A Gentle Introduction to Concurrent TypeState-Oriented Programming

Silvia Crafa Luca Padovani

1 Preamble

An object-oriented programming language that supports *TypeState-Oriented Programming* (TSOP for short) is recognizable by two distinctive features:

1. A suit of built-in constructs for defining objects with state-sensitive interfaces and behaviors.
2. A type system that detects object protocol violations.

In order to detect protocol violations, the type system must be able to track (a suitable abstraction of) the state of objects with structured protocols, meaning that references to these objects must be shared in controlled ways. It follows that languages supporting TSOP rely on some form of aliasing control.

In this tutorial we illustrate *Concurrent TSOP*, namely TSOP in a context where objects are concurrently accessed and possibly modified by several processes. By definition, concurrent objects are generally aliased. This makes it difficult for the type system alone to track the state of concurrent objects, and therefore to determine whether the invocation of a certain operation at a certain point of the program is legal or not. To tackle the challenge, we propose a hybrid technique that combines *static checks* (performed by the type system) and *dynamic synchronizations* (provided by the runtime environment).

We advocate the use of the Objective Join Calculus (OJC for short) as a suitable formal model for studying Concurrent TSOP for the following reasons:

- the OJC features objects, concurrency and high-level synchronization patterns;
- the idiomatic programming of objects in the OJC bears strong similarities with sequential TSOP and supports the modeling of multidimensional states and of partial/concurrent state updates;
- the lack of a formal distinction between state and operations in the OJC permits the definition of a simple, unified type language for describing object interfaces, protocols, and aliasing capabilities.

The type system is detailed in:

- Silvia Crafa and Luca Padovani, **The Chemical Approach to Typestate-Oriented Programming**, Proceedings of OOPSLA'15, `http://dx.doi.org/10.1145/2814270.2814287`.

2 Concurrent Linear Buffer

The theme of this introduction is the modeling of a *concurrent object* in the Objective Join Calculus and the specification of its *type* so that a number of programming errors can be detected by a type checker. The object that we consider, and that we elaborate through a series of steps, is a *concurrent linear buffer*, that is a buffer used for exchanging one value from a producer process to a consumer process:

- the "linear" qualification indicates that the buffer is meant to be discarded after the exchange.
- the "concurrent" qualification means that producer and consumer access the buffer concurrently without coordinating in any way. We have to rely on the runtime synchronization semantics of the OJC to make sure that `Get` is executed only after the `Put` has been completed.

We can describe the life cycle of a concurrent linear buffer as the transition diagram which specifies the three states of `buffer` can be (`EMPTY`, `FULL` or depleted) and the effect of `Put` and `Get` on them:

In the Objective Join Calculus, the buffer is modeled as an object that handles four *messages* tagged `EMPTY`, `FULL`, `Put` and `Get`. The former two messages encode the state of the buffer. The lack of both `EMPTY` and `FULL` means that the buffer is depleted. The `Put` and `Get` messages represent the operations.

```
1 object buffer
2 [ EMPTY   & Put(v) ▶ buffer!FULL(v)
3 | FULL(v) & Get(c) ▶ c!Reply(v) ]
4 buffer!EMPTY &                  // CONSTRUCTOR
5
6 buffer!Put(42) &                // PRODUCER
7 System.Print(buffer.Get); done // CONSUMER
```

- The object behavior is determined by a set of *reaction rules* of the form `pattern ▶ body`: when all the messages in `pattern` reach `buffer` the reaction may fire, causing the atomic consumption of these messages and the production of those in `body`.
- The operators `&` and `!` respectively represent *parallel composition* and *asynchronous output*.
- The operator `.` is *synchronous output*. This operator can only be used in a context where a value or continuation is expected, like *within an expression* (`buffer.Get`) or *before sequential composition* `;` (`System.Print`). Synchronous operations are translated into asynchronous operations with explicit continuation passing. There is no need to manually create and pass continuations (contrast the continuation `c` in the reaction on line 3 and the lack thereof in the invocation on line 7).

In principle, the definition of `buffer` allows producer and consumer to send arbitrary combinations of `EMPTY`, `FULL`, `Put` and `Get` messages. In practice, not all such combinations are desirable or sensible:

- The producer may attempt to `Put` two values in the buffer. Since the first `Put` that is processed changes the state of the buffer, the second `Put` will never trigger a reaction and becomes junk:

```
buffer!Put(42) & buffer!Put(43)
```

- The consumer may attempt to `Get` two values from the buffer violating its linearity. In this case, one `Get` message becomes junk and the consumer will starve:

```
System.Print(buffer.Get + buffer.Get); done
```

- Producer and/or consumer may interfere with the definition of `buffer`, for example by sending state messages directly. This will likely disrupt the intended semantics of the linear buffer and/or generate junk messages and/or cause other undesired behavior:

```
buffer!EMPTY & buffer!FULL(43)
```

3 Types for Concurrent Objects

- We introduce a *type language* for specifying:
  - the *interface* of objects, namely which messages they accept and the type of their arguments;
  - the *protocol* of objects, namely how the messages in their interface are supposed to be sent;
  - whether and how objects can be *shared/aliased*.
- Types are *commutative regular expressions* over message types:

$$t, s, r \quad ::= \quad \mathtt{0} \;\Big|\; \mathtt{1} \;\Big|\; \mathtt{M}(t_1, \ldots, t_n) \;\Big|\; t + s \;\Big|\; t \cdot s \;\Big|\; \star t$$

  - There is no legal way of using an object of type `0`, even discarding the object is illegal.
  - An object of type `1` can be discarded. No other usage is allowed.
  - An object of type $\mathtt{M}(t_1, \ldots, t_n)$ must be used for sending an `M`-tagged message with n arguments having type $t_1, \ldots, t_n$ respectively.
  - An object of type $t + s$ must be used either according to t or according to s. It follows that there must be *one* process that performs the choice.
  - An object of type $t \cdot s$ must be used both according to t and also according to s, possibly concurrently by two processes.
  - An object of type $\star t$ can be used any number of times, each time according to t.
- There is no type for expressing the *sequentiality* of actions since the Objective Join Calculus, which is purely asynchronous, has no native construct for sequential composition. Structured sequential protocols can be modeled by means of continuation passing.
- The following semantic equivalences hold for all types t, s and r:

$\mathtt{1} \cdot t = t \cdot \mathtt{1} = t$	($\mathtt{1}$ is the unit of $\cdot$)	$t \cdot s = s \cdot t$	($\cdot$ is commutative)
$\mathtt{0} \cdot t = t \cdot \mathtt{0} = \mathtt{0}$	($\mathtt{0}$ is absorbing for $\cdot$)	$t \cdot (s + r) = t \cdot s + t \cdot r$	($\cdot$ distributes over $+$)
$\mathtt{0} + t = t + \mathtt{0} = t$	($\mathtt{0}$ is the unit of $+$)	$\star t = \mathtt{1} + t \cdot \star t$	(unlimited usage)

- *Subtyping* is defined according to the usual safe substitution principle: if $t \sqsubseteq s$, then it is safe to use an object of type t wherever an object of type s is expected. Subtyping is characterized by the above equivalences, *inverse language inclusion* and *argument contravariance*:

$$t + s \sqsubseteq t \qquad\qquad t \sqsubseteq s \Rightarrow \mathtt{M}(s) \sqsubseteq \mathtt{M}(t)$$

4 Typed Concurrent Linear Buffer

In order to come up with a type for `buffer`, it may help looking again at its code and transition diagram in Section 2. The transition diagram does not emphasize a fundamental difference between `Put` and `Get`:

- The producer knows that `buffer` is `EMPTY` when the message `Put` is sent.
- The consumer does not (and cannot) know whether `buffer` is `EMPTY` or `FULL` when the `Get` message is sent: the actual state of `buffer` depends on whether its first reaction has already fired (in which case `EMPTY` and `Put` have been consumed, and `buffer` is `FULL`) or not (in which case `buffer` is still `EMPTY` and a `Put` operation will eventually change its state to `FULL`).

In summary, the transition diagram only illustrates the effect of the `Put` and `Get` operations on the state of `buffer`, but does not make it clear that the consumer is allowed to send a `Get` message without knowing for sure whether `buffer` is already `FULL`. The type language we have introduced in the previous section allows us to fill this gap, thus:

```
1 object buffer
2   : (EMPTY·Put(?) + FULL(?))·Get(?) + 1
3 [ EMPTY   & Put(v) ▶ buffer!FULL(v)
4 | FULL(v) & Get(c) ▶ c!Reply(v) ]
5 buffer!EMPTY &
6
7 buffer!Put(42) &
8 System.Print(buffer.Get); done
```

- We specify the type of an object next to its name (line 2).
- We write `?` for unspecified types, which can generally be inferred. It is allowed to write `?` anywhere the type of a message argument is expected.
- The fact that the operation message `Put` is only combined with `EMPTY` indicates that the producer can `Put` a value in `buffer` only when `buffer` is `EMPTY`.
- On the contrary, the fact that the operation message `Get` combines with both `EMPTY` and `FULL` indicates that the consumer is allowed to issue `Get` not knowing whether `buffer` is `EMPTY` or `FULL`. The runtime synchronization mechanism of the OJC ensures that the second reaction (line 4) fires only when `buffer` is `FULL`.
- We use the type `1` to indicate the depleted `buffer`, to which no message is targeted (no junk).
- With this type for `buffer` all the mistakes we have described in Step 2 yield type errors.

5 Digression: Sequential Linear Buffer

Having seen the type of the concurrent buffer, it may be interesting to also have a look at that of the sequential buffer, namely a buffer where the `Get` operation can only be issued when the `buffer` is surely `FULL`. The first attempt is the following

```
1 object buffer
2   : EMPTY·Put(?) + FULL(?)·Get(?) + 1
3 [ EMPTY   & Put(v) ▶ buffer!FULL(v)
4 | FULL(v) & Get(c) ▶ c!Reply(v) ]
5 buffer!EMPTY &
6
7 buffer!Put(42) &
8 System.Print(buffer.Get); done // ERROR //
```

in which we have simply changed the type of `buffer` specifying that `Get` cannot be issued when `buffer` is `EMPTY`. This code is obviously wrong for the consumer issues a `Get` message (line 8) at a time when the state of `buffer` is not guaranteed to be `FULL` (the `EMPTY` and `Put` messages issued at lines 5 and 7 may not have been consumed yet by the reaction on line 3).

As a second attempt, we may just remove the consumer altogether, obtaining:

```
1 object buffer
2   : EMPTY·Put(?) + FULL(?)·Get(?) + 1
3 [ EMPTY   & Put(v) ▶ buffer!FULL(v) // ERROR //
4 | FULL(v) & Get(c) ▶ c!Reply(v) ]
5 buffer!EMPTY &
6
7 buffer!Put(42)
```

Also this version is obviously flawed given that there is no longer a consumer process. What is interesting though is that the error message hints at the solution of the problem. The type error originates from line 3, where the reaction consumes the `EMPTY` and `Put` messages and produces a `FULL` message. At this point, all the outputs targeted to `buffer` (lines 3, 5 and 7) have been performed and `buffer` is in a configuration that contains `FULL` alone. This configuration is illegal according to the type given on line 2, since the sole configuration that contains a `FULL` message should also contain a `Get` message. The idea is that the capability to issue a `Get` message originates as soon as the `FULL` message is issued.

This discussion leads us to equip `Put` with a continuation, and to change the reaction on line 3 so that another reference to `buffer` (having type `Get`) is communicated back to the producer, which can now turn into a consumer and retrieve the value from the buffer:

```
1 object buffer
2   : EMPTY·Put(?,?) + FULL(?)·Get(?) + 1
3 [ EMPTY   & Put(v,p) ▶ buffer!FULL(v) & p!Reply(buffer)
4 | FULL(v) & Get(c)   ▶ c!Reply(v) ]
5 buffer!EMPTY &
6
7 let buffer = buffer.Put(42) in
8 System.Print(buffer.Get); done
```

The code on lines 7–8 is now purely sequential. First of all, a `Put` message is sent to `buffer` (line 7). This operation returns a continuation, which is another reference to the very same `buffer` except that the type of this reference allows sending the `Get` message. This is what happens in line 8, where the buffer protocol comes finally to an end.

Note that we had to slightly retouch the signature of `Put` (line 2) which now has 2 arguments.

6 Adding a Blocking `Put`

In this step we demonstrate the use of complex join patterns to extend the linear buffer with a blocking `Put` operation. The Objective Join Calculus is purely asynchronous, so there is no such thing as a truly "blocking" operation. What we mean here is to realize a version of `Put` that notifies the producer as soon as the value in the buffer is sent to the consumer. This way, after issuing a `Put`, the producer may block waiting for this notification.

```
1  object buffer
2    : (EMPTY·(Put(?) + Put(?,?)) + FULL(?))·Get(?) + 1
3  [ EMPTY   & Put(v)            ▶ buffer!FULL(v)
4  | EMPTY   & Put(v,p) & Get(c) ▶ p!Reply & c!Reply(v)
5  | FULL(v)           & Get(c) ▶ c!Reply(v) ]
6
7  buffer!EMPTY &
8
9  buffer.Put(42);
10 System.Print("OK"); done &
11
12 System.Wait(3);
13 System.Print(buffer.Get); done
```

- We define a new reaction that synchronizes simultaneously on `EMPTY`, `Put` and `Get` (line 4). When all three messages have reached `buffer` and the reaction fires, the producer `p` is notified and the consumer `c` receives the value `v`.
- The new reaction changes the buffer directly from `EMPTY` to the depleted state.
- The non-blocking `Put` is still supported (line 3). From the the consumer's viewpoint the `Get` operation behaves just like before.
- No confusion may arise between the non-blocking `Put` which has 1 argument (line 3) and the blocking `Put` which has 2 arguments (line 4). This is an example of *message overloading*, which is resolved by looking at the number of arguments supplied to the `Put` message.
- The type now specifies that the producer may choose the version of `Put` to use (line 2). It is illegal for the producer to use *both* the blocking and the non-blocking versions of `Put`.
- Using the blocking `Put` requires a continuation which is created and passed implicitly (line 9). The producer will not print `OK` before the value has been delivered to the consumer (line 10).

7 Adding a Non-Blocking `Get`

In this step we further extend the buffer with a non-blocking `Get` operation, which we call `Try`.

```
1  object buffer
2    : (EMPTY·(Put(?) + Put(?,?)) + FULL(?))·(Get(?) + Try(?)) + 1
3  [ EMPTY   & Put(v)            ▶ buffer!FULL(v)
4  | EMPTY   & Put(v,p) & Get(c) ▶ p!Reply & c!Reply(v)
5  | EMPTY              & Try(c) ▶ buffer!EMPTY & c!Empty(buffer)
6  | EMPTY   & Put(v,p) & Try(c) ▶ p!Reply & c!Reply(v)
7  | FULL(v)            & Try(c) ▶ c!Reply(v)
8  | FULL(v)            & Get(c) ▶ c!Reply(v) ]
9
10 buffer!EMPTY &
11
12 System.Wait(3);
13 buffer!Put(42) &
14
15 object Consumer
16 [ Run(buffer, r) ▶
17     case buffer.Try of
18     [ Empty(buffer) ▶
19         System.Print("Buffer still empty");
20         System.Wait(2);
21         Consumer!Run(buffer, r)
22     | Reply(v) ▶
23         System.Print(v);
24         r!Reply
25     ]
26 ]
27
28 Consumer.Run(buffer); done
```

- We add 3 reactions for `Try`, one for each non-depleted configuration of `buffer` (lines 5-6).
- If `buffer` is `FULL` or there is a pending blocking `Put`, `Try` behaves just like `Get` (lines 6–7).
- If `buffer` is `EMPTY` and there is no `Put` message, `Try` notifies the consumer with an `Empty` message (line 5). Two important remarks are in order:
  - the `EMPTY` message is re-generated because the state of `buffer` has not changed;
  - the `Empty` notification sent to the consumer `c` contains a reference to `buffer`. This is because `c` will have to use `buffer` for either another `Try` or a `Get`.
- The type of `buffer` now specifies that the consumer may choose either `Get` or `Try` (line 2).
- We cannot use the tag `Get` and rely on message overloading for the new operation (as we have done for the two versions of `Put`) because `Get` and `Try` have the same number of arguments.
- The `Consumer` is represented as an object (lines 15–26). `Consumer` tries to get a value (line 17) and analyzes the notification returned by `buffer` with a `case` construct (lines 17–25). If `Consumer` receives `Empty` (line 18) it waits for a little while (lines 19–20) before making another attempt (line 21). If `Consumer` receives `Reply` (line 22) it prints the value (line 23) and terminates (line 24).
- The property that the `case` is *exhaustive*, namely that it handles (at least) all the possible responses from `buffer`, is verified by the type checker. Removing either case results in a type error.

8 Complete Type Specification

This step illustrates the use of *type names* to give a complete protocol specification for `buffer`.

```
1  type #Producer = Put(#Number) + Put(#Number, Reply)
2  and  #Consumer = Get(Reply(#Number)) +
3                   Try(Empty(#Consumer) + Reply(#Number)) in
4
5  object buffer
6    : (EMPTY·#Producer + FULL(#Number))·#Consumer + 1
7  [ EMPTY   & Put(v)            ▶ buffer!FULL(v)
8  | EMPTY   & Put(v,p) & Get(c) ▶ p!Reply & c!Reply(v)
9  | EMPTY              & Try(c) ▶ buffer!EMPTY & c!Empty(buffer)
10 | EMPTY   & Put(v,p) & Try(c) ▶ p!Reply & c!Reply(v)
11 | FULL(v)            & Try(c) ▶ c!Reply(v)
12 | FULL(v)            & Get(c) ▶ c!Reply(v) ]
13
14 buffer!EMPTY &
15
16 System.Wait(3);
17 buffer!Put(42) &
18
19 object Consumer
20 [ Run(buffer, r) ▶
21     case buffer.Try of
22     [ Empty(buffer) ▶
23         System.Print("Buffer still empty");
24         System.Wait(2);
25         Consumer!Run(buffer, r)
26     | Reply(v) ▶
27         System.Print(v);
28         r!Reply
29     ]
30 ]
31
32 Consumer.Run(buffer); done
```

- Type names and their expansion can be given at the beginning of the script (lines 1–3).
- The type `#Consumer` is equi-recursive: the name `#Consumer` occurs also on the right hand side of `=`. Mutually recursive types are allowed.
- Non-contractive type definitions such as

  ```
  type #A = #A
  ```

  or

  ```
  type #A = #A + Receive(Reply(#Number))
  ```

  or

  ```
  type #A = #B and #B = #A
  ```

  are illegal.
- Defined type names can be used in the specification of object types (line 6).

9 From Linear to Persistent Buffers

In this step we demonstrate the definition of an object that can be *shared* among – and *used* by – exactly one producer and an arbitrary number of consumers in a possibly concurrent way. We modify the concurrent buffer so that the value stored therein can be retrieved by several consumers. For simplicity, we consider the version of the buffer without the blocking `Put` and the non-blocking `Try`.

A persistent buffer has only two states, `EMPTY` and `FULL`. Once the content of the buffer has been set, the buffer remains `FULL` so that its content can be retrieved an arbitrary number of times.

```
1  object buffer
2    : (EMPTY·Put(?) + FULL(?))·⋆Get(?)
3  [ EMPTY   & Put(v) ▶ buffer!FULL(v)
4  | FULL(v) & Get(c) ▶ buffer!FULL(v) & c!Reply(v) ]
5
6  buffer!EMPTY &
7
8  System.Wait(3);  // Expensive computation
9  buffer!Put(42) &
10
11 System.Print(buffer.Get); done &
12 System.Print(buffer.Get); done &
13 System.Print(buffer.Get); done &
14 System.Print(buffer.Get); done &
15 System.Print(buffer.Get); done &
16 System.Wait(5); done
```

- When the content of the persistent buffer is read (line 4) the `FULL` message is restored, permitting subsequent `Get` operations.
- The type of `buffer` specifies that only one process is entitled to `Put` a value in it and that arbitrarily many consumers can access it (line 2).
- The $\star$ type constructor can be used to specify an object that can be shared/aliased without restrictions. This is made possible by the equivalence

  $$\star t = \star t \cdot \star t$$

  that allows a reference of type $\star t$ to be "duplicated", generating two references of type $\star t$. This relation allows the simultaneous use of `buffer` by several consumers (lines 11–15).
- An object of type $\star t$ need not be used and can be discarded, as indicated by the relation

  $$\star t \sqsubseteq 1$$

10 Properties of Well-Typed Programs

The main property guaranteed for well-typed programs is protocol fidelity:

Protocol Fidelity: If the type of an object prohibits sending a message `M` when the object is in a certain state, then an object in that state will never have a pending `M` message. For example: `Put` is never issued when `buffer` is `FULL`; no operation is issued on a depleted `buffer`.

A straightforward consequence of protocol fidelity is message boundedness:

Boundedness: The type of an object gives precise upper bounds on the number of messages simultaneously targeted to the object. In particular, if there is no $\star$ in the type, then there is a finite upper bound obtained by counting the number of occurrences of tags in the largest valid message configuration. All the versions of `buffer` we have discussed except the persistent one are bounded. Moreover, in the persistent buffer only the `Get` message is unbounded.

The following properties can be determined by means of a reachability analysis combining the information in the type of an object and the structure of its reaction rules:

Junk Detection: It is possible to decide whether there are messages that are never consumed during the entire lifetime of an object, or that will never be consumed after the object reaches a certain state.

Dead Code Detection: It is possible to decide whether there are reactions that will never fire during the entire lifetime of an object, or that will never fire after an object reaches a certain state.

Type information can be used for optimizing objects in well-typed programs:

Garbage Collection: When a reaction changes the type of an object to `1`, the object can be safely deallocated. Even if there exist live references to the object, these must have type `1` meaning that processes with a reference to the object can only discard it. This happens for all versions of the linear buffer, as soon as `Get` is performed.

The type system does *not* guarantee deadlock freedom. Below is a simple example of well-typed program that causes a deadlock.

```
1 object buffer
2   : (EMPTY·Put(?) + FULL(?))·Get(?) + 1
3 [ EMPTY   & Put(v) ▶ buffer!FULL(v)
4 | FULL(v) & Get(c) ▶ c!Reply(v) ]
5 buffer!EMPTY &
6
7 buffer!Put(buffer.Get) // DEADLOCK //
```

Note that `buffer` is correctly initialized (line 5) and the outputs for both `Put` and `Get` do occur in the syntax of the program. However, the argument of `Put` is the very same value that should be returned by `Get`. This circularity is not detected by the type system.