Abstract Machines, Optimal Reduction, and Streams

Marco Pedicini (Roma Tre University)
in collaboration with Anna Chiara Lai (Univ. of Rome La Sapienza) and Mario Piazza (Univ. of Chieti-Pescara)

The first general meeting of the GDRI-LL

Università di Bologna, February 1 – 3, 2016

Implementation of lambda-calculus reduction

- **Optimal reduction** was introduced in J.J. Levy's PhD Thesis in 1978 and defined (by means of sharing graphs) by J. Lamping in 1990.
- **Geometry of Interaction** was introduced by J.-Y. Girard in 1985.
- In 1992, M. Abadi, J. Gonthier and J.-J. Levy made a link between optimal reduction and Geometry of Interaction.
- **Virtual Reduction** (Danos-Regnier 1993) is a way to make GoI close to optimal implementation.
- **Directed Virtual Reduction** (Danos-Pedicini-Regnier 1997) is a modification of the VR to ease implementation.
- **PELCR** (Pedicini-Quaglia 2007) is a complete implementation which permits parallel execution on multi-core machines.

Is a Graph Reduction Technique ?

- The **machine state** is represented by a pair:
  - a **dynamic graph**
  - a list of **pending actions**.
- Any **machine transition** is obtained by applying the action α to the current graph G, from which we get a pair

$$\alpha.G \rightarrow (\Delta_\alpha, G \cup \{\alpha\}).$$

$\Delta_\alpha = \{\alpha_1, \beta'_1, \ldots, \alpha_m, \beta'_m\}$ is a set of residual actions to be added to pending actions already in the state and $G \cup \{\alpha\}$ is the updated dynamic graph.

Machine Transition

More precisely:

Let A be the set of pending actions, so that the couple (A, G) denotes the state of the machine. The transition associated to the action $\alpha \in A$ is then

$$(A, G) \xrightarrow{\tau_\alpha} (A \setminus \{\alpha\} \cup \Delta_\alpha, G \cup \{\alpha\})$$

that is, residual actions Δ_α are added to the list of pending actions.

The basic computational step is borrowed from half-combustion strategy of DVR it includes a symbolic computation in the algebraic structure associated to the graph (the dynamic monoid) and it is a generalisation of the algebraic computations at the base of Geometry of Interaction.

A computation

The memory of the machine is initialized with an empty graph, so that the execution of a terminating program on the abstract machine is represented by the finite sequence of transitions

$$(A^0, \emptyset) \xrightarrow{\tau_{\alpha_0}} (A^1, G^1) \xrightarrow{\tau_{\alpha_1}} \cdots \xrightarrow{\tau_{\alpha_{n-1}}} (A^n, G^n) \xrightarrow{\tau_{\alpha_n}} (\emptyset, G^{n+1})$$

where $\alpha_i \in A^i$ for all $i = 0, \ldots, n$. Note that the initial action set A^0 is the interpretation of the program, and the final graph G^{n+1} represents the result of the evaluation.

In general, the execution of the machine may not terminate and, consequently, be represented by a possibly infinite sequence of elementary steps.

Non termination

When executing on parallel machines, the sequence of residuals produced by non terminating execution on one machine and directed to another one can be infinite.

To cope with this situation we adopt *streams* as data structures for the pending actions.

Description of the Sequential Abstract Machine, whose setting is reminiscent of SECD machines, employed to give the operational semantics of λ-calculus.

A bridging model

We introduce a formal description for **multicore “functional” computation** as a step to **quantitatively study** the behaviour of the PELCR implementation.

A bridging model

We introduce a formal description for **multicore “functional” computation** as a step to **quantitatively study** the behaviour of the PELCR implementation.

As a starting poit we assume (what we already know) that PELCR is sound as a “parallel” operational semantics, this means that we do not care on reordering of actions since the computation of the normal form by using Geometry of interaction rules (shared optimal reduction) is **local and asynchroous**.

A bridging model

We introduce a formal description for **multicore "functional" computation** as a step to **quantitatively study** the behaviour of the PELCR implementation.

As a starting poit we assume (what we already know) that PELCR is sound as a "parallel" operational semantics, this means that we do not care on reordering of actions since the computation of the normal form by using Geometry of interaction rules (shared optimal reduction) is **local and asynchroous**.

Definition (PELCR Actions)

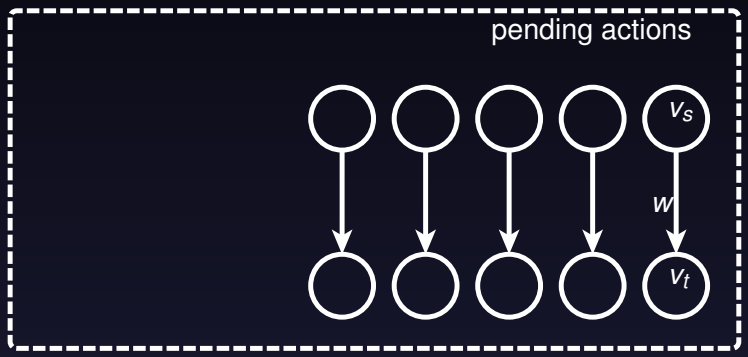
Given a dynamic graph G, which is a graph $G = (V, E \subset V \times V)$ with edges labeled on the Girard dynamic algebra Λ^*, we define an action α on G as $\langle \epsilon, e, w \rangle$ where $\epsilon \in \{+, -\}^2$, $e = (v_t, v_s) \in V^2$ is an edge in G and $w \in \Lambda^*$.

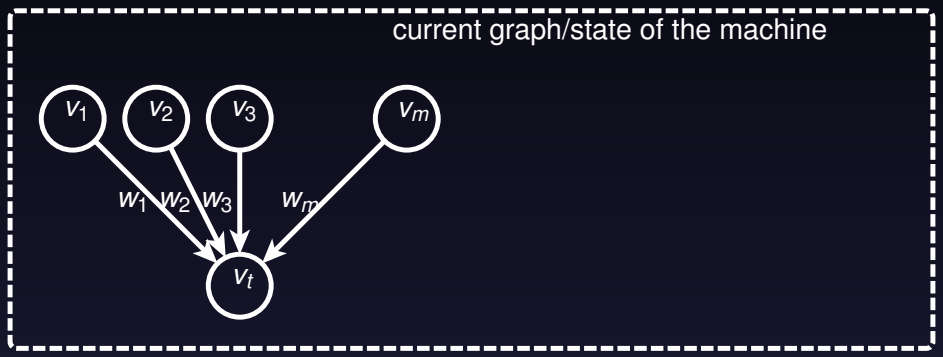
PELCR-VM

We describe the **pelcr virtual machine** (PVM) as an abstract machine working on its state (C, D).

- C contains the **computational task**: a **stream of closures** (FIFO).
  - A closure is a **signed edge**.
  - A signed edge e is represented by a triplet $\langle(\varepsilon_t, \varepsilon_s), (v_t, v_s), w\rangle$, where: $\varepsilon_t, \varepsilon_s \in \{+, -\}$ are the target and source polarities of e; $v_t, v_s \in V$ are the target and source nodes of e and can be considered as their memory addresses; and $w \in M$ is the label of e.
- D represents the **current memory**, and contains **environment elements**.
  - any environment element has a memory address e_i and is called **node**.
  - memory at address e_i contains signed edges.

Machine Representation





Girard dynamic algebra is traditionally used in the execution formula of Geometry of Interaction, which is a power series of matrices. Consequently Λ^* is considered together with a formal sum of its elements, and consequently it is a (monoid) algebra.

Girard dynamic algebra Λ^*

The so-called *Girard dynamic algebra* Λ^* is the dynamic monoid generated by the constants p, q, and a family $W = \{w_i\}_i$ of exponential generators, with a morphism $!(.)$, such that for any $u \in \Lambda^*$:

$$\begin{array}{lrcll} \text{(annihilation)} & x^* y & = & \delta_{xy} & \text{for } x, y = p, q, w_i, \\ \text{(swapping)} & !(u) w_i & = & w_i !^{e_i}(u), & \end{array}$$

where δ_{xy} is the Kronecker operator,e_i is an integer associated with w_i called the *lift* of w_i, i is called the *name* of w_i and we often write w_{i,e_i} to explicitly note the lift of the generator. Notice that swapping and annihilation rules imply that for every $a, b \in \Lambda^*$ either $b^* a = 0$ or it has **a stable form**, that is Λ^* satisfies SF condition.

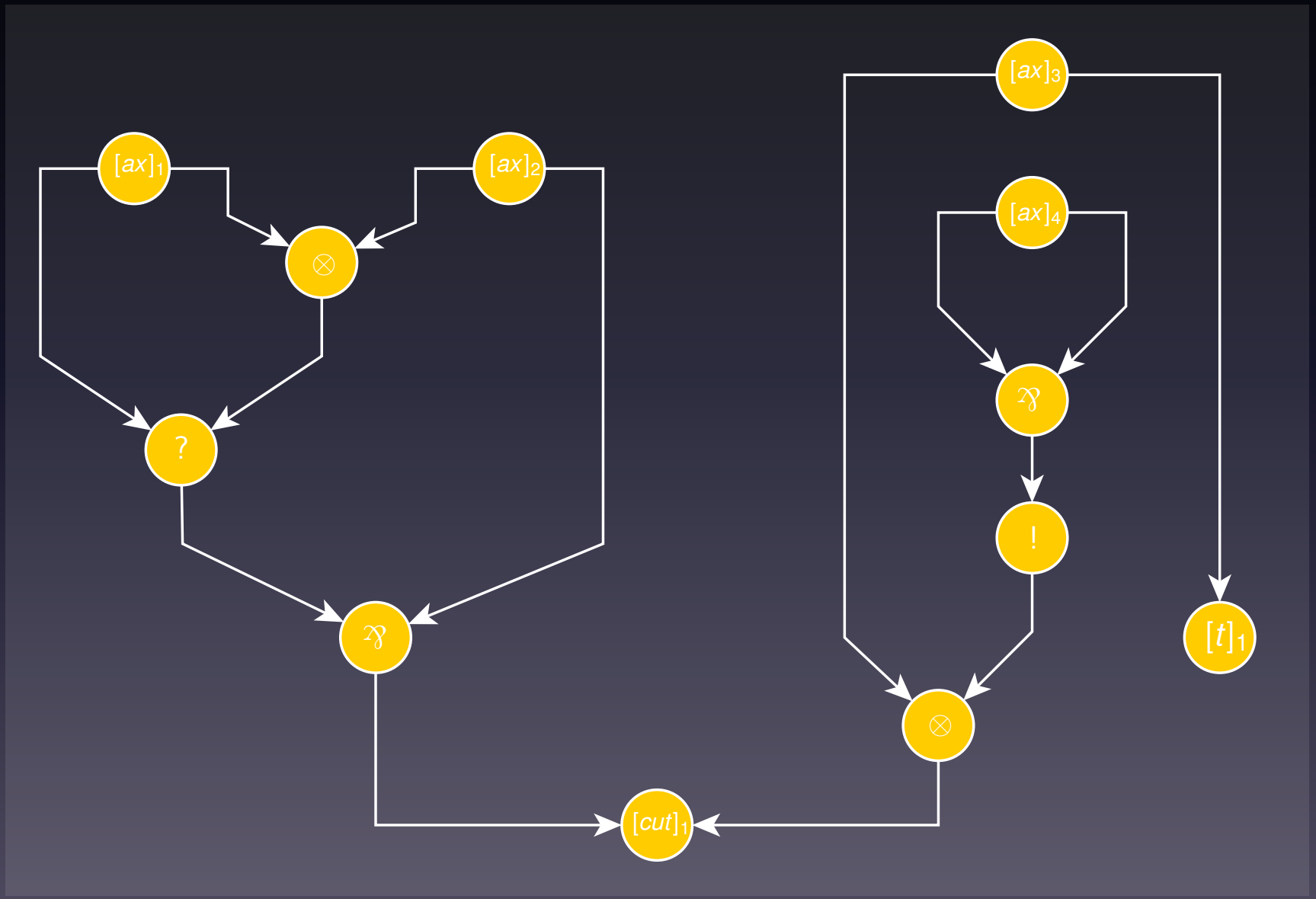
Stable Form

For instance, setting $a = w_{1,2}$ and $b = !^2 q$, by applying the annihilation rule we get:

$$b^* a = (!^2 q)^* w_{1,2} = !(!q^*) w_{1,2} = w_{1,2} !^2 (!q^*) = w_{1,2} (!^3 q)^* = a' b'^*$$

with $a' = a$ and $b' = !b$.

Example: $(\Delta)I$



GoI interpretation

The matrix with entries in the Girard dynamic algebra:

$$
\begin{array}{c}
 \\
[\mathrm{ax}]_1 \\
[\mathrm{ax}]_2 \\
[\mathrm{ax}]_3 \\
[\mathrm{ax}]_4 \\
[\mathrm{cut}]_1 \\
[\mathrm{t}]_1
\end{array}
\begin{array}{c}
\begin{array}{cccccc}
[\mathrm{ax}]_1 & [\mathrm{ax}]_2 & [\mathrm{ax}]_3 & [\mathrm{ax}]_4 & [\mathrm{cut}]_1 & [\mathrm{t}]_1
\end{array} \\
\begin{pmatrix}
0 & 0 & 0 & 0 & qx_2 + qx_1q & 0 \\
0 & 0 & 0 & 0 & qx_1p + p & 0 \\
0 & 0 & 0 & 0 & q!q + q!p & 0 \\
0 & 0 & 0 & 0 & p & 1 \\
x_2^*q^* + q^*x_1^*q^* & p^*x_1^*q^* + p^* & (!q^*)q^* + (!p^*)q^* & p^* & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0
\end{pmatrix}
\end{array}
$$

The corresponding graph

Nodes are axioms, cuts and conclusions (terminal nodes):

$$V = \{[\text{ax}]_1, [\text{ax}]_2, [\text{ax}]_3, [\text{ax}]_4, [\text{cut}]_1, [\text{t}]_1\}$$

and edges $((v_t, v_s), w)$ get a weight $w \in \Lambda^*$ where v_t is the target node and v_s is the source node. In this example, the "sparse" representation, consisting of the list of edges with a non-null weight, is more compact:

$$\begin{aligned} E = \{&(([\text{cut}]_1, [\text{ax}]_1), qx_1q), (([\text{cut}]_1, [\text{ax}]_1), qx_2), (([\text{cut}]_1, [\text{ax}]_2), qx_1p), \\ &(([\text{cut}]_1, [\text{ax}]_2), p), (([\text{cut}]_1, [\text{ax}]_3), q!q), (([\text{cut}]_1, [\text{ax}]_3), q!p), \\ &(([\text{cut}]_1, [\text{ax}]_4), p), (([\text{t}]_1, [\text{ax}]_4), 1)\}. \end{aligned}$$

How to control reduction

In a way similar to that of classical SECD machines we define the state of the machine in terms of four components:

- a **stack** S, which is used to store the current action;
- an **environment** E, is a node of the graph and it provides the local environment where the current action has to be performed;
- a **control** C is the stream of all actions either provided as initial input or created during the execution of other actions, it has to be executed in the context of the graph stored in the memory of the machine;
- a **dump** D corresponds to the current graph and represents the global environment for all future actions.

Transitions are therefore given as

$$(S, E, C, D) \overset{\tau}{\mapsto} (S', E', C', D')$$

Full Combustion

0. If $(S, E, C, D) = (\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset)$ then the machine is in its initial state.
   The initialisation step is

$$(\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset) \overset{\tau}{\mapsto}$$

Full Combustion

0. If $(S, E, C, D) = (\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset)$ then the machine is in its initial state.
   The initialisation step is

$$(\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset) \overset{\tau}{\mapsto} (\langle\rangle, \mathrm{NULL}, \mathrm{read}(), \emptyset).$$

   $\mathrm{read}()$ returns a stream of actions corresponding to the coding of the input in the form of a polarised dynamic graph.

Full Combustion

0. If $(S, E, C, D) = (\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset)$ then the machine is in its initial state.
   The initialisation step is
   $$(\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset) \overset{\tau}{\mapsto} (\langle\rangle, \mathrm{NULL}, \mathrm{read}(), \emptyset).$$
   $\mathrm{read}()$ returns a stream of actions corresponding to the coding of the input in the form of a polarised dynamic graph.
1. If v is a node and **its view is not empty** $C_v \neq \mathbf{0}$,

Full Combustion

0. If $(S, E, C, D) = (\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset)$ then the machine is in its initial state.
   The initialisation step is
   $$(\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset) \overset{\tau}{\mapsto} (\langle\rangle, \mathrm{NULL}, \mathrm{read}(), \emptyset).$$
   $\mathrm{read}()$ returns a stream of actions corresponding to the coding of the input in the form of a polarised dynamic graph.
1. If v is a node and **its view is not empty** $C_v \neq \mathbf{0}$, we have a reordering of actions σ such that $C = \sigma(C' \ltimes C_v)$ and $C' \approx \sigma(C' \ltimes \mathbf{0})$, then

Full Combustion

0. If $(S, E, C, D) = (\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset)$ then the machine is in its initial state.
   The initialisation step is
   $$(\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset) \overset{\tau}{\mapsto} (\langle\rangle, \mathrm{NULL}, \mathrm{read}(), \emptyset).$$
   $\mathrm{read}()$ returns a stream of actions corresponding to the coding of the input in the form of a polarised dynamic graph.
1. If v is a node and **its view is not empty** $C_v \neq \mathbf{0}$, we have a reordering of actions σ such that $C = \sigma(C' \ltimes C_v)$ and $C' \approx \sigma(C' \ltimes \mathbf{0})$, then
   $$(\langle\rangle, \mathrm{NULL}, C, D) \overset{\tau}{\mapsto}$$

Full Combustion

0. If $(S, E, C, D) = (\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset)$ then the machine is in its initial state.
   The initialisation step is
   $$(\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset) \overset{\tau}{\mapsto} (\langle\rangle, \mathrm{NULL}, \mathrm{read}(), \emptyset).$$
   $\mathrm{read}()$ returns a stream of actions corresponding to the coding of the input in the form of a polarised dynamic graph.
1. If v is a node and **its view is not empty** $C_v \neq \mathbf{0}$, we have a reordering of actions σ such that $C = \sigma(C' \ltimes C_v)$ and $C' \approx \sigma(C' \ltimes \mathbf{0})$, then
   $$(\langle\rangle, \mathrm{NULL}, C, D) \overset{\tau}{\mapsto} (C_v, (\{v\}, \emptyset), C', D)$$

Full Combustion

0. If $(S, E, C, D) = (\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset)$ then the machine is in its initial state.
   The initialisation step is
   $$(\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset) \overset{\tau}{\mapsto} (\langle\rangle, \mathrm{NULL}, \mathrm{read}(), \emptyset).$$
   $\mathrm{read}()$ returns a stream of actions corresponding to the coding of the input in the form of a polarised dynamic graph.
1. If v is a node and **its view is not empty** $C_v \neq \mathbf{0}$, we have a reordering of actions σ such that $C = \sigma(C' \ltimes C_v)$ and $C' \approx \sigma(C' \ltimes \mathbf{0})$, then
   $$(\langle\rangle, \mathrm{NULL}, C, D) \overset{\tau}{\mapsto} (C_v, (\{v\}, \emptyset), C', D)$$
2. If $S = \alpha :: S'$, $E = (\{v\}, Y)$, and $\alpha = \langle(\varepsilon, \varepsilon_s), (v, v_s), w\rangle$

Full Combustion

0. If $(S, E, C, D) = (\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset)$ then the machine is in its initial state.
   The initialisation step is
   $$(\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset) \overset{\tau}{\mapsto} (\langle\rangle, \mathrm{NULL}, \mathrm{read}(), \emptyset).$$
   $\mathrm{read}()$ returns a stream of actions corresponding to the coding of the input in the form of a polarised dynamic graph.
1. If v is a node and **its view is not empty** $C_v \neq \mathbf{0}$, we have a reordering of actions σ such that $C = \sigma(C' \ltimes C_v)$ and $C' \approx \sigma(C' \ltimes \mathbf{0})$, then
   $$(\langle\rangle, \mathrm{NULL}, C, D) \overset{\tau}{\mapsto} (C_v, (\{v\}, \emptyset), C', D)$$
2. If $S = \alpha :: S'$, $E = (\{v\}, Y)$, and $\alpha = \langle(\varepsilon, \varepsilon_s), (v, v_s), w\rangle$ then
   $$(S, E, C, D) \overset{\tau}{\mapsto}$$

Full Combustion

0. If $(S, E, C, D) = (\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset)$ then the machine is in its initial state.
   The initialisation step is
   $$(\langle\rangle, \mathrm{NULL}, \mathrm{nil}, \emptyset) \overset{\tau}{\mapsto} (\langle\rangle, \mathrm{NULL}, \mathrm{read}(), \emptyset).$$
   $\mathrm{read}()$ returns a stream of actions corresponding to the coding of the input in the form of a polarised dynamic graph.
1. If v is a node and **its view is not empty** $C_v \neq \mathbf{0}$, we have a reordering of actions σ such that $C = \sigma(C' \ltimes C_v)$ and $C' \approx \sigma(C' \ltimes \mathbf{0})$, then
   $$(\langle\rangle, \mathrm{NULL}, C, D) \overset{\tau}{\mapsto} (C_v, (\{v\}, \emptyset), C', D)$$

2. If $S = \alpha :: S'$, $E = (\{v\}, Y)$, and $\alpha = \langle(\varepsilon, \varepsilon_s), (v, v_s), w\rangle$ then
   $$(S, E, C, D) \overset{\tau}{\mapsto} (S', (\{v\}, Y \cup \{\alpha\}), C \ltimes \mathrm{execute}_v(\alpha), D),$$

3. When the stack is empty we set the environment to NULL:
   $$(\langle\rangle, E, C, D) \overset{\tau}{\mapsto} (\langle\rangle, \mathrm{NULL}, C, D)$$
   and continue with step 1.

Synchronous Machine

0. **read** from input stream

$$(\mathbf{0} \otimes \mathbf{0}, \mathrm{NULL} \otimes \mathrm{NULL}, \mathrm{nil} \otimes \mathrm{nil}, \emptyset \otimes \emptyset) \mapsto (\mathbf{0} \otimes \mathbf{0}, \mathrm{NULL} \otimes \mathrm{NULL}, \mathrm{read}() \otimes \mathrm{nil}, \emptyset \otimes \emptyset)$$

1. actions α_1 and α_2 are synchronously **extracted from streams** C_1 and C_2

$$(\mathbf{0} \otimes \mathbf{0}, \mathrm{NULL} \otimes \mathrm{NULL}, \alpha_1 :: C_1' \otimes \alpha_2 :: C_2', D_1 \otimes D_2) \mapsto (\alpha_1 \otimes \alpha_2, \mathrm{NULL} \otimes \mathrm{NULL}, C_1' \otimes C_2', D_1 \otimes D_2)$$

Synchronous Machine (cont.)

3. simultaneous **environment access** for both actions:

$$(\alpha_1 \otimes \alpha_2, \mathrm{NULL} \otimes \mathrm{NULL}, C_1 \otimes C_2, D_1 \otimes D_2) \mapsto (\alpha_1 \otimes \alpha_2, v_t^1 \otimes v_t^2, C_1 \otimes C_2, D_1' \otimes D_2')$$

when $\alpha_i = \langle \epsilon_i, e_i, w_i \rangle$ and either $e_i = (v_t^i, v_s^i)$ or v_t^i is undefined if $\alpha_i = \mathbf{0}$ then

$$D_i' = \begin{cases} D_i & \text{if } v_t^i \text{ already is a node of } D^i, \\ D_i \cup \{v_t^i\} & \text{if } v_t^i \text{ is a new node to be added to } D^i. \end{cases}$$

4. **actions execution**

$$(\alpha_1 \otimes \alpha_2, v_t^1 \otimes v_t^2, C_1 \otimes C_2, D_1 \otimes D_2) \mapsto$$
$$(\mathbf{0} \otimes \mathbf{0}, \mathrm{NULL} \otimes \mathrm{NULL}, ((C_1 \otimes \mathrm{execute}_1(\alpha_1)) \otimes \mathrm{execute}_1(\alpha_2)) \otimes$$
$$\otimes ((C_2 \otimes \mathrm{execute}_2(\alpha_1)) \otimes \mathrm{execute}_2(\alpha_2)), D_1' \otimes D_2')$$

The graph $D_i' = D_i \cup ((v_t^i, v_s^i)^{\epsilon_i}), w_i)$.

Aynchronous Machine

The state of the asynchronous machine is annotated with the scheduled processing unit:

$$(p, S, E, C, D) = (p, S_1 \otimes S_2, E_1 \otimes E_2, C_1 \otimes C_2, D_1 \otimes D_2)$$

where $p \in \{1, 2\}$ is the order number of the scheduled processor.

The sequence of controls p is by itself a stream (of integers $\{1, 2\}$). We may either choose a random sequence or we may force a particular scheduling by explicitly giving it.

Asynchronous parallel SECD

0. **reading** from the input interface:

$$(1, \mathbf{0} \otimes \mathbf{0}, \mathrm{NULL} \otimes \mathrm{NULL}, \mathrm{nil} \otimes \mathrm{nil}, \emptyset \otimes \emptyset) \mapsto (1, \mathbf{0} \otimes \mathbf{0}, \mathrm{NULL} \otimes \mathrm{NULL}, \mathrm{read}() \otimes \mathrm{nil}, \emptyset \otimes \emptyset)$$

1. action α_p **extraction from the stream** C_p:

$$(p, S_1 \otimes S_2, E_1 \otimes E_2, C_1 \otimes C_2, D_1 \otimes D_2) \mapsto (p', S_1' \otimes S_2', E_1' \otimes E_2', C_1' \otimes C_2', D_1' \otimes D_2')$$

if $S_p = \mathbf{0}$, $E_p = \mathrm{NULL}$, $C_p = \alpha_p :: C_p'$ then

$$S_i' = \begin{cases} S_i & \text{if } i \neq p \\ \alpha_i & \text{if } i = p \end{cases}$$

$E_i' = E_i$, $C_i' = C_i$ if $i \neq p$ and $D_i' = D_i$, finally p' is taken in accord to the scheduling function.

Asynchronous parallel SECD (cont.)

2. action α_p's **environment access**:

$$(p, S_1 \otimes S_2, E_1 \otimes E_2, C_1 \otimes C_2, D_1 \otimes D_2) \mapsto (p', S_1' \otimes S_2', E_1' \otimes E_2', C_1' \otimes C_2', D_1' \otimes D_2')$$

when $S_p = \alpha_p = \langle \epsilon_p, e_p, w_p \rangle$, where

$$E_i' = \begin{cases} E_i & \text{if } i \neq p \\ v_t^p & \text{if } i = p \end{cases}$$

$S_i' = S_i$, $C_i' = C_i$ and

$$D_i' = \begin{cases} D_i & \text{if } i \neq p \text{ or } i = p \text{ and } v_t^p \in D_p, \\ D_i \cup \{v_t^p\} & \text{if } i = p \text{ and } v_t^p \notin D_p. \end{cases}$$

Asynchronous parallel SECD (cont.)

3. **action execution**:

$$(p, S_1 \otimes S_2, E_1 \otimes E_2, C_1 \otimes C_2, D_1 \otimes D_2) \mapsto (p', S_1' \otimes S_2', E_1' \otimes E_2', C_1' \otimes C_2', D_1' \otimes D_2')$$

when $S_p = \alpha_p = \langle \epsilon_p, e_p, w_p \rangle$, $E_p = v_t^p$, then

$$S_i' = \begin{cases} S_i & \text{if } i \neq p \\ \mathbf{0} & \text{if } i = p \end{cases} \qquad E_i' = \begin{cases} E_i & \text{if } i \neq p \\ \text{NULL} & \text{if } i = p \end{cases}$$

$$C_i' = C_i \otimes \text{execute}_i(\alpha_p)$$

and the graph $D_i' = D_i$ for all $i \neq p$ and D_p' is obtained from D_p by adding the edge $((v_t^p, v_s^p)^{\epsilon_p}, w_p)$.

New ideas to extend the computation

The parallel machine can read from two different channels and execute "in parallel" reading the interpretation of two terms:

$$(\langle\rangle, \mathrm{NULL}, \mathsf{nil}, \emptyset) \overset{\tau}{\mapsto}$$

New ideas to extend the computation

The parallel machine can read from two different channels and execute "in parallel" reading the interpretation of two terms:

$$(\langle\rangle, \mathrm{NULL}, \mathsf{nil}, \emptyset) \overset{\tau}{\mapsto} (\langle\rangle, \mathrm{NULL}, \mathrm{read}(0) \oplus \mathrm{read}(1), \emptyset).$$

the only missing part is now the way we can read-back the two results which are independently evaluated by the machine.

Execution in the asynchronous version can now be extended to execute non-deterministic within a processor one of the two terms, so schedules are of type $p.b$ wher p is the **processor** and b is one of the two **subterms**.