Sequential and Parallel Abstract Machines for Optimal Reduction

Marco Pedicini (Roma Tre University) in collaboration with Mario Piazza (Univ. of Chieti-Pescara)

Proofs and Types 25 years later

Università Roma Tre, March 26 – 28, 2015
Alonzo Church in 1930’s introduced lambda-calculus as an alternative (with respect to recursive functions) model of computation.
Alonzo Church in 1930’s introduced lambda-calculus as an alternative (with respect to recursive functions) model of computation.

- Lambda Terms
  - Variables: $x, y, \ldots$ (discrete, denumerable-infinite set)
Alonzo Church in 1930’s introduced lambda-calculus as an alternative (with respect to recursive functions) model of computation.

- **Lambda Terms**
  - **Variables**: $x, y, \ldots$ (discrete, denumerable-infinite set)
  - **Application**: if $T$ and $U$ are lambda-terms then $(T)U$ is a lambda-term
Lambda Calculus

Alonzo Church in 1930’s introduced lambda-calculus as an alternative (with respect to recursive functions) model of computation.

• Lambda Terms
  • Variables: $x, y, \ldots$ (discrete, denumerable-infinite set)
  • Application: if $T$ and $U$ are lambda-terms then

  $$(T)U$$

  is a lambda-term

• Abstraction: if $x$ ia a variable and $U$ is a lambda term then

  $$\lambda x. U$$

  is a lambda term
Lambda Calculus

Alonzo Church in 1930’s introduced lambda-calculus as an alternative (with respect to recursive functions) model of computation.

• Lambda Terms
  • Variables: $x, y, \ldots$ (discrete, denumerable-infinite set)
  • Application: if $T$ and $U$ are lambda-terms then
    $$ (T)U \text{ is a lambda-term} $$

• Abstraction: if $x$ ia a variable and $U$ is a lambda term then
  $$ \lambda x. U \text{ is a lambda term} $$

• Term reduction as computing device:
Lambda Calculus

Alonzo Church in 1930’s introduced lambda-calculus as an alternative (with respect to recursive functions) model of computation.

- **Lambda Terms**
  - **Variables**: $x, y, \ldots$ (discrete, denumerable-infinite set)
  - **Application**: if $T$ and $U$ are lambda-terms then $(T)U$ is a lambda-term

- **Abstraction**: if $x$ ia a variable and $U$ is a lambda term then $\lambda x. U$ is a lambda term

- **Term reduction** as computing device:

  $$(\lambda x. U)V \rightarrow^\beta U[V/x]$$
Turing Completeness

- Lambda Definability of Recursive Functions: by encoding of integers as lambda-terms;

\[
\begin{align*}
0 & = \lambda f. \lambda x. x \\
1 & = \lambda f. \lambda x. (f)x \\
2 & = \lambda f. \lambda x. (f)(f)x \\
& \vdots \\
n & = \lambda f. \lambda x. (f)^n x
\end{align*}
\]
History

• At the beginning of digital computers in the 1950’s one of the first language was **lisp** by Mc Carthy (MIT)
History

- At the beginning of digital computers in the 1950’s one of the first language was **lisp** by Mc Carthy (MIT)
- Then in the 1960’s **functional programming languages** exploiting formal proofs of correctness were studied: **ML, erlang, scheme, clean, caml, …**
History

• At the beginning of digital computers in the 1950’s one of the first language was **lisp** by Mc Carthy (MIT)
• Then in the 1960’s **functional programming languages** exploiting formal proofs of correctness were studied: **ML**, **erlang**, **scheme**, **clean**, **caml**, …
• Nowdays functional languages are enriched with many special constructs which imperative languages cannot support (i.e. **clojure**, **scala**, **F#**).
GOI and PELCR

- **Geometry of Interaction** is the base of (a family of) semantics for programming languages (game semantics).
- GOI is (a kind of) operational semantics.
- GOI realized an algebraic theory for the **sharing of sub-expressions** and permitted the development of optimal lambda calculus reduction and a parallel evaluation mechanism based on a **local and asynchronous** calculus.

Optimal reduction was introduced in J.J. Levy’s PhD Thesis and defined (on sharing graphs) by J. Lamping in 1990.
TERMS as GRAPHS

We use to interpret a lambda term $M$ as its syntactic graph $[M]$:

$[(\lambda x.x)\lambda x.x] = \top$
Reduction Example

Syntactic tree of $\text{(λxx)λxx}$ (with binders).
We orient edges in accord to the five types of nodes and we introduce explicit nodes for variables. We also added sharing operators in order to manage duplications (even if unnecessary in this example for the linearity of $x$ in $\lambda xx$).
We introduce axiom and cut nodes to reconcile edge orientations.
We show one reduction step (the one corresponding to the beta-rule) the cut-node configuration must be removed and replaced by direct connections among the neighborhood nodes.
A reduction step may introduce new cuts (trivial ones in this case) but it consists essentially of the composition of paths in the graph.
Reduction Example
The so-called **Girard dynamic algebra** $\Lambda^*$ is the so-called **GOI monoid**, 
The so-called *Girard dynamic algebra* $\Lambda^*$ is the so-called *GOI monoid*, i.e., the free monoid
The so-called **Girard dynamic algebra** $\Lambda^*$ is the so-called **GOI monoid**, i.e., the free monoid with a morphism $!(.)$,
The so-called **Girard dynamic algebra** $\Lambda^*$ is the so-called **GOI monoid**, i.e., the free monoid with a morphism $!(.)$, an involution $(.)^*$.
The so-called **Girard dynamic algebra** $\Lambda^*$ is the so-called **GOI monoid**, i.e., the free monoid with a morphism $!(.)$, an involution $(.)^*$ and a zero,
The so-called **Girard dynamic algebra** \( \Lambda^* \) is the so-called **GOI monoid**, i.e., the free monoid with a morphism \(!(.).\), an involution \((.\)^*\) and a zero, generated by the following constants:
The so-called **Girard dynamic algebra** $\Lambda^*$ is the so-called **GOI monoid**, i.e., the free monoid with a morphism $!(.)$, an involution $(.)^*$ and a zero, generated by the following constants:

- $p$, $q$, and a family $W = (w_i)$ of exponential generators

Iterated morphism $!$ represents the applicative depth of the target node. The lift of an exponential operator corresponds to the difference of applicative depths between the source and target nodes.
The so-called **Girard dynamic algebra** $\Lambda^*$ is the so-called **GOI monoid**, i.e., the free monoid with a morphism $!(.)$, an involution $(.)^*$ and a zero, generated by the following constants:

- $p$, $q$, and a family $W = (w_i)$ of exponential generators such that for any $u \in \Lambda^*$:

\[(\text{annihilation}) \quad x^* y = \delta_{xy} \quad \text{for } x, y = p, q, w_i,\]
The so-called Girard dynamic algebra $\Lambda^*$ is the so-called GOI monoid, i.e., the free monoid with a morphism $!(.)$, an involution $(.)^*$ and a zero, generated by the following constants: $p$, $q$, and a family $W = (w_i)_i$ of exponential generators such that for any $u \in \Lambda^*$:

\begin{align}
(\text{annihilation}) \quad x^*y &= \delta_{xy} \quad \text{for } x, y = p, q, w_i, \\
(\text{swapping}) \quad !(u)w_i &= w_i!^{e_i}(u),
\end{align}
The so-called **Girard dynamic algebra** $\Lambda^*$ is the so-called **GOI monoid**, i.e., the free monoid with a morphism $!(.)$, an involution $(.)^*$ and a zero, generated by the following constants:

- $p$, $q$, and a family $W = (w_i)_i$ of exponential generators such that for any $u \in \Lambda^*$:

  - (annihilation) $x^* y = \delta_{xy}$ for $x, y = p, q, w_i$,
  - (swapping) $!(u)w_i = w_i!^{e_i}(u)$,

where $\delta_{xy}$ 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 will often write $w_{i,e_i}$ to explicitly note the lift of the generator.
The so-called **Girard dynamic algebra** $\Lambda^*$ is the so-called **GOI monoid**, i.e., the free monoid with a morphism $!(.)$, an involution $(.)^*$ and a zero, generated by the following constants: $p$, $q$, and a family $W = (w_i)_i$ of exponential generators such that for any $u \in \Lambda^*$:

**Annihilation**

$$x^* y = \delta_{xy} \quad \text{for } x, y = p, q, w_i,$$

**Swapping**

$$!(u) w_i = w_i !^{e_i}(u),$$

where $\delta_{xy}$ 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 will often write $w_{i,e_i}$ to explicitly note the lift of the generator.

**Iterated morphism** $!$ represents the **applicative depth** of the target node. The lift of an exponential operator corresponds to the **difference of applicative depths** between the source and target nodes.
Orienting annihilation and swapping equations from left to right, we get a rewriting system which is terminating and confluent.

The non-zero normal forms, known as stable forms, are the terms $ab^*$ where $a$ and $b$ are positive (i.e., written without $*$s).

The fact that all non-zero terms are equal to such an $ab^*$ form is referred to as the “$AB^*$ property”. From this, one easily gets that the word problem is decidable and that $\Lambda^*$ is an inverse monoid.

**Definition (Execution Formula)**

$$EX(R_T) = \sum_{\phi_{ij} \in \mathcal{P}(R_T)} W(\phi_{ij})$$

where $\phi_{ij}$ is the formal sum of all possible paths from node $i$ to node $j$. 
Evaluation as graph reduction technique: in the algebraic interpretation of interaction rules, a lambda term is interpreted as a **weighted graph**.

Parallel evaluation: the graph has to be distributed and we distribute its nodes (and edges), thus a lambda term represents the **program**, the evaluation **state** and the network of communication **channels**.

PELCR stands for **Parallel Environment for optimal Lambda Calculus Reduction** introduced in [PediciniQuaglia2007].
DD4 is the computation of the (shared) normal form of $\delta(\delta)4$ where $\delta := \lambda x (x)x$ and $4 := \lambda f \lambda x (f)^{4}x$. 

![Graph showing the relationship between number of processors and wall-clock time (seconds). The graph indicates a significant decrease in run time as the number of processors increases.](image-url)
DD4 SPEEDUP (speed vs number of PEs)
but... on this job (EXP3)
EXP3 - single CPU workload
EXP3 - run-time vs number of processors

![Graph showing the relationship between wall-clock time (seconds) and the number of processors. The time decreases significantly as the number of processors increases.]
EXP3 - workload on 4 CPUs
Super-linear speedup
Pluggers

Results in parallel execution were obtained by using a directed version of GOI, we shortly describe this version with the help of the algebra of unification.
Pluggers

Results in parallel execution were obtained by using a directed version of GOI, we shortly describe this version with the help of the algebra of unification.

Computation is performed by connecting pluggers:
Pluggers

Results in parallel execution were obtained by using a directed version of GOI, we shortly describe this version with the help of the algebra of unification.

Computation is performed by connecting pluggers:

Two kinds of pluggers, corresponding to polarities: + and −
Double Pluggers

Pluggers are present on both sides. There is a direction for composition.
Combinations of Double Pluggers

Pluggers on the two sides can be of any polarity: therefore we can have the following four types.
Plugging instructions

Pluggers can be connected by following instructions, here represented by terms.

\[ t = p(x) \quad \text{and} \quad u = p(p(x)) \]
Plugging instructions

Pluggers can be connected by following instructions, here represented by terms.

For instance, \( t = p(x) \) and \( u = p(p(x)) \).
Unification of terms is the performing criterion for plugging.
Unification of terms is the performing criterion for plugging.

If \( t = p(x_1) \) and \( u = p(p(x_2)) \), then \( \theta' = x_2 \rightarrow x_3 \) and \( \theta = x_1 \rightarrow p(x_3) \).
Execution Task

The minimal task during execution consists in connecting a plugger $u$ against all compatible pluggers (same color = same node) by following instructions $t_1, t_2, \ldots t_n$. 
Execution Task

The minimal task during execution consists in connecting a plugger $u$ against all compatible pluggers (same color = same node) by following instructions $t_1, t_2, \ldots t_n$. 
Execution Task

The minimal task during execution consists in connecting a plugger $u$ against all compatible pluggers (same color = same node) by following instructions $t_1, t_2, \ldots t_n$.

Pluggers whose connection works become new connection tasks (pairs of new double pluggers, to be connected) decorated with the corresponding residual instructions.
DVR - Directed Virtual Reduction

The way to perform GOI, we shortly introduced, is indeed the so called half-combustion strategy which is derived by DVR introduced by Danos et al. in 1997.

HC-combustion was implemented in PELCR before the PPDP paper (PediciniQuaglia2000) and then presented in PediciniQuaglia2007.
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.

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 asynchronous.
A bridging model

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

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 asynchronous*.

**Definition (PELCR Actions)**

Given a dynamic graph $G$, which is a graph $G = (V, E \subseteq V \times V)$ with edges labeled on the Girard dynamic algebra $\Lambda^*$, we define an action $\alpha$ on $G$ as $\langle \epsilon, e, w \rangle$ where $\epsilon \in \{+, -\}$, $e = (v_t, v_s)$ is a pair of nodes in $G$ and $w \in \Lambda^*$. 
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**.
  - An edge \(\alpha = (s, t, w)\), a signed edge \(\alpha^\varepsilon\) is an edge with a polarity \(\varepsilon \in \{+, -\}\); \(s\) and \(t\) are memory addresses, and \(w\) is a weight in the dynamic algebra.

- \(D\) represents the **current memory**, and contains **environment elements**.
  - any environment element has a memory address \(e_i\) and is called **node**.
  - memory \(e_i\) contains signed edges \(\alpha_i^\varepsilon\).
PELCR in SECD style

0 reading from the input interface:

\[(0, \text{NULL}, \text{nil}, \emptyset) \mapsto (0, \text{NULL}, \text{read}(), \emptyset)\]
PELCR in SECD style

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

\[(0, \text{NULL}, \text{nil}, \emptyset) \mapsto (0, \text{NULL}, \text{read()}, \emptyset)\]

1. **action** \(\alpha\) **extraction from stream** \(C\):

\[(0, \text{NULL}, \alpha :: C', D) \mapsto \begin{cases} 
(\alpha, \text{NULL}, C', D) & \text{if } \alpha \neq 0, \\
(0, \text{NULL}, C', D) & \text{if } \alpha = 0 
\end{cases}\]

where \(\alpha = \langle \epsilon, e, w \rangle\), the edge is \(e = (v_t, v_s)\) and \(D' = \begin{cases} 
D & \text{if } v_t \text{ already is a node of } D, \\
D \cup \{v_t\} & \text{if } v_t \text{ is a new node to be added to } D 
\end{cases}\).
PELCR in SECD style

0 reading from the input interface:

\((0, \text{NULL}, \text{nil}, \emptyset) \mapsto (0, \text{NULL}, \text{read}(), \emptyset)\)

1 action \(\alpha\) extraction from stream \(C\):

\((0, \text{NULL}, \alpha :: C', D) \mapsto \begin{cases} (\alpha, \text{NULL}, C', D) & \text{if } \alpha \neq 0, \\ (0, \text{NULL}, C', D) & \text{if } \alpha = 0 \end{cases}\)

2 action \(\alpha\)'s environment access:

\((\alpha, \text{NULL}, C, D) \mapsto (\alpha, v_t, C, D')\)

where \(\alpha = \langle \epsilon, e, w \rangle\), the edge is \(e = (v_t, v_s)\) and

\[ D' = \begin{cases} D & \text{if } v_t \text{ already is a node of } D, \\ D \cup \{v_t\} & \text{if } v_t \text{ is a new node to be added to } D. \end{cases} \]
4 action execution

\[ (\alpha, v_t, C, D) = \begin{cases} 
(0, \text{NULL}, C, D') & \text{if } X \text{ is empty} \\
(0, \text{NULL}, C \otimes X, D') & \text{if } X \neq \emptyset 
\end{cases} \]

where let be \( X = \text{execute}(\alpha) \) the set of residuals of the action \( \alpha \) on its context \( v_t^{-\epsilon} \) and \( D' = D \cup \{ ((v_t, v_s)^{\epsilon}, w) \} \)

Note that \( v'_i \) are new nodes introduced by the execution step, that can be freely allocated on one of the processing element.
Parallel Abstract Machines

We show a parallel machine with two computing units, whose state is therefore represented by

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

0 read from input stream

\[(0 \otimes 0, \text{NULL} \otimes \text{NULL}, \text{nil} \otimes \text{nil}, \emptyset \otimes \emptyset) \rightarrow (0 \otimes 0, \text{NULL} \otimes \text{NULL}, \text{read}() \otimes \text{nil}, \emptyset \otimes \emptyset)\]

1 actions \(\alpha_1\) and \(\alpha_2\) are synchronously extracted from streams \(C_1\) and \(C_2\)

\[(0 \otimes 0, \text{NULL} \otimes \text{NULL}, \alpha_1 :: C'_1 \otimes \alpha_2 :: C'_2, D_1 \otimes D_2) \rightarrow (\alpha_1 \otimes \alpha_2, \text{NULL} \otimes \text{NULL}, C'_1 \otimes C'_2, D_1 \otimes D_2)\]
Synchronous Machine (cont.)

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

\[(\alpha_1 \otimes \alpha_2, \text{NULL} \otimes \text{NULL}, C_1 \otimes C_2, D_1 \otimes D_2) \mapsto (\alpha_1 \otimes \alpha_2, v_t^1 \otimes v_t^2, C_1 \otimes C_2, D'_1 \otimes D'_2)\]

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 \(\alpha_i = 0\) 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

\[(\alpha_1 \otimes \alpha_2, v_t^1 \otimes v_t^2, C_1 \otimes C_2, D_1 \otimes D_2) \mapsto (0 \otimes 0, \text{NULL} \otimes \text{NULL}, ((C_1 \otimes \text{execute}_1(\alpha_1)) \otimes \text{execute}_1(\alpha_2)) \otimes \otimes ((C_2 \otimes \text{execute}_2(\alpha_1)) \otimes \text{execute}_2(\alpha_2)), D'_1 \otimes D'_2)\]

The graph \(D'_i = D_i \cup ((v_t^i, v_s^i)_{\epsilon_i}), w_i)\).
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:

\( (1, 0 \otimes 0, \text{NULL} \otimes \text{NULL}, \text{nil} \otimes \text{nil}, \emptyset \otimes \emptyset) \mapsto (1, 0 \otimes 0, \text{NULL} \otimes \text{NULL}, \text{read}() \otimes \text{nil}, \emptyset \otimes \emptyset) \)

1 **action** \( \alpha_p \) **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 \( S_p = 0, \ E_p = \text{NULL}, \ C_p = \alpha_p :: C_p' \) 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 $\alpha_p$'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

$$E'_i = \begin{cases} E_i & \text{if } i \neq p \\ v^p_t & \text{if } i = p \end{cases}$$

$S'_i = S_i, \ C'_i = C_i$ and

$$D'_i = \begin{cases} D_i & \text{if } i \neq p \text{ or } i = p \text{ and } v^p_t \in D_p, \\ D_i \cup \{v^p_t\} & \text{if } i = p \text{ and } v^p_t \notin D_p. \end{cases}$$
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 \( S_p = \alpha_p = \langle \epsilon_p, e_p, w_p \rangle \), \( E_p = v^p_t \), then

\[
S'_i = \begin{cases} 
S_i & \text{if } i \neq p \\
0 & \text{if } i = p
\end{cases} \quad E'_i = \begin{cases} 
E_i & \text{if } i \neq p \\
\text{NULL} & \text{if } i = p
\end{cases}
\]

\[C'_i = C_i \otimes \text{execute}_i(\alpha_p)\]

and the graph \( D'_i = D_i \) for all \( i \neq p \) and \( D'_p \) is obtained from \( D_p \) by adding the edge \(((v^p_t, v^p_s) \epsilon_p, w_p)\).
Stream equivalence

Definition (node-view (or view of base ν) of a stream of actions S)

Given a stream of actions S and a node ν we define the stream $S_ν$ by selecting actions with target node ν. More formally:

$$S_ν = \begin{cases} 0 & \text{if } S = 0 \\ S(0) :: \text{shift}(S)_ν & \text{if } S(0) = \langle \epsilon, (ν, ν_s), w \rangle \\ \text{shift}(S)_ν & \text{if } S(0) = \langle \epsilon, (v_t, ν_s), w \rangle \text{ and } ν \neq v_t \\ \text{shift}(S)_ν & \text{or } S(0) = 0 \end{cases}$$

Polarised view of base $ν^ε$ by selecting actions with the opposite polarity with respect to the polarity of the base. Namely:

$$S_{ν^ε} = \begin{cases} 0 & \text{if } S = 0 \\ S(0) :: \text{shift}(S)_{ν^ε} & \text{if } S(0) = \langle -ε, (ν, ν_s), w \rangle \\ \text{shift}(S)_{ν^ε} & \text{if } S(0) = \langle ε, (v_t, ν_s), w \rangle \text{ and } ν \neq v_t \\ \text{shift}(S)_{ν^ε} & \text{or } S(0) = 0 \end{cases}$$
Execution equivalence

**Definition**

The states \((S_1, E_1, C_1, D_1)\) and \((S_2, E_2, C_2, D_2)\) of two machines \(M_1\) and \(M_2\) are ordered w.r.t \(\preceq\) if

1. there is a graph-isomorphism \(\phi\) between \(D_1\) and a sub-graph of \(D_2\) such that the weights and polarities are preserved, and

2. for any node \(w \in \phi(D_1)\) we have that equivalent views on the controls (the two streams of actions) when taking \(v\) and its corresponding node \(\phi(v)\), \((C_1)_v \approx (C_2)_{\phi(v)}\), and

**Theorem**

*Given a (sequential) machine \(M_1\) and a (parallel) machine \(M_2\) such that \(M_1 \simeq_\sigma M_2\) by the isomorphism \(\phi\), then we have that \(v.M_1 \simeq_\sigma \phi(v).M_2\).*
LOAD BALANCING and AGGREGATION

Distribution the evaluation is obtained by

- **Processing Elements** (PE) with separate running PVMs;
- **Global Memory Address Space** for the environments;
- **Message Communication Layer** for streaming among PEs.

Issues we have considered:

- **Granularity**: fine grained vs. coarse grained;
- **Load Balancing**: liveness, avoid deadlocks.
ARCHITECTURE

- **Multicore**: the type of parallelism we considered is MIMD, and it behaves very well on modern multicore machines (super-linear speedup !!);
- **Vectorial**: there is space for further improving the evaluation strategy to cope with vectorial parallelism like in
  - Cell: evolution of the power-pc architecture developed by IBM-SONY-TOSHIBA (and used in BlueGene and PS3);
  - FPGA: arrays of programmable logic gates;
  - GPU: in graphics cards many computational cores can be executed.
Thanks!
Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza.
Distilling abstract machines.

Andrea Asperti and Juliusz Chroboczek.
Safe operators: Brackets closed forever optimizing optimal lambda-calculus implementations.

Andrea Asperti, Cecilia Giovanetti, and Andrea Naletto.
The Bologna Optimal Higher-order Machine.

V. Danos and L. Regnier.
Proof-nets and the hilbert space.


Peter J. Landin.  
The mechanical evaluation of expressions.  

Ian Mackie.  
The geometry of interaction machine.  

Marco Pedicini and Francesco Quaglia.  
PELCR: parallel environment for optimal lambda-calculus reduction.  

Laurent Regnier.  
*Lambda-calcul et réseaux*.  
J.J.M.M. Rutten.  
A tutorial on coinductive stream calculus and signal flow graphs.  

Leslie G. Valiant.  
A bridging model for multi-core computing.  