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

Outline

1. A historical perspective
2. Concurrency and TSOP
3. Behavioral types
4. Practicalities
5. Concluding remarks

Outline

1. A historical perspective
2. Concurrency and TSOP
3. Behavioral types
4. Practicalities
5. Concluding remarks

Typestate: A Programming Language Concept for Enhancing Software Reliability

ROBERT E. STROM AND SHAULA YEMINI

Abstract—We introduce a new programming language concept called *typestate*, which is a refinement of the concept of *type*. Whereas the type of a data object determines the set of operations *ever* permitted on the object, typestate determines the subset of these operations which is permitted in a particular context.

Typestate tracking is a program analysis technique which enhances program reliability by detecting at compile-time syntactically legal but semantically undefined execution sequences. These include, for example, reading a variable before it has been initialized, dereferencing a pointer after the dynamic object has been deallocated, etc. Typestate tracking detects errors that cannot be detected by type checking or by conventional static scope rules. Additionally, typestate tracking makes it possible for compilers to insert appropriate finalization of data at exception points and on program termination, eliminating the need to support finalization by means of either garbage collection or unsafe deallocation operations such as Pascal's dispose operation.

By enforcing typestate invariants at compile-time, it becomes practical to implement a "secure language"—that is, one in which all successfully compiled program modules have fully defined execution-time effects, and the only effects of program errors are incorrect output values.

This paper defines typestate, gives examples of its application, and shows how typestate checking may be embedded into a compiler. We discuss the consequences of typestate checking for software reliability and software structure, and conclude with a discussion of our experience using a high-level language incorporating typestate checking.

scope checking avoid some but not all nonsense. In Section II, we informally present the typestate concept, give examples of its use, and discuss the benefits which accrue from compile-time tracking of typestate. In Section III, we give a more formal definition of typestate, and present an algorithm for verifying the typestate consistency of programs. In Section IV, we discuss the interaction between typestate and other language design issues, such as composite user-defined types, independent compilation, and aliasing. We discuss our experience as designers and users of NIL—a secure programming language incorporating compile-time typestate tracking. Section V presents some conclusions and comparisons with related work.

A. Type Checking

From the perspective of software reliability, one of the most important properties of the concept of type is that it supports the automatic detection of certain kinds of errors.

The *type* of a variable name determines the set of operations which may be applied to that variable. For instance, if X is of type **real** it is allowed to appear in the context

Typestate = Type + Behavior (Strom & Yemini '86)

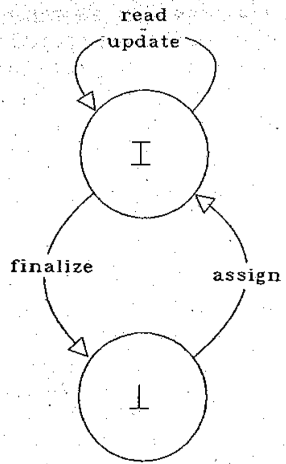


Fig. 1. Typestate transition graph for type **integer:** the scalar type integer illustrates the simplest nontrivial typestate transition graph. There are two typestates: ⊥ (intuitively "uninitialized") and I ("intuitively initialized").

Typestate for objects (DeLine & Fähndrich '04, Microsoft)

```
[ TypeStates("Raw", "Bound", "Connected", "Closed") ]
class Socket {

  [ Post("Raw"), NotAliased ]
  Socket();

  [ Pre("Raw"), Post("Bound"), NotAliased ]
  void Bind(string endpoint);

  [ Pre("Bound"), Post("Connected"), NotAliased ]
  void Connect();

  [ Pre("Connected") ]
  void Send(string data);

  [ Pre("Connected") ]
  string Receive();

  [ Pre("Connected"), Post("Closed"), NotAliased ]
  void Close();
}
```

Typestate-oriented programming in **Plaid**

(Aldrich *et al.* '09, CMU)

```
state File {
  public final String filename;
}

state OpenFile extends File {
  private CFilePtr filePtr;
  public int read() { ... }
  public void close() [OpenFile>>ClosedFile]
    { ... }
}

state ClosedFile extends File {
  public void open() [ClosedFile>>OpenFile]
    { ... }
}
```

- typestate becomes a **native language feature**

Typestate-oriented programming: summary

Objective

- **static** enforcement of object protocols

Mechanisms

- **abstract state** annotations in types — `Closed`, `Open`
- tracking of **state transitions** — `[Closed >> Open]`
- **aliasing control** — linearity

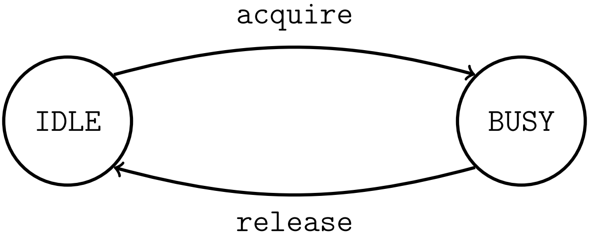
Our contribution: TSOP of **concurrent** objects

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

Outline

1. A historical perspective
2. Concurrency and TSOP
3. Behavioral types
4. Practicalities
5. Concluding remarks

A simple concurrent object: the **lock**



Invoking `acquire`...

- has an effect **if** the lock is `IDLE`
- we **don't (and cannot) know** when the lock is `IDLE`
- must be allowed **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, Laneve, Maranget, Rémy '03)
  (not just because there are objects!)

A behavioral type system

- **New!**
  (interfaces + protocols + aliasing control)

Why the OJC? Intriguing similarities!

Plaid

```
class File
{ public String name; }

state ClosedFile of File {
  public void open() {
    this ← OpenFile {
      ptr = fopen(name);
} } }

state OpenFile of File {
  private FILE* ptr;
  public void close() {
    fclose(ptr);
    this ← ClosedFile {}
} }
```

Objective Join Calculus

```
CLOSED | open() ▷
  let ptr = fopen(name)
  in this.OPEN(ptr)

OPEN(ptr) | close() ▷
  fclose(ptr);
  this.CLOSED
```

Why the OJC? Intriguing similarities!

Plaid

```
class File
{ public String name; }

state ClosedFile of File {
  public void open() {
    this ← OpenFile {
      ptr = fopen(name);
} } }

state OpenFile of File {
  private FILE* ptr;
  public void close() {
    fclose(ptr);
    this ← ClosedFile {}
} }
```

Objective Join Calculus

```
CLOSED | open() ▷
  let ptr = fopen(name)
  in this.OPEN(ptr)

OPEN(ptr) | close() ▷
  fclose(ptr);
  this.CLOSED
```

Example: lock

```
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

Example: lock

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

o.acquire(c1)

o.IDLE

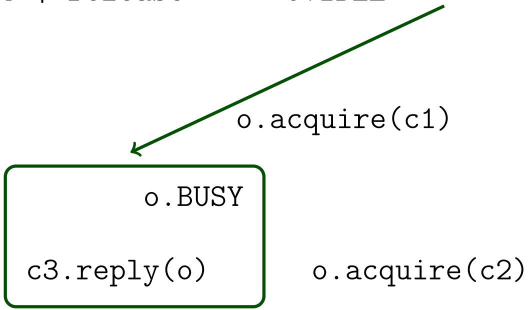
o.acquire(c3)

o.acquire(c2)

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

Example: lock

```
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

Example: lock

```
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

Example: lock

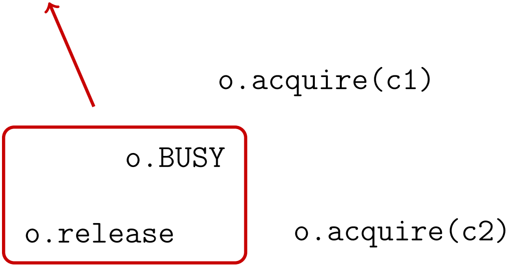
```
def o = IDLE | acquire(c) ▷ o.BUSY | c.reply(o)
 or     BUSY | release    ▷ o.IDLE
in ...
```

```
                         o.acquire(c1)

                    o.BUSY

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

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

Example: lock

```
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

Example: lock

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

```
                     o.acquire(c1)

          o.IDLE

                          o.acquire(c2)
```

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

Outline

1. A historical perspective
2. Concurrency and TSOP
3. Behavioral types
4. Practicalities
5. Concluding remarks

Behavioral types for the OJC

Observation

- both **state** and **operations** are messages
- legal message configuration
  = which operations are permitted in which states

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

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

Behavioral types for the OJC

Observation

- both **state** and **operations** are messages
- legal message configuration
  = which operations are permitted in which states

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

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

Behavioral types for the OJC

Observation

- both **state** and **operations** are messages
- legal message configuration
  = which operations are permitted in which states

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

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

Types ∼ commutative regular expressions

Syntax

$$t, s \quad ::= \quad \mathtt{m}(\tilde{t}) \quad \text{(message)}$$

Types $\sim$ commutative regular expressions

Syntax

$$
\begin{array}{lll}
t, s & ::= & \\
 & \mathtt{m}(\tilde{t}) & \text{(message)} \\
 & t \otimes s & \text{(product)} \\
 & t \oplus s & \text{(sum)} \\
 & *t & \text{(exponential)}
\end{array}
$$

Types $\sim$ commutative regular expressions

Syntax

$$
\begin{array}{llll}
t, s & ::= & \mathbb{0} & \text{(unit for } \oplus\text{)} \\
 & & \mathbb{1} & \text{(unit for } \otimes\text{)} \\
 & & \mathtt{m}(\tilde{t}) & \text{(message)} \\
 & & t \otimes s & \text{(product)} \\
 & & t \oplus s & \text{(sum)} \\
 & & *t & \text{(exponential)}
\end{array}
$$

Types ∼ commutative regular expressions

Syntax

t, s	$::=$	$\mathbb{0}$	(unit for $\oplus$)
		$\mathbb{1}$	(unit for $\otimes$)
		$\mathtt{m}(\tilde{t})$	(message)
		$t \otimes s$	(product)
		$t \oplus s$	(sum)
		$*t$	(exponential)

Semantics

t	$[\![t]\!]$	
$\mathtt{m}$	$\{\mathtt{m}\}$	must send $\mathtt{m}$
$\mathtt{a} \oplus \mathtt{b}$	$\{\mathtt{a}, \mathtt{b}\}$	must send either $\mathtt{a}$ or $\mathtt{b}$
$\mathtt{a} \otimes \mathtt{b}$	$\{\mathtt{a} \cdot \mathtt{b}\}$	must send both $\mathtt{a}$ and $\mathtt{b}$
$\mathbb{1} \oplus \mathtt{m}$	$\{\varepsilon, \mathtt{m}\}$	can (but need not) send $\mathtt{m}$
$\mathbb{0}$	$\emptyset$	☠

Subtyping $\sim$ inverse language inclusion

$\mathtt{a} \oplus \mathtt{b} \leqslant \mathtt{a}$	generalizes OO subtyping
$\mathtt{m(real)} \leqslant \mathtt{m(int)}$	contravariance on arguments
$t \leqslant \mathbb{1}$	top object without obligations
$t \leqslant \mathbb{0}$	top object
$\mathbb{0} \not\leqslant t$	usable object

A few (simple) typing rules

Output $$\frac{}{u : \mathtt{m} \qquad\qquad \vdash u.\mathtt{m}}$$

A few (simple) typing rules

Output

$$\frac{}{u : \mathtt{m}(t), v : t \vdash u\,.\,\mathtt{m}(v)}$$

A few (simple) typing rules

Output $$\frac{}{u : \mathtt{m}(t), v : t \vdash u\,.\,\mathtt{m}(v)} \quad \mathbb{0} \not\leqslant t$$

A few (simple) typing rules

Output

$$\frac{}{u : \mathtt{m}(t), v : t \vdash u.\mathtt{m}(v)} \qquad \mathbb{0} \not\leq t$$

Parallel

$$\frac{u : t \vdash P \qquad u : s \vdash Q}{u : t \otimes s \vdash P \mid Q}$$

A few (simple) typing rules

Output
$$\frac{}{u : \mathtt{m}(t), v : t \vdash u\,.\,\mathtt{m}(v)} \qquad \mathbb{0} \not\leqslant t$$

Parallel
$$\frac{u : t \vdash P \qquad u : s \vdash Q}{u : t \otimes s \vdash P \mid Q}$$

Subsumption
$$\frac{u : s \vdash P}{u : t \vdash P} \qquad t \leqslant s$$

A few (simple) typing rules

Output
$$\frac{}{u : \mathtt{m}(t), v : t \vdash u\,.\,\mathtt{m}(v)} \qquad \mathbb{0} \not\leqslant t$$

Parallel
$$\frac{u : t \vdash P \qquad u : s \vdash Q}{u : t \otimes s \vdash P \mid Q}$$

Subsumption
$$\frac{u : s \vdash P}{u : t \vdash P} \qquad t \leqslant s$$

Reaction
$$\frac{u : s \vdash P}{u : t \vdash \mathtt{m}_1 \mid \cdots \mid \mathtt{m}_k \triangleright P} \qquad t \leqslant t[\mathtt{m}_1 \cdots \mathtt{m}_k] \otimes s$$

A few (simple) typing rules

Output

$$\frac{}{u : \mathtt{m}(t), v : t \vdash u.\mathtt{m}(v)} \quad \mathbb{0} \not\leqslant t$$

Parallel

$$\frac{u : t \vdash P \qquad u : s \vdash Q}{u : t \otimes s \vdash P \mid Q}$$

Subsumption

$$\frac{u : s \vdash P}{u : t \vdash P} \quad t \leqslant s$$

Brzozowski's derivative

Reaction

$$\frac{u : s \vdash P}{u : t \vdash \mathtt{m}_1 \mid \cdots \mid \mathtt{m}_k \triangleright P} \quad t \leqslant t[\mathtt{m}_1 \cdots \mathtt{m}_k] \otimes s$$

Example: the lock protocol

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

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

Example: the lock protocol

o : BUSY

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

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

Example: the lock protocol

o : BUSY   o : release

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

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

Example: the lock protocol

o : BUSY

o : release

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

c : reply(release)

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

Example: violation of the lock protocol

```
def Lock = create(r) ▷
  def o = IDLE | acquire(c) ▷ o.BUSY | c.reply(o)
   or     BUSY | release    ▷ o.IDLE
  in
    o.IDLE | r.reply(o)
in
  let l = Lock.create in
  let l = l.acquire in
    l.release | l.release
```

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

Example: violation of the lock protocol

```
def Lock = create(r) ▷
  def o = IDLE | acquire(c) ▷ o.BUSY | c.reply(o)
   or     BUSY | release    ▷ o.IDLE
  in
 l : *acquire r.reply(o)
in
  let l = Lock.create in
  let l = l.acquire in
    l.release | l.release
```

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

Example: violation of the lock protocol

l : release

```
def Lock = create(r) ▷
  def o = IDLE | acquire(c) ▷ o.BUSY | c.reply(o)
   or     BUSY | release    ▷ o.IDLE
  in
    o.IDLE | r.reply(o)
in
  let l = Lock.create in
  let l = l.acquire in
    l.release | l.release
```

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

Example: violation of the lock protocol

```
def Lock = create(r) ▷
  def o = IDLE | acquire(c) ▷ o.BUSY | c.reply(o)
   or     BUSY | release    ▷ o.IDLE
  in
    o.IDLE | r.reply(o)
in
  let l = Lock.create in
  let l = l.acquire in
    l.release | l.release
```

l : release

l : release   l : release

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

Example: violation of the lock protocol

```
def Lock = create(r) ▷
  def o = IDLE | acquire(c) ▷ o.BUSY | c.reply(o)
   or     BUSY | release    ▷ o.IDLE
  in
    o.IDLE | r.reply(o)
in
  let l = Lock.create in
  let l = l.acquire in
    l.release | l.release
```

$\mathtt{l} : \mathtt{release}$

$\mathtt{l} : \mathtt{release} \otimes \mathtt{release}$

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

Example: sharing locks

```
def Lock = create(r) ▷
  def o = IDLE | acquire(c) ▷ o.BUSY | c.reply(o)
   or     BUSY | release    ▷ o.IDLE
  in
    o.IDLE | r.reply(o)
in
  let l = Lock.create in
          { let l = l.acquire in l.release }
        | { let l = l.acquire in l.release }
```

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

Example: sharing locks

l : *acquire

```
def Lock = create(r) ▷
  def o = IDLE | acquire(c) ▷ o.BUSY | c.reply(o)
   or     BUSY | release    ▷ o.IDLE
  in
          r.reply(o)
in
  let l = Lock.create in
          { let l = l.acquire in l.release }
        | { let l = l.acquire in l.release }
```

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

Example: sharing locks

```
def Lock = create(r) ▷
  def o = IDLE | acquire(c) ▷ o.BUSY | c.reply(o)
   or     BUSY | release    ▷ o.IDLE
  in
  l : *acquire  r.reply(o)
in
                l : *acquire
  let l = Lock.creat   in
          { let l = l.acquire in l.release }
        | { let l = l.acquire in l.release }

                l : *acquire
```

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

Example: sharing locks

```
def Lock = create(r) ▷
  def o = IDLE | acquire(c) ▷ o.BUSY | c.reply(o)
   or     BUSY | release    ▷ o.IDLE
  in
                 r.reply(o)
in
  let l = Lock.create in
          { let l = l.acquire in l.release }
        | { let l = l.acquire in l.release }
```

l : ∗acquire

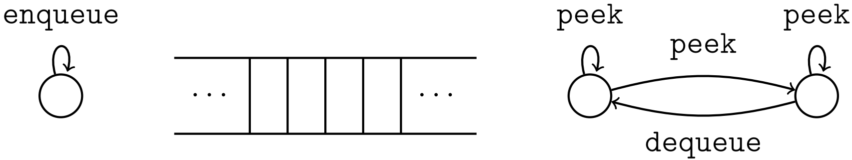
l : ∗acquire ⊗ ∗acquire

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

Example: concurrent queue

Producer

Consumer



Example: 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,o)
in o.NONE | ...
```

Example: type of the concurrent queue

Producer protocol

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

Consumer protocol

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

Queue type

$$
\begin{array}{ll}
 & (\ \mathtt{NONE} \qquad\qquad\quad \otimes\ t_{\mathtt{prod}} \otimes t_{\mathtt{none}}\ ) \\
\oplus & (\ \mathtt{HEAD} \otimes \mathtt{TAIL} \otimes t_{\mathtt{prod}} \otimes t_{\mathtt{some}}\ )
\end{array}
$$

Example: type of the concurrent queue

Producer protocol

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

Consumer protocol

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

Queue type

$$
\begin{array}{cl}
 & (\,\texttt{NONE} \qquad\qquad\quad \otimes\ t_{\texttt{prod}} \otimes t_{\texttt{none}}\,) \\
\oplus & (\,\texttt{HEAD} \otimes \texttt{TAIL} \otimes t_{\texttt{prod}} \otimes t_{\texttt{some}}\,)
\end{array}
$$

Example: type of the concurrent queue

Producer protocol

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

Consumer protocol

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

Queue type

$$
\begin{array}{lllllllll}
 & (\ \texttt{NONE} & & & \otimes & t_{\texttt{prod}} & \otimes & t_{\texttt{none}} &) \\
\oplus & (\ \texttt{HEAD} & \otimes & \texttt{TAIL} & \otimes & t_{\texttt{prod}} & \otimes & t_{\texttt{some}} &)
\end{array}
$$

Example: type of the concurrent queue

Producer protocol

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

Consumer protocol

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

Queue type

$$
\begin{array}{lllllll}
 & (\mathtt{NONE} & & \otimes & t_{\mathrm{prod}} & \otimes & t_{\mathrm{none}}) \\
\oplus & (\mathtt{HEAD} \otimes & \mathtt{TAIL} & \otimes & t_{\mathrm{prod}} & \otimes & t_{\mathrm{some}})
\end{array}
$$

Example: type of the concurrent queue

Producer protocol

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

Consumer protocol

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

Queue type

$$
\begin{aligned}
& (\ \texttt{NONE} && \otimes\ t_{\texttt{prod}} \otimes t_{\texttt{none}}\ ) \\
\oplus\ & (\ \texttt{HEAD} \otimes \texttt{TAIL} && \otimes\ t_{\texttt{prod}} \otimes t_{\texttt{some}}\ )
\end{aligned}
$$

Well-typed programs respect object protocols

Theorem (soundness)

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

Examples

- $\mathtt{o} : t_{\mathtt{lock}} \vdash \mathtt{o.IDLE} \mid P$ and $\mathtt{IDLE}, \mathtt{release} \notin [\![t_{\mathtt{lock}}]\!]$
- $\mathtt{o} : t_{\mathtt{lock}} \vdash \mathtt{o.BUSY} \mid P$ and $\mathtt{BUSY}, \mathtt{release}, \mathtt{release} \notin [\![t_{\mathtt{lock}}]\!]$

Outline

1 A historical perspective

2 Concurrency and TSOP

3 Behavioral types

4 Practicalities

5 Concluding remarks

From theory (OJC) to practice (Java)

Guiding principles

- ad-hoc languages are nice and clean but seldom popular
- better to piggyback on a mainstream programming language

Runtime support for join definitions

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

Protocol enforcement

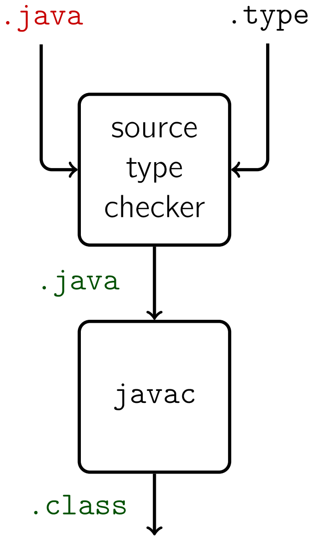
- user-provided behavioral type annotations, and
- behavioral type checker as a pre- (or post-) processor

From the Objective Join Calculus to Java

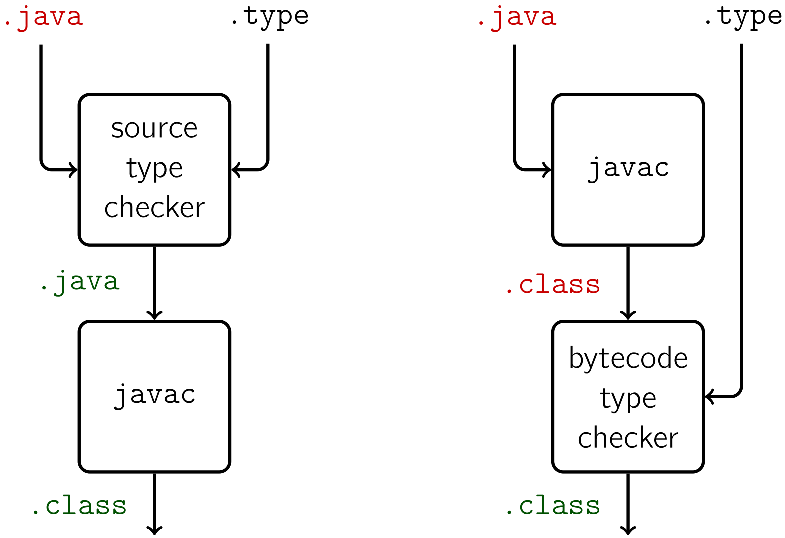
OJC	**Java**
object	object
state message	private method
operation message	public method
chemical reaction	message queues + condition vars
continuation	∼ sequential composition

```
class Lock {
  private void IDLE()     { ... }
  private void BUSY()     { ... }
  public  void acquire() { ... }
  public  void release() { ... }
  Lock()                  { IDLE(); }
}
```

From theory (OJC) to practice (Java)



From theory (OJC) to practice (Java)

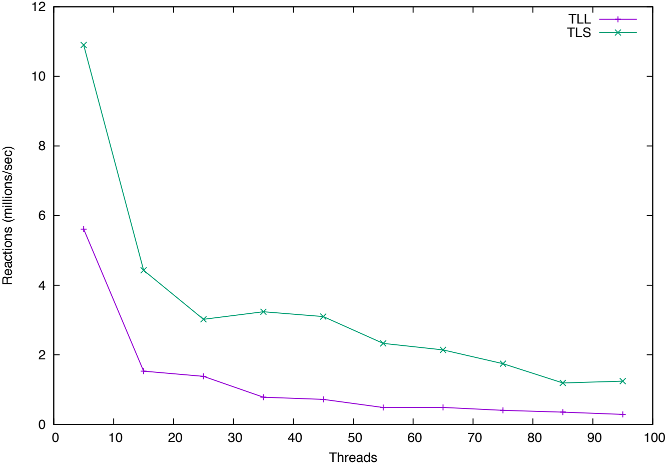


Protocol-aware compilation of join patterns

- knowing that objects will be used according to a specific protocol may help producing better code (fewer locks. . .)

Protocol-aware compilation of join patterns

- knowing that objects will be used according to a specific protocol may help producing better code (fewer locks. . .)



Outline

1. A historical perspective
2. Concurrency and TSOP
3. Behavioral types
4. Practicalities
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
   - TSOP = how you implement objects in the OJC
3. first **behavioral type theory** for OJC
   - interface + protocols + sharing control

In the paper

- more examples (iterators, full-duplex channels)
- formal definitions (OOPSLA proceedings) and proofs (HAL)

Example: only usable objects can be sent

$$\frac{}{u : \texttt{foo} \vdash u.\texttt{bar}}$$

Example: only usable objects can be sent

$$\frac{}{u : \texttt{foo} \vdash u\texttt{.bar}} \quad \texttt{foo} \leqslant \mathbb{0}$$

Example: only usable objects can be sent

$$\dfrac{}{u : \texttt{foo} \vdash u\texttt{.bar}} \quad \texttt{foo} \leqslant \mathbb{0} \simeq \texttt{bar} \otimes \mathbb{0}$$

Example: only usable objects can be sent

$$\dfrac{u : \texttt{bar} \vdash u.\texttt{bar} \qquad \dfrac{\vdots}{u : \mathbb{0} \vdash P}}{\dfrac{u : \texttt{bar} \otimes \mathbb{0} \vdash u.\texttt{bar} \mid P}{u : \texttt{foo} \vdash u.\texttt{bar} \mid P}} \quad \texttt{foo} \leqslant \mathbb{0} \simeq \texttt{bar} \otimes \mathbb{0}$$

$$P \stackrel{\text{def}}{=} \texttt{def}\ v = \cdots\ \texttt{in}\ v.\texttt{trash}(u)$$