Locally Static, Globally Dynamic Session Types for Active Objects

Reiner Hähnle[1] Anton W. Haubner[1] Eduard Kamburjan[1,2]

27.11.2020

[1]Technische Universität Darmstadt [2]University of Oslo

Introduction

Three aspects of our paper to make Maurizio happy:

Introduction

Three aspects of our paper to make Maurizio happy:

- Behavioral types related to dynamic choreographies

Introduction

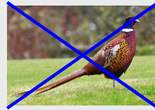
Three aspects of our paper to make Maurizio happy:

- Behavioral types related to dynamic choreographies
- We included an Oulipian riddle

Introduction

Three aspects of our paper to make Maurizio happy:

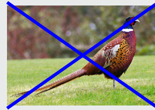
- Behavioral types related to dynamic choreographies
- We included an Oulipian riddle
- No pheasants!



Introduction

Three aspects of our paper to make Maurizio happy:

- Behavioral types related to dynamic choreographies
- We included an Oulipian riddle
- No pheasants!



Happy birthday, Maurizio! — Enjoy your grapes!

Endpoints in Session Types

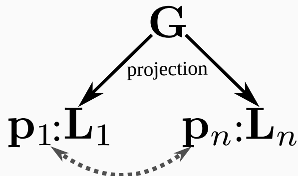
Structure of A Session Type System

- A communication medium with a set of endpoints $\mathbf{p}_i$
- A global specification of this communication $\mathbf{G}$
- A projection mechanism that generates local specifications $\mathbf{L}_i$

Endpoints in Session Types

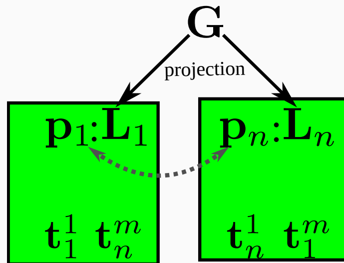
Structure of A Session Type System

- A communication medium with a set of endpoints $\mathbf{p}_i$
- A global specification of this communication $\mathbf{G}$
- A projection mechanism that generates local specifications $\mathbf{L}_i$



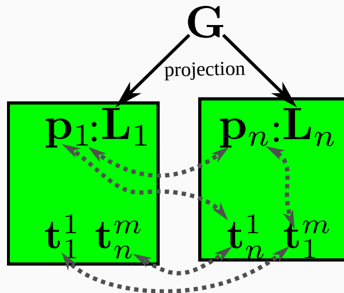
Structure of A Session Type System

- A communication medium with a set of endpoints $\mathbf{p}_i$
- A global specification of this communication $\mathbf{G}$
- A projection mechanism that generates local specifications $\mathbf{L}_i$



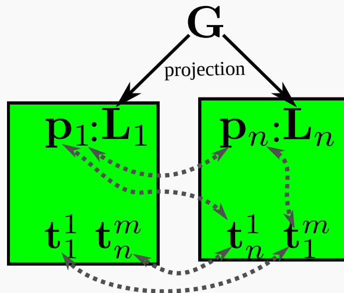
Structure of A Session Type System

- A communication medium with a set of endpoints $\mathbf{p}_i$
- A global specification of this communication $\mathbf{G}$
- A projection mechanism that generates local specifications $\mathbf{L}_i$



Structure of A Session Type System

- A communication medium with a set of endpoints $\mathbf{p}_i$
- A global specification of this communication $\mathbf{G}$
- A projection mechanism that generates local specifications $\mathbf{L}_i$



What are the Endpoints in Active Objects (AO)?

Active Objects

Active Objects by Example (ABS)

Active Objects

Object-Oriented Actors with Futures and Cooperative Scheduling.

```
1  class Notify (MailI mf, UII u) implements NotifyI  {
2    Fut<Msg> fMail;
3    Unit init(int bound) {
4      Int i = 0;
5      while (i < bound) {
6        fMail = mf!checkMail();
7        await fMail?;
8        Msg res = fMail.get;
9        ...
10     }
11   }
12 }
```

Active Objects by Example (ABS)

Active Objects

Object-Oriented Actors with Futures and Cooperative Scheduling.

```
1 class Notify (MailI mf, UII u) implements NotifyI {
2   Fut<Msg> fMail;
3   Unit init(int bound) {
4     Int i = 0;
5     while (i < bound) {
6       fMail = mf!checkMail();
7       await fMail?;
8       Msg res = fMail.get;
9       ...
10    }
11  }
12 }
```

Object-Oriented Actors

Active Objects

Object-Oriented Actors with Futures and Cooperative Scheduling.

```
1  class Notify (MailI mf, U
2    Fut<Msg> fMail;
3    Unit init(int bound) {
4      Int i = 0;
5      while (i < bound) {
6        fMail = mf!checkMail();
7        await fMail?;
8        Msg res = fMail.get;
9        ...
10     }
11   }
12 }
```

Strongly Encapsulated:

Every field is object-private.

Active Objects by Example (ABS)

Active Objects

Object-Oriented Actors with Futures and Cooperative Scheduling.

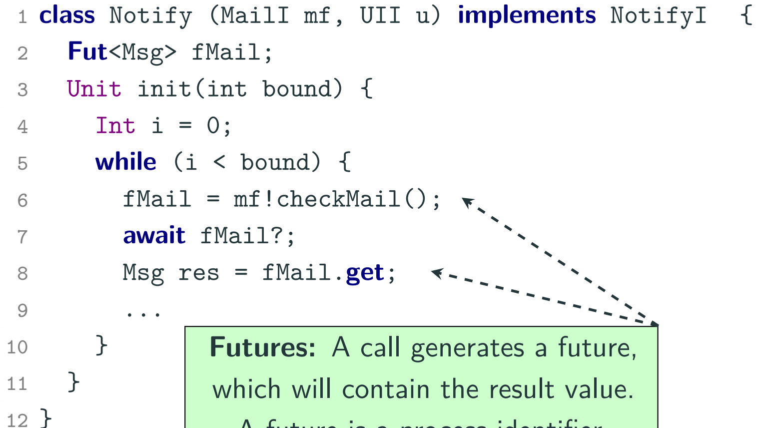
```
1  class Notify (MailI mf, UII u) implements NotifyI  {
2    Fut<Msg> fMail;
3    Unit init(int bound) {
4      Int i = 0;
5      while (i < bound) {
6        fMail = mf!checkMail();
7        await fMail?;
8        Msg res = fMail.get;
9        ...
10     }
11   }
12 }
```

Asynchronous Method Calls as Messages:
Calls are asynchronous and are checked against the callee interface.

Active Objects by Example (ABS)

Active Objects

Object-Oriented Actors with Futures and Cooperative Scheduling.



Active Objects

Object-Oriented Actors with Futures and Cooperative Scheduling.

```
1  class Notify (MailI mf, UII u) implements NotifyI  {
2    Fut<Msg> fMail;
3    Unit init(int bound) {
4      Int i = 0;
5      while (i < bound) {
6        fMail = mf!checkMai
7        await fMail?;  <---
8        Msg res = fMail.get
9        ...
10     }
11   }
12 }
```

Cooperative Scheduling: Only one process is active per object. It must explicitly release control.

Active Objects by Example (ABS)

Active Objects

Object-Oriented Actors with Futures and Cooperative Scheduling.

Endpoints in Active Objects are nested:

Both processes and objects are endpoints for communication.

How to specify such communication?

How to verify such communication?

```
1
2
3
4
5     while (i < bound) {
6       fMail = mf!checkMail();
7       await fMail?;
8       Msg res = fMail.get;
9       ...
10    }
11  }
12 }
```

Active Objects

User-Defined Schedulers

To define a scheduler, the target language represents the process queue has an explicit representation at runtim.

```
1 def Maybe<Process> default(List<Process> l,
2                              MailI mf, UII u/* fields */)
3     = head(filter(l, {it -> method(it) == "init"}));
4
5 [Scheduler : default(mf, u)]
6 class Notify (MailI mf, UII u)...
```

- A process has a name, a future, a sender and a method.
- The scheduler may delay execution
- The scheduler may read, but not write the fields of the object.

Session Types for Active Objects

Global Types for Active Objects

Communication

- Between two objects: Method calls
- Between object and future: Read and write access
- Between object and process: Suspension

$\mathbf{G} ::= \mathbf{p} \xrightarrow{\mathtt{t}} \mathbf{q} : \mathtt{m}$	**p** calls **q**.$\mathtt{m}$ with future $\mathtt{t}$
$\mid \mathbf{p} \downarrow \mathtt{t}$	**p** resolves $\mathtt{t}$
$\mid \mathbf{p} \uparrow \mathtt{t}$	**p** reads from $\mathtt{t}$
$\mid \mathsf{Rel}(\mathbf{p}, \mathtt{t})$	**p** suspends active process on $\mathtt{t}$
$\mid \mathbf{p}\{\mathbf{G}_i\}_{i \in I} \mid (\mathbf{G})^*$	Branching and Repetition
$\mid \mathbf{G} \,.\, \mathbf{G} \mid \textbf{skip}$	Sequence and Empty Statement

Simple Local Types

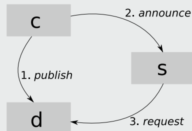
Layered Local Types

We use *object*-local types for the view of an object/role, *method*-local types for the view of a method/process. Both use the same syntax.

$\mathbf{L} ::= \mathbf{q}!_{\mathtt{t}}\mathtt{m} \mid \mathbf{p}?_{\mathtt{t}}\mathtt{m}$	Local view on method calls
$\mid \text{Put } \mathtt{t} \mid \text{Get } \mathtt{f}$	Terminating and reading a future $\mathtt{t}$
$\mid \text{Susp}(\mathtt{t}, \mathtt{t}')$	Suspending $\mathtt{t}$ until $\mathtt{t}'$ is resolved
$\mid \text{React } \mathtt{t}$	Reactivating $\mathtt{t}$
$\mid \mathbf{L} \,.\, \mathbf{L} \mid (\mathbf{L})^*$	Sequence and Repetition
$\mid \&_{\mathtt{t}}\{\mathbf{L}_i\}_{i \in I} \mid \oplus\{\mathbf{L}_i\}_{i \in I}$	Passive and Active Choice

Example

Publishing and Accessing Grades



$$\mathsf{G} =$$

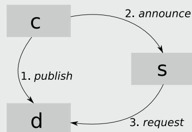
$$\mathbf{0} \xrightarrow{\mathtt{t}_0} \mathbf{c} : \mathtt{pubGrd} \,.\, \mathbf{c} \xrightarrow{\mathtt{t}_1} \mathbf{d} : \mathtt{publish} \,.\, \mathbf{d} \downarrow \mathtt{t}_1 \,.$$

$$\mathbf{c} \xrightarrow{\mathtt{t}_2} \mathbf{s} : \mathtt{announce} \,.\, \mathbf{s} \xrightarrow{\mathtt{t}_3} \mathbf{d} : \mathtt{request} \,.$$

$$\mathbf{d} \downarrow \mathtt{t}_3 \,.\, \mathbf{s} \uparrow \mathtt{t}_3 \,.\, \mathbf{s} \downarrow \mathtt{t}_2 \,.\, \mathbf{c} \downarrow \mathtt{t}_0$$

Example

Publishing and Accessing Grades



$\mathsf{G} =$

$\mathbf{0} \xrightarrow{\mathtt{t}_0} \mathbf{c}:\mathtt{pubGrd}\,.\,\mathbf{c} \xrightarrow{\mathtt{t}_1} \mathbf{d}:\mathtt{publish}\,.\,\mathbf{d}{\downarrow}\mathtt{t}_1\,.$

$\mathbf{c} \xrightarrow{\mathtt{t}_2} \mathbf{s}:\mathtt{announce}\,.\,\mathbf{s} \xrightarrow{\mathtt{t}_3} \mathbf{d}:\mathtt{request}\,.$

$\mathbf{d}{\downarrow}\mathtt{t}_3\,.\,\mathbf{s}{\uparrow}\mathtt{t}_3\,.\,\mathbf{s}{\downarrow}\mathtt{t}_2\,.\,\mathbf{c}{\downarrow}\mathtt{t}_0$

Generated Local Types and Scheduling

Object-Local: $\mathbf{c}?_{\mathtt{t}_1}\mathtt{publish}\,.\,\mathsf{Put}\ \mathtt{t}_1\,.\,\mathbf{s}?_{\mathtt{t}_3}\mathtt{request}\,.\,\mathsf{Put}\ \mathtt{t}_3$

Method-Local: $\mathbf{c}?_{\mathtt{t}_1}\mathtt{publish}\,.\,\mathsf{Put}\ \mathtt{t}_1$

Scheduling: $\mathbf{c}?_{\mathtt{t}_1}\mathtt{publish}\,.\,\mathbf{s}?_{\mathtt{t}_3}\mathtt{request}$

Locally Static, Globally Dynamic Session Types for Active Objects

Enforcing Object-Local Types

Checking Object-Local Types

Method-local types can be statically type-checked against methods, but what about the scheduling type?

Enforcing Object-Local Types

Checking Object-Local Types

Method-local types can be statically type-checked against methods, but what about the scheduling type?

Checking vs. Enforcing Object-Local Types

Prior work [iFM'18]: statically check causality of *global* type.
Most likely, no user-defined scheduler is available.
Here: Represent scheduling type as a register automata and inject the automaton as a user-defined scheduler into the code.

[iFM'18] Kamburjan and Chen, "Stateful Behavioral Types for Active Objects"

Register Automata

Definition

A k-register automaton is a NFA over a an alphabet $\Sigma \times D$, where D is infinite. It has k register to store elements from D and can only (1) store the current letter and (2) compare with equality.

Register Automata

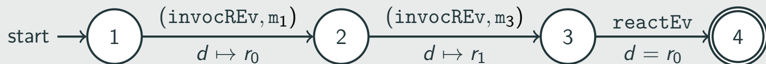
Definition

A k-register automaton is a NFA over a an alphabet $\Sigma \times D$, where D is infinite. It has k register to store elements from D and can only (1) store the current letter and (2) compare with equality.

Example

Each future tracked in the type is one register.

$$\mathbf{p}?_{\mathtt{t}}\mathtt{m}_1 \,.\, \mathsf{Susp}(\mathtt{t},\mathtt{t}') \,.\, \mathbf{p}?_{\mathtt{t}''}\mathtt{m}_3 \,.\, \mathsf{Put}\ \mathtt{t}'' \,.\, \mathsf{React}(\mathtt{t}) \,.\, \mathsf{Put}\ \mathtt{t}$$



Code Injection

- Add field `state` for the RA state and k fields for the registers

Code Injection

- Add field `state` for the RA state and k fields for the registers
- Each method and **await** stores its future and changes the state

```
1     case (this.state) {
2         1 -> {r_i = destiny; this.state = n;}
3         2 -> {r_j = destiny; this.state = m;} \
  dots
4     }
5     //original code;
```

- A scheduler function that encodes the transition relation

Code Injection

- Add field `state` for the RA state and k fields for the registers
- Each method and **await** stores its future and changes the state

```
1      case (this.state) {
2          1 -> {r_i = destiny; this.state = n;}
3          2 -> {r_j = destiny; this.state = m;} \
   dots
4      }
5      //original code;
```

- A scheduler function that encodes the transition relation

```
1 ra(list, state, r1, ...) = case state {
2   1 -> head(filter(l, { it -> /*out−edges 1*/}));
3   ⋮
4 });
```

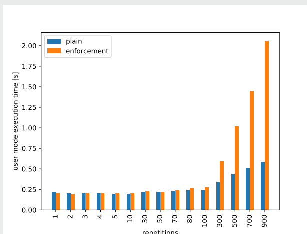
Evaluation

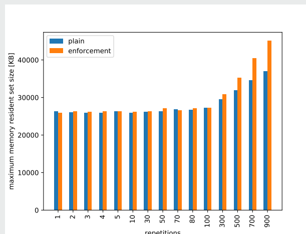
Synthetic benchmark for n calls on $\mathtt{m}$ for implementation of $\mathbf{G}_{bench}$

$$\mathbf{0} \xrightarrow{\mathtt{t}_1} \mathbf{p}:\mathtt{m}_1 \,.\, \left(\mathbf{p} \xrightarrow{\mathtt{t}_2} \mathbf{q}:\mathtt{m}_2 \,.\, \mathbf{q}\downarrow \mathtt{t}_2 \,.\, \mathbf{p} \xrightarrow{\mathtt{t}_3} \mathbf{q}:\mathtt{m}_3 \,.\, \mathbf{q}\downarrow \mathtt{t}_3\right)^* . \mathbf{p}\downarrow \mathtt{t}_1$$

Generated Code vs. Unmodified Code

Explicit representation of process bag does not scale





Conclusion

Theorem (Protocol Adherence/Session Fidelity)

If type-checking for methods succeeds and each scheduler follows its type, then the system follows the global protocol.

Conclusion

Theorem (Protocol Adherence/Session Fidelity)

If type-checking for methods succeeds and each scheduler follows its type, then the system follows the global protocol.

Summary

- Heterogenous handling of types for endpoints
- Methods are statically checked
- Objects are dynamically enforced to schedule correctly

Future work: Verification of (half-)open systems with AOs

Conclusion

Theorem (Protocol Adherence/Session Fidelity)

If type-checking for methods succeeds and each scheduler follows its type, then the system follows the global protocol.

Summary

- Heterogenous handling of types for endpoints
- Methods are statically checked
- Objects are dynamically enforced to schedule correctly

Future work: Verification of (half-)open systems with AOs

Thank you for your attention