Concurrent Typestate-Oriented Programming in Java

Rosita Gerbo and Luca Padovani

University of Torino, Italy

TypeState-Oriented Programming

objects with a state-sensitive interface

- File — don't read if closed
- TCP socket — don't send if disconnected
- Stack — don't pop if empty
- Bounded buffer — don't put if full
- ...

*"...approximately **7.2%** of all types defined protocols, while **13%** of classes were clients of types defining protocols."*

[Beckman et al., 2011]

typestate-oriented programming in Plaid [Aldrich et al., 2009]

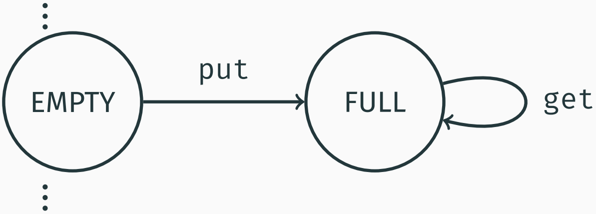
```
class Buffer { }

state EMPTY of Buffer {
  public void put(int x) {      [EMPTY >> FULL]
    this <- FULL { this.value = x; }
} }

state FULL of Buffer {
  private int value;
  public int get() {            [FULL >> EMPTY]
    int x = this.value;
    this <- EMPTY {}
    return x;
} }
```

goal: concurrent typestate-oriented programming in Java

```
public class CompletableFuture<T> {
```




```
}
```

- can be **completed once** and **read many times**
- if read while uncompleted, the reader **suspends**

A model for concurrent TSOP

TSOP with the Join Calculus [Crafa and Padovani, 2017]

```
new obj = EMPTY   & put(x) ▷ obj!FULL(x)
        | FULL(x) & get(u) ▷ obj!EMPTY() & u!reply(x)
```

join patterns	⇒	paired states and operations
reactions	⇒	state transitions

About missing reactions

- `EMPTY   & get(u)` — OK, reader suspends
- `FULL(x) & put(y)` — ☠ protocol violation

Type of a completable future

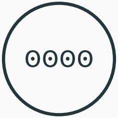
$$*\mathtt{get} \cdot (\mathtt{EMPTY} \cdot \mathtt{put} + \mathtt{FULL})$$

a generative approach to concurrent TSOP in Java

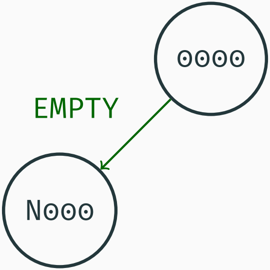
```
public class CompletableFuture<T> {
    ⋮
  EMPTY   & put(x) ▷ this!FULL(x)
  FULL(x) & get(u) ▷ this!EMPTY() & u!reply(x)
    ⋮
}
```

Compiling (typed!) join patterns

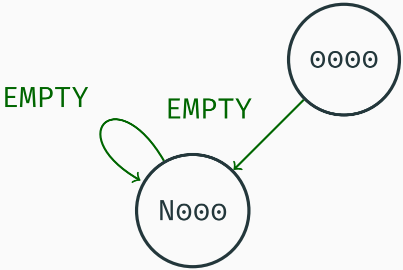
matching automata [Le Fessant and Maranget, 1998]



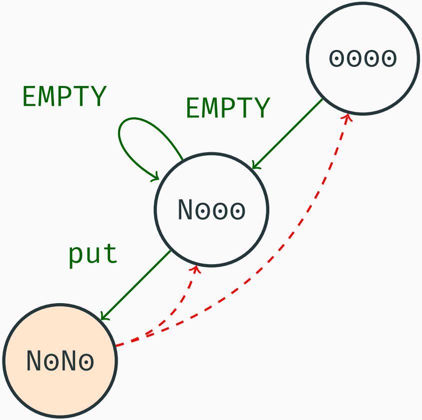
- automaton state = approximate description of mailbox



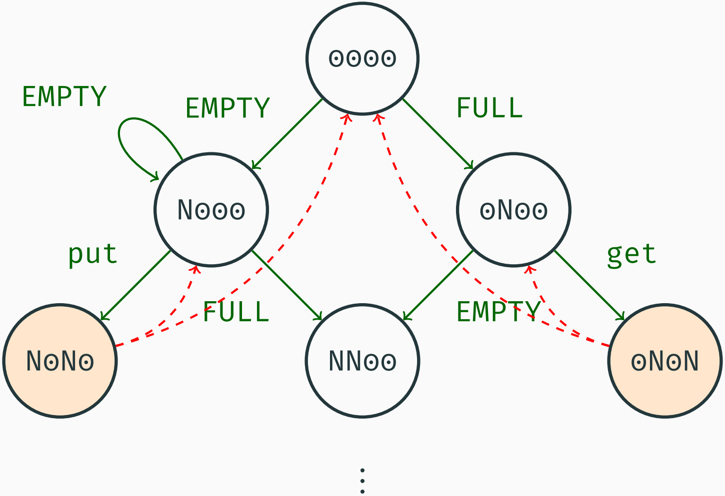
- automaton state = approximate description of mailbox
- automaton transition = receive



- automaton state = approximate description of mailbox
- automaton transition = receive



- automaton state = approximate description of mailbox
- automaton transition = receive or react



- automaton state = approximate description of mailbox
- automaton transition = receive or react

matching automata and behavioral types

$$*\texttt{get} \cdot (\texttt{EMPTY} \cdot \texttt{put} + \texttt{FULL})$$

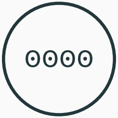
- `EMPTY`, `FULL` and `put` are 1**-bounded**
- `get` is **unbounded**
- `EMPTY` and `FULL` are **mutually exclusive**
- ...

Refining the matching automaton

- bounded messages can be counted precisely
- not all automaton states are meaningful

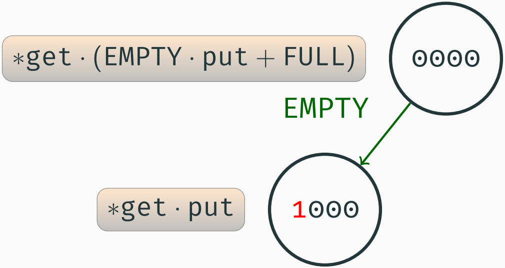
compiling behaviorally typed join patterns

$*\mathtt{get} \cdot (\mathtt{EMPTY} \cdot \mathtt{put} + \mathtt{FULL})$



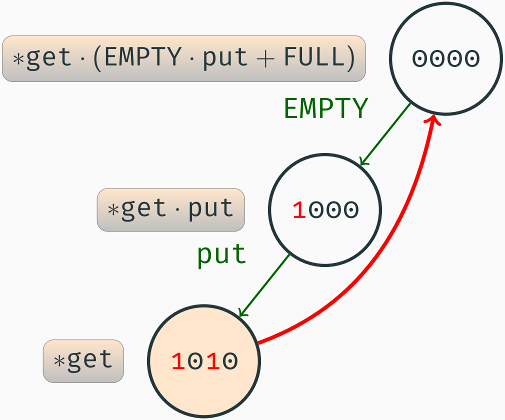
- initial state $\Rightarrow$ use object type

compiling behaviorally typed join patterns



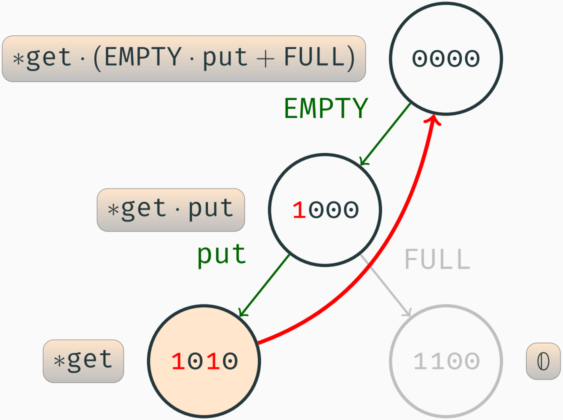
- initial state ⇒ use object type
- other states ⇒ compute with derivative [Brzozowski, 1964]

compiling behaviorally typed join patterns



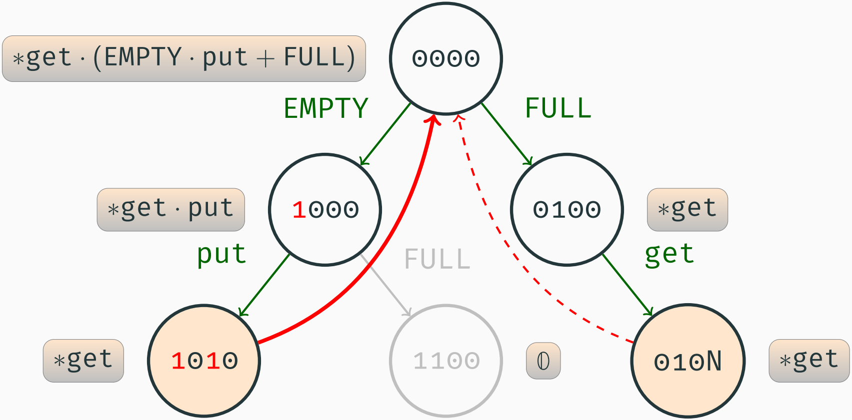
- initial state ⇒ use object type
- other states ⇒ compute with derivative [Brzozowski, 1964]

compiling behaviorally typed join patterns



- initial state ⇒ use object type
- other states ⇒ compute with derivative [Brzozowski, 1964]
- derived type is 𝟘 ⇒ state is illegal ⇒ discard

compiling behaviorally typed join patterns



- initial state ⇒ use object type
- other states ⇒ compute with derivative [Brzozowski, 1964]
- derived type is 𝟘 ⇒ state is illegal ⇒ discard

Generating Java code

describing typed join patterns with Java annotations

```
new obj = EMPTY   & put(x) ▷ obj!FULL(x)
        | FULL(x) & get(u) ▷ obj!FULL(x) & u!reply(x)

@Protocol("*get·(EMPTY·put + FULL)")
public class CompletableFuture<T> {
  @State     void EMPTY();
  @State     void FULL(T x);
  @Operation T    get();
  @Operation void put(T x);
  @Reaction  void when_EMPTY_put(T x) { this.FULL(x); }
  @Reaction  T    when_FULL_get(T x)  { this.FULL(x);
                                        return x; }
  public CompletableFuture()          { this.EMPTY(); }
}
```

automaton and mailbox representation

```
public class CompletableFuture<T> {
  private int state      = 0;    // initial state
  private T   queue_FULL = null; // no FULL message
  private int queue_get  = 0;    // no get messages
  ...
}
```

	no arguments	argument of type T
bounded message	—	T
unbounded message	int	Queue<T>

code generated for state methods

```
synchronized private void FULL(T x) {
  queue_FULL = x; // store message
  ...             // update automaton state
  if (illegal state reached)
    throw new IllegalStateException();
  if (reaction state reached) notify();
}
```

- `private` ⇒ state changes allowed only from within class
- **no blocking actions** ⇒ “asynchronous” message

code generated for operation methods

```
synchronized public T get() {
  queue_get++;      // store message
  ...               // update automaton state
  if (illegal state reached)
    throw new IllegalStateException();
  while (!reaction state) wait();
  T x = queue_FULL; // consume message
  ...               // update automaton state
  return when_FULL_get(x); // invoke reaction
}
```

- **blocking actions** $\Rightarrow$ “synchronous” message

Conclusions

This approach in a nutshell

- write Java classes almost as if concurrent TSOP was native
- a **code generator** takes care of the low-level details

Key insight

- **states as messages**: no need for TSOP-specific constructs

Benefits of behavioral types

- prune states of Le Fessant and Maranget's matching automaton
- reduce non-determinism
- detect protocol violations (at runtime)
- reduce overhead due to message queues

Thank you. Questions?

Q: Why Java and not XYZ?

Mostly practical reasons:

- there is a working Java parser for Haskell
- Java has official annotations

Q: Is it available?

Search for **EasyJoin** on Zenodo

Q: Is it portable?

No strong dependency on Java, having official annotations helps

References

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of OOPSLA'09*, pages 1015–1022. ACM, 2009.

Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In *Proceedings of ECOOP'11*, volume LNCS 6813, pages 2–26. Springer, 2011.

Janusz A. Brzozowski. Derivatives of Regular Expressions. *Journal of ACM*, 11(4): 481–494, 1964.

Silvia Crafa and Luca Padovani. The Chemical Approach to Typestate-Oriented Programming. *ACM Transactions on Programming Languages and Systems*, 39: 13:1–13:45, 2017.

Fabrice Le Fessant and Luc Maranget. Compiling join-patterns. *Electr. Notes Theor. Comput. Sci.*, 16(3):205–224, 1998.