chemistry of typestates

Silvia Crafa[1] Luca Padovani[2]

[1] Dipartimento di Matematica, Università di Padova

[2] Dipartimento di Informatica, Università di Torino

typestate-oriented programming (Aldrich et al. '09)

```
class File {
  public final String fileName;



  public method open() {

      handle = fopen(fileName);
  }


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

typestate-oriented programming (Aldrich et al. '09)

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

state ClosedFile of File {          // explicit state
  public method open() {

      handle = fopen(fileName);
} }

state OpenFile of File {            // explicit state
  private FILE* handle;             ~~// meaningful if open~~
  public method close() { ... }
  public method read()
  { ...fread(handle)... }           ~~// valid if open~~
}
```

typestate-oriented programming (Aldrich et al. '09)

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

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

      handle = fopen(fileName);
} }

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

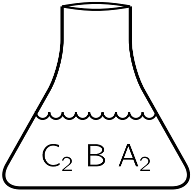
typestate-oriented programming (Aldrich et al. ’09)

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

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

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

the chemical metaphor (Berry & Boudol'92)

program behavior
= chemical reaction

$$A \mid B \mid C \quad \triangleright \quad D \mid E$$

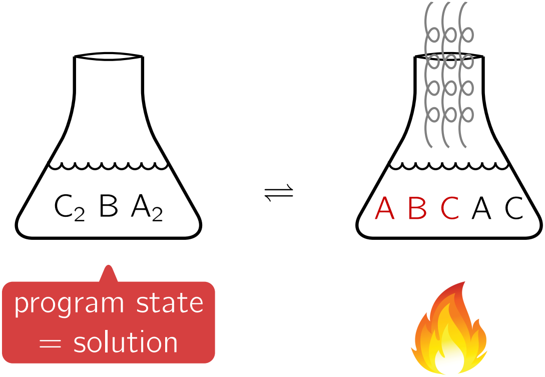


program state
= solution

the chemical metaphor (Berry & Boudol'92)

program behavior
= chemical reaction

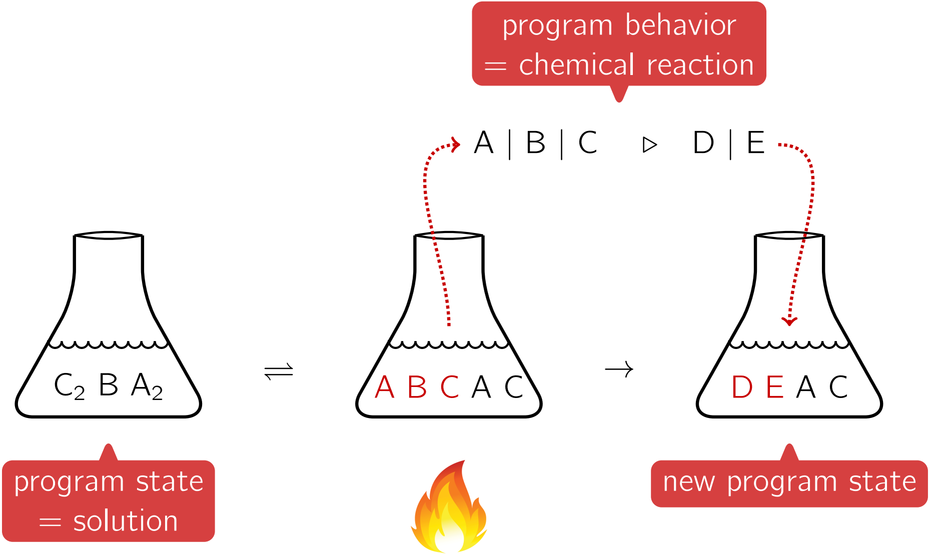
$$A \mid B \mid C \quad \triangleright \quad D \mid E$$



$$C_2 \; B \; A_2 \quad \rightleftharpoons \quad A \; B \; C \; A \; C$$

program state
= solution

the chemical metaphor (Berry & Boudol'92)



program behavior = chemical reaction

$A \mid B \mid C \quad \triangleright \quad D \mid E$

$C_2\ B\ A_2 \quad \rightleftharpoons \quad A\ B\ C\ A\ C \quad \rightarrow \quad D\ E\ A\ C$

program state = solution

new program state

the file object revisited

```
def file =






in
```

Objective Join Calculus (Fournet, Laneve, Maranget, Rémy '03)

the file object revisited

```
def file =
   CLOSED  | open(n,r) ▷



in
```

compound molecule
= state + operation

Objective Join Calculus (Fournet, Laneve, Maranget, Rémy ’03)

the file object revisited

```
def file =
   CLOSED  | open(n,r) ▷ let h = fopen(n) in
                         file.OPEN(h) | r.reply(file)

in
```

state change

Objective Join Calculus (Fournet, Laneve, Maranget, Rémy ’03)

the file object revisited

```
def file =
   CLOSED  | open(n,r) ▷ let h = fopen(n) in
                         file.OPEN(h) | r.reply(file)
or OPEN(h) | close(r)  ▷ fclose(h);
                         file.CLOSED  | r.reply(file)

in
```

scoping rules prevent invalid field access

Objective Join Calculus (Fournet, Laneve, Maranget, Rémy ’03)

the file object revisited

```
def file =
   CLOSED  | open(n,r) ▷ let h = fopen(n) in
                         file.OPEN(h) | r.reply(file)
or OPEN(h) | close(r)  ▷ fclose(h);
                         file.CLOSED  | r.reply(file)
or OPEN(h) | read(r)   ▷ let v = fread(h) in
                         file.OPEN(h) | r.reply(v,file)
in file.CLOSED
```

no state change

Objective Join Calculus (Fournet, Laneve, Maranget, Rémy ’03)

the file object revisited

```
def file =
   CLOSED  | open(n,r) ▷ let h = fopen(n) in
                         file.OPEN(h) | r.reply(file)
or OPEN(h) | close(r)  ▷ fclose(h);
                         file.CLOSED  | r.reply(file)
or OPEN(h) | read(r)   ▷ let v = fread(h) in
                         file.OPEN(h) | r.reply(v,file)
in file.CLOSED | let file    = file.open("a.txt") in
                 let v, file = file.read in
                 let file    = file.close in ...
```

🗎 **Objective Join Calculus** (Fournet, Laneve, Maranget, Rémy ’03)

types

$$t_{\texttt{CLOSED}} \;=\; \texttt{open}(\texttt{string}, \texttt{reply}(t_{\texttt{OPEN}}))$$

types

$$t_{\mathtt{CLOSED}} = \mathtt{open}(\mathtt{string}, \mathtt{reply}(t_{\mathtt{OPEN}})) \oplus \mathbb{1}$$

behavioral disjunction

types

$$\begin{aligned} t_{\texttt{CLOSED}} &= \texttt{open}(\texttt{string}, \texttt{reply}(t_{\texttt{OPEN}})) \oplus \mathbb{1} \\ t_{\texttt{OPEN}} &= \texttt{close}(\texttt{reply}(t_{\texttt{CLOSED}})) \oplus \texttt{read}(\texttt{reply}(\texttt{int}, t_{\texttt{OPEN}})) \end{aligned}$$

types

$$\begin{aligned} t_{\mathtt{CLOSED}} &= \mathtt{open}(\mathtt{string}, \mathtt{reply}(t_{\mathtt{OPEN}})) \oplus \mathbb{1} \\ t_{\mathtt{OPEN}} &= \mathtt{close}(\mathtt{reply}(t_{\mathtt{CLOSED}})) \oplus \mathtt{read}(\mathtt{reply}(\mathtt{int}, t_{\mathtt{OPEN}})) \end{aligned}$$

$$\mathtt{file} \; : \; (\mathtt{CLOSED} \otimes t_{\mathtt{CLOSED}}) \oplus (\mathtt{OPEN}(\mathtt{FILE*}) \otimes t_{\mathtt{OPEN}})$$

behavioral conjunction

- type = set of **valid message molecules** targeted to object
- e.g. “reading from a closed file is forbidden”

types

$$\begin{aligned} t_{\mathtt{CLOSED}} &= \mathtt{open}(\mathtt{string}, \mathtt{reply}(t_{\mathtt{OPEN}})) \oplus \mathbb{1} \\ t_{\mathtt{OPEN}} &= \mathtt{close}(\mathtt{reply}(t_{\mathtt{CLOSED}})) \oplus \mathtt{read}(\mathtt{reply}(\mathtt{int}, t_{\mathtt{OPEN}})) \end{aligned}$$

$$\mathtt{file} \;:\; (\mathtt{CLOSED} \otimes t_{\mathtt{CLOSED}}) \oplus (\mathtt{OPEN}(\mathtt{FILE*}) \otimes t_{\mathtt{OPEN}})$$

- type = set of **valid message molecules** targeted to object
- e.g. "reading from a closed file is forbidden"

Theorem (type preservation)

Messages targeted to `file` *are always described by its type*

Corollary (protocol compliance)

A well-typed program will not try to read from a closed file

the fork

```
def fork =
   FREE | acquire(r) ▷ fork.BUSY | r.reply(fork)
or BUSY | release    ▷ fork.FREE
in fork.FREE | Phil.new(fork) | Phil.new(fork)
```

the fork

```
def fork =
   FREE | acquire(r) ▷ fork.BUSY | r.reply(fork)
or BUSY | release    ▷ fork.FREE
in fork.FREE | Phil.new(fork) | Phil.new(fork)
```

- − the state of the fork cannot be tracked statically
- + invocation to `acquire` **blocks** until the fork is released

the fork

```
def fork =
   FREE | acquire(r) ▷ fork.BUSY | r.reply(fork)
or BUSY | release    ▷ fork.FREE
in fork.FREE | Phil.new(fork) | Phil.new(fork)
```

- ⊖ the state of the fork cannot be tracked statically
- ⊕ invocation to `acquire` **blocks** until the fork is released

$$\texttt{fork} : *\texttt{acquire(reply(release))} \otimes (\texttt{FREE} \oplus (\texttt{BUSY} \otimes \texttt{release}))$$

$$*t = \mathbb{1} \oplus t \oplus (t \otimes t) \cdots$$

on state (un)awareness and subtyping

```
def iter =
   SOME(p) | next(r) ▷
     r.reply(p->data, iter) |
     if p->next != null then iter.SOME(p->next)
                         else iter.NONE



in ...
```

on state (un)awareness and subtyping

```
def iter =
   SOME(p) | next(r) ▷
     r.reply(p->data, iter) |
     if p->next != null then iter.SOME(p->next)
                         else iter.NONE
or NONE    | hasNext(r) ▷ iter.NONE    | r.no(iter)
or SOME(p) | hasNext(r) ▷ iter.SOME(p) | r.yes(iter)
in ...
```

on state (un)awareness and subtyping

```
def iter =
   SOME(p) | next(r) ▷
     r.reply(p->data, iter) |
     if p->next != null then iter.SOME(p->next)
                         else iter.NONE
or NONE    | hasNext(r) ▷ iter.NONE    | r.no(iter)
or SOME(p) | hasNext(r) ▷ iter.SOME(p) | r.yes(iter)
in ...
```

$$\begin{aligned}
t_{\mathtt{NONE}} &= \mathtt{hasNext}(\mathtt{no}(t_{\mathtt{NONE}})) \oplus \mathbb{1} \\
t_{\mathtt{SOME}} &= \mathtt{hasNext}(\mathtt{yes}(t_{\mathtt{SOME}})) \oplus \mathtt{next}(\mathtt{reply}(\mathtt{int}, t_{\mathtt{UNKNOWN}})) \\
t_{\mathtt{UNKNOWN}} &= \mathtt{hasNext}(\mathtt{no}(t_{\mathtt{NONE}}) \oplus \mathtt{yes}(t_{\mathtt{SOME}}))
\end{aligned}$$

OJC for (concurrent) TSOP: wrap-up

- state-dependent fields and operations
- explicit **state change**
- state unawareness 1: runtime **synchronization** (`acquire`)
- state unawareness 2: runtime **introspection** (`hasNext`)
- **multidimensional states** (not illustrated)
- **partial/concurrent** state update (not illustrated)

$$\mathbb{0} \quad | \quad \mathbb{1} \quad | \quad \mathtt{m}(\tilde{t}) \quad | \quad t \oplus s \quad | \quad t \otimes s \quad | \quad *t$$

- **one type language** for state, operations, protocols, sharing
- **state-dependent** field/method **types** (`hasNext`)
- type **preservation** = protocol **compliance**