0)
       result∼Error;
     else  if(x == 0)
       result∼Num(1);
     else
       Fact[Mult(x)[result]]∼Num(x-1) ;

Mult(int x)[res] >< Num(int y) =>
      counter = counter + 1;
      res∼Num(x*y);
main(){
  counter = 0;
  Fact[result]∼Num(6);
  print "total number of interactions ",counter;
}
```

Fig. 4. Example program: factorial with state

However, if we restrict the use of states in our programs, we preserve this property.

Proposition 3.1 (Strong Confluence) *Let each interaction rule be free from state transformers. For a net N without states, if $N \Longrightarrow N_1$ and $N \Longrightarrow N_2$ with $N_1 \neq N_2$, then there is a net N_3 such that $N_1 \Longrightarrow N_3$ and $N_2 \Longrightarrow N_3$.*

Proof. Assume that $N \Longrightarrow N_1$ and $N \Longrightarrow N_2$ with $N_1 \neq N_2$ by using the rules r_1 and r_2 respectively.

In the reduction steps, active pairs to which the rules are applied do not overlap because there is at most one rule for each pair of agents. The result of applying a rule to an active pair can be uniquely determined because: the operations for the attributes are deterministic, and conditions in a conditional rewrite rule are all disjoint. In addition, connections of auxiliary ports are preserved during rewritings. Therefore, $N_1 \Longrightarrow N_3$ and $N_2 \Longrightarrow N_3$ by using rules r_2 and r_1 respectively. □

3.3 Evaluation Strategy

A net may contain more than one active pair, so reduction can proceed by alternative routes. In order to guarantee uniqueness of normal forms (in the sense that the same net will always give the same result), we give one simple reduction strategy which reduces a net to full normal form using a last-in-first-out (LIFO) order. We keep all active pairs in a stack and reduce them according to LIFO. We simulate this strategy using a big-step structured operational semantics:

$$\frac{\langle t \sim u \rangle \Downarrow \langle \Gamma \rangle}{\langle \Delta, t \sim u \rangle \Downarrow \langle \Delta, \Gamma \rangle}$$

where Δ, Γ are sequences of active pairs. Given a sequence of active pairs with the pair $t \sim u$ as the last in the sequence, we first reduce $t \sim u$. If the reduction of $t \sim u$ creates a sequence Γ of new active pairs, we append these to the end of the original sequence. We assume that the active pairs in Γ are created in a deterministic way: agents connected to the auxiliary ports of an active pair are *wired* to the corresponding rhs net in a linear order and the next connection that creates an active pair is added to the top/end of the sequence.

4 Representing interaction nets in C

Definition 4.1 Let $Loc \subseteq \mathbb{N}$ be a set of memory locations. We define a set of agent nodes $Agent = \{(Id \times P \times V)\}$ where $Id \in \Sigma$ is an identifier that represents the name of the agent, V is the set of values that an agent holds, and P is a set of ports. Each port $p \in P$ is a pair (l_a, n) where $l_a \in Loc$ is a pointer to another agent node, n is the port that the other node connects to this. The heap $H : Loc \rightarrow Agent$ returns a node $a \in Agent$ given some location $l \in Loc$.

We represent an agent node graphically using:

Id	p_0	p_1	$\cdots$	p_{ar}	v_1	$\cdots$	v_k

where ar is the arity of the agent. The port p_0 represents the principal port. It is straightforward to represent this memory model in the C language. We use the following C structure to represent *agent* nodes [1]:

```
typedef struct Agent{
 unsigned int Id;
 struct Port *port;
 union Val *val;
}Agent;
```

```
typedef struct Port{
 unsigned int portNum;
 unsigned int agent;
}Port;
```

```
typedef union Val{
 char char_value;
 int int_value;
 float float_value;
}Val;
```

We use a function `mkAgent` : $\Sigma \times \mathbb{N} \rightarrow Loc$ that given an Id and the arity will construct an agent node in the heap and return its location $l \in dom(H)$. We define two functions that manipulate agent nodes:

(i) `connect` : $Loc \times \mathbb{N} \times Loc \times \mathbb{N} \rightarrow$ Void that connects two agent ports. For example, if we have the agents:

$H(l_\alpha) =$

α	p_0	p_1	$\cdots$	p_n

$H(l_\beta) =$

β	p_0	p_1	$\cdots$	p_m

[1] there exists alternative representations which are more efficient in terms of memory.

then `connect`$(l_\alpha, 1, l_\beta, 0)$ will transform the structure of the nodes to:

$$H(l_\alpha) = \boxed{\alpha}\boxed{p_0}\boxed{(l_\beta, 0)}\boxed{\cdots}\boxed{p_n}$$

$$H(l_\beta) = \boxed{\beta}\boxed{(l_\alpha, 1)}\boxed{p_1}\boxed{\cdots}\boxed{p_m}$$

where $ar(\alpha) = n$, $ar(\beta) = m$. The updated nodes above represent the net $\alpha[\beta[s_1, \ldots, s_m], t_2, \ldots, t_n]$. The function `connect` updates the connection information in the ports of two agent nodes. We can represent this function using the following C macro definition:

```
#define connect(a1,p1,a2,p2)          \
   heap[a1].port[p1].agent   = a2;    \
   heap[a1].port[p1].portNum = p2;    \
   heap[a2].port[p2].agent   = a1;    \
   heap[a2].port[p2].portNum = p1;    \
   if(p1 == 0 && p2 == 0) pushActive(a1,a2)
```

The conditional statement checks if we are connecting principal ports and subsequently pushes the two (interacting) agents into a stack S of active pairs.

We can build nets using the instructions `mkAgent` and `connect` defined above. Below we give an example sequence of instructions that will construct the net `p`$\sim$`Add[S[Z],y]` in memory.

l_{Add} `= mkAgent (`Add`,2)`
l_S `= mkAgent (`S`,1)`
l_Z `= mkAgent (`Z`,0)`
`connect(`l_{Add}`,1,l_S,0)`
`connect(`l_S`,1,l_Z,0)`

Note that the ports which have no connections represent the interface of the net. The ports `p` and `y` are the free ports in the example net above. As another example, the cyclic net `x`$\sim$`B[x]` can be represented using:

l_B `= mkAgent` $(B, 1)$
`connect`$(l_B, 0, l_B, 1)$

(ii) The function `getPort` : $Loc \times \mathbb{N} \rightarrow Loc \times \mathbb{N}$ returns the connection information stored in a port of some agent. `getPort`$(l_a, n) = (l_b, m)$ where $l_a, n \in Loc \times \mathbb{N}$ and $l_b, m \in Loc \times \mathbb{N}$. As a consequence of the `connect` function, if `getPort`$(l_a, n) = (l_b, m)$ then `getPort`$(l_b, m) = (l_a, n)$. We can represent the function `getPort` using the following C macro definition:

```
#define getPort(a,p) (heap[a].port[p])
```

To represent a rule, we construct the rhs net of the rule and connect it to the auxiliary agents of the active pair. The rewiring is accomplished with the help of the function `getPort` which is used to *fetch* the auxiliary agents that will connect to the rhs net. As an example, the rule:

```
Add[x,y] >< S[a] => x~S[b],a~Add[b,y]
```

can be represented by the following sequence of instructions:

```
(l_x, p_x) = getPort(l_aAdd,1)
l_s = mkAgent(S,1)
connect(l_s,0,l_x,p_x)
(l_a, p_a) = getPort(l_aS,1)
l_Add = mkAgent(Add,2)
connect(l_Add,0,l_a,p_a)
connect(l_Add,1,l_s,1)
(l_y, p_y) = getPort(l_aAdd,2)
connect(l_Add,2,l_y,p_y)
```

where $l_{a_{Add}}$ and l_{a_S} are the locations of the active pair agents `Add` and `S` respectively.

5 Compilation

In this section we define the compilation schemes from our source language to C source code. We use existing C compilers to translate the generated C source files to native codes. When executed, the generated codes will build the corresponding net in memory and reduce it to full normal form.

The basic model is that we compile each rule and each net to a C function. The functions generated for rules take a pair of (active) agents as parameters. They contain code that will build the rhs net of a rule and wire it to the agents that are connected to the auxiliary ports of the active pair.

5.1 Runtime Environment

The compiled C files need to be linked with a run-time library. That library contains the internal INETS primitives, runtime error reporting routines and definition of the runtime data areas. The generated files contain a main method that calls for the initialisation of the three runtime data areas:

- the heap H. All agents are allocated in the heap.
- the evaluation stack S, contains pointers to active pairs. Evaluation pops a pointer to an active pair, evaluates the pair and pushes any newly created pairs into S. Thus, our default reduction strategy is based on a stack (LIFO).
- the *rule table* R maps pointers to functions generated for the rules. The evaluation function examines this table to select the appropriate function to reduce a given pair.

We define R in C using:

```
typedef void(*RuleFun)();
RuleFun R[MAX_RULES];
```

For simplicity, we assume a pre-defined constant `MAX_RULES` that gives the maximal number of rules in a given program. When we compile a rule to a C function, we create an entry of the function name in the table `R`. We shall see later that function

names are formed from an ordered concatenation of active pair names. Since there can only be one conditional interaction rule for any pair of agents, all function names that are generated for a rule are unique. The injective function *hash* takes a string (function name) and returns a unique integer i, $0 \leq i \leq$ `MAX_RULES`. The *hash* function ensures that each entry in `R` has a unique index.

The evaluation function *eval* reduces a given net to normal form. It pops an active pair (α, β) from S, examines the rule table R (by computing *hash* of the ordered concatenation of the active pair names (α, β)) to determine the appropriate function to invoke. The function is invoked with the arguments (α, β). Evaluation stops once S is empty.

5.2 Compilation schemes

Here, we present the compilation function $\mathcal{T}$ that will translate our source language into the intermediate language C. The environment Γ maps identifiers to memory locations $l \in Loc$. We write [] for the empty map and $\Gamma(x) = \perp$ when there is no entry for x in Γ. We use the notation: $\Gamma[x \mapsto s](z) = s$ (if $z = x$), or $\Gamma(z)$ otherwise. The function *fresh* returns a unique string and is used to generate fresh variables for our target language. The notation $\{x\}$ will replace the variable x with its actual value. For example, if $x =$ "*abc*", then "123$\{x\}$456" = "123*abc*456". The given value of x may be an integer. We write $\{p_1\} +$ "*str*" to concatenate the value of the string variable p_1 to the sequence of characters *str*.

The compilation of INETS into C is governed by the schemes: $\mathcal{T}_{inet}$ compiles a program, $\mathcal{T}_{exps}$ compiles expressions, $\mathcal{T}_{ns}$ compiles nets and $\mathcal{T}_{rs}$ compiles rules. We will now look at each of these schemes in turn.

The scheme $\mathcal{T}_{inet}$ calls for the compilation of an INETS program composed of a set of rules, nets and state declarations.

$$\mathcal{T}_{inet}(\Sigma, \langle e_1, \ldots, e_k \rangle, \langle n_1, \ldots, n_n \rangle, \mathcal{R}) = \begin{cases} \mathcal{T}_{exps}(e_1, \ldots e_k,);\ \mathcal{T}_{ns}(n_1, \ldots, n_n); \\ \mathcal{T}_{rs}(r_1); \ldots; \mathcal{T}_{rs}(r_n);\ \texttt{initRules} \end{cases}$$

where Σ is a set of symbols, $r_1, ..., r_n = \mathcal{R}$ are instances of rules, each e_i is a state declarations and each n_i is a net definition. The function `initRules` generates code that will fill the table R with function pointers.

$$\begin{aligned} \texttt{initRules} = \text{let} \\ & p_1 = \text{“}\texttt{R[}\text{”} + hash(\{\alpha_1\} + \{\beta_1\}) + \text{“}\texttt{] =}\text{”} + \{\alpha_1\} + \{\beta_1\} + \text{“}\texttt{;}\text{”}; \\ & \ldots \\ & p_n = \text{“}\texttt{R[}\text{”} + hash(\{\alpha_n\} + \{\beta_n\}) + \text{“}\texttt{] =}\text{”} + \{\alpha_n\} + \{\beta_n\} + \text{“}\texttt{;}\text{”}; \\ & \text{in “}\texttt{void initRules() \{}\text{”} + \{p_1\} + \ldots + \{p_n\} + \text{“}\texttt{\}}\text{” end} \end{aligned}$$

where α_i, β_i are the interacting agents for each rule $r_i \in \mathcal{R}$. The string $\{\alpha\} + \{\beta\}$ is an ordered concatenation of the active pair names.

The compilation scheme $\mathcal{T}_{ns}(n_1, \ldots, n_n)$ compiles a sequence of net definitions:

$$\mathcal{T}_{ns}(n_1, \ldots, n_n) = \mathcal{T}_n(n_1); \ldots; \mathcal{T}_n(n_n);$$

A net named `myNet` with formal parameters $x_1, \ldots, x_n$ and body P translates to a C function named `myNet`:

$$\mathcal{T}_n(\texttt{myNet}(x_1 : T_1, \ldots, x_n : T_n)\{xs, es\}) = \text{let } p_x = \mathcal{T}_{exps}(xs);\ p_e = \mathcal{T}_{eqs}(es, \emptyset);$$
$$\text{in } \text{“}\texttt{void myNet(}T_1\ x_1, \ldots, T_n\ x_n\texttt{)\{}\text{”} + \{p_x\} + \{p_e\} + \text{“}\texttt{\}}\text{”} \text{ end}$$

where the parameter x_i is of type T_i and the body of the net contains a list of expressions xs and/or a list of active pairs es. We use the scheme $\mathcal{T}_{exps}$ to translate the list of expressions and $\mathcal{T}_{eqs}$ to translate the list of active pairs. A C function named `main` will be generated: “`int main() {...}`”, which is the entry point for the execution of our generated program and allow the codes to run as stand alone programs. Our main function simply calls for the execution of the codes generated for the main net.

The scheme $\mathcal{T}_t$ emits codes that will construct a term in memory. Given a term T which is not a variable, the scheme $\mathcal{T}_t$ generates code that will: create the root agent, construct the sub-terms t_i that connect to the auxiliary port of the root agent, and finally connect the sub-terms to the appropriate auxiliary port. If t_i is an agent $\alpha \in \Sigma$, it will connect to its *parent* agent via its principal port (port 0).

$$\mathcal{T}_t(\alpha(e_1 : T_1, ..., e_m : T_m)[t_1, \ldots, t_n], l, p, \Gamma) = \text{let } a = \mathit{fresh};$$
$$pr_0 = \text{“}\texttt{int \{}a\texttt{\}= mkAgent("\{}\alpha\texttt{\}",}n\texttt{,}m\texttt{);}\text{”};$$
$$pr_v = \text{“}\texttt{heap[\{}a\texttt{\}].val[1].}T_1\texttt{=\{}e_1\texttt{\};}\text{”} + \cdots + \text{“}\texttt{heap[\{}a\texttt{\}].val[}m\texttt{].}T_m\texttt{=\{}e_m\texttt{\};}\text{”};$$
$$(pr_1, (l_1, p_1), \Gamma_1) = \mathcal{T}_t(t_1, a, 1, \Gamma); \ldots;$$
$$(pr_n, (l_n, p_n), \Gamma_n) = \mathcal{T}_t(t_n, a, n, \Gamma_{n-1});$$
$$\text{in } (\{pr_0\} + \{pr_v\} + \{pr_1\} + \text{“}\texttt{connect(\{}a\texttt{\},1,\{}l_1\texttt{\},\{}p_1\texttt{\});}\text{”} + \cdots +$$
$$\{pr_n\} + \text{“}\texttt{connect(\{}a\texttt{\},}n\texttt{,\{}l_n\texttt{\},\{}p_n\texttt{\});}\text{”}, a, 0, \Gamma_n) \text{ end}$$

where the expression e_i is of type T_i.

The compilation of a variable x does not generate any code. We consider two cases to compile variable nodes: 1) when $\Gamma(x) = \perp$, we create an entry $\Gamma[x \mapsto (l, p)]$ that maps the variable name x to the location and port number of the (*parent*) agent node that the variable wants to connect to. 2) when $\Gamma(x) = (l_2, p_2)$, we use the pair (l_2, p_2) to connect to the parent node of the variable. This scheme provides a mechanism to connect the ports of agents in a direct way other than through variable nodes.

$$\mathcal{T}_t(x, l, p, \Gamma) = \begin{cases} \text{if } (\Gamma(x) = \perp \vee x \notin \mathit{dom}(\Gamma)) \text{ then } (-, l, p, \Gamma[x \mapsto (l, p)]) \\ \text{else let } (l_x, p_x) = \Gamma(x) \text{ in } (-, l_x, p_x, \Gamma[x \mapsto \perp]) \text{ end} \end{cases}$$

We use the scheme $\mathcal{T}_{exps}$ to translate a list of expressions.

$$\mathcal{T}_{exps}(e_1, \ldots, e_n) = \mathcal{T}_{exp}(e_1); \ldots; \mathcal{T}_{exp}(e_n);$$

$$\mathcal{T}_{exp}(e1\ op\ e2) = \begin{cases} \text{let } r_1 = \mathcal{T}_{exp}(e1); \quad r_2 = \mathcal{T}_{exp}(e2); \text{ in } \{r_1\} + op + \{r_2\} \\ \text{where } op \in \{+, -, /, *, =, <, >, <=, >=, \%, !=, ==\} \text{ end} \end{cases}$$

$$\mathcal{T}_{exp}(\texttt{print}\ e_1, \ldots, e_n) =$$

$$\begin{cases} \text{let } r_1 = \mathcal{T}_{exp}(e_1); \ldots; r_n = \mathcal{T}_{exp}(e_n); \\ \text{in } \text{“}\texttt{printf(}F\texttt{,}\{r_1\}\texttt{);”} + \ldots + \text{“}\texttt{printf(}F\texttt{,}\{r_n\}\texttt{);”} \\ \quad \text{where } F = \text{“}\%d\text{”} \text{ if } \mathrm{r_i} \text{ is an integer, } F = \text{“}\%c\text{”} \text{ if } r_i \text{ is a character} \\ \text{end} \end{cases}$$

$$\mathcal{T}_{exp}(n) = \text{“}\{n\}\text{”} \text{ where } n \in \mathbb{Z} \text{ or } n \text{ is a variable name}$$

$$\mathcal{T}_{exp}(T\ n) = \text{“}\{T\}\ \{n\}\texttt{;”} \text{ where } T \in \{\texttt{int}, \texttt{char}, \texttt{bool}\}$$

The scheme $\mathcal{T}_{eqs}$ translates a list of active pairs. We use $\mathcal{T}_{eq}$ to generate the code for each active pair.

$$\mathcal{T}_{eqs}(u_1 \sim v_1, \ldots u_n \sim v_n, \Gamma) = \begin{cases} \text{let } (pr_1, l, p, \Gamma_1) = \mathcal{T}_{eq}(u_1 \sim v_1, 0, 0, \Gamma); \ldots; \\ \quad (pr_n, l, p, \Gamma_n) = \mathcal{T}_{eq}(u_n \sim v_n, 0, 0, \Gamma_{n-1}); \\ \text{in } (\{pr_1\} + \{pr_2\} + \ldots + \{pr_n\}, \Gamma_n) \text{ end} \end{cases}$$

For each active pair, we use the scheme $\mathcal{T}_{eq}$ to generate the code for each interacting agent, which is given in Figure 5.

$$\mathcal{T}_{eq}(\alpha(V_\alpha)[u_1, \ldots, u_n] \sim \beta(V_\beta)[v_1, \ldots, v_y], l, p, \Gamma) =$$

$$\text{let } (pr_1, (l_1, p_1), \Gamma_1) = \mathcal{T}_t(\alpha(V_\alpha)[u_1, \ldots, u_n], l, p, \Gamma);$$

$$(pr_2, (l_2, p_2), \Gamma_2) = \mathcal{T}_t(\beta(V_\beta)[v_1, \ldots, v_y], l, p, \Gamma_1);$$

$$\text{in } (\{pr_1\} + \{pr_2\} + \text{“}\texttt{connect(}\{l_1\}\texttt{,}\{p_1\}\texttt{,}\{l_2\}\texttt{,}\{p_2\}\texttt{); eval();”}, l, p, \Gamma_2) \text{ end}$$

$$\mathcal{T}_{eq}(x \sim \alpha(V_\alpha)[u_1, \ldots, u_n], l, p, \Gamma) =$$

$$\text{let } (pr_1, (l_1, p_1), \Gamma_1) = \mathcal{T}_t(\alpha(V_\alpha)[u_1, \ldots, u_n], l, p, \Gamma);$$

$$(pr_2, (l_2, p_2), \Gamma_2) = \mathcal{T}_t(x, l_1, p_1, \Gamma_1);$$

$$\text{in } (\{pr_1\} + \{pr_2\} + \text{“}\texttt{connect(}\{l_1\}\texttt{,}\{p_1\}\texttt{,}\{l_2\}\texttt{,}\{p_2\}\texttt{);”}, l, p, \Gamma_2) \text{ end}$$

Fig. 5. Compilation scheme $\mathcal{T}_{eq}$

The compilation scheme $\mathcal{T}_{rs}$ compiles a rule definition. We use the scheme $\mathcal{T}_r$ to generate a C function for each binary rule. This function contains code that will

build the rhs net and wire it to the corresponding auxiliary ports of the active pair.

$$\mathcal{T}_{rs}(\alpha(x_1 : T_1, \ldots, x_j : T_j)[t_1, \ldots, t_n] \;\texttt{><}\; e_1 \ldots, e_m \; \{r_1, \ldots, r_k\}) =$$

$$\begin{cases} \mathcal{T}_r(r_1, \alpha(x_1 : T_1, \ldots, x_j : T_j)[t_1, \ldots, t_n], e_1, \ldots, e_n); \\ \quad \vdots \\ \mathcal{T}_r(r_k, \alpha(x_1 : T_1, \ldots, x_j : T_j)[t_1, \ldots, t_n], e_1, \ldots, e_n); \end{cases}$$

where each r_i is either of the form

$$\beta(y_1 : T_{y_1}, \ldots, y_n : T_{y_n})[s_1, \ldots, s_n] \Rightarrow e_1, \ldots, e_k, eqs, If$$

or

$$\{e_1, \ldots, e_m, \beta(y_1 : T_{y_1}, \ldots, y_n : T_{y_n})[s_1, \ldots, s_n] \Rightarrow e_{m+1}, \ldots, e_n, eqs, If\},$$

$eqs = u_1 \sim v_1, \ldots, u_n \sim v_n,$

$If = \texttt{If}\ (b)\ eqs_1\ \texttt{else}\ If, eqs_2.$

In Figures 6 and 7 we gather together the remaining schemes that will generate the codes for a rule. We conclude this section with an example hand-compiled code generated for the main net given in the example program in Section 3.2:

```
void main(){
  counter = 0;
  int Fact = mkAgent("Fact",1,0);
  int Num = mkAgent("Num",0,1);
  heap[Num].val[1].int_value = 6;
  connect(Fact,0,Num,0);
  eval();
  printf("total number of interactions");
  printf("%d",counter);
}
```

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 can be downloaded from the project's web page [2]. The compiler reads a source program and outputs an executable file. 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 (INETS) with Amine

[2] http://www.interaction-nets.org/

$\mathcal{T}_r(\beta(y_1 : T_{y_1}, \ldots, y_k : T_{y_k})[u_1, \ldots, u_n] \Rightarrow e_1, \ldots, e_k,\ eqs,\ If,$

$\quad \alpha(x_1 : T_1, \ldots, x_l : T_l)[t_1, \ldots, t_m], e_{k+1}, \ldots, e_l) =$

let $a = fresh;\ b = fresh;$

$\quad pr_0 =$ "$\{T_{y_1}\}\ \{y_1\};$" $+ \ldots +$ "$\{T_{y_k}\}\ \{y_k\};$" $+$ "$\{T_1\}\ \{x_1\};$" $+ \ldots +$ "$\{T_l\}\ \{x_l\};$";

$\quad pr_1 = \mathcal{T}_{exps}(e_l, \ldots, e_k, \ldots, e_1);$

$\quad pr_2 = rewrite(\beta[u_1, \ldots, u_n], \alpha[t_1, \ldots, t_m],\ eqs,\ a, b, []);$

$\quad pr_3 = \mathcal{T}_{if}(If, \beta[u_1, \ldots, u_n], \alpha[t_1, \ldots, t_m], a, b);$

in "`void` $\{\alpha\} + \{\beta\}$`(int` $\{a\}$`, int` $\{b\}$`){`" $+ \{pr_0\} +$

$\quad$ "$\{y_1\}$ `= heap[{`b`}].val[1].`T_{y_1}`;`" $+ \ldots +$ "$\{y_k\}$ `= heap[{`b`}].val[`k`].`T_{y_k}`;`" $+$

$\quad$ "$\{x_1\}$ `= heap[{`a`}].val[1].`T_1`;`" $+ \ldots +$ "$\{x_l\}$ `= heap[{`a`}].val[`l`].`T_l`;`" $+$

$\quad \{pr_1\} + \{pr_2\} + \{pr_3\} +$ "`}`"

where $If =$ `if` $(b)\ eqs_1$ `else` If, eqs_2

$eqs = u_1 \sim v_1, \ldots, u_n \sim v_n$ end

$\mathcal{T}_r(\{e_1, \ldots, e_m\ , r, e_{m+1}, \ldots, e_n\}\alpha(x_1 : T_1, \ldots, x_n : T_n)[t_1, \ldots, t_m]) =$

$\quad \mathcal{T}_{exps}(e_1, \ldots, e_m);$

$\quad \mathcal{T}_r(r, \alpha(x_1 : T_1, \ldots, x_n : T_n)[t_1, \ldots, t_m]);$

$\quad \mathcal{T}_{exps}(e_{m+1}, \ldots, e_n);$

$\mathcal{T}_{if}($`if` $(b)\ eqs_1$ `else` $If, eqs_2, \alpha[t_1, \ldots, t_n], \beta[s_1, \ldots, s_n], a, b) =$

let $label = fresh;\ next = fresh;\ pr_1 = \mathcal{T}_{exp}(b);$

$\quad pr_2 = rewrite(\alpha[t_1, \ldots, t_n], \beta[s_1, \ldots, s_n], eqs_1, a, b, []);$

$\quad pr_3 = \mathcal{T}_{if}(If, \alpha[t_1, \ldots, t_n], \beta[s_1, \ldots, s_n], a, b);$

$\quad pr_4 = rewrite(\alpha[t_1, \ldots, t_n], \beta[s_1, \ldots, s_n], eqs_2, a, b, []);$

in "`if (!{`pr_1`}) goto {`$label$`};`" $+ \{pr_2\} +$ "`goto {`$next$`};`"

$\quad$ "$\{label\}$`:`" $+ \{pr_3\} + \{pr_4\} +$ "`goto {`$next$`};`" $+$ "$\{next\}$`:`"

end

Fig. 6. Compilation of Rules

$$
\begin{array}{l}
\mathit{rewrite}(\alpha[x_1,\ldots,x_k],\beta[y_1,\ldots,y_m],t_1 \sim s_1,\ldots,t_n \sim s_n,a,b,\Gamma) = \\
\quad \text{let } \mathcal{N} = \{x_1 \ldots, x_n\} \cup \{y_1,\ldots,y_m\}; \\
\qquad \Gamma[x_1 \mapsto (a,1),\ldots,x_n \mapsto (a,n), y_1 \mapsto (b,1),\ldots,y_m \mapsto (b,m)]; \\
\qquad (pr_{t_1},(l_{t_1},p_{t_1}),\Gamma_1) = \mathcal{T}_{t_r}(t_1,-,-,\Gamma,\mathcal{N}); \\
\qquad (pr_{s_1},(l_{t_n},p_{t_n}),\Gamma_2) = \mathcal{T}_{t_r}(s_1,-,-,\Gamma_1,\mathcal{N}); \\
\qquad\qquad \vdots \\
\qquad (pr_{t_n},(l_{s_1},p_{s_1}),\Gamma_{j+1}) = \mathcal{T}_{t_r}(t_n,-,-,\Gamma_j,\mathcal{N}); \\
\qquad (pr_{s_n},(l_{s_n},p_{s_n}),\Gamma_k) = \mathcal{T}_{t_r}(s_n,-,-,\Gamma_{k-1},\mathcal{N}); \\
\quad \text{in} \\
\qquad (\{pr_{t_1}\} + \{pr_{s_1}\} + \text{“}\texttt{connect(\{}l_{t_1}\texttt{\},\{}p_{t_1}\texttt{\},\{}l_{s_1}\texttt{\},\{}p_{s_2}\texttt{\});”} + \ldots + \\
\qquad \{pr_{t_n}\} + \{pr_{s_n}\} + \text{“}\texttt{connect(\{}l_{t_n}\texttt{\},\{}p_{t_n}\texttt{\},\{}l_{s_n}\texttt{\},\{}p_{s_n}\texttt{\});”}, l, p, \Gamma_2)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{T}_{t_r}(x,l,p,\Gamma,\mathcal{N}) = \text{let } (l_x,p_x) = \Gamma(x) \\
\quad pr_1 = \text{“}\texttt{Port p = getPort(}l_x,p_x\texttt{);”}; \; l_a = \text{“}\texttt{p.agent”}; \; p_l = \text{“}\texttt{p.portNum”}; \\
\text{in} \\
\quad \text{if } (\Gamma(x) = \bot \vee x \notin \mathit{dom}(\Gamma)) \text{ then } (-,l,p,\Gamma[x \mapsto (l,p)]) \\
\quad \text{else if } (x \in \mathcal{N}) \text{ then } (pr_1,l_a,p_l,\Gamma) \\
\quad \text{else } (-,l_x,p_x,\Gamma[x \mapsto \bot]) \\
\text{end}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{T}_{t_r}(\alpha[t_1,\ldots,t_n],l,p,\Gamma,\mathcal{N}) = \\
\text{let } a = \mathit{fresh}; \\
\quad pr_0 = \text{“}\{a\}\texttt{= mkAgent(\{}\alpha\texttt{\},}n\texttt{);”}; \\
\quad (pr_1,(l_1,p_1),\Gamma_1) = \mathcal{T}_{t_r}(t_1,a,1,\Gamma,\mathcal{N}); \ldots; \\
\quad (pr_n,(l_n,p_n),\Gamma_n) = \mathcal{T}_{t_r}(t_n,a,n,\Gamma_{n-1},\mathcal{N}); \\
\text{in} \\
\quad (\{pr_0\} + \{pr_1\} + \text{“}\texttt{connect(\{}a\texttt{\},1,\{}l_1\texttt{\},\{}p_1\texttt{\});”} + \ldots + \\
\quad \{pr_n\} + \text{“}\texttt{connect(\{}a\texttt{\},}n\texttt{,\{}l_n\texttt{\},\{}p_n\texttt{\})”}, a, 0, \Gamma_n)
\end{array}
$$

Fig. 7. Compilation of Rules, continued

[11]—an interaction net interpreter. The last column gives the number of interactions performed by both INETS 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 [9]. 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 $\texttt{A[S[S[S[S[S[S[S[Z]]]]]]],r]} \sim \texttt{S[S[S[Z]]]}$.

Program	INETS	Amine	Interactions
334II	4.5	23.8	6292779
245II	0.02	0.3	20211
A(3,8)	4	26	8360028
A(3,10)	66	340	134103148

We can see from the table that the ratio of the average number of interactions/sec of Pin to Amine is approximately 6 : 1. This performance is achieved partly because in our implementation, the *indirection rule* (see [2]) is performed at compile time. We represent interaction nets using undirected graph structure. In Amine, interaction nets are represented using tree data structures which introduces extra computational rules. Another major factor for this performance comes from the optimisations of the C code generator. C compilers have matured over many years and their optimisation techniques have advanced significantly. By using a C compiler to generate native codes, we benefit from the improvements of the compilers code generator. We have identified various optimisations (source to source optimisations, code and memory optimisations, etc.) for our source language and our compiler. With these optimisations in place, we anticipate to obtain better results.

7 Conclusions

In this paper we have presented our language for interaction nets, and given a compilation into C. We have implemented this language, which is available from the

web page: http://www.interaction-nets.org/, and is one of the main building blocks for building a programming environment for interaction nets. Current work is focussed on giving a formal operational semantics of this language, and also building a richer set of programming tools.

References

[1] D. Diaz. Wamcc: Compiling prolog to c. In *In 12th International Conference on Logic Programming*, pages 317–331. MIT PRess, 1995.

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

[3] 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.

[4] A. Hassan, I. Mackie, and S. Sato. Interaction nets: programming language design and implementation. In *Proceedings of the seventh international workshop on Graph Transformation and Visual Modeling Techniques*, March 2008.

[5] M. J.B. Almeida, J.S.Pinto. A tool for programming with interaction nets. Technical report, University of Minho, 2006.

[6] 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.

[7] 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.

[8] 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.

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

[10] B. C. Pierce and D. N. Turner. Pict: a programming language based on the pi-calculus. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction*, pages 455–494. The MIT Press, 2000.

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

[12] 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.

[13] D. M. Ritchie. The c programming language, 1988.

[14] D. Tarditi, P. Lee, and A. Acharya. No assembly required: Compiling Standard ML to C. Technical report, ACM Letters on Programming Languages and Systems, 1990.

[15] G. Wong. Compiling erlang via C. Technical report, 1998.