choreographies in practice

luís cruz-filipe

(joint work with fabrizio montesi)

department of mathematics and computer science
university of southern denmark

forte 2016
june 7th, 2016

outline

what are choreographies?

⇝ a model for distributed computation based on what is done "in practice"

- used for modeling interactions between web services
- high-level languages, alice-and-bob notation
- good properties: message pairing, deadlock-freedom
- projectable to adequate process calculi

a simple example

alice. “hi” $\rightarrow$ bob; bob. “hello” $\rightarrow$ alice

a simple example

$$\text{alice}.\text{“hi”} \rightarrow \text{bob};\ \text{bob}.\text{“hello”} \rightarrow \text{alice}$$

- all messages are correctly paired
- synthetizable process implementation

$$\underbrace{!\text{bob}.\text{“hi”};\ ?\text{bob}}_{\text{alice}} \mid \underbrace{?\text{alice};\ !\text{alice}.\text{“hello”}}_{\text{bob}}$$

a simple example

$$\text{alice}.\text{“hi”} \to \text{bob};\ \text{bob}.\text{“hello”} \to \text{alice}$$

- all messages are correctly paired
- synthetizable process implementation

$$\underbrace{!\text{bob}.\text{“hi”};\ ?\text{bob}}_{\text{alice}} \mid \underbrace{?\text{alice};\ !\text{alice}.\text{“hello”}}_{\text{bob}}$$

⇝ non-interfering communications in choreographies are allowed to *swap*, reflecting the process implementation

$$\text{alice}.\text{“hi”} \to \text{bob};\ \text{carol}.\text{“bye”} \to \text{di};\ \text{bob}.\text{“hello”} \to \text{alice}$$

a simple example

$$\text{alice.“hi”} \rightarrow \text{bob; bob.“hello”} \rightarrow \text{alice}$$

- all messages are correctly paired
- synthetizable process implementation

$$\underbrace{\text{!bob.“hi”; ?bob}}_{\text{alice}} \mid \underbrace{\text{?alice; !alice.“hello”}}_{\text{bob}}$$

$\rightsquigarrow$ non-interfering communications in choreographies are allowed to *swap*, reflecting the process implementation

$$\text{alice.“hi”} \rightarrow \text{bob; carol.“bye”} \rightarrow \text{di; bob.“hello”} \rightarrow \text{alice}$$
$$\equiv$$
$$\text{carol.“bye”} \rightarrow \text{di; alice.“hi”} \rightarrow \text{bob; bob.“hello”} \rightarrow \text{alice}$$
$$\equiv$$
$$\text{alice.“hi”} \rightarrow \text{bob; bob.“hello”} \rightarrow \text{alice; carol.“bye”} \rightarrow \text{di}$$

a simple example

$$\text{alice}.\text{“hi”} \to \text{bob};\ \text{bob}.\text{“hello”} \to \text{alice}$$

- all messages are correctly paired
- synthetizable process implementation

$$\underbrace{!\text{bob}.\text{“hi”};\ ?\text{bob}}_{\text{alice}} \mid \underbrace{?\text{alice};\ !\text{alice}.\text{“hello”}}_{\text{bob}}$$

⇝ non-interfering communications in choreographies are allowed to *swap*, reflecting the process implementation

$$\text{alice}.\text{“hi”} \to \text{bob};\ \text{carol}.\text{“bye”} \to \text{di};\ \text{bob}.\text{“hello”} \to \text{alice}$$

is implemented as

$$\underbrace{!\text{bob}.\text{“hi”};\ ?\text{bob}}_{\text{alice}} \mid \underbrace{?\text{alice};\ !\text{alice}.\text{“hello”}}_{\text{bob}} \mid \underbrace{!\text{di}.\text{“bye”}}_{\text{carol}} \mid \underbrace{?\text{carol}}_{\text{di}}$$

the world of choreographies

- common features (present in most languages)
- message passing
- conditional
- (tail) recursion
- label selection

the world of choreographies

- common features (present in most languages)
  - message passing
  - conditional
  - (tail) recursion
  - label selection
- additional features (only in particular languages)
  - channel passing
  - process spawning
  - asynchrony
  - web services
  - ...

the world of choreographies

- ⇝ common features (present in most languages)
  - message passing
  - conditional
  - (tail) recursion
  - label selection
- ⇝ additional features (only in particular languages)
  - channel passing
  - process spawning
  - asynchrony
  - web services
  - ...
- ⇝ the target process calculi reflect these design choices

our motivation

goal study foundational aspects of choreographies & identify minimal primitives required for particular constructions

- computational completeness
- asynchronous communication

our motivation

goal study foundational aspects of choreographies & identify minimal primitives required for particular constructions

- computational completeness
- asynchronous communication

$\rightsquigarrow$ "bottom-up" approach, rather than "top-down"

our motivation

goal study foundational aspects of choreographies & identify minimal primitives required for particular constructions

- computational completeness
- asynchronous communication

$\leadsto$ "bottom-up" approach, rather than "top-down"

this work

- what algorithms can we implement with current-day choreography languages?
- what primitives do we need to go beyond these limits?
- what can we *not* do?

outline

a model for programming

⇝ motivated by the intuition of parallel algorithms

a model for programming

$\rightsquigarrow$ motivated by the intuition of parallel algorithms

merge sort given a list ℓ

1. split ℓ into ℓ_1 and ℓ_2
2. compute mergesort(ℓ_1) and mergesort(ℓ_2)
3. merge mergesort(ℓ_1) and mergesort(ℓ_2)

a model for programming

$\rightsquigarrow$ motivated by the intuition of parallel algorithms

merge sort given a list ℓ

1. split ℓ into ℓ_1 and ℓ_2
2. compute mergesort(ℓ_1) and mergesort(ℓ_2)
3. merge mergesort(ℓ_1) and mergesort(ℓ_2)

$\rightsquigarrow$ step 2 should be done in two parallel computations

$\rightsquigarrow$ it is not clear how to do this with only tail recursion...

procedural choreographies

design options

- typed processes, hold only one value
- communication allows for computation by both parties
- general sequential composition
- parameterized global procedures
- process spawning

procedural choreographies

design options

- typed processes, hold only one value
- communication allows for computation by both parties
- general sequential composition
- parameterized global procedures
- process spawning

$$C ::= \eta; C \mid I; C \mid \mathbf{0} \qquad \mathcal{D} ::= \mathtt{X}(\tilde{\mathtt{q}}) = C, \mathcal{D} \mid \emptyset$$

$$\eta ::= \mathtt{p}.e \to \mathtt{q}.f \mid \mathtt{p} \to \mathtt{q}[\ell] \mid \mathtt{p}\ \mathtt{start}\ \mathtt{q} \mid \mathtt{p} : \mathtt{q} \leftrightarrow \mathtt{r}$$

$$I ::= \mathtt{if}\ \mathtt{p}.e\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 \mid \mathtt{X}\langle \tilde{\mathtt{p}} \rangle \mid \mathbf{0}$$

procedural choreographies

design options

- typed processes, hold only one value
- communication allows for computation by both parties
- general sequential composition
- parameterized global procedures
- process spawning

$$C ::= \eta; C \mid I; C \mid \mathbf{0} \qquad \mathcal{D} ::= \mathtt{X}(\tilde{\mathtt{q}}) = C, \mathcal{D} \mid \emptyset$$

$$\eta ::= \mathtt{p}.e \rightarrow \mathtt{q}.f \mid \mathtt{p} \rightarrow \mathtt{q}[\ell] \mid \mathtt{p}\ \mathtt{start}\ \mathtt{q} \mid \mathtt{p} : \mathtt{q} \leftrightarrow \mathtt{r}$$

$$I ::= \mathtt{if}\ \mathtt{p}.e\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 \mid \mathtt{X}\langle\tilde{\mathtt{p}}\rangle \mid \mathbf{0}$$

omitted

- type system to ensure correctness
- endpoint projection to procedural processes

outline

merge sort revisited

choreography

```
MS(p) = if p.is_small then 0
        else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
             MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

execution

```
MS<p>
```

merge sort revisited

choreography

```
MS(p) = if p.is_small then 0
        else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
             MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

execution

```
MS<p>
```

projection

```
MS(p) = if is_small then 0
        else start (q1 ▷ p?id; MS<q1>; p!c);
             start (q2 ▷ p?id; MS<q2>; p!c);
             q1!split1; q2!split2; q1?id; q2?merge
```

execution

```
p ▷ MS<p>
```

merge sort revisited

choreography

```
MS(p) = if p.is_small then 0
        else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
             MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

execution

```
if p.is_small then 0
else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
     MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

projection

```
MS(p) = if is_small then 0
        else start (q1 ▷ p?id; MS<q1>; p!c);
             start (q2 ▷ p?id; MS<q2>; p!c);
             q1!split1; q2!split2; q1?id; q2?merge
```

execution

```
p ▷ if is_small then 0
    else start (q1 ▷ p?id; MS<q1>; p!c);
         start (q2 ▷ p?id; MS<q2>; p!c);
         q1!split1; q2!split2; q1?id; q2?merge
```

merge sort revisited

choreography

```
MS(p) = if p.is_small then 0
        else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
             MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

execution

```
p start q1,q2; p.split1 -> q1; p.split2 -> q2;
MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

projection

```
MS(p) = if is_small then 0
        else start (q1 ▷ p?id; MS<q1>; p!c);
             start (q2 ▷ p?id; MS<q2>; p!c);
             q1!split1; q2!split2; q1?id; q2?merge
```

execution

```
p ▷ start (q1 ▷ p?id; MS<q1>; p!c);
    start (q2 ▷ p?id; MS<q2>; p!c);
    q1!split1; q2!split2; q1?id; q2?merge
```

merge sort revisited

choreography

```
MS(p) = if p.is_small then 0
        else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
             MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

execution

```
p.split1 -> q1; p.split2 -> q2;
MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

projection

```
MS(p) = if is_small then 0
        else start (q1 ▷ p?id; MS<q1>; p!c);
             start (q2 ▷ p?id; MS<q2>; p!c);
             q1!split1; q2!split2; q1?id; q2?merge
```

execution

```
p ▷ q1!split1; q2!split2; q1?id; q2?merge
q1 ▷ p?id; MS<q1>; p!c
q2 ▷ p?id; MS<q2>; p!c
```

merge sort revisited

choreography

```
MS(p) = if p.is_small then 0
        else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
             MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

execution

```
p.split1 -> q1; p.split2 -> q2;
MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

projection

```
MS(p) = if is_small then 0
        else start (q1 ▷ p?id; MS<q1>; p!c);
             start (q2 ▷ p?id; MS<q2>; p!c);
             q1!split1; q2!split2; q1?id; q2?merge
```

execution

```
p ▷ q1!split1; q2!split2; q1?id; q2?merge
q1 ▷ p?id; MS<q1>; p!c
q2 ▷ p?id; MS<q2>; p!c
```

merge sort revisited

choreography

```
MS(p) = if p.is_small then 0
        else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
             MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

execution

```
p.split2 -> q2;
MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

projection

```
MS(p) = if is_small then 0
        else start (q1 ▷ p?id; MS<q1>; p!c);
             start (q2 ▷ p?id; MS<q2>; p!c);
             q1!split1; q2!split2; q1?id; q2?merge
```

execution

```
p ▷ q2!split2; q1?id; q2?merge
q1 ▷ MS<q1>; p!c
q2 ▷ p?id; MS<q2>; p!c
```

merge sort revisited

choreography

```
MS(p) = if p.is_small then 0
        else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
             MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

execution

```
p.split2 -> q2;
MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

projection

```
MS(p) = if is_small then 0
        else start (q1 ▷ p?id; MS<q1>; p!c);
             start (q2 ▷ p?id; MS<q2>; p!c);
             q1!split1; q2!split2; q1?id; q2?merge
```

execution

```
p ▷ q2!split2; q1?id; q2?merge
q1 ▷ MS<q1>; p!c
q2 ▷ p?id; MS<q2>; p!c
```

merge sort revisited

choreography

```
MS(p) = if p.is_small then 0
        else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
             MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

execution

```
p.split2 -> q2;
MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

projection

```
MS(p) = if is_small then 0
        else start (q1 ▷ p?id; MS<q1>; p!c);
             start (q2 ▷ p?id; MS<q2>; p!c);
             q1!split1; q2!split2; q1?id; q2?merge
```

execution

```
p ▷ q2!split2; q1?id; q2?merge
q1 ▷ MS<q1>; p!c
q2 ▷ p?id; MS<q2>; p!c
```

merge sort revisited

choreography

```
MS(p) = if p.is_small then 0
        else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
             MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

execution

```
p.split2 -> q2; if q1.is_small then 0
else q1 start q11,q12; q1.split1 -> q11; q1.split2 -> q12;
     MS<q11>; MS<q12>; q11.c -> q1; q12.c -> q1.merge
MS<q2>; q1.c -> p; q2.c -> p.merge
```

projection

```
MS(p) = if is_small then 0
        else start (q1 ▷ p?id; MS<q1>; p!c);
             start (q2 ▷ p?id; MS<q2>; p!c);
             q1!split1; q2!split2; q1?id; q2?merge
```

execution

```
p ▷ q2!split2; q1?id; q2?merge
q1 ▷ if is_small then 0
     else start (q11 ▷ ...); start (q12 ▷ ...); ...
     p!c
q2 ▷ p?id; MS<q2>; p!c
```

merge sort revisited

choreography

```
MS(p) = if p.is_small then 0
        else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
             MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

execution

```
p.split2 -> q2;
q1 start q11,q12; q1.split1 -> q11; q1.split2 -> q12;
MS<q11>; MS<q12>; q11.c -> q1; q12.c -> q1.merge
MS<q2>; q1.c -> p; q2.c -> p.merge
```

projection

```
MS(p) = if is_small then 0
        else start (q1 ▷ p?id; MS<q1>; p!c);
             start (q2 ▷ p?id; MS<q2>; p!c);
             q1!split1; q2!split2; q1?id; q2?merge
```

execution

```
p ▷ q2!split2; q1?id; q2?merge
q1 ▷ start (q11 ▷ ...); start (q12 ▷ ...);
     q11!split1; q12!split2; q11?id; q12?merge; p!c

q2 ▷ p?id; MS<q2>; p!c
```

merge sort revisited

choreography

```
MS(p) = if p.is_small then 0
        else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
             MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

execution

```
p.split2 -> q2;
q1.split1 -> q11; q1.split2 -> q12;
MS<q11>; MS<q12>; q11.c -> q1; q12.c -> q1.merge
MS<q2>; q1.c -> p; q2.c -> p.merge
```

projection

```
MS(p) = if is_small then 0
        else start (q1 ▷ p?id; MS<q1>; p!c);
             start (q2 ▷ p?id; MS<q2>; p!c);
             q1!split1; q2!split2; q1?id; q2?merge
```

execution

```
p ▷ q2!split2; q1?id; q2?merge
q1 ▷ q11!split1; q12!split2
q11 ▷ q1?id; MS<q11>; q1!c
q12 ▷ q1?id; MS<q12>; q1!c
q2 ▷ p?id; MS<q2>; p!c
```

merge sort revisited

choreography

```
MS(p) = if p.is_small then 0
        else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
             MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

execution

```
MS<q11>; MS<q12>; q11.c -> q1; q12.c -> q1.merge
MS<q2>; q1.c -> p; q2.c -> p.merge
```

projection

```
MS(p) = if is_small then 0
        else start (q1 ▷ p?id; MS<q1>; p!c);
             start (q2 ▷ p?id; MS<q2>; p!c);
             q1!split1; q2!split2; q1?id; q2?merge
```

execution

```
p ▷ q1?id; q2?merge
q1 ▷ q11?id; q12?merge; p!c
q11 ▷ MS<q11>; q1!c
q12 ▷ MS<q12>; q1!c
q2 ▷ MS<q2>; p!c
```

merge sort revisited

choreography

```
MS(p) = if p.is_small then 0
        else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
             MS<q1>; MS<q2>; q1.c -> p; q2.c -> p.merge
```

execution

```
MS<q11>; MS<q12>; q11.c -> q1; q12.c -> q1.merge
MS<q2>; q1.c -> p; q2.c -> p.merge
```

projection

```
MS(p) = if is_small then 0
        else start (q1 ▷ p?id; MS<q1>; p!c);
             start (q2 ▷ p?id; MS<q2>; p!c);
             q1!split1; q2!split2; q1?id; q2?merge
```

execution

```
p ▷ q1?id; q2?merge
q1 ▷ q11?id; q12?merge; p!c
q11 ▷ MS<q11>; q1!c
q12 ▷ MS<q12>; q1!c
q2 ▷ MS<q2>; p!c
```

gaussian elimination

goal given a system of linear equations in matrix form

$$A\vec{x} = \vec{b}$$

transform it into an equivalent one

$$U\vec{x} = \vec{b}'$$

where U is upper triangular

gaussian elimination

goal given a system of linear equations in matrix form

$$A\vec{x} = \vec{b}$$

transform it into an equivalent one

$$U\vec{x} = \vec{b}'$$

where U is upper triangular

algorithm gauss elimination (naive)

1. divide the first row of A^+ by a_{11}
2. subtract $a_{k1} \times A_1^+$ from A_k^+
3. ignore the first row and column of A^+ and repeat

$\rightsquigarrow$ A^+ is the extended matrix $\left[A|\vec{b}\right]$

thoughts on implementation

main idea each entry is stored in one process

thoughts on implementation

main idea each entry is stored in one process

- the size of the matrix changes
- need procedures with variable number of arguments

thoughts on implementation

main idea each entry is stored in one process

- the size of the matrix changes
- need procedures with variable number of arguments

extension we need to enrich the choreography language

- procedures can have *lists* as arguments
- lists are uniform in type
- lists are uniform in connections
- lists can only be used as arguments to pure functions in procedure calls
- calling a procedure with an empty list terminates

a naive implementation

implementation

```
gauss(A) =        solve(fst_row(A)); elim(fst_row(A),rest(A));
                  gauss(minor(A))

solve(A) =        divide_all(hd(A),tl(A)); set_1(hd(A))

divide_all(a,A) = divide(a,hd(A)); divide_all(a,tl(A))
divide(a,b) =     a.c → b.div

elim(A,B) =       elim_row(A,fst_row(B)); elim(A,rest(B))
elim_row(A,B) =   elim_all(tl(A),hd(B),tl(B)); set_0(hd(B))
elim_all(A,m,B) = elim_1(hd(A),m,hd(B)); elim_all(tl(A),m,tl(B))
elim_1(a,m,b) =   b start x; b:x ↔ a; b:x ↔ m;
                  a.c → x.id; m.c → x.mult; x.c → b.minus

set_0(a) =        a start p; p.0 → a.id
set_1(a) =        a start p; p.1 → a.id
```

a naive implementation

implementation

```
gauss(A) =         solve(fst_row(A)); elim(fst_row(A),rest(A));
                   gauss(minor(A))

solve(A) =         divide_all(hd(A),tl(A)); set_1(hd(A))

divide_all(a,A) =  divide(a,hd(A)); divide_all(a,tl(A))
divide(a,b) =      a.c → b.div

elim(A,B) =        elim_row(A,fst_row(B)); elim(A,rest(B))
elim_row(A,B) =    elim_all(tl(A),hd(B),tl(B)); set_0(hd(B))
elim_all(A,m,B) =  elim_1(hd(A),m,hd(B)); elim_all(tl(A),m,tl(B))
elim_1(a,m,b) =    b start x; b:x ↔ a; b:x ↔ m;
                   a.c → x.id; m.c → x.mult; x.c → b.minus

set_0(a) =         a start p; p.0 → a.id
set_1(a) =         a start p; p.1 → a.id
```

- standard programming techniques
- implicit parallelism yields concurrent semantics
- pipelined communication and computation

next step: graphs

goal implement standard graph algorithms

- broadcast to all
- minimum spanning tree
- vertex coloring

next step: graphs

goal implement standard graph algorithms

- broadcast to all
- minimum spanning tree
- vertex coloring

problem no primitives to access connections at runtime

- requires encoding the graph in the algorithm
- does not allow for changes in the network
- (see example in paper for more details)

next step: graphs

goal implement standard graph algorithms

- broadcast to all
- minimum spanning tree
- vertex coloring

problem no primitives to access connections at runtime

- requires encoding the graph in the algorithm
- does not allow for changes in the network
- (see example in paper for more details)

solution? further extend communication primitives
$\rightsquigarrow$ requires drastic changes to the theory
(type system, underlying calculus, endpoint projection)

outline

conclusions

- concurrent algorithms in choreographies
  - mergesort and quicksort
  - gaussian elimination
  - fast fourier transform
  - broadcasting on graphs
- implicit parallelism often yields “good” behaviour
- next step: dynamically accessing the network structure

thank you!