Sequential and Parallel Abstract Machines for Optimal Reduction

Marco Pedicini (Roma Tre University)
in collaboration with Mario Piazza (Univ. of Chieti) and
Giulio Pellitta (Univ. of Bologna)

DEIK – TCS Seminar 2014

DEIK, Debrecen (HU), November 11, 2014

Lambda Calculus

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

Lambda Calculus

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

- **Lambda Terms** (3 rules):
  - **Variables**: $x, y, \ldots$ (discrete, denumerable-infinite set)

Lambda Calculus

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

- **Lambda Terms** (3 rules):
  - **Variables**: $x, y, \dots$ (discrete, denumerable-infinite set)
  - **Application**: if T and U are lambda-terms then

$$(T)U \text{ is a lambda-term}$$

Lambda Calculus

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

- **Lambda Terms** (3 rules):
  - **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 $\lambda x.U$ 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** (3 rules):
  - **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 $\lambda x.U$ 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** (3 rules):
  - **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 $\lambda x.U$ 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;
-

$$\begin{aligned} \underline{0} &= \lambda f.\lambda x.x \\ \underline{1} &= \lambda f.\lambda x.(f)x \\ \underline{2} &= \lambda f.\lambda x.(f)(f)x \\ &\vdots \\ \underline{n} &= \lambda f.\lambda x.(f)^n x \end{aligned}$$

History

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

History

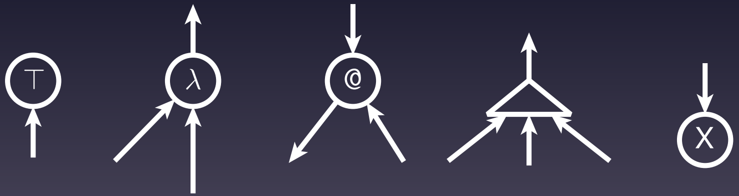
- At the beginning of digital computers in the 1950's one of the first language was **lisp** by Mc Carthy (MIT)
- Then in the 1960's **functional programming languages** exploiting formal proofs of correctness were studied: **ML**, **erlang**, **scheme**, **clean**, **caml**, …

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 familiy 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.

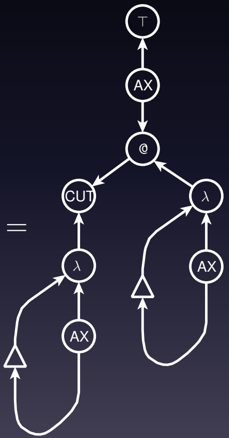


Optimal reduction was defined by J. Lamping in 1990.

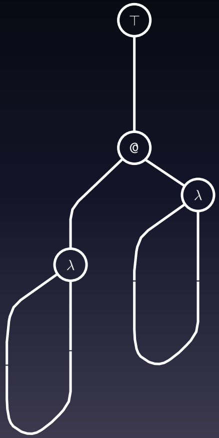
TERMS as GRAPHS

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

$[(\lambda x.x)\lambda x.x] \quad =$

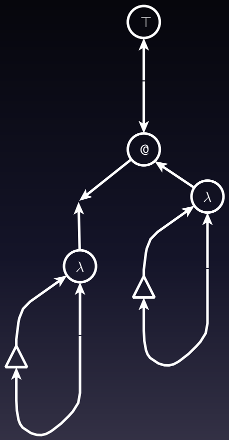


Reduction Example



Syntactic tree of $(\lambda xx)\lambda xx$ (with binders).

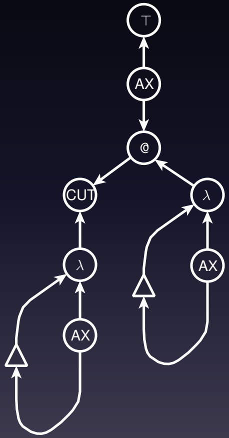
Reduction Example



We orient edges in accord to the five types of nodes and we introduce explicit nodes for variables.

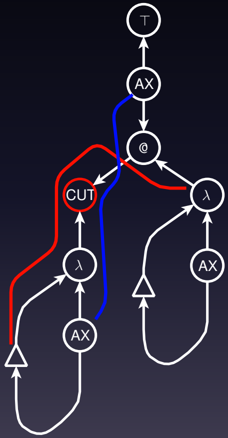
We also added sharing operators in order to manage duplications (even if unnecessary in this example for the linearity of x in λxx).

Reduction Example



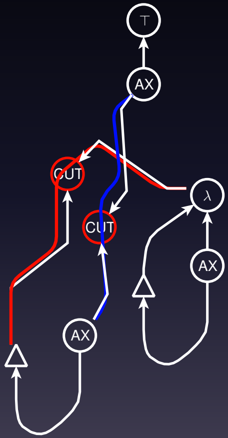
We introduce axiom and cut nodes to reconcile edge orientations.

Reduction Example



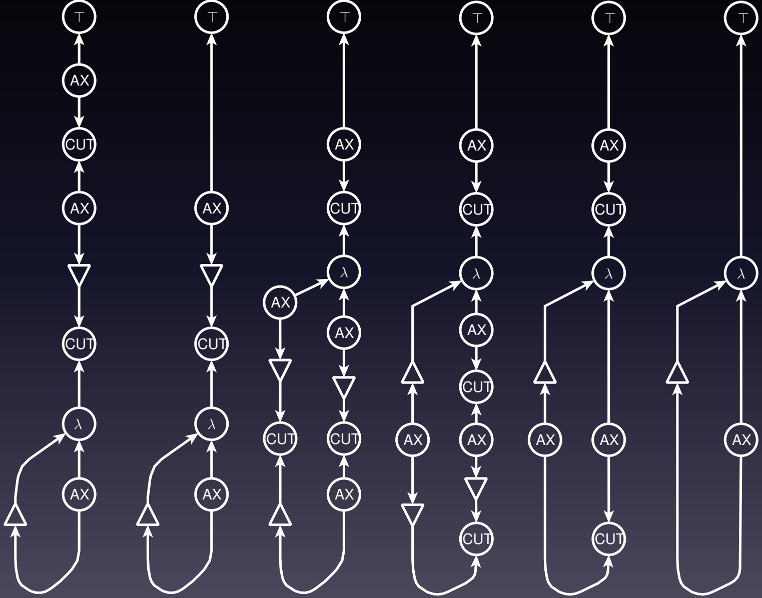
We show one reduction step (the one corresponding to the beta-rule) the cut-node configuration must be removed and replaced by direct connections among the neighborhood nodes.

Reduction Example



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



LAMBDA STAR

The so-called **Girard dynamic algebra** Λ^* is the so-called

GOI monoid,

LAMBDA STAR

The so-called **Girard dynamic algebra** Λ^* is the so-called

GOI monoid,

i.e., the free monoid

LAMBDA STAR

The so-called **Girard dynamic algebra** Λ^* is the so-called

GOI monoid,

i.e., the free monoid with a morphism $!(.)$,

LAMBDA STAR

The so-called **Girard dynamic algebra** Λ^* is the so-called

GOI monoid,

i.e., the free monoid with a morphism $!(.)$, an involution $(.)^*$

LAMBDA STAR

The so-called **Girard dynamic algebra** Λ^* is the so-called

GOI monoid,

i.e., the free monoid with a morphism $!(.)$, an involution $(.)^*$ and a zero,

LAMBDA STAR

The so-called **Girard dynamic algebra** Λ^* is the so-called

GOI monoid,

i.e., the free monoid with a morphism $!(.)$, an involution $(.)^*$ and a zero, generated by the following constants:

LAMBDA STAR

The so-called **Girard dynamic algebra** Λ^* 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

LAMBDA STAR

The so-called **Girard dynamic algebra** Λ^* 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 $u \in \Lambda^*$:

$$\textbf{(annihilation)} \qquad x^* y \quad = \quad \delta_{xy} \qquad \text{for } x, y = p, q, w_i,$$

LAMBDA STAR

The so-called **Girard dynamic algebra** Λ^* 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 $u \in \Lambda^*$:

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

LAMBDA STAR

The so-called **Girard dynamic algebra** Λ^* 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 $u \in \Lambda^*$:

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

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 will often write w_{i,e_i} to explicitly note the lift of the generator.

LAMBDA STAR

The so-called **Girard dynamic algebra** Λ^* 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 $u \in \Lambda^*$:

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

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 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.

STABLE FORMS and EXECUTION FORMULA

- 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 Λ^* is an inverse monoid.

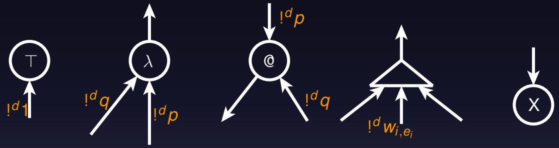
Definition (Execution Formula)

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

where ϕ_{ij} is the formal sum of all possible paths from node i to node j.

PELCR EVALUATION

- **Evaluation as graph reduction technique**: in the algebraic interpretation of interaction rules, a lambda term is interpreted as a **weighted graph**.

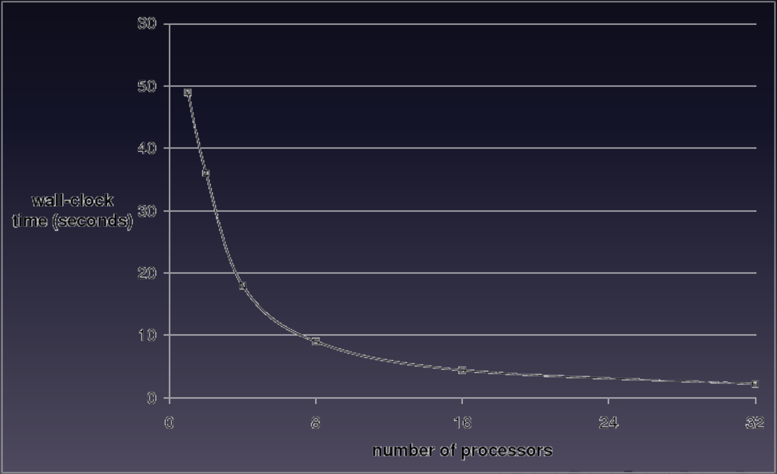


- **Parallel evaluation**: the graph has to be distributed and we distribute its nodes (and edges), thus a lambda term represents the **program**, the evaluation **state** and the network of communication **channels**.

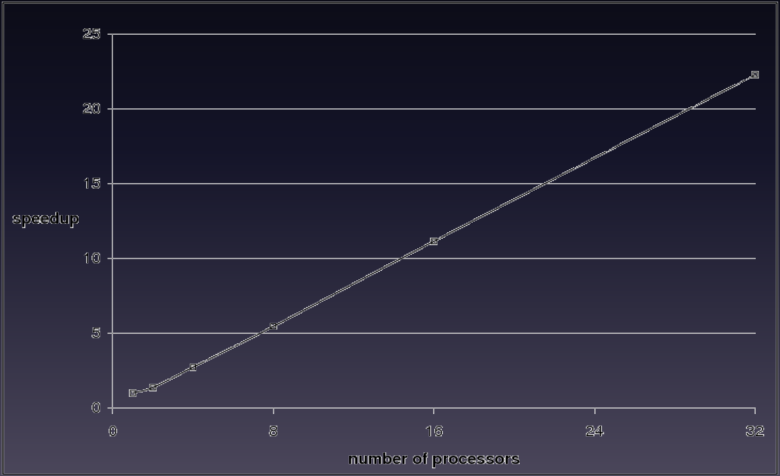
PELCR stands for **Parallel Environment for optimal Lambda Calculus Reduction** introduced in [PediciniQuaglia2007].

PELCR SPEEDUP (DD4 run time)

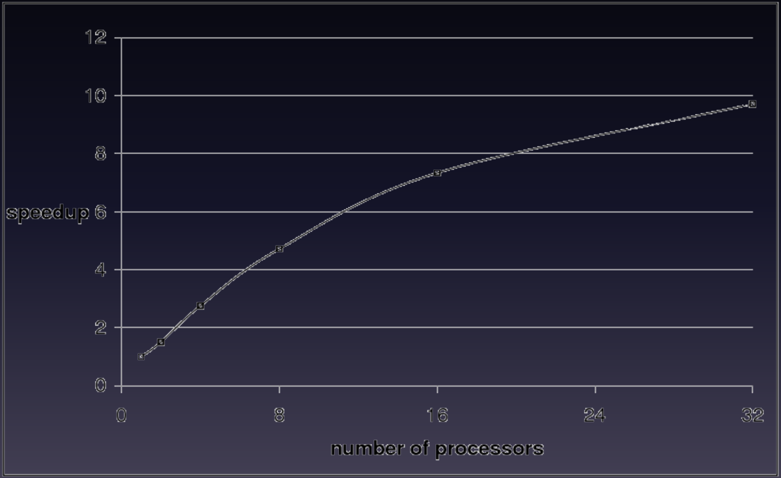
DD4 is the computation of the (shared) normal form of $(\delta)(\delta)\underline{4}$ where $\delta := \lambda x(x)x$ and $\underline{4} := \lambda f \lambda x(f)^4 x$.



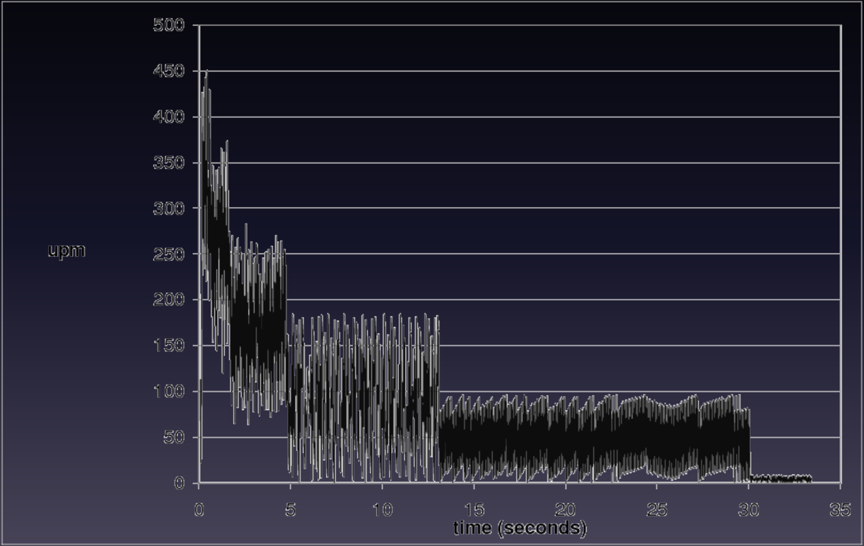
DD4 SPEEDUP (speed vs number of PEs)



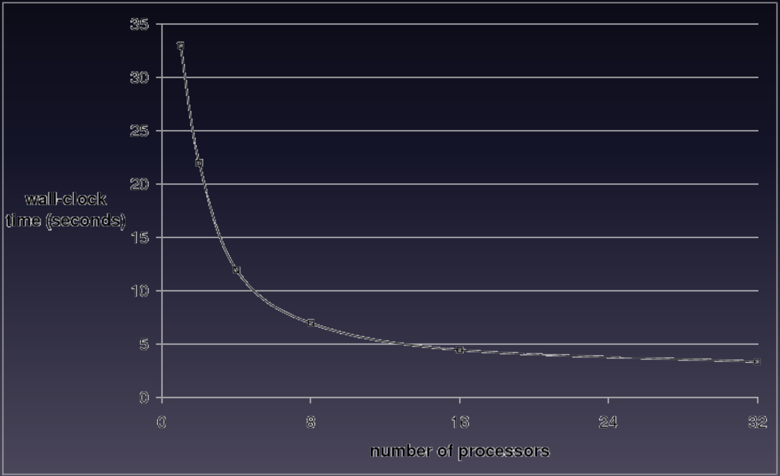
but... on this job (EXP3)



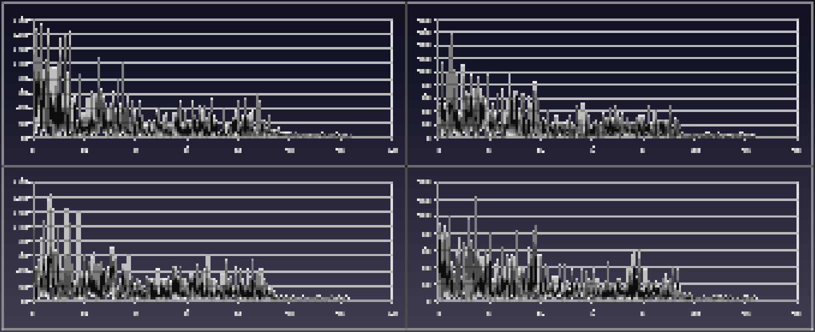
EXP3 - sigle CPU workload



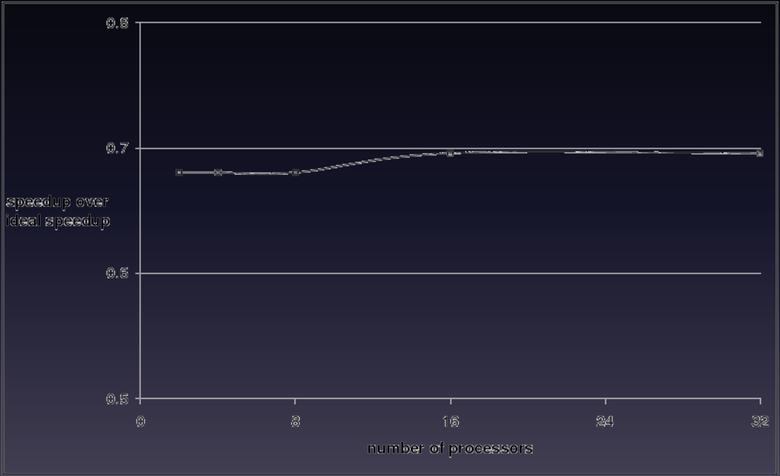
EXP3 - run-time vs number of processors



EXP3 - wordload on 4 CPUs



Super-linear speedup



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 asynchroous**.

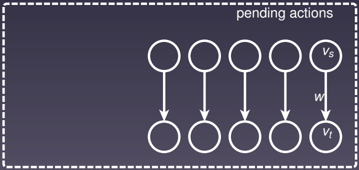
Definition (PELCR Actions)

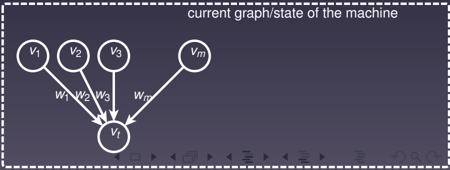
Given a dynamic graph G, which is a graph $G = (V, E \subset V \times V)$ with edges labeled on the Girard dynamic algebra Λ^*, we define an action α on G as $\langle \epsilon, e, w \rangle$ where $\epsilon \in \{+, -\}$, $e = (v_t, v_s)$ is a pair of nodes in G and $w \in \Lambda^*$.

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**.
  - An edge $\alpha = (s, t, w)$, a signed edge α^{ε} 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 $\alpha_i^{\varepsilon_i}$.





PELCR in SECD style

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

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

PELCR in SECD style

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

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

1. action α **extraction from stream** C:

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

PELCR in SECD style

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

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

1. action α **extraction from stream** C:

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

2. action α's **environment access**:

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

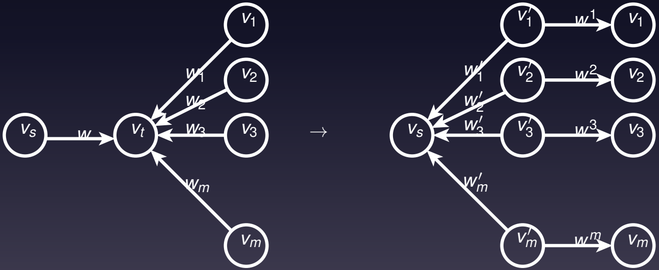
where $\alpha = \langle \epsilon, e, w \rangle$, the edge is $e = (v_t, v_s)$ and

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

4 action execution

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

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

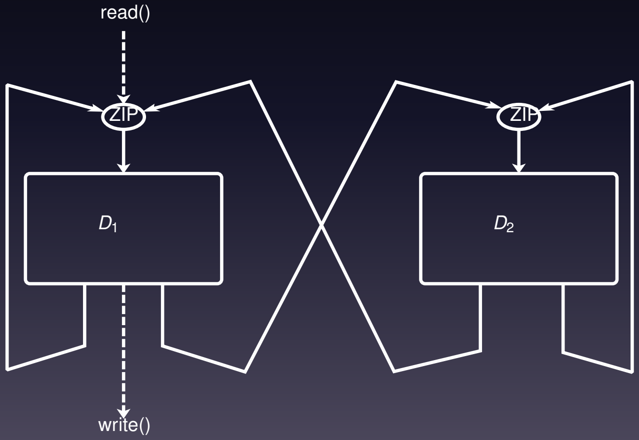


Note that v'_i are **new nodes** introduced by the execution step, that can be freely allocated on one of the processing element.

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

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

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:

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

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

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

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

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

Stream equivalence

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

Given a stream of actions S and a node v we define the stream S_v by selecting actions with target node v. More formally:

$$S_v = \begin{cases} \mathbf{0} & \text{if } S = \mathbf{0} \\ S(0) :: \mathrm{shift}(S)_v & \text{if } S(0) = \langle \epsilon, (v, v_s), w \rangle \\ \mathrm{shift}(S)_v & \text{if } S(0) = \langle \epsilon, (v_t, v_s), w \rangle \text{ and } v \neq v_t \\ & \text{ or } S(0) = \mathbf{0} \end{cases}$$

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

$$S_{v^\epsilon} = \begin{cases} \mathbf{0} & \text{if } S = \mathbf{0} \\ S(0) :: \mathrm{shift}(S)_{v^\epsilon} & \text{if } S(0) = \langle -\epsilon, (v, v_s), w \rangle \\ \mathrm{shift}(S)_{v^\epsilon} & \text{if } S(0) = \langle \epsilon, (v_t, v_s), w \rangle \text{ and } v \neq v_t \\ & \text{or } S(0) = \mathbf{0} \end{cases}$$

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 ϕ 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 ϕ, 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.

Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza.
Distilling abstract machines.
In *Proceedings of The 19th ACM SIGPLAN International Conference on Functional Programming*, 2014.

Andrea Asperti and Juliusz Chroboczek.
Safe operators: Brackets closed forever optimizing optimal lambda-calculus implementations.
Appl. Algebra Eng. Commun. Comput., 8(6):437–468, 1997.

Andrea Asperti, Cecilia Giovanetti, and Andrea Naletto.
The Bologna Optimal Higher-order Machine.
Journal of Functional Programming, 6(6):763–810, 1996.

V. Danos and L. Regnier.
Proof-nets and the hilbert space.
In *Advances in Linear Logic*, pages 307–328. Cambridge University Press, 1995.

Vincent Danos, Marco Pedicini, and Laurent Regnier.

Directed virtual reductions.
In *Computer science logic (Utrecht, 1996)*, volume 1258 of *Lecture Notes in Comput. Sci.*, pages 76–88. Springer, Berlin, 1997.

Vincent Danos and Laurent Regnier.
Local and asynchronous beta-reduction (an analysis of Girard's execution formula).
In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science (LICS 1993)*, pages 296–306. IEEE Computer Society Press, June 1993.

Jean-Yves Girard.
Geometry of interaction I. Interpretation of system **F**.
In *Logic Colloquium '88 (Padova, 1988)*, volume 127 of *Stud. Logic Found. Math.*, pages 221–260. North-Holland, Amsterdam, 1989.

Jean-Yves Girard.
Geometry of interaction II. Deadlock-free algorithms.
In *COLOG-88 (Tallinn, 1988)*, volume 417 of *Lecture Notes in Comput. Sci.*, pages 76–93. Springer, Berlin, 1990.

Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy.
The geometry of optimal lambda reduction.
In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–26, Albequerque, New Mexico, 1992.

J. Roger Hindley and Jonathan P. Seldin.
Introduction to combinators and λ-calculus, volume 1 of *London Mathematical Society Student Texts*.
Cambridge University Press, Cambridge, 1986.

Peter J. Landin.
The mechanical evaluation of expressions.
Computer Journal, 6(4):308–320, January 1964.

Ian Mackie.
The geometry of interaction machine.
In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 198–208. ACM, 1995.

- Marco Pedicini and Francesco Quaglia.
  PELCR: parallel environment for optimal lambda-calculus reduction.
  ACM Trans. Comput. Log., 8(3):Art. 14, 36, 2007.
- Laurent Regnier.
  Lambda-calcul et réseaux.
  PhD thesis, Paris 7, 1992.
- J.J.M.M. Rutten.
  A tutorial on coinductive stream calculus and signal flow graphs.
  Theoretical Computer Science, 343(3):443 – 481, 2005.
- Leslie G. Valiant.
  A bridging model for multi-core computing.
  J. Comput. System Sci., 77(1):154–166, 2011.