A Simple Parallel Implementation of Interaction Nets in Haskell

Wolfram Kahl
Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada
kahl@cas.mcmaster.ca

Due to their "inherent parallelism", interaction nets have since their introduction been considered as an attractive implementation mechanism for functional programming. We show that a simple highly-concurrent implementation in Haskell can achieve promising speed-ups on multiple cores.

1 Introduction

The *interaction nets* introduced by Lafont [Laf90] can be considered as a variant of term graphs, and therewith as a kind of graphs used as representation of terms. Interaction nets are equipped with an "inherently parallel" local and confluent reduction mechanism that makes them an, at least conceptually, attractive target for (functional) programming language implementation. However, to date there have been only limited experiments with parallel implementations of interaction nets, and no easily-usable parallel implementation is publicly available. In addition, the nature of the parallelism of interaction net reduction is in general rather fine-grained, so that the question of distribution strategies arises naturally.

In this paper, we report on an experiment that bypasses the question of distribution strategies, and instead investigates whether a fine-grained threading mechanism with parallel execution on shared-memory multi-core systems, as provided by the run-time system of the Glasgow Haskell Compiler (GHC), can already realise the potential of parallelisation offered by interaction nets. Our implementation is publicly available (at `http://www.cas.mcmaster.ca/~kahl/Haskell/HINet/`) and accepts a slightly restricted version of the Inets file format, enabling further experiments also by other interaction net researchers. In the benchmarking section, we provide a lot of data, and also discuss the potential pitfalls of benchmarking Haskell programs with large heap requirements, in order to aid potential users of our system to avoid these pitfalls.

1.1 From Term Graphs via Jungles and Code Graphs to Interaction Nets

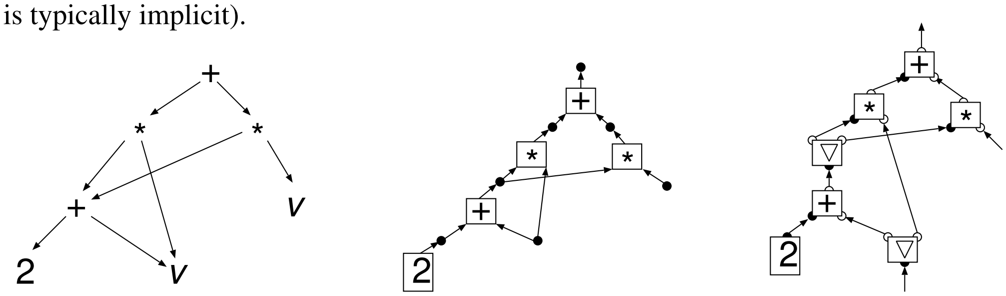
We now give an introduction to interaction nets that puts them into the context of different term graph representations. We do this for two reasons: First, to make interaction nets more accessible for readers interested in functional programming language implementation, who may already be familiar with graph reduction, but might find the principal-port orientation of most of the interaction net literature rather obscure, and second, to give a clear understanding of polarities, which have almost disappeared from the interaction net literature.

Conventional term graphs (see e.g. [KKSV93]) are node-labelled directed graphs, where each node has a sequence of outgoing edge the length of which is determined (or sometimes part of) the label. Node labels of these term graphs correspond to function symbols in terms; variables do not need labels: Different variable nodes (labelled "V" below) represent different variables.

The "jungle" approach of Hoffmann and Plump [HP91] moves the function symbols into hyperedges, with a sequence of "argument tentacles" (or "input tentacles") extending to argument nodes, and (normally) exactly one "result tentacle" (or "output tentacle") extending to the hyperedge's result node (or

U. Dal Lago and R. Harmer (Eds.): Developments
in Computational Models 2014 (DCM 2014).
EPTCS 179, 2015, pp. 33–47, doi:10.4204/EPTCS.179.3

output node). In both approaches, there is no restriction on the number of edges (resp. input tentacles) incoming into each node; multiple incoming edges implement *sharing* (and zero incoming edges into a non-root node implement (uncollected) "garbage", where in term graph and jungle rewriting, garbage collection is typically implicit).



Term Graph Jungle Interaction Net

The drawing above shows a conventional term graph, a jungle, and an interaction net each representing the term $(2+x)*x+(2+x)*y$ with the same degree of sharing. In all three drawings, the sequence of the outgoing or incoming edges, respectively tentacles, or ports, of each node or hyperedge is part of the structure, but is, as customary, not made more explicit.

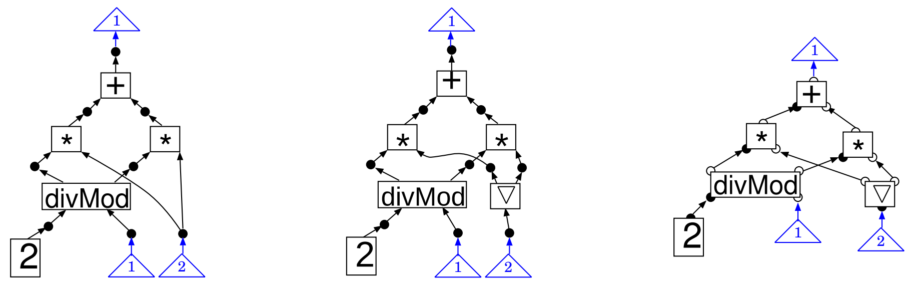
Interaction nets are different from jungles in several ways. First of all, a different terminology is used: Instead of "hyperedge", the terms "node" or "agent" are used, the nodes of jungles turn into "connections" and the tentacle labels and directions turn into "ports". In interaction nets, connections must be incident with exactly one or two ports; those incident with only one port make up the interface of the net. Because of this, sharing and garbage must be made explicit via duplicator ("∇") and terminator ("!") nodes. Each interaction net node label determines one *principal port* for its nodes. We draw principal ports as filled-in circles attached to the rectangular nodes, while auxiliary ports are hollow. Interaction net rules only replace pairs of nodes connected via their principal ports.

The directions of edges in termgraphs, and of tentacles in jungles, are motivated by denotational semantics; the corresponding directions of connections in interaction nets were introduced under the name *polarities* by Lafont [Laf90], but are omitted in a large part of the interaction net literature, where interaction nets are drawn with undirected connections. Instead, the operationally motivated direction of nodes ("actors") from auxiliary ports to the principal port is typically emphasised. We follow Lafont [Laf90] to distinguish output ports (with positive *polarity*) and input ports (negative polarity), and draw connections as directed arrows from output to input ports. Note that besides Lafont [Laf90], most of the interaction net literature does *not* draw nets in a way that easily corresponds to a jungle reading.

Whereas jungle hyperedges have only one output tentacle, the duplicator (∇) nodes of the interaction net above have two output ports — a feature that also occurs in the *code graphs* of [KAC06, AK09]. We illustrate this with a second example; the term $(2/x)*y+(2\%x)*y$ represented with sharing as a term graph has two variable nodes corresponding to x and y; represented as jungle these turn into two input nodes.

Term Graph Jungle

In code graphs, the sequence of these input nodes is explicitly visualised via triangular tags with arrows towards the input nodes; code graphs also have a sequence of output nodes visualised via triangular tags with arrows from the output nodes. Code graph hyperedges also have as interface a sequence of input nodes (as in jungles) and a sequence of output nodes, which in contrast to jungles is not constrained to contain exactly one element. For the sake of an example, we can therefore use a two-output operation “divMod” to obtain a code graph that uses a single operation to produce the same result as the two separate operations / and % in the term and jungle above. (The sequences of input and output nodes of hyperedges are still indicated implicitly via the graphical arrangement.)



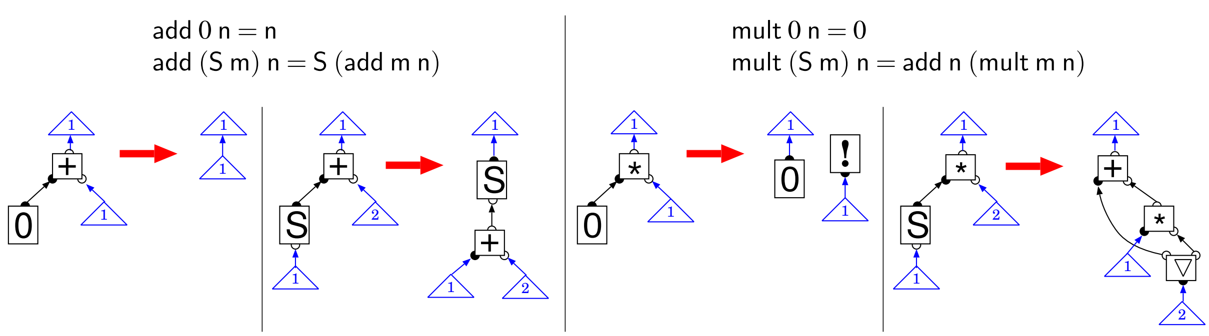
Code Graph   Code Graph with Duplicator   Interaction Net

Since code graphs allow multi-output nodes, duplicators (“∇”) do not need to be given any special status, and interaction net languages can be understood as code graph languages without node-based sharing (and without “garbage”), which allows us to replace the code graph nodes with their single incoming and outgoing tentacles with simple connections. Input and output nodes of code graphs turn into input and output ports of interaction nets — these are the ports of negative, respectively positive polarity that have no connection attached to them. As for code graphs, we will assume the input and output ports to be organised into two sequences, and tag them using the same triangles.

1.2 Interaction Net Rules and Reduction

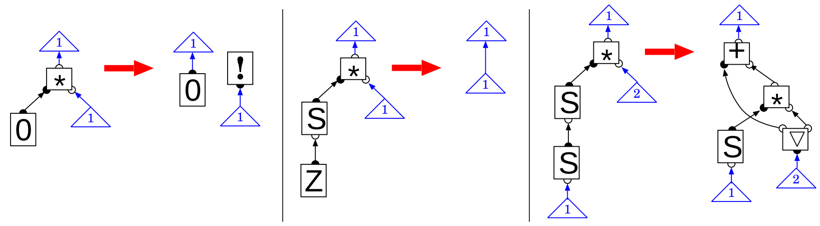
Application of rules is defined as subnet replacement, where the input and output ports of the rule sides may map to arbitrary ports in the application net. Due to the constraints on the left-hand sides of rules, the resulting reduction has no critical pairs; it is therefore confluent and has a deterministic normalisation relation. Since left-hand sides match only to subnets induced by two nodes connected via their principal ports, reduction exhibits extreme locality, and is frequently considered as “inherently parallel”.

Below, we show rules for addition and multiplication of natural numbers built up from the constructors for zero (“0”) and successor (“S”). The first multiplication rule, “mult 0 n = 0”, turns n into “garbage” by attaching a terminator (“!”) node; the second multiplication rule “duplicates” n for use both by the addition and by the recursive call.



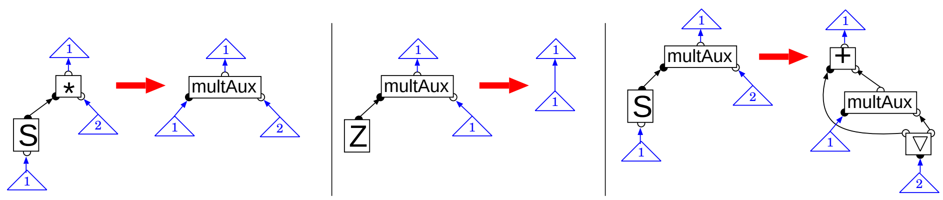
The reader may notice that the multiplication rules provided above always perform a "superfluous" last addition to zero if the first factor is non-zero. One might consider the following starting point instead:

```
mult 0 n = 0
mult (S 0) n = n
mult (S m) n = add n (mult m n)
```



However, the "deep pattern matching" here cannot be implemented directly by conventional interaction net reduction; the rules drawn above to the right are however allowed in the extension proposed by Hassan *et al.* [HJS09] which translates them into conventional interaction net rules by adding an auxiliary function:

```
mult 0 n = 0
mult (S m) n
   = multAux m n
multAux 0 n = n
multAux (S m) n
   = add n (multAux m n)
```



Such encoding issues are not relevant to the current study, which considers interaction nets as an execution model, rather than as a programming language. Compilation to interaction net rules is a separate topic, and has been studied for example by H. Cirstea and others [CFF$^+$07] using the ρ-calculus as intermediate language.

1.3 Related Work

Pedicini and Quaglia [PQ07] describe PELCR, a distributed parallel environment for optimal λ-calculus reduction, which uses a specialised fixed interaction net language and implements sophisticated distribution strategies. (I found no trace of this being or having been publicly available.) Besides such specialised systems, we are aware of only a small number of parallel implementations of interaction nets, in particular [BP97, Pin01, Jir14]. Of all these, only the last seems to be (still) available; it is an experimental GPU implementation that requires new rules to be implemented manually in C/CUDA at a very low level.

A general interaction net implementation that is still available is part of the Inets project of Mackie *et al.* [HMS09, HJ12]. This it is a compiler for the interaction net definition language Inets, which is considered as a programming language; the compiler is implemented in Java, and compiles via C to non-parallel executables. While Inets implements nets as pointer structures, the (apparently unavailable) successor system "Light" [HMS10], as well as the systems of Pinto [Pin01] and Jiresch [Jir14] are based on a term representation of interaction nets (based on the fact already pointed out by Lafont [Laf90] that "well-behaved" fully reduced nets always can be represented via pairs of terms with common variables and further constraints). Lippi's implementation called "in^2" [Lip02] was apparently close in spirit, but not directly based on terms.

Other available implementations are geared more towards graphical interaction directly with interaction nets (and also don't support parallel execution), including de Falco's "Interaction Nets Laboratory" [Fal06], the "interaction net IDE" INblobs of Almeida *et al.* [APV08], and the graph rewriting system IDE "PORGY" [AFK$^+$11] which can also be used for interaction nets. By emphasising visualisation of net transformations, these tools by design cannot target efficient parallel implementation.

1.4 Contribution and Overview

We present a design for highly concurrent interaction net implementations that is at the same time surprisingly simple and very close to the graph understanding of the interaction net definition. The parallel implementation of concurrency in the Glasgow Haskell Compiler (GHC) is a good fit for this kind of design; our implementation obtains satisfactory speed-ups even for simple examples.

While most current non-graphical implementations of interaction nets are based on a term-based calculus, we explain our more direct approach in Sect. 2. The actual (literate) Haskell source code of the kernel of our implementation is then presented in Sect. 3 — the full source code is available on-line at `http://www.cas.mcmaster.ca/~kahl/Haskell/HINet/`. In Sect. 4 we summarise our implementation of a language similar to that of Inets [HJ12]. Measurements and relevant observations are in Sect. 5.

2 Implementation Design

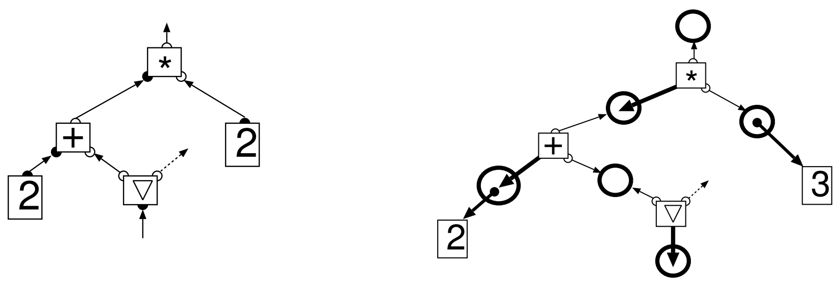
Our implementation essentially follows the main ideas of Banach and Papadopoulos [BP97]:

- Two-way connections, which easily introduce opportunities for deadlock and race conditions, can be avoided by using polarities to direct the connections between ports (which, in a large part of the literature, are treated as undirected, and implemented as two-way connections).
- These *directed* connections hold mutable state.
- The connection with the principal port of a constructor does not need to be known to the constructor node if the connection state refers to the node.

The following main decisions then determine most of our implementation details:

- Connections (drawn below as thick circles) are initially “empty”, and each node has references to the connections attached to its auxiliary ports.
- Attaching the principal port of a constructor to a connection deposits a reference to the constructor node in the connection (which is then “full”). (This reference is drawn below with a thick arrow with a bullet tail.)
- Attaching the principal port of a function to a connection starts a concurrent thread that waits for a constructor reference in that connection, and if/when it finds one, starts the corresponding rule application. (This is drawn below with an even thicker arrow ending inside the connection.)

The following shows a net fragment first in the same style as the previous example, and to the right with implementation details added.



3 Implementation in Concurrent Haskell

We implement connections using the Concurrent Haskell synchronisation primitive MVar, which can be created empty; putMVar waits for empty state to fill, and takeMVar waits for full state to empty [PJGF96]. The GHC version of Concurrent Haskell has an extremely light-weight thread implementation that makes it feasible to create millions of threads; we therefore directly create new threads for functions as mentioned above, and even smaller threads for short-circuiting two interface ports that are directly connected by rule applications: These threads only wait for a constructor on the originally negative port of the LHS, and copy it to the positive side.

The run-time implementation of nets, based on MVars, is introduced in Sect. 3.2. For the static representation of rules, our implementation uses a non mutable datatype NetDescription to represent right-hand sides (RHSs) of reduction rules; these are introduced in Sect. 3.3. At run-time, these NetDescriptions are instantiated into new parts of the mutable run-time net, as fully defined in Sect. 3.4 following the principles outlined in Sect. 2.

3.1 Polarity

Lafont [Laf90] and Banach and Papadopoulos [BP97] use typed connections in their interaction nets, where the two ports incident in a connection have the same type, but different polarity. Since we design our interaction net implementation as a run-time system, types are currently not important, and will be assumed to have been taken care of before net generation. Polarity, however, drives several run-time decisions; for the sake of readability, we define a special-purpose data-type for it (and let Haskell's "**deriving**" mechanism provide us with the default implementation of equality and ordering tests, and of conversion to strings):

```
data Polarity = Neg | Pos
  deriving (Eq, Ord, Show)
opposite :: Polarity → Polarity
opposite Neg = Pos
opposite Pos = Neg
```

We will follow Lafont's convention of letting "constructors" have positive polarity, and "functions" negative polarity.

3.2 Mutable Net Representation

A connection between two ports is implemented as a single MVar that is either empty, or contains the constructor node for which the connection is at the principal port. (To allow different node label types to be used, we use the type variable nLab throughout.)

```
type Conn nLab = MVar (Node nLab)
```

For an auxiliary port of a node, besides its connection we also record the port's polarity to make it available efficiently at run-time. (In Haskell, data constructors for simple record types habitually are given the same name as the type constructor; the fields pol and conn here are declared strict using "!", and the "UNPACK" pragma declares an "unpacking" optimisation as desired to the compiler.)

```
data Port nLab = Port
  {pol  ::                  ! Polarity
```

```
  , conn ::  {-# UNPACK #-}  ! (Conn nLab)
  }
```

We introduce the type synonym Ports to abbreviate the type of port arrays.

```
type Ports nLab = Vector (Port nLab)
```

Given a port p, the port at the other end of its connection is obtained as opPort p by flipping the polarity:

```
opPort :: Port nLab → Port nLab
opPort p = p { pol = opposite $ pol p }
```

A node contains a label, and the array of its *non-principal* ports. We do not include the principal port in ports since

- the principal port of a constructor is connected to the MVar pointing back to the constructor, and
- the principal port of a function is connected to the MVar the function's thread is waiting on.

```
data Node nLab = Node
  { label :: nLab
  , ports :: Ports nLab
  }
```

3.3 Net Descriptions

Whereas in Sect. 3.2, we introduced types for nets considered as run-time states, here we introduce *net description* for static representation of, in particular, rule right-hand sides.

The following types are dictated by our current choice of array implementation (Data.Vector from the vector package, for efficiency), but aliased for readability:

```
type PI = Int   -- "port index"
type NI = Int   -- "node index"
```

The port index type PI will be used also in actual nets, while the node index type NI is needed only for right-hand side nodes in descriptions and during creation. We arbitrarily call the two nodes engaged in an interaction "source" and "target"; the "source" interface consists of the auxiliary ports of the node with the "function" label with negative principal port, and the "target" interface consists of the auxiliary ports of the "constructor" node with positive principal port. The following data type serves to identify all ports in a rule's right-hand side (the "!" specifies strict constructor argument positions for efficiency):

```
data PortTargetDescription
  = SourcePort   ! PI
  | InternalPort ! NI ! PI   -- node, port
  | TargetPort   ! PI     deriving (Eq, Ord, Show)
```

Therefore, each RHS node is described by its label and by the connections of *all* its ports:

```
data NodeDescription nLab = NodeDescription
  { nLab             ::                 ! nLab
  , portDescriptions ::  {-# UNPACK #-}  ! (Vector PortTargetDescription)
  }
```

A NetDescription is intended as description of the RHS of interaction rules:

```
data NetDescription nLab = NetDescription
  {source :: {-# UNPACK #-} !(Vector PortTargetDescription)
  ,target :: {-# UNPACK #-} !(Vector PortTargetDescription)
  ,nodes  :: {-# UNPACK #-} !(Vector (NodeDescription nLab))
  }
```

A *language* for interaction nets consists of a type of node labels together with arity and polarity information defining *all* ports for each node label, and for any “function” node label f and any “constructor” node label c that can occur as “argument” to f a rule, specified by a right-hand side ruleRHS f c, which needs to be a net description having a source compatible with the *auxiliary* ports of f, and a target compatible with the auxiliary ports of c.

```
data INetLang nLab = INetLang {polarity :: !(nLab → Vector Polarity)
                              ,ruleRHS :: !(nLab → nLab → NetDescription nLab)
                              }
```

3.4 Interaction Net Reduction

The main purpose of the function replaceNet is to implement the instantiation part of the rule application step. It is a separate function because it also serves the secondary purpose of constructing the start net.

The function replaceNet takes as arguments a NetDescription (defined in Sect. 3.3) for the rule’s RHS, and arrays src and trg containing the non-principal connections of the two nodes of the image of rule’s LHS in the mutable net representation (Sect. 3.2) of the run-time state.

The mdo is a “recursive do” as introduced by [EL02], and the use here essentially corresponds to the imperative programming pattern of allocating an array of uninitialised cells, and creating references to the array cells possibly before initialising them. (Functions prefix with “V.” operate on Vectors.)

```
replaceNet :: forall nLab ∘ INetLang nLab → NetDescription nLab
           → Ports nLab → Ports nLab → IO ()
replaceNet lang descr src trg = mdo
  nps ← let mkNode (NodeDescription lab pds) = do
                 ps ← V.zipWithM mkPort (polarity lang lab) pds
                 return (Node {label = lab, ports = V.tail ps}
                        ,V.head ps
                        )
              where mkPort Pos (InternalPort _ _) = fmap (Port Pos) newEmptyMVar
                    mkPort _   ptd                = return (portTarget ptd)
         in V.mapM mkNode (nodes descr)
```

The first step above creates descr image nodes, taking over interface ports from src and trg, creating new internal connections at positive ports, and lazily connecting negative ports with internal connections located via the function portTarget defined below.

Note that the prose explanations here are interspersed within the scope of the mdo above, since all code before the definition of reduce below remains indented below the mdo.

```
  let portTarget :: PortTargetDescription → Port nLab
      portTarget (SourcePort i) = atErr "portTarget: SourcePort S" src (pred i)
```

```
portTarget (TargetPort i) = atErr "portTarget: TargetPort S" trg (pred i)
portTarget (InternalPort n i) = let e = "portTarget: InternalPort "
                                    (n′, pp) = atErr e nps n
  in opPort (if i ≡ 0 then pp else atErr (e ++ shows n " S") (ports n′) (pred i))
```

We traverse the newly created nodes and "connect" their principal ports.

```
let doNode (n@(Node lab prts), Port pl c) = case pl of
      Neg → forkIO (reduce lang (ruleRHS lang lab) c prts) ≫ return ()
      Pos → putMVar c n
  in V.mapM_ doNode nps
```

For source and target ports, we only need to take care of short-circuits:

```
let doIfacePort (Port Pos c) ptd = return ()    -- will be done from the other side if necessary
    doIfacePort (Port Neg c) ptd = let           -- original port of the LHS node
      Port _pl′ c′ = portTarget ptd              -- connecting port in image of RHS
      in if c ≡ c′    then return ()             -- empty cycle
                      else case ptd of
                        InternalPort n i′ → return ()   -- already dealt with
                        _                 → do forkIO (moveMVar c c′)
                                               return ()
  in do V.zipWithM_ doIfacePort src $ source descr
        V.zipWithM_ doIfacePort trg $ target descr
```

Whenever a function node is created, i.e., a node with positive principal port, a reduce thread is started (via forkIO). This thread waits on the connection (pconn) between the principal ports of the rule until this contains the constructor node (the principal port of which has positive polarity). The array src contains the auxiliary ports of the function node (the principal port of which has negative polarity).

```
reduce :: INetLang nLab → (nLab → NetDescription nLab) → Conn nLab → Ports nLab → IO ()
reduce lang rules pconn src = do
  Node clab trg ← takeMVar pconn
  replaceNet lang (rules clab) src trg
```

4 Reading .inet Files

The Inets project led by Ian Mackie has implemented the only publicly available general implementation of interaction nets, the compiler [HJ12] for the interaction net programming language "Inets". This language was introduced by Mackie [Mac05], with the core of the Inets implementation described in [HMS09].

We implemented a front-end to our interaction net reduction system for the core sublanguage of Inets, leaving out in particular the extension of nested pattern matching described in [HMS10], and generic rules and variadic agents.

Since our system depends on polarity for its directed implementation of connections, but Inets has no concept of polarity, we adopted the convention that the first-mentioned agent of each rule has negative principal port (that is, is considered as a function), and the second agent has positive principal port (constructor). This convention is adopted in most of the Inets examples anyways; only two rules in fibonacci.inet had been written the other way around. From this starting point we attempt to deduce the polarities of all other ports; for the examples accessible to us so far, we only needed to add a single additional heuristic: A function for which all other ports except one are known to have negative polarity is assumed to have positive polarity on the last port. (Unfortunately the λ-calculus evaluator yale.inet [Mac98] is defined in a way that does not allow a consistent assignment of polarities.)

Inets supports "parameters", that is, agent attributes of the primitive types int, bool, float, char, and String. The description in [HMS09] suggests that only a single parameter is allowed per agent; our implementation allows arbitrary numbers, but expects the number and types of attributes to be determined by the agent label. We also interpret type int as Haskell's arbitrary-precision Integer type. Our current interpreting implementation uses a parameterised agent label type:

```
data NLab arg = NLab { nLabName :: Name, nLabAttrs :: [arg] }
```

When reading a .inet file, the nets on the rule RHSs are translated into NetDescription (NLab Expression) and stored in a finite map for lookup by the rule LHS agent label pair; in the run-time net, agent labels of type NLab Value are used, and the variable bindings induced by the attributes of the interacting nodes are used at the time of rule application to evaluate the expressions in the RHSs (and the condition expressions for the conditional structure of Inets RHSs).

Inets modules can contain global variables, which are used in the examples to implement reduction counts etc.; since in a parallel implementation such global variables would require synchronisation (and thus would destroy the independence of parallel reduction), we did not implement any feature related to global variables.

5 Benchmarks

For our first examples, we use a cascading recursion for calculating Fibonacci numbers, and the Ackermann function, both computing with unary natural numbers constructed from zero Z and the unary successor constructor S:

```
fib 0 = 0
fib (S n) = fibAux n
fibAux 0 = 1
fibAux (S n) = fib n + fibAux n
```

```
ack 0 n = S n
ack (S m) n = ackAux m n
ackAux m 0 = ack m 1
ackAux m (S n) = ack m (ack (S m) n)
```

These rules were directly encoded using NetDescriptions (see Sect. 3.3); we will refer to these implementations now as fibND and ackND.

We timed the actual code of Sect. 3 on a six-core 2.8GHz Phenom 2 with 16GB main memory; our implementation achieved the timings in Table 1, where the GHC run-time system is instructed by "$-$Nk" to use k cores for parallel processing. The user-space time of a Haskell process is divided into "mutation" time and garbage collection time. The run-time system can be made to report these times and further information; in Tables 1 and 3 we include, after the elapsed time for each process (which is the "real" time as reported by "`time`" BASH built-in), the "allocation rate", which measures how many megabytes are allocated on the Haskell heap per second of mutation time, and the "productivity", which is the result of dividing the mutation time by the elapsed time. For example, a productivity of 240% for a three-core ("-N3") run means that each core spent on average 20% of its time on garbage collection, since $240\% + 3 \times 20\% = 300\%$. The last column in each of the groups for "-N2" to "-N6" contains the speedup over single-core execution.

By default, the GHC run-time system starts execution with a small heap and grows it by relatively small increments on demand; we indicate use of this this default setting by "**dft.**" in the third column (“heap”). Where a size is specified in this column, this size was given to the run-time system as fixed heaps size (with options -H *and* -M).

expr.	result	heap	-N1			-N2				-N3				-N4				-N5				-N6			
			time (s)	allocation rate (MB per mutation second)	productivity (% of elapsed)	time (s)	allocation rate	productivity	**speedup**	time (s)	allocation rate	productivity	**speedup**	time (s)	allocation rate	productivity	**speedup**	time (s)	allocation rate	productivity	**speedup**	time (s)	allocation rate	productivity	**speedup**
ackND 3 6	509	**dft.**	1.078	1699	48	0.630	1189	118	**1.71**	0.483	952	192	**2.23**	0.431	779	264	**2.50**	0.427	611	340	**2.52**	0.426	506	411	**2.53**
ackND 3 6	509	2M	1.004	1716	51	0.633	1188	118	**1.59**	0.489	960	189	**2.05**	0.452	738	266	**2.22**	0.411	647	334	**2.44**	0.421	516	409	**2.38**
ackND 3 6	509	3M	0.800	1693	65	0.568	1196	130	**1.41**	0.478	960	193	**1.67**	0.445	741	269	**1.80**	0.409	652	333	**1.96**	0.429	509	405	**1.86**
ackND 3 6	509	4M	0.694	1678	76	0.504	1202	146	**1.38**	0.444	942	212	**1.56**	0.433	747	274	**1.60**	0.405	646	339	**1.71**	0.425	512	407	**1.63**
ackND 3 6	509	5M	0.652	1652	82	0.475	1194	156	**1.37**	0.415	948	225	**1.57**	0.404	751	293	**1.61**	0.385	652	354	**1.69**	0.422	510	413	**1.55**
ackND 3 6	509	6M	0.647	1604	85.3	0.462	1181	163	**1.40**	0.400	945	235	**1.62**	0.387	745	308	**1.67**	0.375	647	366	**1.73**	0.395	522	431	**1.64**
ackND 3 6	509	7M	0.644	1575	87	0.459	1159	167	**1.40**	0.389	943	242	**1.66**	0.382	739	315	**1.69**	**0.361**	652	377	**1.78**	0.395	521	431	**1.63**
ackND 3 6	509	8M	0.659	1521	88	0.472	1108	170	**1.40**	0.396	918	244	**1.66**	0.384	727	319	**1.72**	0.363	640	383	**1.81**	0.388	511	447	**1.69**
ackND 3 6	509	9M	0.675	1469	89	0.482	1070	172	**1.40**	0.411	873	248	**1.64**	0.393	705	321	**1.72**	0.374	615	386	**1.80**	0.392	501	452	**1.72**
ackND 3 6	509	10M	0.686	1437	90	0.485	1061	172	**1.41**	0.420	846	250	**1.63**	0.404	678	324	**1.70**	0.379	600	391	**1.81**	0.399	489	456	**1.72**
ackND 3 6	509	0.1G	0.749	1305	92	0.522	982	175	**1.43**	0.445	796	253	**1.68**	0.430	635	329	**1.74**	0.416	544	397	**1.80**	0.435	452	458	**1.72**
ackND 3 7	1021	**dft.**	5.866	1676	36	3.177	1185	94	**1.85**	2.287	982	158	**2.56**	1.990	802	223	**2.95**	1.845	642	300	**3.18**	1.771	547	367	**3.31**
ackND 3 7	1021	6M	3.335	1585	67	2.288	1181	131	**1.46**	1.877	979	193	**1.78**	1.815	764	256	**1.84**	1.661	681	314	**2.01**	1.728	542	380	**1.93**
ackND 3 7	1021	8M	3.024	1514	78	2.115	1133	148	**1.43**	1.723	953	217	**1.76**	1.666	759	281	**1.82**	1.521	683	342	**1.99**	1.591	551	406	**1.90**
ackND 3 7	1021	9M	3.000	1474	80	2.076	1122	152	**1.45**	1.715	930	223	**1.75**	1.651	743	290	**1.82**	**1.496**	670	355	**2.01**	1.545	547	422	**1.94**
ackND 3 7	1021	10M	3.010	1438	82	2.107	1078	156	**1.43**	1.735	902	227	**1.74**	1.638	729	298	**1.84**	1.520	649	361	**1.98**	1.579	531	424	**1.90**
ackND 3 7	1021	20M	2.932	1377	88	2.022	1035	170	**1.45**	1.712	848	245	**1.71**	1.637	682	319	**1.79**	1.539	592	390	**1.91**	1.579	490	460	**1.86**
ackND 3 7	1021	1G	3.508	1187	91	2.472	881	171	**1.41**	2.085	727	246	**1.68**	1.971	590	319	**1.78**	1.835	527	384	**1.91**	1.840	448	449	**1.90**
ackND 3 8	2045	**dft.**	30.034	1557	30	16.857	1138	74	**1.78**	11.412	956	131	**2.63**	9.640	791	187	**3.12**	8.597	646	256	**3.49**	8.061	550	322	**3.73**
ackND 3 8	2045	10M	17.000	1423	59	11.116	1065	120	**1.53**	8.802	899	180	**1.93**	8.195	727	239	**2.07**	7.320	657	296	**2.32**	7.306	540	361	**2.33**
ackND 3 8	2045	40M	13.089	1248	87	9.171	929	167	**1.43**	7.450	789	243	**1.76**	7.079	633	318	**1.85**	6.503	564	389	**2.01**	6.539	473	461	**2.00**
ackND 3 8	2045	60M	13.057	1228	89	9.019	923	171	**1.45**	7.373	778	248	**1.77**	6.929	634	324	**1.88**	6.372	565	396	**2.05**	6.375	475	471	**2.05**
ackND 3 8	2045	80M	13.110	1212	90	8.989	920	173	**1.46**	7.292	781	250	**1.80**	6.904	630	328	**1.90**	**6.353**	562	399	**2.06**	6.364	478	469	**2.06**
ackND 3 8	2045	100M	13.043	1215	90	9.042	913	173	**1.44**	7.345	772	251	**1.76**	6.917	628	328	**1.88**	6.372	559	400	**2.05**	6.376	475	470	**2.05**
ackND 3 8	2045	1G	13.849	1154	90	9.588	869	173	**1.44**	7.824	737	250	**1.77**	7.288	606	326	**1.90**	6.665	547	395	**2.08**	6.627	472	460	**2.09**
ackND 3 8	2045	8G	15.521	1043	91	11.200	819	169	**1.39**	9.204	703	239	**1.69**	8.686	573	309	**1.79**	7.947	523	370	**1.95**	7.822	453	432	**1.98**
ackND 3 9	4093	**dft.**	141.662	1415	28	85.999	1041	64	**1.65**	60.032	904	105	**2.36**	49.941	755	151	**2.84**	42.996	625	212	**3.29**	38.546	547	271	**3.68**
ackND 3 9	4093	8G	62.920	1016	91	41.032	815	174	**1.53**	32.717	708	251	**1.92**	30.653	577	328	**2.05**	27.727	526	398	**2.27**	**27.087**	459	466	**2.32**
ackND 3 10	8189	8G	300.687	837	91	184.245	716	174	**1.63**	141.858	643	252	**2.12**	128.381	546	328	**2.34**	115.164	501	398	**2.61**	110.754	447	464	**2.71**
fibND 20	6765	1G	0.513	667	94	0.339	558	168	**1.51**	0.283	444	232	**1.81**	0.251	440	294	**2.04**	0.232	404	338	**2.21**	0.225	358	403	**2.28**
fibND 25	75025	4G	6.651	621	85	4.354	527	153	**1.53**	3.437	410	212	**1.93**	3.097	410	276	**2.15**	2.883	370	327	**2.31**	2.634	332	387	**2.53**
fibND 28	317811	8G	32.24	732	64	21.92	619	112	**1.47**	16.34	544	172	**1.97**	14.85	454	226	**2.17**	13.41	412	278	**2.40**	12.85	375	317	**2.51**
fibND 30	832040	8G	139.617	762	38	101.619	647	62	**1.37**	62.626	557	116	**2.23**	56.072	438	158	**2.49**	49.494	423	193	**2.82**	44.478	371	247	**3.14**

Table 1: Benchmarks for directly-programmed NetDescriptions

In general, as long as the heap is small in comparison with the space requirements of the current run, the run-time system spends a much higher part of its time performing garbage collection — this manifests itself in low “productivity” entries in the tables below in the rows with small fixed heap sizes and with “**dft.**”. (The amount of space that is allocated on the heap by any given task varies only minimally with different heap and parallelism settings.) Not limiting the heap size (with (“-M*size*”) on longer-running tasks may lead the run-time system to use a heap that is larger than the available physical memory, leading to drastic performance loss dues to swapping of memory pages to peripheral storage. For tasks that actually do use large heap space, not fixing the start heap size (with (“-H*size*”) lets the run-time system adopt the default behaviour at the start of the program, leading to slow-down of actually acquiring the needed large heap. Therefore, optimal time is typically obtained using a fixed heap size, that is, with both -H and -M set to the same size, which is what we adopted for our benchmarking.

(The GHC run-time system also provides finer control over the initial heap size, and over the size of the increments; we did not experiment with these here.)

Over its whole run-time, ackND 3 6 allocates 880MB on the heap, and ackND 3 7 allocates 3.5GB. If such small tasks are given large heaps, this leads to significant slow-down. As can be seen for ackND 3 8, which allocates 14GB, giving larger processes a generous fixed heap produces a performance that is closer to the optimum than using the default settings.

On an 8-core 16-hyperthread 2.4GHz Xeon 8870, each of the examples we tried so far has a maximum number of cores beyond which adding cores slows down reduction, see Table 2. This is an example

	time (s)								speedup factor over -N1						
expr.	-N1	-N2	-N5	-N8	-N9	-N10	-N11	-N12	-N2	-N5	-N8	-N9	-N10	-N11	-N12
fib 28	63.581	40.173	22.495	19.389	16.572	17.640	16.618	17.234	1.58	2.83	3.28	**3.84**	3.60	**3.83**	3.69
fib 30	223.291			68.377	63.488	58.204	60.160	62.559			3.27	3.52	**3.84**	3.71	3.57
ack 3 7	5.900	4.177	3.234	3.889	3.786	4.042	4.033	4.170	1.41	**1.82**	1.52	1.56	1.46	1.46	1.41

Table 2: 16-core Benchmarks for directly-programmed NetDescriptions

of the effect of diminishing gains of adding processors to a parallel workload that does not split into a sufficient number of sufficiently large independent pieces: The overhead of synchronisation in such a context makes it unfeasible to profit from the computing power of added cores beyond a task-dependent threshold.

Table 3 contains timings for running our RunInets interpreter on a collection of Inets programs mostly derived from programs in [HJ12] by replacing the main nets with larger examples. The last two columns contain timings for running the compiled programs using the Inets compiler of [HJ12], and the quotient of our "-N1" time with this run-time.

Ackerman.inet from [HJ12] uses a (totalised) predecessor function; Ack.inet is a direct translation of the rules in ackND. The counts reported by the Inets implementation indicate that Ackerman.inet requires almost exactly 1.5 times the number of rule applications of Ack.inet; Inets-compiled executables and our RunInets take roughly 1.6 times the time.

fib.inet is a direct translation of our fibND implementation into Inets, and works, like both Ackerman functions, on unary natural numbers constructed from S and Z. We found that fib.inet performs roughly 20% more allocation than fibND, which will be due to the overhead of transforming an Expression-based NetDescription into Value-based for each rule application (even though there are no expressions to evaluate in this example that does not use attributes). However, it appears that the difference in run times is, as for Ack.inet versus ackND, much less — this should be due to the fact that the overhead is not slowed down by concurrency synchronisation.

fibonacci.inet from [HJ12] carries arguments and results in node attributes, and uses implementation-provided addition of integer attributes instead of recursing over predecessors like fibND. It therefore has significantly less work to do than fibND.

sort.inet from Inets is an implementation of bubble sort on lists; it uses an int-valued agent attribute to carry the list elements, so element comparisons are performed as part of choosing the RHS of conditional rules. The counter results of the Inets runs show that this performs exactly $(n/2+1)\cdot(n+1)$ interactions for a randomly generated start list with even length n. This pattern fits some of the RunInets times in Table 3 exactly, while other RunInets times appear to exhibit a worse asymptotic behaviour; I suggest that this is due to the fact that I used the same heap sizes for different sort argument sizes instead of trying to identify respective optimal heap sizes.

On the whole, on a single core, RunInets typically takes about 10 to 20 times the time of the Inets-compiled executables, which is to be expected for an interpreted implementation.

expr.	result	heap	time (s), allocation rate (MB/MUT-s), productivity (% of elapsed), **speedup**																							Inets	
			-N1			-N2				-N3				-N4				-N5				-N6				time	speedup over -N1
Ackerman 3 6	509	**dft.**	2.150	2100	38	1.191	1440	100	**1.80**	0.850	1214	166	**2.53**	0.770	932	238	**2.79**	0.722	758	312	**2.98**	0.716	624	382	**3.00**	0.125	17.2
Ackerman 3 6	509	20M	1.204	1597	89	0.861	1163	171	**1.40**	0.744	920	251	**1.62**	0.711	745	323	**1.69**	**0.700**	618	396	**1.72**	0.740	497	466	**1.63**		9.63
Ackerman 3 6	509	8G	2.163	1083	88	1.536	850	156	**1.41**	1.323	712	216	**1.63**	1.236	608	268	**1.75**	1.189	528	318	**1.82**	1.209	437	374	**1.79**		17.3
Ackerman 3 7	1021	**dft.**	11.432	2022	30	6.251	1415	77	**1.83**	4.158	1226	134	**1.75**	3.668	936	200	**3.12**	3.249	779	271	**3.52**	3.080	655	339	**3.71**	0.511	22.37
Ackerman 3 7	1021	40M	5.211	1498	88	3.601	1127	169	**1.48**	3.036	915	247	**1.72**	2.799	759	322	**1.86**	**2.738**	632	396	**1.90**	2.843	516	467	**1.83**		10.2
Ackerman 3 7	1021	8G	8.359	1034	92	5.678	827	169	**1.47**	4.741	701	238	**1.76**	4.291	609	302	**1.95**	4.051	535	362	**2.06**	4.023	457	423	**2.08**		16.4
Ackerman 3 8	2045	**dft.**	55.740	1899	26	32.567	1329	63	**1.71**	21.601	1173	108	**2.58**	18.430	915	163	**3.02**	15.819	772	225	**3.52**	14.883	652	283	**3.75**	2.050	27.2
Ackerman 3 8	2045	60M	22.716	1408	86	15.671	1062	165	**1.45**	12.997	878	241	**1.75**	11.922	732	314	**1.91**	11.419	620	389	**1.99**	**11.258**	536	455	**2.02**		11.1
Ackerman 3 8	2045	8G	25.963	1230	90	18.016	938	170	**1.44**	14.724	803	242	**1.76**	13.623	673	312	**1.90**	12.904	585	379	**2.01**	13.042	490	446	**1.99**		12.7
Ackerman 3 9	4093	**dft.**	274.154	1583	25	161.029	1207	57	**1.70**	114.140	1072	90	**2.42**	98.409	842	133	**2.79**	82.098	725	185	**3.34**	72.683	638	237	**3.77**	7.167	38.3
Ackerman 3 9	4093	100M	107.704	1215	84	68.144	1018	158	**1.58**	54.532	872	231	**1.98**	49.224	741	301	**2.19**	47.475	618	375	**2.27**	**47.038**	525	445	**2.29**		15.03
Ackerman 3 9	4093	8G	116.946	1055	90	73.624	883	172	**1.59**	59.421	756	248	**1.97**	52.597	658	322	**2.22**	49.011	579	392	**2.39**	47.341	513	458	**2.47**		16.3
Ackerman 3 10	8189	8G	501.817	971	91	326.369	774	175	**1.54**	254.119	686	253	**1.97**	222.919	606	327	**2.25**	201.495	550	398	**2.49**	191.309	500	461	**2.62**	28.677	17.5
Ack 3 6	509	**dft.**	1.161	2201	42	0.686	1456	109	**1.70**	0.526	1189	174	**2.21**	0.475	934	245	**2.44**	0.439	780	317	**2.64**	0.443	642	383	**2.62**	0.078	14.9
Ack 3 6	509	20M	0.710	1691	90	0.524	1196	174	**1.35**	0.463	926	254	**1.53**	0.433	774	325	**1.64**	**0.428**	645	394	**1.66**	0.502	575	377	**1.41**		9.10
Ack 3 7	1021	**dft.**	6.412	2117	32	3.552	1427	85	**1.81**	2.473	1224	143	**2.59**	2.187	956	207	**2.93**	1.933	815	275	**3.32**	1.835	686	344	**3.49**	0.310	20.7
Ack 3 7	1021	40M	3.023	1612	89	2.180	1157	172	**1.39**	1.909	908	250	**1.58**	1.709	782	325	**1.77**	**1.698**	646	395	**1.78**	1.896	590	388	**1.59**		9.75
Ack 3 8	2045	**dft.**	31.762	2039	27	18.775	1357	68	**1.69**	12.488	1206	115	**2.54**	10.555	954	172	**3.01**	9.132	810	235	**3.48**	8.543	695	292	**3.72**	1.109	28.64
Ack 3 8	2045	60M	13.675	1445	88	9.743	1053	169	**1.40**	8.044	875	247	**1.70**	7.291	742	321	**1.88**	**6.914**	637	394	**1.98**	7.027	529	467	**1.95**		12.3
fib 20	6765	1G	0.587	714	94	0.390	602	169	**1.50**	0.325	527	234	**1.80**	0.291	470	293	**2.02**	0.266	428	353	**2.21**	0.260	401	385	**2.26**	0.030	19.6
fib 25	75025	1G	8.832	838	56	5.531	674	112	**1.60**	4.250	576	171	**2.08**	3.754	515	216	**2.35**	3.373	456	272	**2.62**	3.141	413	323	**2.81**	0.513	17.2
fib 25	75025	**2G**	7.096	797	74	4.603	637	144	**1.54**	3.685	562	205	**1.93**	3.398	473	264	**2.09**	3.082	446	310	**2.30**	**2.958**	402	359	**2.40**		13.8
fib 25	75025	4G	7.321	677	85	4.709	568	160	**1.55**	3.910	486	228	**1.87**	3.445	432	291	**2.13**	3.226	404	333	**2.27**	**2.958**	365	406	**2.47**		14.3
fib 25	75025	8G	7.538	667	86	4.907	563	157	**1.54**	4.026	501	217	**1.87**	3.667	442	271	**2.06**	3.318	401	331	**2.27**	3.329	359	372	**2.26**		14.7
fib 25	75025	12G	7.589	684	84	5.023	580	151	**1.51**	4.218	502	209	**1.80**	3.804	443	264	**2.00**	3.428	409	318	**2.21**	3.226	382	362	**2.35**		14.8
fib 28	317811	2G	89.117	834	25	54.464	682	51	**1.64**	54.409	532	63	**1.64**	48.101	493	79	**1.85**	35.036	452	115	**2.54**	44.302	406	101	**2.011**	SegFault	
fib 28	317811	4G	42.922	826	52	25.890	649	109	**1.66**	21.034	567	141	**2.04**	19.416	501	189	**2.21**	16.953	425	256	**2.53**	16.310	411	283	**2.63**		
fib 28	317811	8G	35.092	777	68	22.117	627	133	**1.59**	18.213	541	189	**1.93**	16.366	482	236	**2.144**	14.749	444	290	**2.38**	14.093	397	335	**2.49**		
fib 28	317811	**12G**	34.283	750	72	22.027	613	138	**1.56**	17.603	551	193	**1.95**	16.007	465	253	**2.14**	14.546	430	301	**2.36**	**13.759**	386	355	**2.49**		
fib 30	832040	12G	116.444	804	53	76.595	624	104	**1.52**	62.474	559	142	**1.86**	55.308	495	182	**2.11**	49.664	454	222	**2.34**	45.192	406	273	**2.58**	SegFault	
fibonacci 20	6765	1G	0.269	759	91	0.185	619	163	**1.45**	0.154	551	223	**1.75**	0.141	486	275	**1.91**	0.130	456	320	**2.07**	0.125	417	366	**2.15**	0.018	14.9
fibonacci 25	75025	2G	2.740	742	95	1.841	590	178	**1.49**	1.459	526	255	**1.88**	1.324	451	327	**2.07**	1.172	404	400	**2.34**	1.134	377	461	**2.42**	0.172	15.9
fibonacci 28	317811	8G	11.752	724	90	7.926	583	167	**1.48**	5.928	508	258	**1.98**	5.460	446	321	**2.15**	4.934	404	395	**2.38**	4.519	374	469	**2.60**	0.721	16.3
fibonacci 30	832040	8G	47.947	1048	45	30.137	707	96	**1.59**	23.071	608	144	**2.08**	17.589	513	222	**2.73**	15.243	464	282	**3.15**	16.191	422	292	**2.96**	1.858	25.8
sort200		**dft.**	0.123	2226	46	0.092	1443	94	**1.34**	0.085	1171	126	**1.45**	0.080	925	168	**1.54**	0.080	766	204	**1.54**	0.080	662	236	**1.54**	0.012	10.3
sort300		**dft.**	0.255	2114	44	0.191	1341	93	**1.34**	0.155	1115	138	**1.65**	0.152	874	180	**1.68**	0.159	695	217	**1.60**	0.145	626	264	**1.76**	0.023	11.1
sort400		**dft.**	0.466	2039	41	0.328	1291	92	**1.42**	0.254	1084	142	**1.83**	0.243	851	190	**1.92**	0.223	741	237	**2.09**	0.218	622	289	**2.13**	0.037	12.6
sort500		**dft.**	0.802	1924	38	0.543	1233	87	**1.48**	0.400	1057	137	**2.01**	0.376	826	187	**2.13**	0.345	709	238	**2.32**	0.332	619	283	**2.41**	0.060	13.4
sortC600		50M	0.570	1504	91	0.439	1054	169	**1.30**	0.367	875	243	**1.55**	0.349	714	313	**1.63**	0.335	607	384	**1.70**	0.326	534	449	**1.75**	0.071	8.03
sortC700		50M	0.765	1495	91	0.586	1044	170	**1.31**	0.492	862	246	**1.55**	0.460	708	319	**1.66**	0.430	627	387	**1.78**	0.429	530	458	**1.78**	0.095	8.05
sortC800		50M	1.002	1474	91	0.756	1039	170	**1.31**	0.625	869	246	**1.55**	0.586	714	320	**1.66**	0.558	613	391	**1.78**	0.557	526	458	**1.78**	0.121	8.05
sortC900		50M	1.274	1451	90	0.955	1032	170	**1.33**	0.798	850	248	**1.60**	0.741	707	320	**1.72**	0.698	620	393	**1.83**	0.688	530	460	**1.85**	0.155	8.22
sortC1000		**dft.**	3.810	1741	31	2.307	1158	77	**1.65**	1.602	995	128	**2.38**	1.404	790	184	**2.71**	1.215	682	247	**3.14**	1.105	600	309	**3.45**	0.196	19.4
sortC1000		100M	1.599	1410	91	1.196	997	173	**1.34**	0.999	827	249	**1.60**	0.902	700	326	**1.77**	0.853	605	399	**1.87**	0.845	520	469	**1.89**	0.196	8.16
sortC2000		100M	6.805	1292	90	4.915	936	172	**1.38**	3.939	800	251	**1.73**	3.599	671	328	**1.89**	3.323	592	404	**2.05**	3.239	510	480	**2.10**	StackOverflow	
sortC3000		100M	16.932	1174	88	11.656	892	169	**1.45**	9.261	768	248	**1.83**	8.344	654	323	**2.03**	7.674	579	397	**2.21**	7.430	503	472	**2.28**	StackOverflow	
sortC4000		100M	32.337	1108	87	21.991	854	166	**1.47**	17.454	737	242	**1.85**	15.580	634	315	**2.08**	14.112	567	390	**2.29**	13.638	497	460	**2.37**	StackOverflow	
sortC5000		100M	54.673	1037	85	36.384	830	161	**1.50**	28.286	722	238	**1.93**	25.387	620	308	**2.15**	22.902	558	380	**2.38**	21.701	492	455	**2.52**	StackOverflow	
sortC10000		1G	247.709	839	93	165.529	650	180	**1.50**	131.459	554	265	**1.88**	110.329	506	346	**2.25**	97.315	472	420	**2.55**	89.930s	424	506	**2.75**	StackOverflow	

Table 3: Benchmarks for Inets programs

Just adding cores to a RunInets run without any heap settings (see the "**dft.**" rows) appears to produce relatively nice speed-ups for fine-grained parallelism, but one has to be aware that the single-core execution in that case typically was spending a far larger portion of its time in garbage collection than the multi-core versions. (This applies also for “relatively small” fixed heaps.)

It appears to be more honest to consider the speed-ups compared to single-core executions with a “good” fixed heap setting; the fastest runs on our six-core machine with our parallel interpreter all use five or six cores, and tend to take only about five to six times as long as the compiled Inets runs on a single core.

(For reasons I have not investigated, the Inets-compiled executables crashed for the larger fib.inet runs after producing partial output; on a modified version (fibNat.inet) that converts results from unary representation to int attributes, all Inets runs crashed. For sort.inet, the Inets version was originally changed only by adding longer argument lists to the start net; beyond 500 elements, this lead to stack overflow errors in the javacc-generated parser. Changing the start net definition to a sequence of equations each adding a smaller chunk to the list allowed us to make some progress, but beyond 1000 elements, a different stack overflow occurred.)

6 Conclusion

Interaction nets as an “inherently parallel” execution model promise large speed-ups via parallelisation, but accessible platforms for experimentation are still missing.

Using Concurrent Haskell to implement interaction nets understood as an execution mechanism, we achieved a simple and easily understandable implementation, the entire core of which could be presented in just a bit more than three pages of literate code. By having added support for the Inets file format, we enable experimentation with interaction net definitions in the shape used by most of the current interaction net literature — with the restriction that a consistent polarity assignment must be possible (which is also one of the conditions of Lafont [Laf90] for deadlock safety).

Keeping in mind that, in our straight-forward ultrafine-grained implementation, the concurrent interaction net rules reduce a heavily shared structure, and given that we made no effort to enable coarse-grain parallelism, the speed-ups achieved on the usual microbenchmarks are actually surprisingly good, and we expect even better behaviour on rules with larger right-hand sides that give rise to more sparsely connected nets.

References

[AFK+11] Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, Olivier Namet & Bruno Pinaud (2011): *PORGY: Strategy-driven interactive transformation of graphs*. In Rachid Echahed, editor: *TERMGRAPH 2011, International Workshop on Term Graph Rewriting*, *Electronic Proceedings in Computer Science* 48, pp. 54–68, doi:10.4204/EPTCS.48.7.

[AK09] Christopher K. Anand & Wolfram Kahl (2009): *Synthesizing and Verifying Multicore Parallelism in Categories of Nested Code Graphs*. In Michael Alexander & William Gardner, editors: *Process Algebra for Parallel and Distributed Processing*, chapter 1, *CRC Computational Science Series* 2, Chapman & Hall, pp. 3–45, doi:10.1201/9781420064872.pt1.

[APV08] José Bacelar Almeida, Jorge Sousa Pinto & Miguel Vilaça (2008): *A Tool for Programming with Interaction Nets*. *ENTCS* 219, pp. 83–96, doi:10.1016/j.entcs.2008.10.036. Proc. Eighth International Workshop on Rule Based Programming (RULE 2007).

[BP97] Richard Banach & George A. Papadopoulos (1997): *A Study of Two Graph Rewriting Formalisms: Interaction Nets and MONSTR*. *Journal of Programming Languages* 5, pp. 210–231.

[CFF+07] Horatiu Cirstea, Germain Faure, Maribel Fernández, Ian Mackie & François-Régis Sinot (2007): *From Functional Programs to Interaction Nets via the Rewriting Calculus*. *ENTCS* 174(10), pp. 39–56, doi:10.1016/j.entcs.2007.02.046. Proceedings of the Sixth International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2006).

[EL02] Levent Erkök & John Launchbury (2002): *A recursive do for Haskell*. In Manuel Chakravarty, editor: *Proc. Haskell Workshop 2002*, ACM Press, pp. 29–37, doi:10.1145/581690.581693.

[Fal06] Marc de Falco (2006): *Interaction Nets Laboratory*. Available at `http://inl.sourceforge.net/`.

[HJ12] Abubakar Hassan & Eugen Jiresch (2012): *Interaction Nets Programming Language*. `https://gna.org/projects/inets/`, `https://gna.org/svn/?group=inets`, last accessed 2015-02-06. (Source code for the "Inets" system.).

[HJS09] Abubakar Hassan, Eugen Jiresch & Shinya Sato (2009): *An Implementation of Nested Pattern Matching in Interaction Nets*. In Ian Mackie & Anamaria Martins Moreira, editors: *Proceedings Tenth International Workshop on Rule-Based Programming, RULE 2009, Brasília, Brazil, 28th June 2009.*, *EPTCS* 21, pp. 13–25, doi:10.4204/EPTCS.21.2.

[HMS09] Abubakar Hassan, Ian Mackie & Shinya Sato (2009): *Compilation of Interaction Nets*. *ENTCS* 253(4), pp. 73–90, doi:10.1016/j.entcs.2009.10.018. Proc. TERMGRAPH 2009.

[HMS10] Abubakar Hassan, Ian Mackie & Shinya Sato (2010): *A lightweight abstract machine for interaction nets*. In Jochen Küster & Emilio Tuosto, editors: *Proc. GT-VMT 2010*, *ECEASST* 29, pp. 9.1–9.12. Available at `http://journal.ub.tu-berlin.de/eceasst/article/view/416`.

[HP91] Berthold Hoffmann & Detlef Plump (1991): *Implementing Term Rewriting by Jungle Evaluation*. *Informatique théorique et applications/Theoretical Informatics and Applications* 25(5), pp. 445–472.

[Jir14] Eugen Jiresch (2014): *Towards a GPU-based Implementation of Interaction Nets*. In Benedikt Löwe & Glynn Winskel, editors: *8th International Workshop on Developments in Computational Models, DCM 2012*, *EPTCS* 143, pp. 41–53, doi:10.4204/EPTCS.143.4.

[KAC06] Wolfram Kahl, Christopher Kumar Anand & Jacques Carette (2006): *Control-Flow Semantics for Assembly-Level Data-Flow Graphs*. In Wendy McCaull, Michael Winter & Ivo Düntsch, editors: *8th Intl. Seminar on Relational Methods in Computer Science, RelMiCS 8, Feb. 2005*, *LNCS* 3929, Springer, pp. 147–160, doi:10.1007/11734673_12.

[KKSV93] J.R. Kennaway, J.W. Klop, M.R. Sleep & F.J. de Vries (1993): *An Introduction to Term Graph Rewriting*. In M.R. Sleep, M.J. Plasmeijer & M.C.J.D. van Eekelen, editors: *Term Graph Rewriting: Theory and Practice*, chapter 1, Wiley, pp. 1–14.

[Laf90] Yves Lafont (1990): *Interaction Nets*. In: *17th POPL*, ACM, New York, NY, USA, pp. 95–108, doi:10.1145/96709.96718.

[Lip02] Sylvain Lippi (2002): *in^2: A Graphical Interpreter for Interaction Nets*. In Sophie Tison, editor: *RTA 2002*, *LNCS* 2378, Springer, Berlin Heidelberg, pp. 380–385, doi:10.1007/3-540-45610-4_29.

[Mac98] Ian Mackie (1998): *YALE: Yet Another Lambda Evaluator Based on Interaction Nets*. In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, ACM, New York, NY, USA, pp. 117–128, doi:10.1145/289423.289434.

[Mac05] Ian Mackie (2005): *Towards a Programming Language for Interaction Nets*. *ENTCS* 127(5), pp. 133–151, doi:10.1016/j.entcs.2005.02.015. Proc. TERMGRAPH 2004.

[Pin01] Jorge Sousa Pinto (2001): *Parallel Evaluation of Interaction Nets with MPINE*. In Aart Middeldorp, editor: *Rewriting Techniques and Applications, RTA 2001*, *LNCS* 2051, Springer, pp. 353–356, doi:10.1007/3-540-45127-7_26.

[PJGF96] Simon L. Peyton Jones, Andrew Gordon & Sigbjorn Finne (1996): *Concurrent Haskell*. In: *23rd POPL*, acm press, pp. 295–308, doi:10.1145/237721.237794.

[PQ07] Marco Pedicini & Francesco Quaglia (2007): *PELCR: Parallel Environment for Optimal Lambda-calculus Reduction*. *ACM Trans. Computational Logic* 8(3), doi:10.1145/1243996.1243997.