procedural choreographic programming

luís cruz-filipe

(joint work with fabrizio montesi)

department of mathematics and computer science university of southern denmark

forte 2017, neuchâtel, switzerland june 20th, 2017

outline



1 choreographic programming





choreographies

chore ographies

a model for distributed computation based on "common 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

different usages

- choreographies as specifications (types)
- choreographies as programs (our approach)

$$examples$$
 (i/ii)

a simple example

alice. "hi" \rightarrow bob; bob. "hello" \rightarrow alice

all messages are correctly paired

synthetizable process implementation

non-interfering communications can swap

alice. "hi"
$$\rightarrow$$
 bob; carol. "bye" \rightarrow dan; bob. "hello" \rightarrow alice
 \equiv
carol. "bye" \rightarrow dan; alice. "hi" \rightarrow bob; bob. "hello" \rightarrow alice
 \equiv
alice. "hi" \rightarrow bob; bob. "hello" \rightarrow alice; carol. "bye" \rightarrow dan

is implemented as

$$\underbrace{!bob. "hi"; ?bob}_{alice} | \underbrace{?alice; !alice. "hello"}_{bob} | \underbrace{!dan. "bye"}_{carol} | \underbrace{?carol}_{dan}$$

the world of choreographies

common features (present in most languages)

◆□▶ ◆□▶ ◆三▶ ◆三▶ 三三 のへぐ

- message passing/method selection
- conditional and (tail) recursion

the world of choreographies

common features (present in most languages)

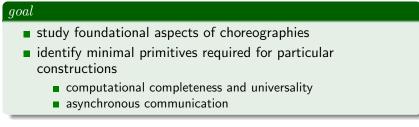
- message passing/method selection
- conditional and (tail) recursion

additional features (only in particular languages)

- channel passing
- process spawning
- asynchrony
- web services
- **.**..

 \rightsquigarrow the target process calculi reflect these design choices

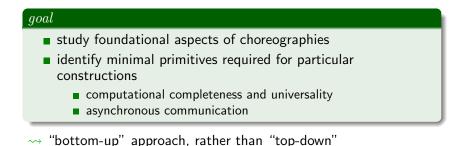
our motivation



◆□▶ ◆□▶ ◆三▶ ◆三▶ 三三 のへぐ

→ "bottom-up" approach, rather than "top-down"

our motivation



this work

- procedures in choreographies
- arbitrary composition and recursion
- runtime process spawning and name mobility

outline



choreographic programming

2 procedural choreographies

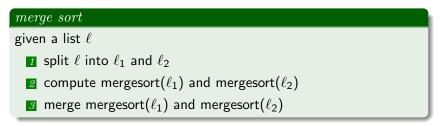
3 choreographies in practice





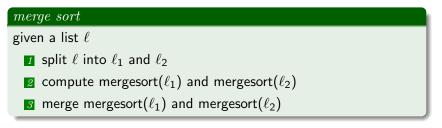
motivation

 \rightsquigarrow we want to be able to write intuitive parallel algorithms



motivation

 \rightsquigarrow we want to be able to write intuitive parallel algorithms



step 2 should be done in two parallel computations

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

$procedural\ choreographies$

design options

- typed processes
- each process holds only one value (but might be a record)

- communication allows for computation by both parties
- parameterized global procedures
- process spawning
- name mobility (three-way communication)

syntax

$procedural\ choreographies$

a procedural choreography is a pair $\langle \mathcal{D}, {\it C} \rangle$

$$\begin{split} \mathcal{C} &::= \eta; \mathcal{C} \mid I; \mathcal{C} \mid \boldsymbol{0} \qquad \mathcal{D} ::= \mathtt{X}(\tilde{\mathtt{q}}) = \mathcal{C}, \mathcal{D} \mid \emptyset \\ \eta &::= \mathtt{p}.e \rightarrow \mathtt{q}.f \mid \mathtt{p} \rightarrow \mathtt{q}[\ell] \mid \mathtt{p} \; \mathtt{start} \; \mathtt{q}^{\mathcal{T}} \mid \mathtt{p} : \mathtt{q} \leftrightarrow \mathtt{r} \\ \mathcal{I} &::= \mathtt{if} \; \mathtt{p}.e \; \mathtt{then} \; \mathcal{C}_1 \; \mathtt{else} \; \mathcal{C}_2 \mid \mathtt{X} \left\langle \widetilde{\mathtt{p}^{\mathsf{T}}} \right\rangle \mid \boldsymbol{0} \end{split}$$

▲□▶ ▲□▶ ▲□▶ ▲□▶ ▲□ ● ● ●

syntax

procedural choreographies

a procedural choreography is a pair $\langle \mathcal{D}, \mathcal{C} \rangle$

$$\begin{split} C &::= \eta; C \mid I; C \mid \mathbf{0} \qquad \mathcal{D} ::= \mathtt{X}(\tilde{\mathtt{q}}) = C, \mathcal{D} \mid \emptyset \\ \eta &::= \mathtt{p}.e \rightarrow \mathtt{q}.f \mid \mathtt{p} \rightarrow \mathtt{q}[\ell] \mid \mathtt{p} \; \mathtt{start} \; \mathtt{q}^T \mid \mathtt{p} : \mathtt{q} \leftrightarrow \mathtt{r} \\ I &::= \mathtt{if} \; \mathtt{p}.e \; \mathtt{then} \; C_1 \; \mathtt{else} \; C_2 \mid \mathtt{X} \left\langle \widetilde{\mathtt{p}^T} \right\rangle \mid \mathbf{0} \end{split}$$

▲□▶ ▲□▶ ▲□▶ ▲□▶ ▲□ ● ● ●

D is a set of procedure definitions (choreographies)
C is a distinguished ("main") choreography

syntax

procedural choreographies

a procedural choreography is a pair $\langle \mathcal{D}, \mathcal{C} \rangle$

$$\begin{split} \mathcal{C} &::= \eta; \mathcal{C} \mid I; \mathcal{C} \mid \boldsymbol{0} \qquad \mathcal{D} :::= \mathtt{X}(\tilde{\mathtt{q}}) = \mathcal{C}, \mathcal{D} \mid \emptyset \\ \eta &::= \mathtt{p}.e \rightarrow \mathtt{q}.f \mid \mathtt{p} \rightarrow \mathtt{q}[\ell] \mid \mathtt{p} \; \mathtt{start} \; \mathtt{q}^{\mathcal{T}} \mid \mathtt{p} : \mathtt{q} \leftrightarrow \mathtt{r} \\ \mathcal{I} &::= \mathtt{if} \; \mathtt{p}.e \; \mathtt{then} \; \mathcal{C}_1 \; \mathtt{else} \; \mathcal{C}_2 \mid \mathtt{X} \left\langle \widetilde{\mathtt{p}^{\mathtt{T}}} \right\rangle \mid \boldsymbol{0} \end{split}$$

- η are communication actions (values, label selection, spawning, name communication)
- I are instructions (conditionals, procedure calls) composed sequentially

syntax

procedural choreographies

a procedural choreography is a pair $\langle \mathcal{D}, \mathcal{C} \rangle$

$$\begin{split} \mathcal{C} &::= \eta; \mathcal{C} \mid I; \mathcal{C} \mid \boldsymbol{0} \qquad \mathcal{D} :::= \mathtt{X}(\tilde{\mathtt{q}}) = \mathcal{C}, \mathcal{D} \mid \emptyset \\ \eta &::= \mathtt{p}.e \rightarrow \mathtt{q}.f \mid \mathtt{p} \rightarrow \mathtt{q}[\ell] \mid \mathtt{p} \; \mathtt{start} \; \mathtt{q}^{\mathcal{T}} \mid \mathtt{p} : \mathtt{q} \leftrightarrow \mathtt{r} \\ \mathcal{I} &::= \mathtt{if} \; \mathtt{p}.e \; \mathtt{then} \; \mathcal{C}_1 \; \mathtt{else} \; \mathcal{C}_2 \mid \mathtt{X} \left\langle \widetilde{\mathtt{p}^{\mathtt{T}}} \right\rangle \mid \boldsymbol{0} \end{split}$$

• in $p.e \rightarrow q.f$, p evaluates expression e and sends its result to q

- e may refer to the value stored at p
- q applies function f to the value received and stores the result
- f may refer to the value stored at q

syntax

procedural choreographies

a procedural choreography is a pair $\langle \mathcal{D}, \mathcal{C} \rangle$

$$\begin{split} \mathcal{C} &::= \eta; \, \mathcal{C} \mid I; \, \mathcal{C} \mid \mathbf{0} \qquad \mathcal{D} ::= \mathtt{X}(\tilde{\mathtt{q}}) = \mathcal{C}, \mathcal{D} \mid \emptyset \\ \eta &::= \mathtt{p}.e \rightarrow \mathtt{q}.f \mid \mathtt{p} \rightarrow \mathtt{q}[\ell] \mid \mathtt{p} \; \mathtt{start} \; \mathtt{q}^T \mid \mathtt{p} : \mathtt{q} \leftrightarrow \mathtt{r} \\ I &::= \mathtt{if} \; \mathtt{p}.e \; \mathtt{then} \; \mathcal{C}_1 \; \mathtt{else} \; \mathcal{C}_2 \mid \mathtt{X} \left< \widetilde{\mathtt{p}^{\mathtt{T}}} \right> \mid \mathbf{0} \end{split}$$

- in $p : q \leftrightarrow r$, p *introduces* processes q and r
- afterwards, q and r can communicate directly
- these actions are required for distributed implementations

semantics

components

transition semantics over triples $\langle \mathcal{G}, \mathcal{C}, \sigma \rangle$, parameterized by \mathcal{D}

- \mathcal{G} is a graph of connections (who knows who)
- σ is a (total) state function (what is stored at each process)

◆□▶ ◆□▶ ◆三▶ ◆三▶ 三三 のへぐ

semantics

components

transition semantics over triples $\langle \mathcal{G}, \mathcal{C}, \sigma \rangle$, parameterized by \mathcal{D}

- $\blacksquare \mathcal{G}$ is a graph of connections (who knows who)
- σ is a (total) state function (what is stored at each process)

$$\frac{\mathbf{p} \xleftarrow{\mathbf{G}} \mathbf{q} \quad e \downarrow_{\mathbf{p}} v \quad f(v) \downarrow_{\mathbf{q}} w}{G, \mathbf{p}.e \to \mathbf{q}.f; C, \sigma \rightarrow_{\mathcal{D}} G, C, \sigma[\mathbf{q} \mapsto w]} \ \lfloor \mathsf{C} |\mathsf{Com}|$$

possible errors

- p and q do not know each other
- e or f is not well-typed
- v does not have the type expected by f

semantics

components

transition semantics over triples $\langle \mathcal{G}, \mathcal{C}, \sigma \rangle$, parameterized by \mathcal{D}

- \mathcal{G} is a graph of connections (who knows who)
- σ is a (total) state function (what is stored at each process)

$$\overline{\mathsf{G}, \texttt{p start } \texttt{q}^{\mathsf{T}}; \mathsf{C}, \sigma} \xrightarrow{}_{\mathcal{D}} \mathsf{G} \cup \{\texttt{p} \leftrightarrow \texttt{q}\}, \mathsf{C}, \sigma[\texttt{q} \mapsto \bot_{\mathsf{T}}]} [\mathsf{C}|\mathsf{Start}]$$

◆□▶ ◆□▶ ◆三▶ ◆三▶ 三三 のへぐ

semantics

components

transition semantics over triples $\langle \mathcal{G}, \mathcal{C}, \sigma \rangle$, parameterized by \mathcal{D}

- \mathcal{G} is a graph of connections (who knows who)
- σ is a (total) state function (what is stored at each process)

$$\frac{\mathsf{p} \xleftarrow{\mathsf{G}} \mathsf{q} \quad \mathsf{p} \xleftarrow{\mathsf{G}} \mathsf{r}}{\mathsf{G}, \mathsf{p}: \mathsf{q} \leftrightarrow \mathsf{r}; \mathsf{C}, \sigma \rightarrow_{\mathcal{D}} \mathsf{G} \cup \{\mathsf{q} \leftrightarrow \mathsf{r}\}, \mathsf{C}, \sigma} \ \lfloor \mathsf{C} |\mathsf{Tell}|$$

possible errors

p does not know q or r

semantics

components

transition semantics over triples $\langle \mathcal{G}, \mathcal{C}, \sigma \rangle$, parameterized by \mathcal{D}

- G is a graph of connections (who knows who)
- σ is a (total) state function (what is stored at each process)

$$\frac{X(\widetilde{q^{\mathsf{T}}}) = \mathsf{C}_X \in \mathcal{D}}{\mathtt{X}\left\langle \widetilde{p^{\mathsf{T}}} \right\rangle; \mathsf{C} \preceq_{\mathcal{D}} \mathsf{C}_X[\widetilde{p}/\widetilde{q}] \, ; \mathsf{C}}} \, \left\lfloor \mathsf{C} |\mathsf{Unfold} \right\rceil$$

possible errors

- procedure X is not defined
- the types of the ps and qs do not match

semantics

components

transition semantics over triples $\langle \mathcal{G}, \mathcal{C}, \sigma \rangle$, parameterized by \mathcal{D}

- \mathcal{G} is a graph of connections (who knows who)
- σ is a (total) state function (what is stored at each process)

implicit parallelism

swapping relation extended to instructions (including procedure calls)

examples to come

$typing \ system$

avoiding errors

well-typed choreographies are guaranteed never to encounter an error (deadlock-freedom by design)

- symbolic execution
- over-approximation (analyzes potentially unreachable code)

 judgements include connection "requirements" and "guarantees" for procedure calls

$typing \ system$

avoiding errors

well-typed choreographies are guaranteed never to encounter an error (deadlock-freedom by design)

- symbolic execution
- over-approximation (analyzes potentially unreachable code)

- judgements include connection "requirements" and "guarantees" for procedure calls
- type checking is decidable
- type inference is decidable
- typable choreographies may be unprojectable

outline



choreographic programming

2 procedural choreographies

3 choreographies in practice



・ロト・日本・ キョ・ キョ・ しょうくの

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.* -> p; q2.* -> p.merge
```

projection

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

merge sort revisited

chore ography

MS

projection

p ⊳ MS

▲□▶ ▲圖▶ ▲国▶ ▲国▶ 三国 - 釣ぬの

merge sort revisited

choreography

```
if p.is_small then 0
```

```
else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
```

```
MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge
```

projection

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

◆□ > ◆□ > ◆豆 > ◆豆 > ̄豆 = のへで

merge sort revisited

choreography

```
p start q1,q2; p.split1 -> q1; p.split2 -> q2;
```

```
MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge
```

projection

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

merge sort revisited

choreography

```
p.split1 -> q1; p.split2 -> q2;
```

```
MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge
```

projection

```
p > q1!split1; q2!split2; q1?id; q2?merge
```

```
q1 ▷ p?id; MS<q1>; p!*
```

q2 ▷ p?id; MS<q2>; p!*

```
ヘロア ヘロア ヘビア ヘビア
                æ
```

merge sort revisited

choreography

p.split1 -> q1 ; p.split2 -> q2;

MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge

projection

q2 ▷ p?id; MS<q2>; p!*

- p > q1!split1 ; q2!split2; q1?id; q2?merge

- q1 ▷ p?id ; MS<q1>; p!*

ヘロア ヘロア ヘビア ヘビア æ

merge sort revisited

choreography

 $p.split2 \rightarrow q2;$

MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge

projection

```
p > q2!split2; q1?id; q2?merge
```

```
. . . . . . .
```

```
q1 ⊳ MS<q1>; p!*
```

```
q2 ▷ p?id; MS<q2>; p!*
```

```
▲ロト ▲圖 と ▲目 と ▲目 と りつ
```

procedural choreographic programming

choreographies in practice

merge sort revisited

choreography

p.split2 \rightarrow q2;

MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge

p ▷ q2!split2 ; q1?id; q2?merge

projection

q1 ▷ MS<q1>; p!*

q2 ▷ p?id ; MS<q2>; p!*

merge sort revisited

choreography

 $p.split2 \rightarrow q2;$

MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge

projection

- p⊳ q2!split2 ; q1?id; q2?merge

q2 ▷ p?id ; MS<q2>; p!*

- q1 ⊳ MS<q1>; p!*

ヘロア 人間ア 人間ア 人間ア æ

merge sort revisited

chore ography

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.* -> q1; q12.* -> q1.merge
```

```
MS<q2>; q1.* -> p; q2.* -> p.merge
```

projection

```
p ▷ q2!split2 ; q1?id; q2?merge
```

```
q1 \triangleright if is_small then 0
```

else start (q11 ▷ ...); start (q12 ▷ ...); ...

p!*

merge sort revisited

choreography

 $p.split2 \rightarrow q2;$

```
q1 start q11 , q12 ; q1.split1 -> q11; q1.split2 -> q12;
```

```
MS<q11>; MS<q12>; q11.* -> q1; q12.* -> q1.merge
```

MS<q2>; q1.* -> p; q2.* -> p.merge

projection

```
p ▷ q2!split2 ; q1?id; q2?merge
```

```
q1 ▷ start (q11 ▷ ...); start (q12 ▷ ...);
```

q11!split1; q12!split2; q11?id; q12?merge; p!*

q2 ▷ p?id ; MS<q2>; p!*

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

projection

MS<q2>; q1.* -> p; q2.* -> p.merge

MS<q11>; MS<q12>; q11.* -> q1; q12.* -> q1.merge

q1.split1 -> q11 ; q1.split2 -> q12 ;

 $p.split2 \rightarrow q2;$

merge sort revisited

choreography

- choreographies in practice

q2 ⊳ MS<q2>; p!*

q12 ▷ MS<q12>; q1!*

q11 ▷ MS<q11>; q1!*

q1 > q11?id; q12?merge; p!*

p ⊳ q1?id; q2?merge

projection

```
MS<q2>; q1.* -> p; q2.* -> p.merge
```

MS<q11>; MS<q12>; q11.* -> q1; q12.* -> q1.merge

```
choreography
```

merge sort revisited

procedural choreographic programming — choreographies in practice

procedural choreographic programming

choreographies in practice

merge sort revisited

choreography

MS<q11>; MS<q12>; q11.* -> q1; q12.* -> q1.merge

MS<q2>; q1.* -> p; q2.* -> p.merge

p ▷ q1?id; q2?merge

q11 ⊳ MS<q11>; q1!* q12 ▷ MS<q12>; q1!* q2 ⊳ MS<q2> ; p!*

q1 > q11?id; q12?merge; p!*

projection

other examples

see our forte'16 paper

- gaussian elimination
 - naive implementation gives pipelined communication and computation

- fast fourier transform
 - naive implementation using an orchestrator

 \rightsquigarrow these examples use simple language extensions

- conclusions

outline



choreographic programming

2 procedural choreographies

3 choreographies in practice

4 conclusions

▲□▶ ▲圖▶ ▲≣▶ ▲≣▶ ▲国 ● ● ●

- conclusions

conclusions

- a minimalist choreography language including key primitives
 - procedure definition
 - runtime process spawning
 - name mobility
- type system (also) keeping track of connections between processes
 - decidable type checking
 - decidable type inference
- easy to extend for practical applications
- not included: asynchronous semantics (see our ice'17 paper)

- conclusions

thank you!

◆□▶ ◆□▶ ◆臣▶ ◆臣▶ 臣 のへぐ