the chemical approach to typestate-oriented programming

Silvia Crafa[1] Luca Padovani[2]

[1]Dipartimento di Matematica, Università di Padova, Italy

[2]Dipartimento di Informatica, Università di Torino, Italy

OOPSLA 2015

Typestate-oriented programming (Aldrich *et al.* '09)

```
class File {
  public final String fileName;



  public method open() {

      handle = fopen(fileName); // OK if closed
  }


  private FILE* handle;         // meaningful if open
  public method close() { ... }
  public method read()
  { ...fread(handle)... }       // OK if open
}
```

Typestate-oriented programming (Aldrich *et al.* '09)

```
class File {
  public final String fileName;
}

state ClosedFile of File {      // Closed File
  public method open() {        [Closed >> Open]

      handle = fopen(fileName); // OK if closed
} }

state OpenFile of File {        // Open File
  private FILE* handle;         // meaningful if open
  public method close() { ... } [Open >> Closed]
  public method read()
  { ...fread(handle)... }       // OK if open
}
```

Typestate-oriented programming (Aldrich *et al.* '09)

```
class File {
  public final String fileName;
}

state ClosedFile of File {      // Closed File
  public method open() {        [Closed >> Open]
    this <- OpenFile {          // explicit state change
      handle = fopen(fileName); // OK if closed
} } }

state OpenFile of File {        // Open File
  private FILE* handle;         // meaningful if open
  public method close() { ... } [Open >> Closed]
  public method read()
  { ...fread(handle)... }       // OK if open
}
```

Typestate-oriented programming

Objective

- enforcement of object protocols

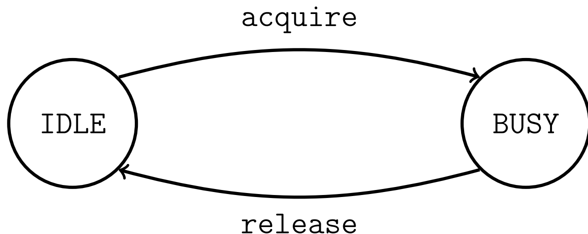
Features

- **static** tracking of **state transitions** `[Closed >> Open]`
- **restricted aliasing**

This talk: an approach for TSOP of **concurrent** objects

- state transitions aren't always statically trackable
- concurrent objects are typically shared

A simple concurrent object: the **lock**



Invoking `acquire`...

- has an effect if the lock is `IDLE`
- must be possible regardless of the lock state
- **suspends the invoker** if the lock is `BUSY`

Our recipe for concurrent TSOP

General idea

- static checking of protocol compliance, whenever possible
- runtime synchronization, if necessary

A model of concurrent objects

- **Objective Join Calculus** (Fournet *et al.* '03)
- Chemical Abstract Machine (Berry & Boudol '92)

A behavioral type system

- object interface
- object protocols
- sharing control

Outline

1 Introduction

2 TSOP in the Objective Join Calculus

3 Behavioral types for concurrent TSOP

4 A concurrent queue

5 Concluding remarks

Outline

1. Introduction
2. TSOP in the Objective Join Calculus
3. Behavioral types for concurrent TSOP
4. A concurrent queue
5. Concluding remarks

The **lock** in the Objective Join Calculus

```
def o = IDLE | acquire(c)       ▷ o.BUSY | c.reply(o)
 or     BUSY | release          ▷ o.IDLE
in ...
```

- explicit association of state and operations
- explicit state changing
- pending `acquires` are suspended

The **lock** in the Objective Join Calculus

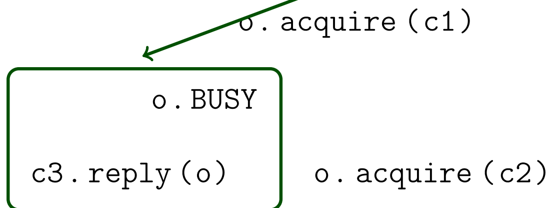
```
def o = IDLE | acquire(c)      ▷ o.BUSY | c.reply(o)
 or     BUSY | release         ▷ o.IDLE
in ...
```

```
o.acquire(c1)

o.IDLE

o.acquire(c3)      o.acquire(c2)
```

- explicit association of state and operations
- explicit state changing
- pending acquires are suspended

The **lock** in the Objective Join Calculus

```
def o = IDLE | acquire(c)    ▷ o.BUSY | c.reply(o)
 or     BUSY | release       ▷ o.IDLE
in ...
```



- explicit association of state and operations
- explicit state changing
- pending acquires are suspended

The **lock** in the Objective Join Calculus

```
def o = IDLE | acquire(c)      ▷ o.BUSY | c.reply(o)
 or     BUSY | release         ▷ o.IDLE
in ...
```

```
o.acquire(c1)

o.BUSY

c3.reply(o)      o.acquire(c2)
```

- explicit association of state and operations
- explicit state changing
- pending `acquires` are suspended

The **lock** in the Objective Join Calculus

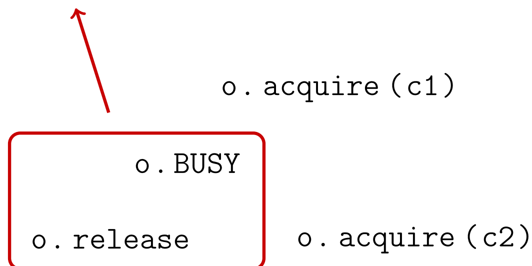
```
def o = IDLE | acquire(c)      ▷ o.BUSY | c.reply(o)
 or     BUSY | release         ▷ o.IDLE
in ...
```

```
o.acquire(c1)

o.BUSY

c3.reply(o)      o.acquire(c2)
```

- explicit association of state and operations
- explicit state changing
- pending `acquires` are suspended

The **lock** in the Objective Join Calculus

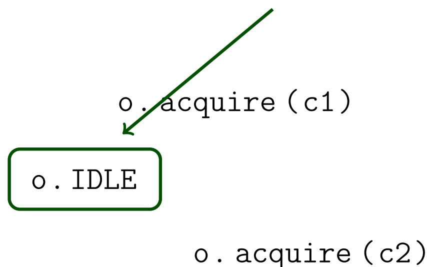
```
def o = IDLE | acquire(c)      ▷ o.BUSY | c.reply(o)
 or     BUSY | release         ▷ o.IDLE
in ...
```



- explicit association of state and operations
- explicit state changing
- pending `acquires` are suspended

The **lock** in the Objective Join Calculus

```
def o = IDLE | acquire(c)     ▷ o.BUSY | c.reply(o)
 or     BUSY | release        ▷ o.IDLE
in ...
```



- explicit association of state and operations
- explicit state changing
- pending `acquires` are suspended

Outline

1. Introduction
2. TSOP in the Objective Join Calculus
3. Behavioral types for concurrent TSOP
4. A concurrent queue
5. Concluding remarks

Behavioral types for the OJC

Observation

- both **state** and **operations** are messages

Which message configurations are legal for the lock?

- there must always be either an `IDLE` or a `BUSY` message
- there can be any number of `acquire`, regardless of state
- there must be one `release` in state `BUSY`, eventually

Behavioral types for the OJC

Observation

- both **state** and **operations** are messages

Which message configurations are legal for the lock?

- there must always be either an `IDLE` or a `BUSY` message
- there can be any number of `acquire`, regardless of state
- there must be one `release` in state `BUSY`, eventually

Types $\sim$ commutative Kleene algebra

$$*\texttt{acquire} \otimes (\texttt{IDLE} \oplus (\texttt{BUSY} \otimes \texttt{release}))$$

Locks, files and their protocols

Which message configurations are legal for files?

- a file is always either `CLOSED` or `OPEN`
- a `CLOSED` file can (but need not) be opened
- an `OPEN` file must be either read or closed

$$(\texttt{CLOSED} \otimes (\mathbb{1} \oplus \texttt{open})) \oplus (\texttt{OPEN} \otimes (\texttt{read} \oplus \texttt{close}))$$

Locks, files and their protocols

Which message configurations are legal for files?

- a file is always either `CLOSED` or `OPEN`
- a `CLOSED` file can (but need not) be opened
- an `OPEN` file must be either read or closed

$$(\mathtt{CLOSED} \otimes (\mathbb{1} \oplus \mathtt{open})) \oplus (\mathtt{OPEN} \otimes (\mathtt{read} \oplus \mathtt{close}))$$

Where's the protocol?

Continuations and protocols

- the Objective Join Calculus is purely **asynchronous**
- sequential composition ⇒ **continuation passing**

```
def o = IDLE | acquire(c)   ▷ o.BUSY | c.reply(o)
 or     BUSY | release      ▷ o.IDLE
in ...
```

$$*\mathtt{acquire}(\ldots\ldots\ldots?\ldots\ldots\ldots) \otimes (\mathtt{IDLE} \oplus (\mathtt{BUSY} \otimes \mathtt{release}))$$

Continuations and protocols

- the Objective Join Calculus is purely **asynchronous**
- sequential composition ⇒ **continuation passing**

o : BUSY

```
def o = IDLE | acquire(c)   ▷ o.BUSY | c.reply(o)
 or     BUSY | release      ▷ o.IDLE
in ...
```

$$*\texttt{acquire}(\ldots\ldots\ldots?\ldots\ldots\ldots) \otimes (\texttt{IDLE} \oplus (\texttt{BUSY} \otimes \texttt{release}))$$

Continuations and protocols

- the Objective Join Calculus is purely **asynchronous**
- sequential composition ⇒ **continuation passing**


```
def o = IDLE | acquire(c)   ▷ o.BUSY | c.reply(o)
 or     BUSY | release      ▷ o.IDLE
in ...
```

$$*\texttt{acquire}(\texttt{.......?.......}) \otimes (\texttt{IDLE} \oplus (\texttt{BUSY} \otimes \texttt{release}))$$

Continuations and protocols

- the Objective Join Calculus is purely **asynchronous**
- sequential composition ⇒ **continuation passing**

o : BUSY

o : release

```
def o = IDLE | acquire(c)  ▷ o.BUSY | c.reply(o)
 or     BUSY | release     ▷ o.IDLE
in ...
```

c : reply(release)

$$*\texttt{acquire}(\texttt{reply}(\texttt{release})) \otimes (\texttt{IDLE} \oplus (\texttt{BUSY} \otimes \texttt{release}))$$

Continuations and protocols

- the Objective Join Calculus is purely **asynchronous**
- sequential composition ⇒ **continuation passing**

o : BUSY

o : release

```
def o = IDLE | acquire(c)   ▷ o.BUSY | c.reply(o)
 or     BUSY | release      ▷ o.IDLE
in ...
```

$$*\texttt{acquire(reply(release))} \otimes (\texttt{IDLE} \oplus (\texttt{BUSY} \otimes \texttt{release}))$$

$$*\texttt{acquire;release} \otimes \cdots$$

Well-typed programs respect object protocols

Theorem (soundness)

If $\texttt{o} : t \vdash P$ *and* $\texttt{m}_1 \cdots \texttt{m}_k \notin t$*,* P **is not sending** $\texttt{m}_1 \cdots \texttt{m}_k$ *to* $\texttt{o}$*.*

Examples

- $\texttt{o} : t_{\texttt{lock}} \vdash \texttt{o.IDLE} \mid P$ and $\texttt{IDLE}, \texttt{release} \notin t_{\texttt{lock}}$
- $\texttt{o} : t_{\texttt{lock}} \vdash \texttt{o.BUSY} \mid P$ and $\texttt{BUSY}, \texttt{release}, \texttt{release} \notin t_{\texttt{lock}}$

Outline

1 Introduction

2 TSOP in the Objective Join Calculus

3 Behavioral types for concurrent TSOP

4 A concurrent queue

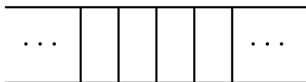
5 Concluding remarks

A concurrent queue

Producer

Consumer

enqueue



peek

dequeue

Protocols/types of the concurrent queue

Producer protocol

- $t_{\text{prod}} = \texttt{enqueue}(\texttt{reply}(t_{\text{prod}}))$

Consumer protocol

- $t_{\text{cons}} = \texttt{peek}(\texttt{none}(t_{\text{cons}}) \oplus \texttt{some}(t_{\text{some}}))$
- $t_{\text{some}} = \texttt{dequeue}(\texttt{reply}(t_{\text{cons}}))$

Queue type

$$
\begin{array}{rl}
& (\,\texttt{NONE} \qquad\qquad \otimes\ t_{\text{prod}} \otimes t_{\text{cons}}\,) \\
\oplus & (\,\texttt{HEAD} \otimes \texttt{TAIL} \otimes t_{\text{prod}} \otimes t_{\text{some}}\,)
\end{array}
$$

Protocols/types of the concurrent queue

Producer protocol

- $t_{\mathtt{prod}} = \mathtt{enqueue}(\mathtt{reply}(t_{\mathtt{prod}}))$

Consumer protocol

- $t_{\mathtt{cons}} = \mathtt{peek}(\mathtt{none}(t_{\mathtt{cons}}) \oplus \mathtt{some}(t_{\mathtt{some}}))$
- $t_{\mathtt{some}} = \mathtt{dequeue}(\mathtt{reply}(t_{\mathtt{cons}}))$

Queue type

producer and consumer share the queue

$$
\begin{array}{cccccccc}
 & (\ \mathtt{NONE} & & & \otimes\ t_{\mathtt{prod}} & \otimes\ t_{\mathtt{cons}}\ ) \\
\oplus & (\ \mathtt{HEAD} & \otimes & \mathtt{TAIL} & \otimes\ t_{\mathtt{prod}} & \otimes\ t_{\mathtt{some}}\ )
\end{array}
$$

Outline

1 Introduction

2 TSOP in the Objective Join Calculus

3 Behavioral types for concurrent TSOP

4 A concurrent queue

5 Concluding remarks

Wrap-up

1. TSOP in a **concurrent** setting
   - static protocol enforcement + runtime support

2. the OJC is a **natural model** for concurrent TSOP
   - pairing state with operations + explicit state changing

3. first **behavioral type theory** for OJC
   - interface + protocols + sharing control

In the paper

- more examples (iterators, full-duplex channels)
- formal definitions (proceedings) and proofs (tech report)

Ongoing work

Implementation

- integration into existing programming language

Runtime support for join definitions

- libraries for various programming languages, or
- direct implementation (message queues + condition vars)

Protocol enforcement

- behavioral type checker as a pre- (or post-) processor

A concurrent queue

```
def o =
   NONE    | enqueue(m,c) ▷
     let x = new Node(m) in
     o.HEAD(x) | o.TAIL(x) | c.reply(o)
or TAIL(x) | enqueue(m,c) ▷
     let y = new Node(m) in
     x↑next := y | o.TAIL(y) | c.reply(o)
or NONE    | peek(c) ▷ o.NONE | c.none(o)
or HEAD(x) | peek(c) ▷ o.HEAD(x) | c.some(o)
or HEAD(x) | TAIL(y) | dequeue(c) ▷
     if x = y then
       o.NONE | c.reply(x↑val, o)
     else
       o.HEAD(x↑next) | o.TAIL(y) | c.reply(x↑val,
in o.NONE | ...
```