# Compiling Interaction Nets

Abubakar.Hassan at kcl.ac.uk

Department of Computer Science
Kings College London

April 4, 2006

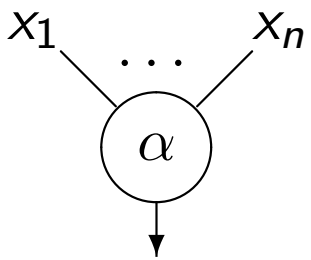
# Outline

# Part I

# Interaction Nets Programming Language

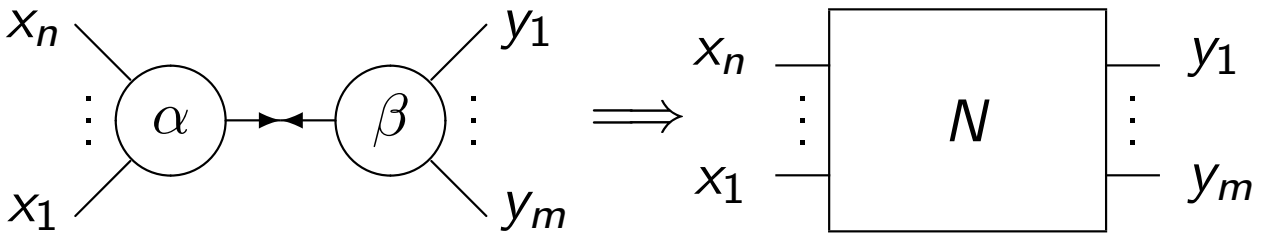
# Interaction nets system

Define a set $\Sigma$ of Agents



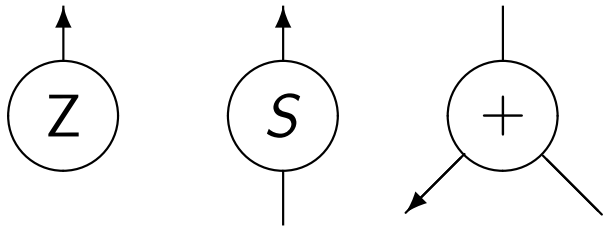

$$y = \alpha(x_1, \ldots, x_n)$$

and a set $\mathcal{R}$ of Rules



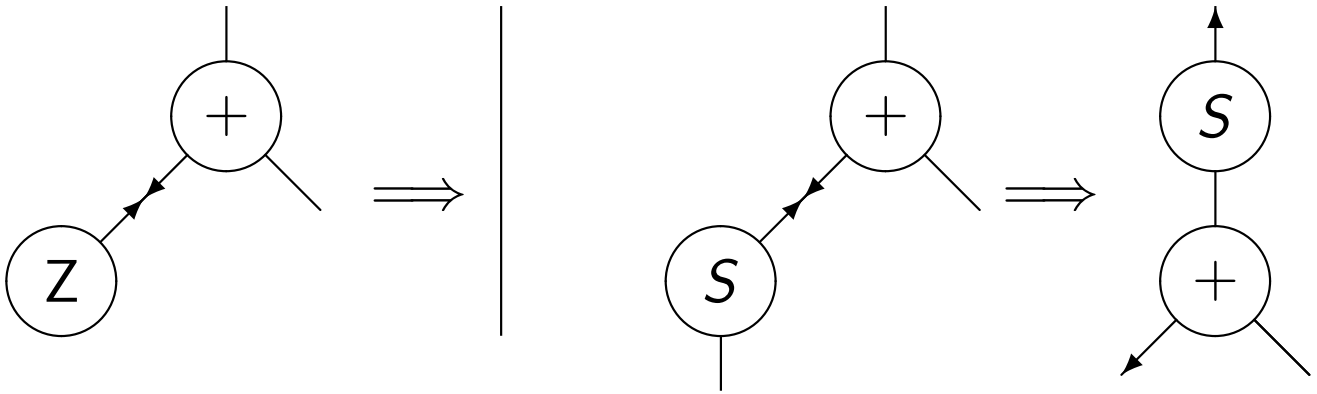
# Example

$\Sigma = \{Z, S, +\}$:




$$x = Z() \quad y = S(a) \quad w = +(d, e)$$

$\mathcal{R} =$



$$\begin{array}{lcl} a = \mathrm{Add}(x, y), a = \mathrm{Z} & \Longrightarrow & x = y \\ a = \mathrm{Add}(x, y), a = \mathrm{S}(z) & \Longrightarrow & x = \mathrm{S}(b), z = \mathrm{Add}(b, y) \end{array}$$

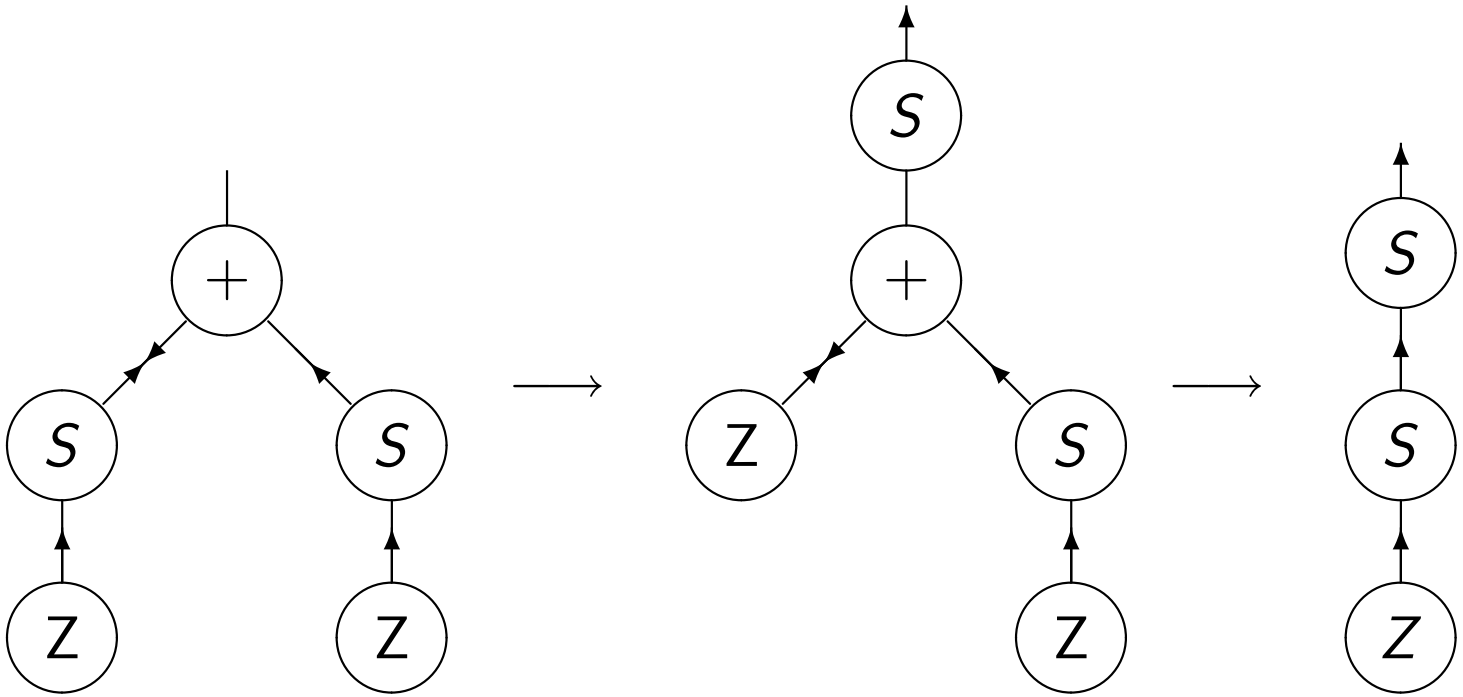
Replace equals for equals:

$$\begin{array}{ll} \mathrm{Z} = \mathrm{Add}(x, x) & \Longrightarrow \\ \mathrm{S}(\mathrm{Add}(b, y)) = \mathrm{Add}(\mathrm{S}(b), y) & \Longrightarrow \end{array}$$

we replace the '$=$' by '$><$' and remove $\Longrightarrow$ if empty.

$$\begin{array}{l} \mathrm{Z} >< \mathrm{Add}(x, x) \\ \mathrm{S}(\mathrm{Add}(b, y)) >< \mathrm{Add}(\mathrm{S}(b), y) \end{array}$$

Net with active pair and Reduction




After replacing equals for equals:

$$\begin{aligned} \mathrm{S(Z)} = \mathrm{Add}(z, \mathrm{S(Z)}) &\longrightarrow \mathrm{Z} = \mathrm{Add}(S, \mathrm{S(Z)}) \\ &\longrightarrow \mathrm{S(S(Z))} \end{aligned}$$

# Language Syntax

We can see interaction nets as a configuration
$c = (\Sigma, \langle \vec{t} \mid u_1 = v_1, \ldots u_n = v_n \rangle, \mathcal{R})$

$$
\begin{array}{lll}
P & ::= & (agent \mid rule)^*\ net \\
agent & ::= & agentName\ :\ arity \\
rule & ::= & [ruleName:]\ term\ ><\ term\ (\ =>\ net \mid ) \\
net & ::= & [netName:]\ equation\ (,\ equation)^* \mid \\
equation & ::= & term\ =\ term \\
term & ::= & agentName(term, ..., term) \mid var
\end{array}
$$

# Example

Z:0 S:1
Add:2
AddZ: Z $>\!<$ Add(x,x)
AddS: S(Add(b,y)) $>\!<$ Add(S(b),y)
S(Z) = Add(b,S(Z))

# Foreign Language Interface

$$
\begin{array}{lll}
P & ::= & (agent \mid rule)^* \; net \\
agent & ::= & agentName \; : \; arity \\
rule & ::= & [ruleName :] \; term \; >< \; term \; (\; => net \mid ) \\
net & ::= & [netName :] \; equation \; (, \; equation)^* \mid \\
equation & ::= & term \; = \; term \\
term & ::= & agentName\{\ell\}(term, ..., term)
\end{array}
$$

# Example

fact:1 num:0 ans:0

num{*int n*} >< fact(ans{Factorial.fact(n)})

fact(x) = num{8}

# Module System

$$
\begin{array}{lll}
P & ::= & \texttt{module} \quad \{\ [\texttt{import}\ \mathit{modName}[.\ \mathit{compName}]] \\
 & & (\mathit{agent} \mid \mathit{rule})^*\ \mathit{net} \quad \} \\
\mathit{agent} & ::= & [*]\mathit{agentName}\ :\ \mathit{arity} \\
\mathit{rule} & ::= & [[*]\mathit{ruleName}:\ ]\ \mathit{term}\ ><\ \mathit{term}\ (\ =>\ \mathit{net} \mid\ ) \\
\mathit{net} & ::= & [[*]\mathit{netName}:\ ]\ \mathit{equation}\ (,\ \mathit{equation})^* \mid \\
\mathit{equation} & ::= & \mathit{term}\ =\ \mathit{term} \\
\mathit{term} & ::= & [*]\mathit{agentName}\{\ell\}(\mathit{term}, ..., \mathit{term})
\end{array}
$$

# Example

module add {
import arith.Z
import arith.S
*Add:2
*AddZ: Z >< Add(x,x)
*AddS: S(Add(b,y)) >< Add(S(b),y)
S(Z) = Add(b,S(Z)) }

Part II

# Compilation of Interaction nets

# Compilation

Compiling a program *p.net* gives a *linkable* `p.l` and a `p.sym`. We link `p.l` to get a *loadable* `p.o` which is ready for execution by the abstract machine.
Stack based machine that executes the bcodes.

$$\begin{array}{lll} \mathit{bcodes} & ::= & S \mid A \mid M \\ S & ::= & \mathit{loadc} \mid \mathit{store} \mid \mathit{load} \mid \mathit{pop} \\ A & ::= & \mathit{mkAgent} \mid \mathit{mkVar} \mid \mathit{mkNet} \mid \mathit{mkRule} \\ M & ::= & \mathit{getVar} \mid \mathit{connectPorts} \mid \mathit{eval} \end{array}$$

A state of the machine is a configuration:

$$\langle PC, \rho, S, A, R \rangle$$

# Compilation schemes

A program may consist of a list of modules $M_1, ...M_n$. The compilation scheme $\mathcal{C}_p$ compiles each module using the scheme $\mathcal{C}_m$. This in turn compiles each of the component

$$c = (\Sigma, \langle \vec{t} \mid u_1 = v_1, \ldots u_n = v_n \rangle, \mathcal{R})$$

in the module by using the appropriate scheme:

$$\mathcal{C}_p = \mathcal{C}_m[\![M_1]\!], ..., \mathcal{C}_m[\![M_n]\!]$$

The compilation of a module generates the following code:

$$\mathcal{C}_m[\![(\Sigma, \langle \vec{t} \mid u_1 = v_1, \ldots u_n = v_n \rangle, \mathcal{R})]\!] = \begin{cases} ruleName : \\ \mathcal{C}_R[\![r_1]\!] \\ \vdots \\ ruleName : \\ \mathcal{C}_R[\![r_n]\!] \\ netName : \\ \mathcal{C}_N[\![u_1 = v_1]\!] \\ \vdots \\ netName : \\ \mathcal{C}_N[\![u_n = v_n]\!] \\ interface : \\ \mathcal{C}_T[\![t]\!] \end{cases}$$

# Example

The code generated by compiling:

```
module Erase{
Eps:0
EpsEpsRule:  Eps >< Eps
EpsNet:  Eps = Eps
}
```

is:

```
EpsEpsRule:     loadc 2         loadc 1
loadc eps       load 0          mkAgent
loadc 1         load 1          store 0
mkAgent         mkRule          load 0
store 0         pop             store 0
load 0          EpsNet:         load 0
store 1         loadc eps       load 1
loadc eps       loadc 1         mkNet
```

Execution starts from the initial configuration:

$$P \vdash \langle 0, [], [], [], [] \rangle$$

The machine starts executing the instructions under the label `EpsEpsRule`. After interpreting all the code in this label, we obtain the configuration:

$$p \vdash \langle 16, [], [], [], [\texttt{Eps}, \texttt{Eps}] \rangle$$

The configuration obtained after executing the code in the label `EpsNet`, before the instruction `eval`:

$$p \vdash \langle 32, [], [], [\texttt{Eps}, \texttt{Eps}], [\texttt{Eps}, \texttt{Eps}] \rangle$$

The final configuration obtained after the `eval` instruction:

$$p \vdash \langle 33, [], [0 \mapsto \epsilon], [], [\texttt{Eps}, \texttt{Eps}] \rangle$$

# Future work

We have seen for the first time a compilation scheme for Interaction nets.

- extend the core language
- study alternative compilation schemes
- compilation for parallel hardware
- optimisations
- type system

# Compilation Schemes

$$\mathcal{C}_v[\![v]\!] = \begin{cases} \texttt{loadc } v \\ \texttt{mkVar} \\ \texttt{store\_}i \end{cases}$$

$$\mathcal{C}_t[\![\alpha(t_1 \cdots t_n)]\!] \begin{cases} \mathcal{C}_a[\![\alpha]\!] \\ \texttt{store\_}i \\ \mathcal{C}_a[\![t_1]\!] \\ \texttt{store\_}i+1 \\ \texttt{load } i \\ \texttt{loadc } 1 \\ \texttt{load } i+1 \end{cases}$$

$$\mathcal{C}_n[\![\alpha(u_1,...,u_n) = \beta(v_1,...,v_n)]\!] = \begin{cases} \mathcal{C}_t[\![\alpha(u_1 \cdots u_n)]\!] \\ \mathcal{C}_t[\![\beta(v_1 \cdots v_n)]\!] \\ \texttt{mkNet} \\ \texttt{eval} \\ \texttt{pop} \end{cases}$$

$$\mathcal{C}_r[\![\alpha(t_1,...,t_n) >< \beta(u_1,...,u_n)]\!] = \begin{cases} \mathcal{C}_t[\![\alpha(t_1 \cdots t_n)]\!] \\ \mathcal{C}_t[\![\beta(u_1 \cdots u_n)]\!] \\ \texttt{mkRule} \\ \texttt{pop} \end{cases}$$

$$\mathcal{C}_a[\![\alpha]\!] = \begin{cases} \texttt{loadc}\ \alpha & \texttt{store\_}i+1 \\ \texttt{loadc}\ ar & \texttt{load}\ i \\ \texttt{mkAgent} & \texttt{loadc}\ n \\ \texttt{store\_}i & \texttt{load}\ i+1 \\ \texttt{loadc}\ 0 & \texttt{loadc}\ 0 \\ \texttt{connectPorts} & \texttt{connectPorts} \\ \texttt{store\_}i & \texttt{store\_}i \\ \vdots & \\ \mathcal{C}_a[\![t_n]\!] & \end{cases}$$