Inferring Session Types in ML

Imperial Concurrency Workshop 2015

Carlo Spaccasassi, Vasileios Koutavas
Trinity College Dublin

Research supported, in part, by Science Foundation Ireland grant 13/RC/2094, and by MSR (MRL 2011-039).

- 

Session types

Binary session types guarantee that interactions at the two endpoints of a channel always follow some protocol

- K. Honda, *Types for dyadic interaction*. [CONCUR'93]
- S. Gay, M. Hole. *Types and subtypes for client-server interactions*. [ESOP'99]

Very active community:

- K. Honda, N. Yoshida, M. Carbone, *Multiparty asynchronous session types*. [POPL'08]
- C. Fournet, C. A. R. Hoare, S. K. Rajamani, J. Rehof. *Stuck-Free Conformance*. [CAV'04]
- N. Kobayashi, *A partially deadlock-free typed process calculus.* [TOPLAS'98]
- S. J. Gay, V. T. Vasconcelos. *Linear type theory for asynchronous session types.* [JFP'10]
- …

MLs primitives

- Core ML: `fn x => e, let x=e in e, if e then e else e…`
- Session primitives:

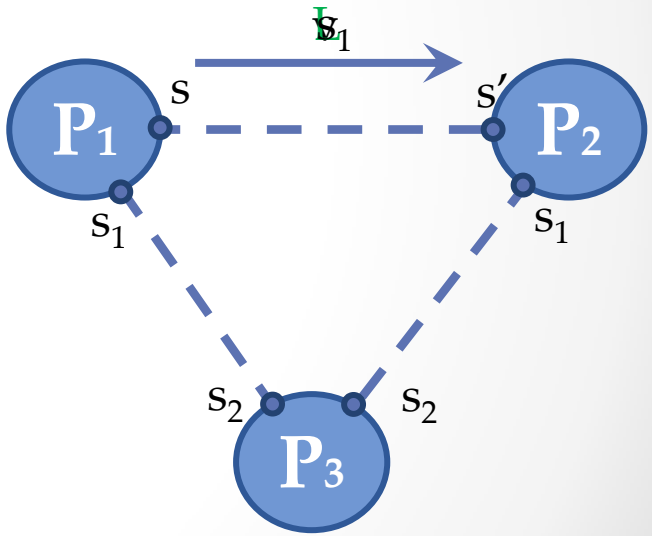
```
spawn (fn x => e)

request-c   accept-c
 c :: S     s :: S   s’:: S̄

  send s v    recv s’
 s :: !T.S     s’ :: ?T.S

select-L s    match s’ {Li => ei}
s :: !L.S      s’ :: <Li.Si> i in I

delegate s s’’   resume s’
s :: ![S’’].S     s’ :: ?[S’’].S’
s1 :: S’’
```



Swap channel

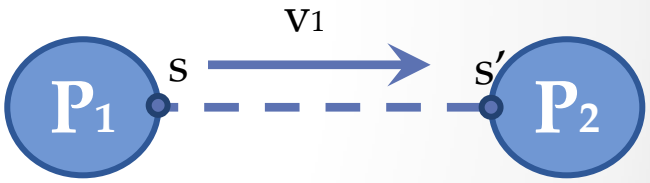
```
1 let doSwap v = let s = request-swap
2                in send s v; recv s
4 in
5 let doSwap' v = let s' = accept-swap
6                       v' = recv s'
7                 in send s' v; v'
8 in
9 spawn (fn x => doSwap v1); (doSwap' v2)
```

Swap channel

```
1 let doSwap v = let s = request-swap
2                in send s v; recv s
4 in
5 let doSwap' v = let s' = accept-swap
6                         v' = recv s'
7                 in send s' v; v'
8 in
9 spawn (fn x => doSwap v1); (doSwap' v2)
```

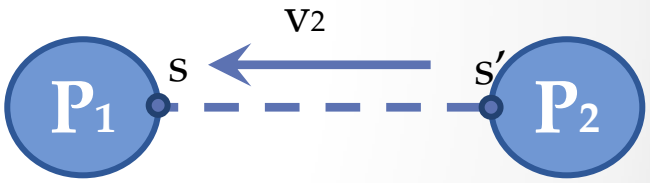
Swap channel

```
1 let doSwap v = let s = request-swap
2                in send s v; recv s
4 in
5 let doSwap' v = let s' = accept-swap
6                         v' = recv s'
7                 in send s' v; v'
8 in
9 spawn (fn x => doSwap v1); (doSwap' v2)
```



Swap channel

```
1 let doSwap v = let s = request-swap
2                in send s v; recv s
4 in
5 let doSwap' v = let s' = accept-swap
6                         v' = recv s'
7                 in send s' v; v'
8 in
9 spawn (fn x => doSwap v1); (doSwap' v2)
```

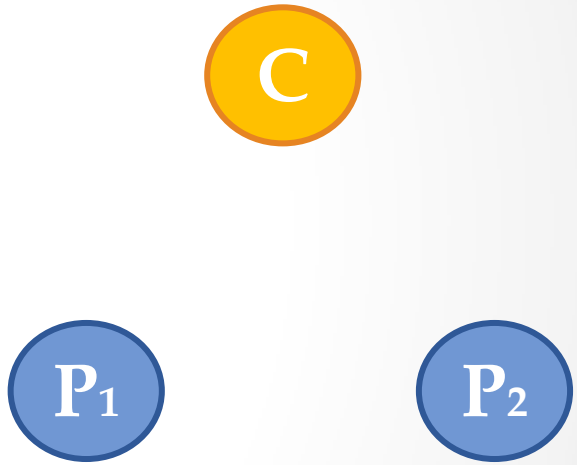


Swap channel

```
1 let doSwap v = let s = request-swap
2                in send s v; recv s
4 in
5 let doSwap' v = let s' = accept-swap
6                       v' = recv s'
7                 in send s' v; v'
8 in
9 spawn (fn x => doSwap v1); (doSwap' v2)
```

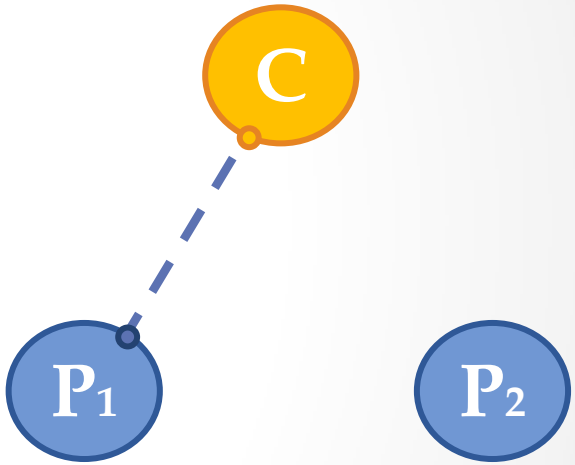
Swap service

```
1  let doSwap v = let s = request-swap
2                 in send s v;
3                    recv s
4  in
5  letrec coord _ = let s1 = accept-swap
6                           v1 = recv s1
7                           s2 = accept-swap
8                           v2 = recv s2
9                   in  send s2 v1;
10                      send s1 v2;
11                      coord ()
12 in
13 spawn coord;
14 map (fn x => spawn (fn y => doSwap x)) [0..10]
```



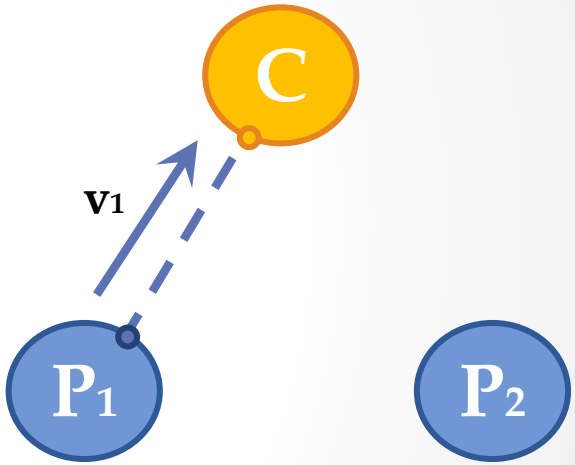
Swap service

```
1  let doSwap v = let s = request-swap
2                 in send s v;
3                    recv s
4  in
5  letrec coord _ = let s1 = accept-swap
6                           v1 = recv s1
7                           s2 = accept-swap
8                           v2 = recv s2
9                   in  send s2 v1;
10                      send s1 v2;
11                      coord ()
12 in
13 spawn coord;
14 map (fn x => spawn (fn y => doSwap x)) [0..10]
```



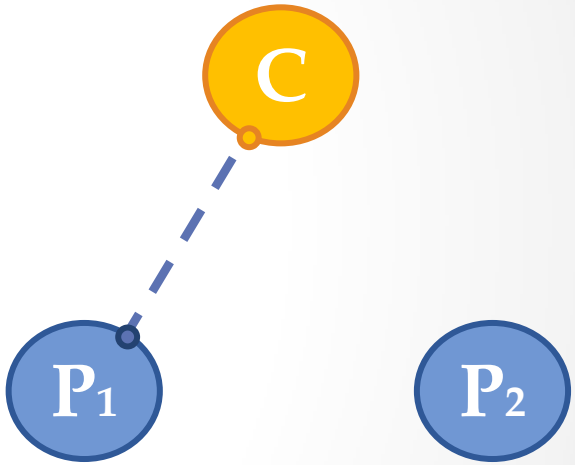
Swap service

```
1  let doSwap v = let s = request-swap
2                 in send s v;
3                    recv s
4  in
5  letrec coord _ = let s1 = accept-swap
6                           v1 = recv s1
7                           s2 = accept-swap
8                           v2 = recv s2
9                   in  send s2 v1;
10                      send s1 v2;
11                      coord ()
12 in
13 spawn coord;
14 map (fn x => spawn (fn y => doSwap x)) [0..10]
```



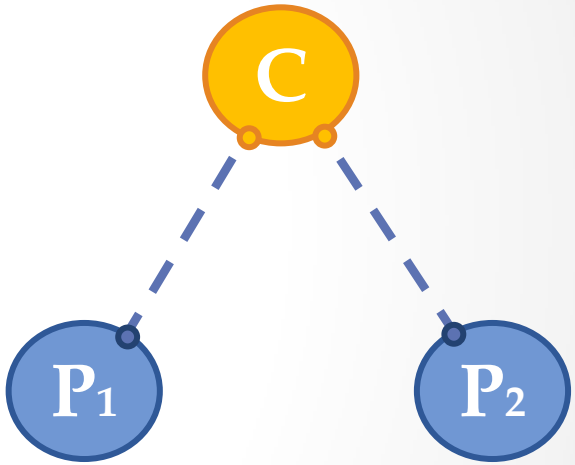
Swap service

```
1  let doSwap v = let s = request-swap
2                 in send s v;
3                    recv s
4  in
5  letrec coord _ = let s1 = accept-swap
6                           v1 = recv s1
7                           s2 = accept-swap
8                           v2 = recv s2
9                   in  send s2 v1;
10                      send s1 v2;
11                      coord ()
12 in
13 spawn coord;
14 map (fn x => spawn (fn y => doSwap x)) [0..10]
```



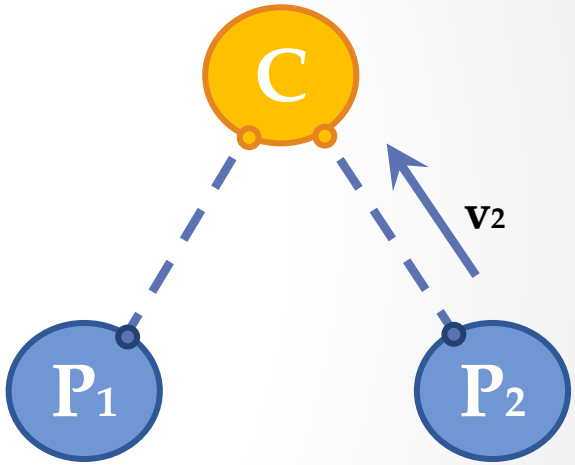
Swap service

```
1  let doSwap v = let s = request-swap
2                 in send s v;
3                    recv s
4  in
5  letrec coord _ = let s1 = accept-swap
6                           v1 = recv s1
7                           s2 = accept-swap
8                           v2 = recv s2
9                   in  send s2 v1;
10                      send s1 v2;
11                      coord ()
12 in
13 spawn coord;
14 map (fn x => spawn (fn y => doSwap x)) [0..10]
```



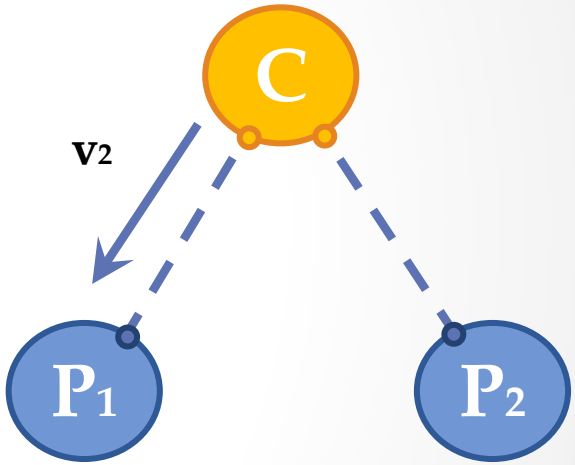
Swap service

```
1  let doSwap v = let s = request-swap
2                 in send s v;
3                    recv s
4  in
5  letrec coord _ = let s1 = accept-swap
6                           v1 = recv s1
7                           s2 = accept-swap
8                           v2 = recv s2
9                   in  send s2 v1;
10                      send s1 v2;
11                      coord ()
12 in
13 spawn coord;
14 map (fn x => spawn (fn y => doSwap x)) [0..10]
```



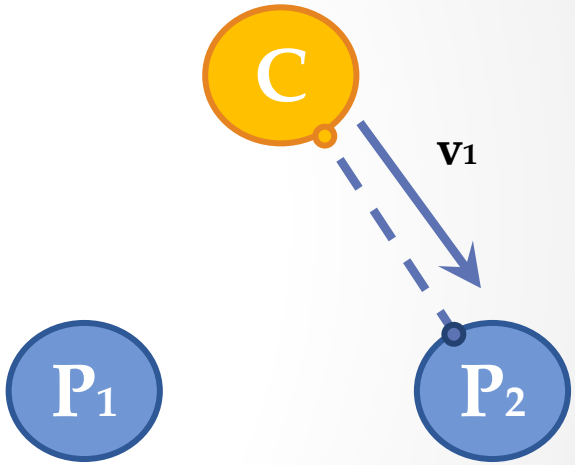
Swap service

```
1  let doSwap v = let s = request-swap
2                 in send s v;
3                    recv s
4  in
5  letrec coord _ = let s1 = accept-swap
6                           v1 = recv s1
7                           s2 = accept-swap
8                           v2 = recv s2
9                   in  send s2 v1;
10                      send s1 v2;
11                      coord ()
12 in
13 spawn coord;
14 map (fn x => spawn (fn y => doSwap x)) [0..10]
```



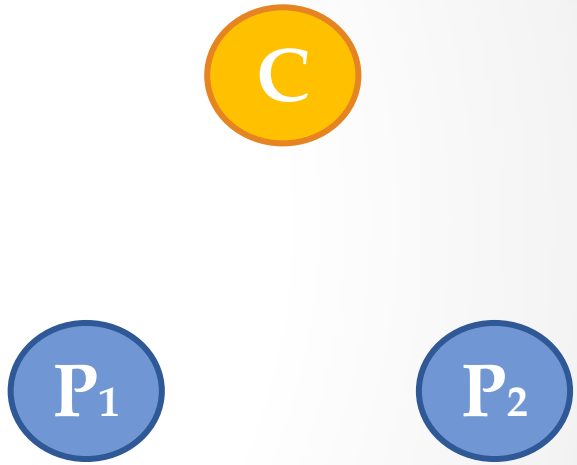
Swap service

```
1  let doSwap v = let s = request-swap
2                 in send s v;
3                    recv s
4  in
5  letrec coord _ = let s1 = accept-swap
6                           v1 = recv s1
7                           s2 = accept-swap
8                           v2 = recv s2
9                   in  send s2 v1;
10                      send s1 v2;
11                      coord ()
12 in
13 spawn coord;
14 map (fn x => spawn (fn y => doSwap x)) [0..10]
```



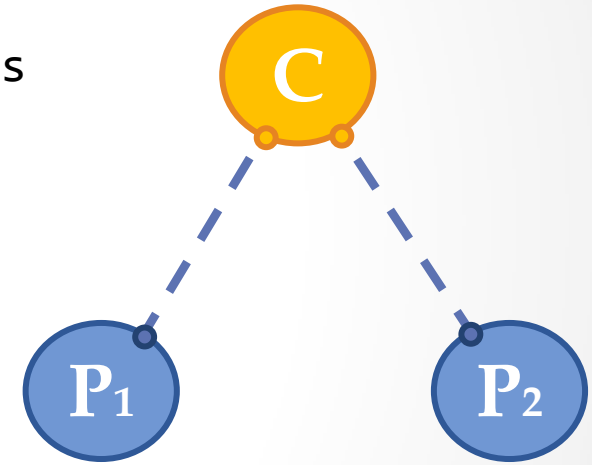
Swap service

```
1  let doSwap v = let s = request-swap
2                 in send s v;
3                    recv s
4  in
5  letrec coord _ = let s1 = accept-swap
6                           v1 = recv s1
7                           s2 = accept-swap
8                           v2 = recv s2
9                   in  send s2 v1;
10                      send s1 v2;
11                      coord ()
12 in
13 spawn coord;
14 map (fn x => spawn (fn y => doSwap x)) [0..10]
```



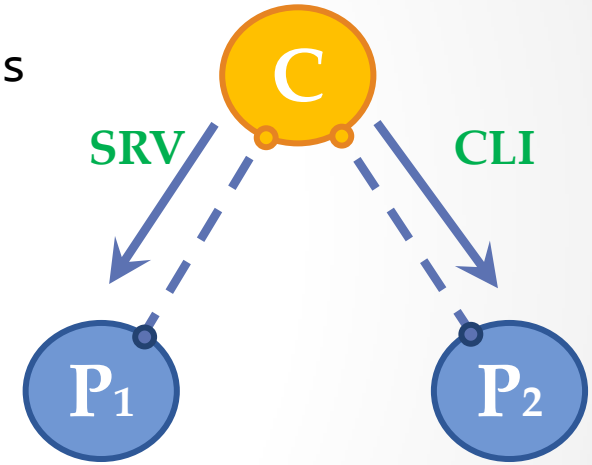
P2P swap service

```
1  let doSwap x = let s = request-swap
2                 in match s with
3                       CLI   -> send s x;
4                                recv s
5                       SRV   -> let s' = resume s
6                                    y  = recv s'
7                                in send s' x;
8                                y
9
10 letrec coord _ = let s1 = accept-swap
11                  in  select-CLI s1;
12                      let s2 = accept-swap
13                  in  select-SRV s2;
14                      delegate p2 s1;
15                      coord ()
16
17 spawn coord;
18 map (fn x => spawn (fn y => doSwap x)) [0..10]
```



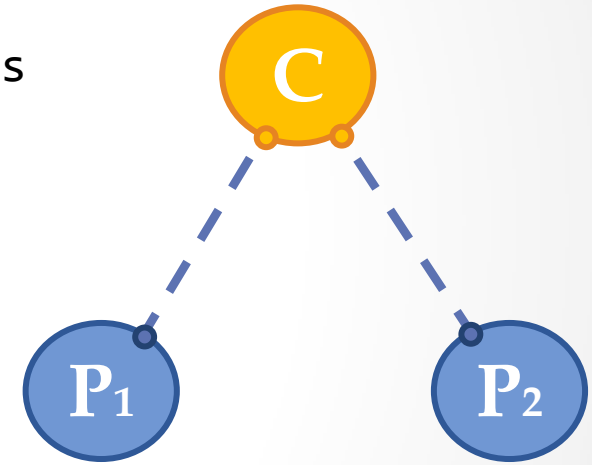
P2P swap service

```
1  let doSwap x = let s = request-swap
2                 in match s with
3                       CLI   -> send s x;
4                                recv s
5                       SRV   -> let s' = resume s
6                                    y  = recv s'
7                                in send s' x;
8                                y
9
10 letrec coord _ = let s1 = accept-swap
11                  in  select-CLI s1;
12                      let s2 = accept-swap
13                  in  select-SRV s2;
14                      delegate p2 s1;
15                      coord ()
16
17 spawn coord;
18 map (fn x => spawn (fn y => doSwap x)) [0..10]
```



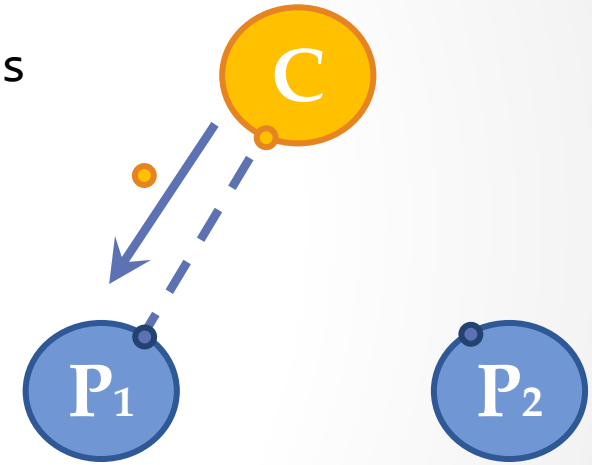
P2P swap service

```
1  let doSwap x = let s = request-swap
2                 in match s with
3                       CLI   -> send s x;
4                                recv s
5                       SRV   -> let s' = resume s
6                                    y  = recv s'
7                                in send s' x;
8                                y
9
10 letrec coord _ = let s1 = accept-swap
11                  in  select-CLI s1;
12                      let s2 = accept-swap
13                  in  select-SRV s2;
14                      delegate p2 s1;
15                      coord ()
16
17 spawn coord;
18 map (fn x => spawn (fn y => doSwap x)) [0..10]
```



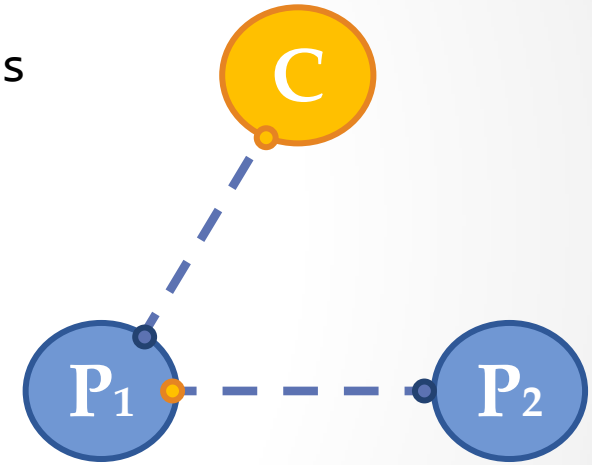
P2P swap service

```
1  let doSwap x = let s = request-swap
2                 in match s with
3                       CLI   -> send s x;
4                                recv s
5                       SRV   -> let s' = resume s
6                                    y  = recv s'
7                                in send s' x;
8                                y
9
10 letrec coord _ = let s1 = accept-swap
11                  in  select-CLI s1;
12                      let s2 = accept-swap
13                  in  select-SRV s2;
14                      delegate p2 s1;
15                      coord ()
16
17 spawn coord;
18 map (fn x => spawn (fn y => doSwap x)) [0..10]
```



P2P swap service

```
1  let doSwap x = let s = request-swap
2                 in match s with
3                       CLI   -> send s x;
4                                recv s
5                       SRV   -> let s' = resume s
6                                    y  = recv s'
7                                in send s' x;
8                                y
9
10 letrec coord _ = let s1 = accept-swap
11                  in  select-CLI s1;
12                      let s2 = accept-swap
13                  in  select-SRV s2;
14                      delegate p2 s1;
15                      coord ()
16
17 spawn coord;
18 map (fn x => spawn (fn y => doSwap x)) [0..10]
```



P2P swap service

```
1  let doSwap x = let s = request-swap
2                 in match s with
3                       CLI   -> send s x;
4                                recv s
5                       SRV   -> let s' = resume s
6                                    y  = recv s'
7                                in send s' x;
8                                   y
9
10 letrec coord _ = let s1 = accept-swap
11                  in  select-CLI s1;
12                      let s2 = accept-swap
13                  in  select-SRV s2;
14                      delegate p2 s1;
15                      coord ()
16
17 spawn coord;
18 map (fn x => spawn (fn y => doSwap x)) [0..10]
```

Goals and challenges

Our goals:

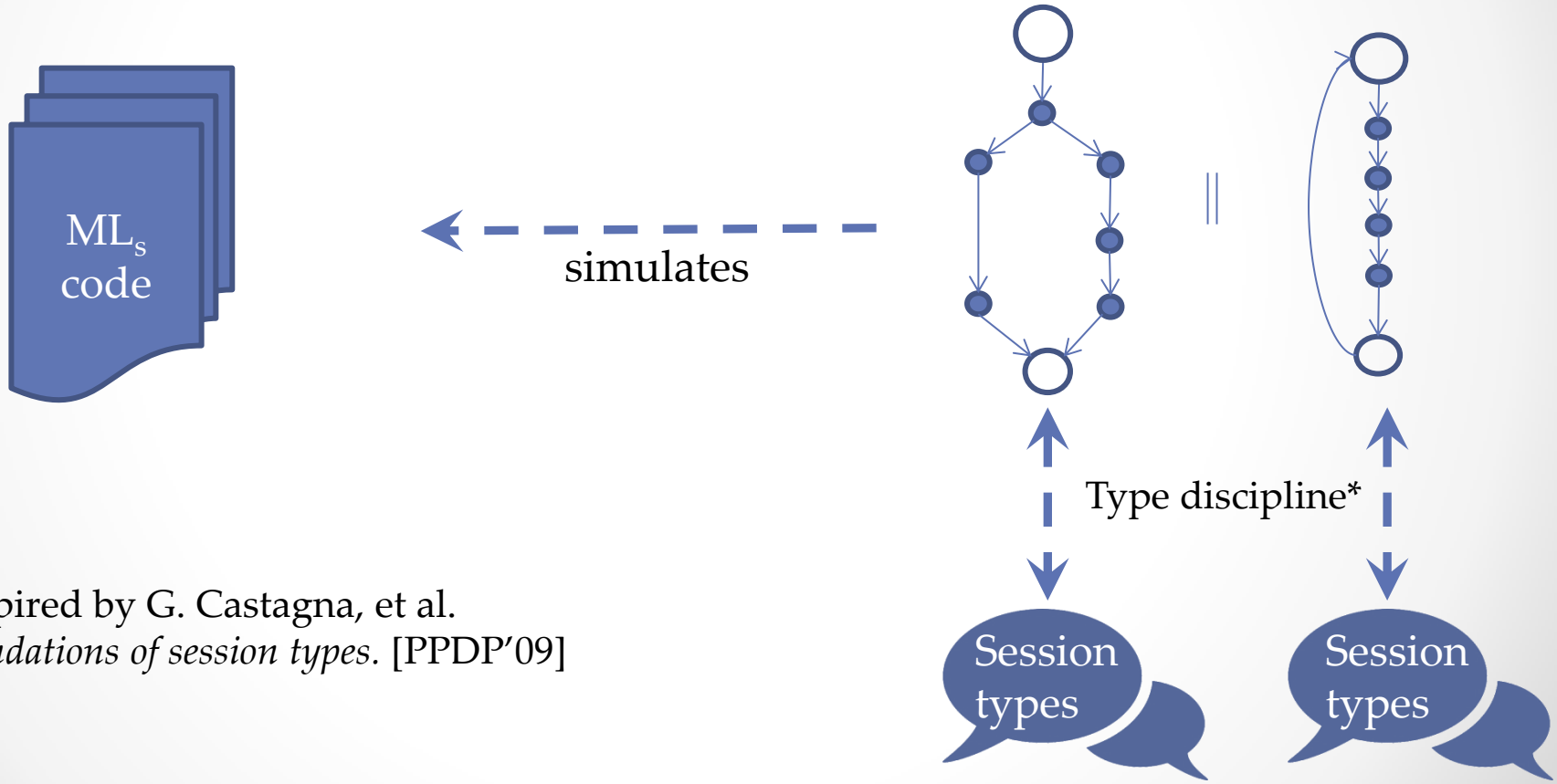
1. Fully automatic type inference for MLs
   - sound and complete algorithm (with no annotations)
2. Type guarantees
   - (Session) type safety, deadlock freedom

Challenges:

1. Code complexity
   - function calls, delegation, aliasing of endpoints…
2. No best approach to inferring session types
   - conservative extension of ML
   - scales with ML features (references, exceptions)

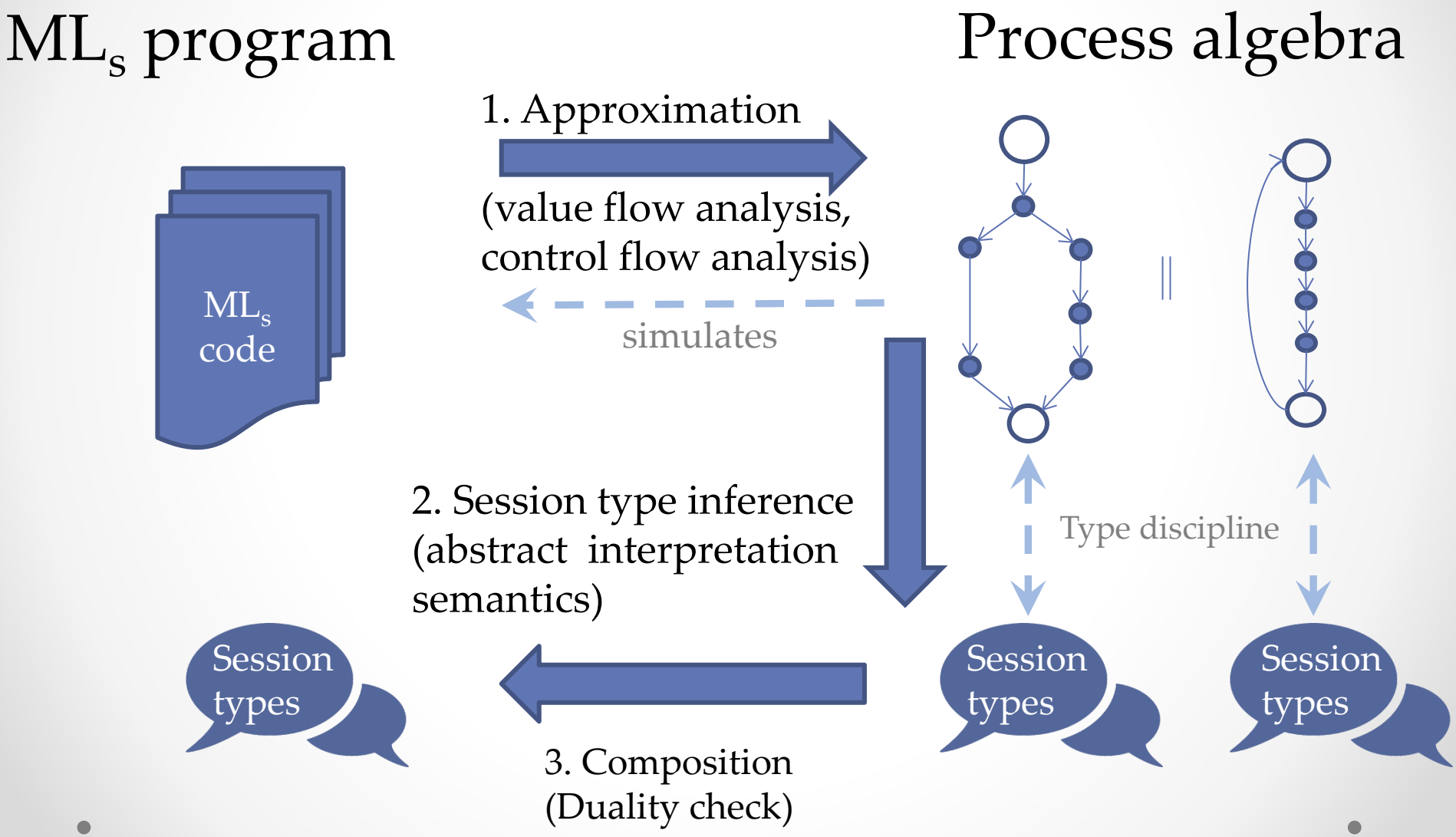
Type checking framework

ML_s program : $ML_{(s)}$ type » Process algebra



* Inspired by G. Castagna, et al.
Foundations of session types. [PPDP'09]

Type inference framework

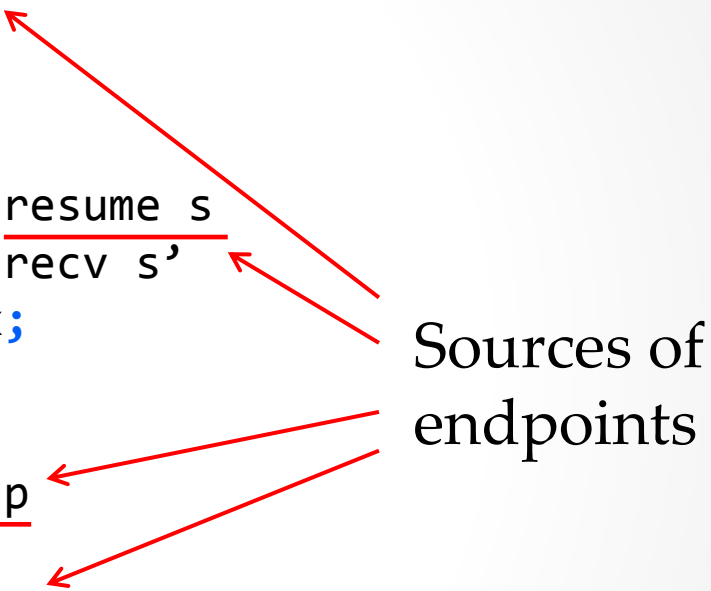


Approximating endpoints (region analysis)

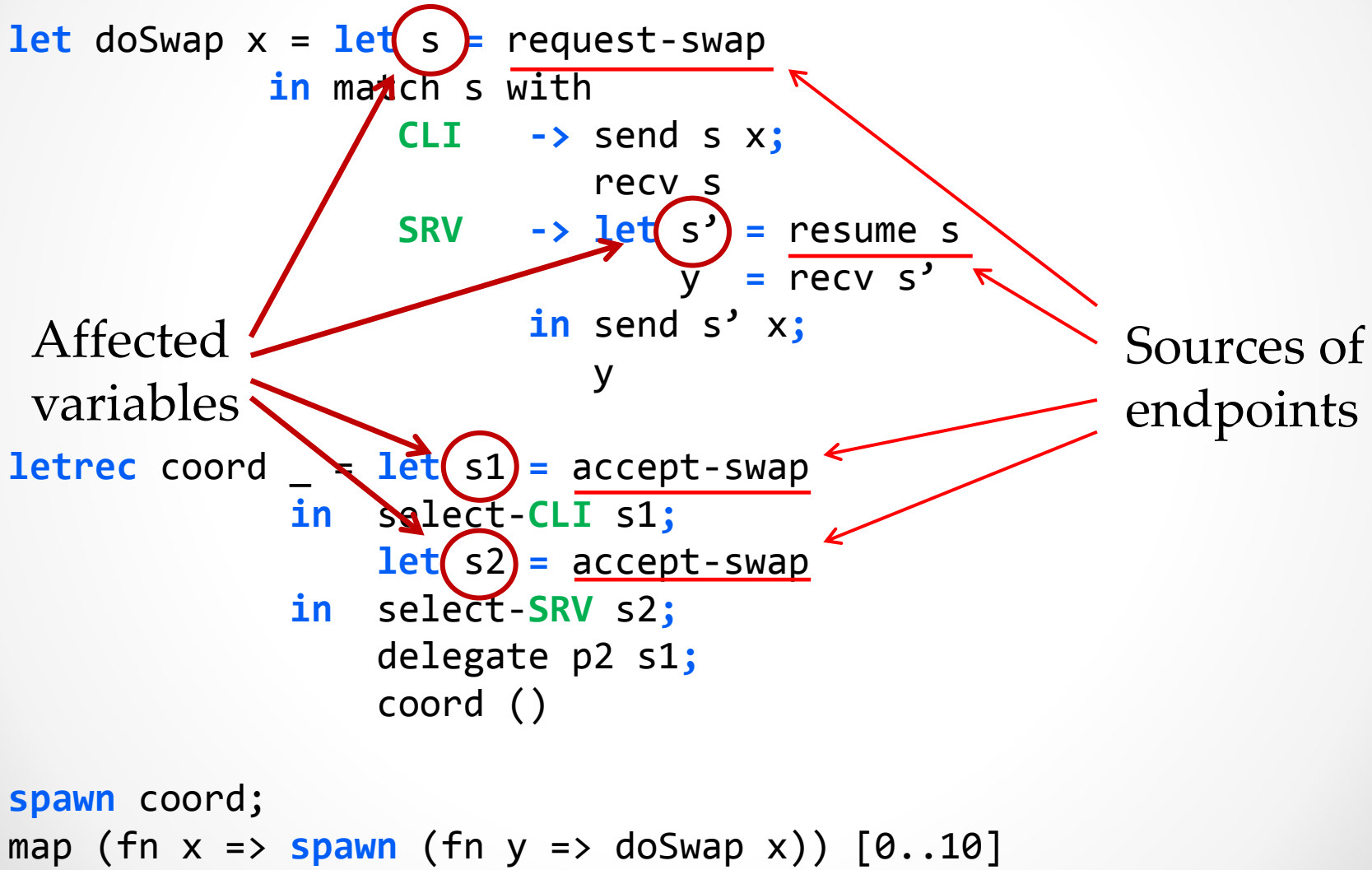
```
1  let doSwap x = let s = request-swap
2                 in match s with
3                       CLI   -> send s x;
4                                recv s
5                       SRV   -> let s' = resume s
6                                    y  = recv s'
7                                in send s' x;
8                                y
9
10 letrec coord _ = let s1 = accept-swap
11                  in  select-CLI s1;
12                      let s2 = accept-swap
13                  in  select-SRV s2;
14                      delegate p2 s1;
15                      coord ()
16
17 spawn coord;
18 map (fn x => spawn (fn y => doSwap x)) [0..10]
```

Approximating endpoints (region analysis)

```
1  let doSwap x = let s = request-swap
2                 in match s with
3                       CLI   -> send s x;
4                                recv s
5                       SRV   -> let s' = resume s
6                                    y  = recv s'
7                                in send s' x;
8                                y
9
10 letrec coord _ = let s1 = accept-swap
11                  in  select-CLI s1;
12                      let s2 = accept-swap
13                  in  select-SRV s2;
14                      delegate p2 s1;
15                      coord ()
16
17 spawn coord;
18 map (fn x => spawn (fn y => doSwap x)) [0..10]
```

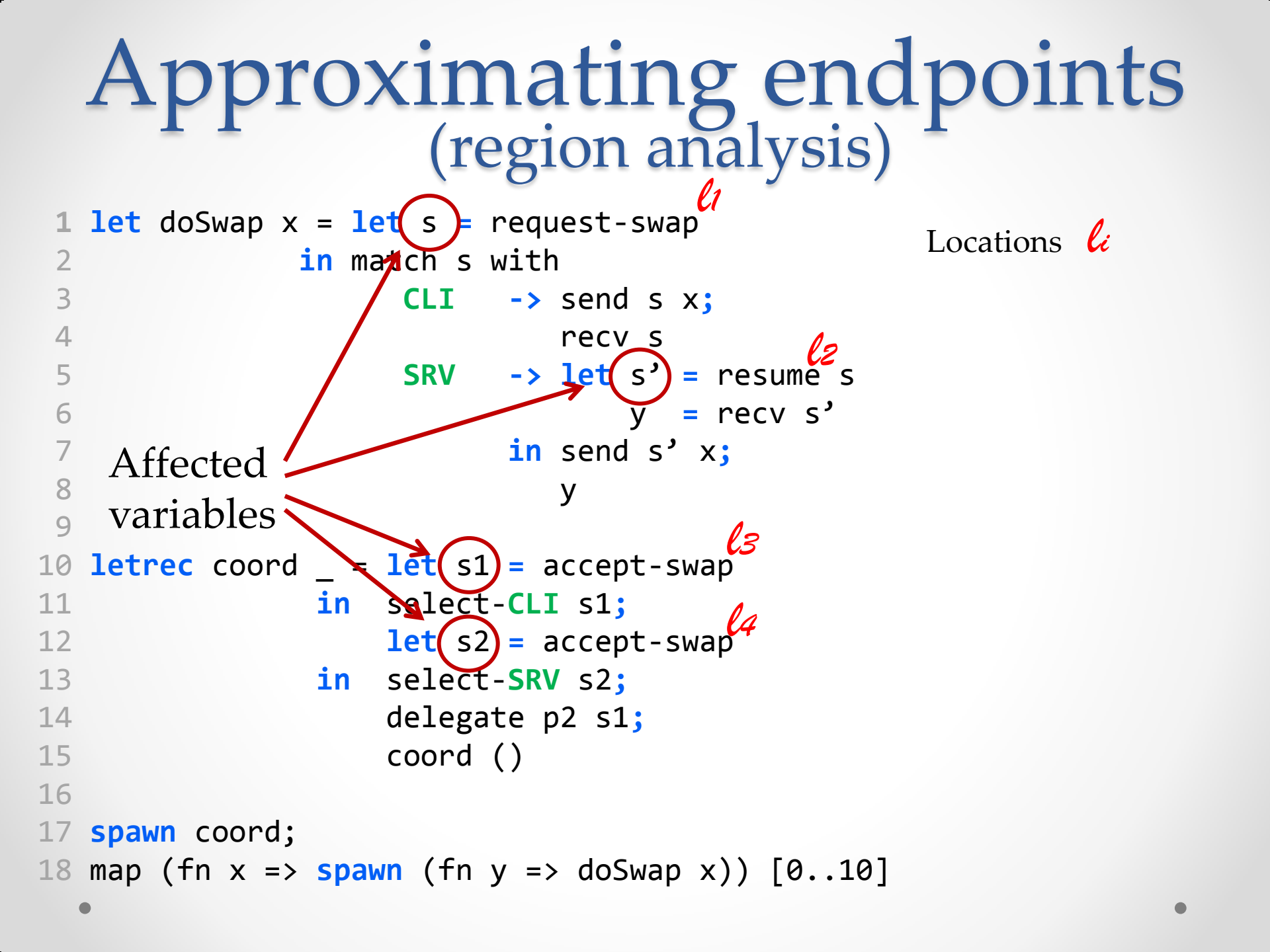


Approximating endpoints (region analysis)



Approximating endpoints (region analysis)

```
1  let doSwap x = let s = request-swap
2                 in match s with
3                      CLI   -> send s x;
4                               recv s
5                      SRV   -> let s' = resume s
6                                   y  = recv s'
7                               in send s' x;
8                               y
9
10 letrec coord _ = let s1 = accept-swap
11                  in  select-CLI s1;
12                      let s2 = accept-swap
13                  in  select-SRV s2;
14                      delegate p2 s1;
15                      coord ()
16
17 spawn coord;
18 map (fn x => spawn (fn y => doSwap x)) [0..10]
```



Approximating endpoints (region analysis)

```
1  let doSwap x = let r1 = request-swap                    ℓ1
2                 in match r1 with
3                       CLI   -> send r1 x;
4                                recv r1
5                       SRV   -> let r2 = resume r1       ℓ2
6                                    y  = recv r2
7                                in send r2 x;
8                                   y
9
10 letrec coord _ = let r3 = accept-swap                   ℓ3
11                  in  select-CLI r3;
12                      let r4 = accept-swap               ℓ4
13                  in  select-SRV r4;
14                      delegate r4 r3;
15                      coord ()
16
17 spawn coord;
18 map (fn x => spawn (fn y => doSwap x)) [0..10]
```

Locations ℓ_i
Regions r_i

Approximating endpoints (region analysis)

```
1  let doSwap x = let ℓ1 = request-swap ℓ1
2                 in match ℓ1 with
3                       CLI   -> send ℓ1 x;
4                                recv ℓ1
5                       SRV   -> let ℓ2 = resume ℓ2 ℓ1
6                                    y  = recv ℓ2
7                                in send ℓ2 x;
8                                   y
9
10 letrec coord _ = let ℓ3 = accept-swap ℓ3
11                  in  select-CLI ℓ3;
12                      let ℓ4 = accept-swap ℓ4
13                  in  select-SRV ℓ4;
14                      delegate ℓ4 ℓ3;
15                      coord ()
16
17 spawn coord;
18 map (fn x => spawn (fn y => doSwap x)) [0..10]
```

Locations ℓ_i
Regions r_i

[M. Tofte, J. Talpin, Implementation of the typed call-by-value lambda-calculus using a stack of regions. POPL 94]

Approximating behaviour
(control-flow analysis)

```
1  let doSwap x = let ℓ1 = request-swap ℓ1
2                 in match ℓ1 with
3                       CLI   -> send ℓ1 x;
4                                recv ℓ1
5                       SRV   -> let ℓ2 = resume ℓ2 ℓ1
6                                    y  = recv ℓ2
7                                in send ℓ2 x;
8                                   y
9
10 letrec coord _ = let ℓ3 = accept-swap ℓ3
11                  in  select-CLI ℓ3;
12                      let ℓ4 = accept-swap ℓ4
13                  in  select-SRV ℓ4;
14                      delegate ℓ4 ℓ3;
15                      coord ()
16
17 spawn coord;
18 map (fn x => spawn (fn y => doSwap x)) [0..10]
```

Approximating behaviour
(control-flow analysis)

```
1  let doSwap x = let ℓ1 = request-swap
2                 in
3
4
5
6
7                        in send ℓ2 x;
8                           y
9
10 letrec coord _ = l
11                  in  s
12                      l
13                  in  s
14                      delegate ℓ4 ℓ3;
15                      coord ()
16
17 spawn coord;
18 map (fn x => spawn (fn y => doSwap x)) [0..10]
```

$$\beta_{doSwap} = new(\ell_1 :: Swap_{req}).\ \Sigma \begin{cases} ?CLI \rightarrow \ell_1!T_1.\ \ell_1?T_2 \\ ?SRV \rightarrow \ell_1?\ell_2.\ \ell_2?T_2.\ \ell_2!T_1 \end{cases}$$

$$\beta_{coord} = new(\ell_3 :: Swap_{acc}).\ \ell_3!CLI.\ new(\ell_4 :: Swap_{acc}).\ \ell_4!SRV.\ \ell_4!\ell_3.\ rec\ \beta_{coord}$$

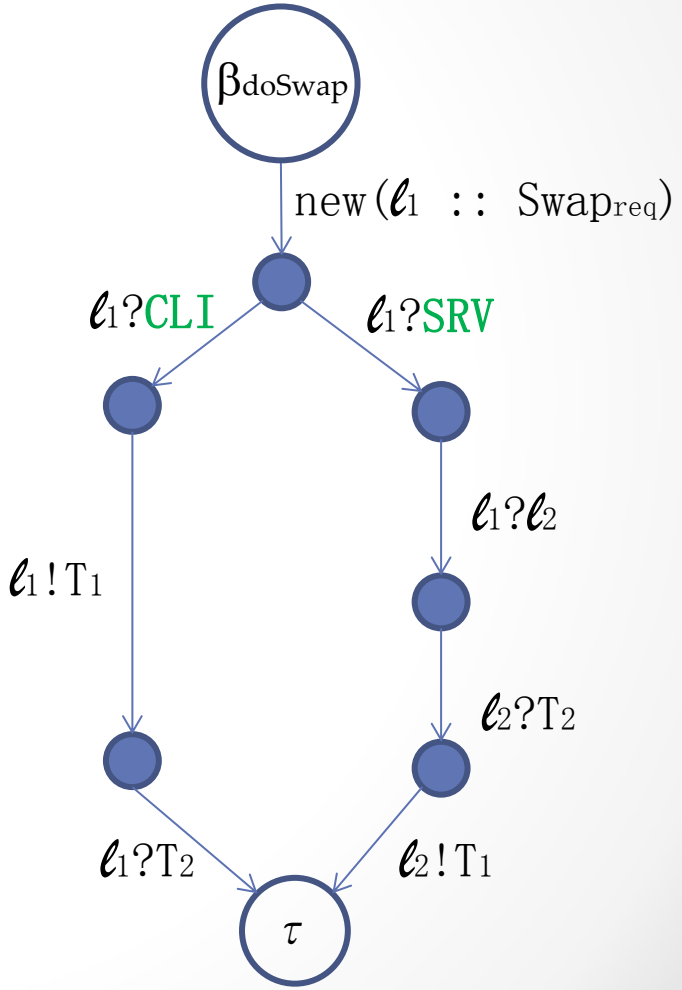
[T. Amtoft, H. R. Nielson, and F. Nielson. *Type and effect systems - behaviours for concurrency*. Imperial College Press, 1999]

Session type inference (abstract interpretation semantics)

- Abstract interpretation semantics for behaviours
- Explore all local paths in the LTS
- Derive session types

- Finite state LTSs
  - finite session types
  - but also recursive session types

- Number of states proportional to source code size
  - no state explosion

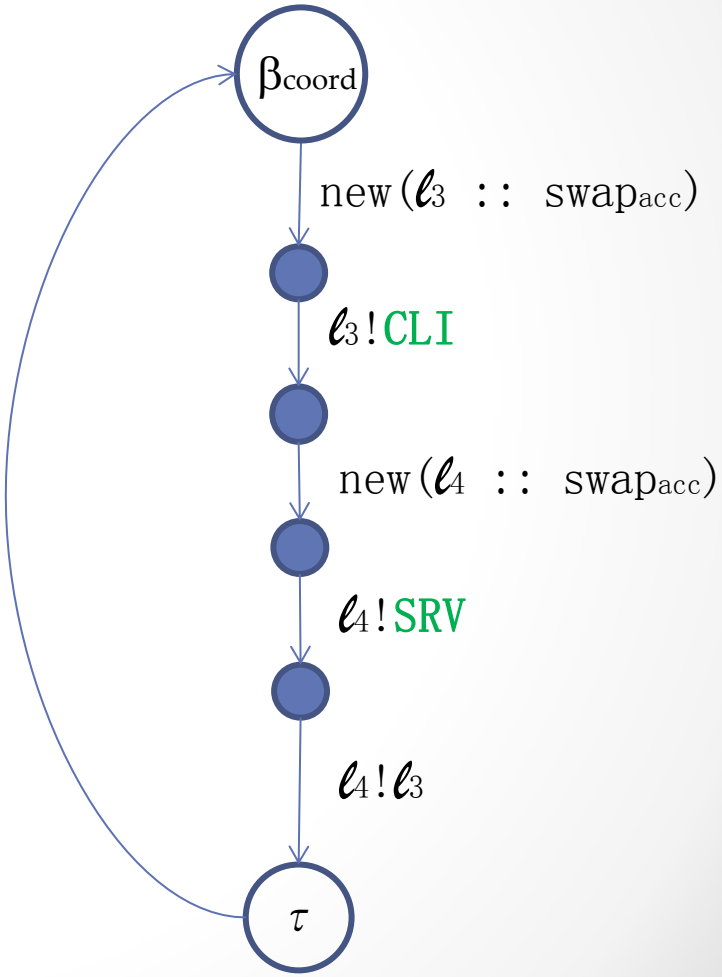


Inferred type:

$$\text{Swap}_{\text{req}} :: \Sigma \begin{cases} ?\text{CLI}.\ !T_1.\ ?T_2.\ \text{end} \\ ?\text{SRV}.\ ?[?T_2.\ !T_1.\ \text{end}].\ \text{end} \end{cases}$$

Session type inference (abstract interpretation semantics)

- Recursive behaviours have to be *self contained*



Inferred type:

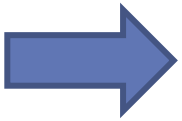
$\mathrm{Swap}_{\mathrm{acc}} :: !\mathtt{CLI}.\, S_2 \oplus !\mathtt{SRV}.\, ![S_2].\, \mathrm{end}$

where S_2 is unknown

Composition
(duality check)

Local session types are composed back:

$$\mathrm{Swap}_{\mathrm{req}} :: \Sigma \begin{cases} ?\mathtt{CLI}.\ !T_1.\ ?T_2.\ \mathrm{end} \\ ?\mathtt{SRV}.\ ?[?T_2.\ !T_1.\ \mathrm{end}].\ \mathrm{end} \end{cases}$$



$$S_2 = ?T_2.\ !T_1.\ \mathrm{end},\quad T_1 <: T_2$$

$$\mathrm{Swap}_{\mathrm{acc}} :: !\mathtt{CLI}.\ S_2 \oplus !\mathtt{SRV}.\ ![S_2].\ \mathrm{end}$$

Theorem: well-typed processes are (weakly) deadlock free

Conclusions

We presented a method to

- add session communications to ML
  - extract communication behaviour from code
  - check session types on the behaviour
- infer types automatically (sound and complete)
- guarantee type safety and (some) deadlock freedom

We are currently implementing a prototype (on top of the OCaml compiler)

Questions?

(thank you!)