Law and Order for Typestate

Hannes Saffrich, Yuki Nishida, Peter Thiemann

Typestate

- objects that change type as they are updated
- interface to the object changes according to current typestate
- (problem: aliasing)

Example: file handle API

```
f = open("foo")
x = read(f)
g = f
close(g)
x = read(f) # <--- NOOOOOO
            # statically rejected by typestate
```

Abstract Typestate

- objects described by linear types of the form

T,U,V ::= ... | [$\mathcal{L}$]

- $\mathcal{L}$ is a non-empty language $\subseteq \Xi^*$
- Ξ an alphabet of primitive operations
- the protocol of the object
- operations on objects (capability passing)

$\mathtt{op}_a$: [$\mathcal{L}$] $\multimap$ [a \ $\mathcal{L}$]
where a $\in \Xi$ and a \ $\mathcal{L} \neq \emptyset$

- alternative object API

$\mathtt{op!}_a$: [a] $\multimap$ ()
where a $\in \Xi$

Borrowing

Given

`x : [` $\mathcal{L}$ `]`
`f : [` $\mathcal{B}$ `]` $\multimap \mathbb{1}$

would like to write

```
f (&x); ... x ...
```

- `...` `x` `...` is linear in the continuation
- with type “leftover of $\mathcal{L}$ after $\mathcal{B}$”

Encoding

```
let (x1; x2) = split B x in
  f (x1); ... x2 ...
```

- must consume x_1 before x_2
- Typing the output of `split`

x_1 : [$\mathcal{B}$]
x_2 : [$\mathcal{R}$]
where $\mathcal{R} = \mathcal{B} \setminus\setminus \mathcal{L}$
$= \{ w \in \Xi^* \mid \forall\ v \in \mathcal{B},\ vw \in \mathcal{L} \}$

- $\mathcal{B} \setminus\setminus \mathcal{L}$ is “strong left quotient” aka product derivative [Suzuki & Okui, 2008]
- side condition: $\mathcal{B}, \mathcal{R} \neq \emptyset$

Interlude: File handles in abstract typestate

- operations: **r**ead, **w**rite, **c**lose
- type `Handle[` $\mathcal{L}$ `]` where $\mathcal{L} \subseteq$ (r|w)*c

```
open  : String ⊸ Handle[ (r|w)*c ]
read  : Handle[ r ] ⊸ Byte
write : Byte ⊗ Handle[ w ] ⊸ Unit
close : Handle[ c ] ⊸ Unit
```

reading from `f : Handle[ (r|w)*c ]`

```
let (f1; f2) = split r f in
let b = read f1 in
... f2 : Handle[ (r|w)*c ] ...
```

Interlude file handles (cont'd)

closing `f : Handle[ (r|w)*c ]`

```
let (f1; f2) = split c f in
let _ = close f1 in
... f2 : Handle[ ε ] ...
```

A resource of type `Handle[ ε ]` is inactive and can be dropped.

take read-only borrow from `f`

```
let just_read( rob : Handle[ r* ] ) ⊸ Unit = ... in
let (f1; f2) = split r* f in
let _ = just_read f1 in
... f2 : Handle[ (r|w)*c ] ...
```

Digression

- abstract type state

$\texttt{op}_a$: [$\mathcal{L}$] $\multimap$ [a \ $\mathcal{L}$]
where a $\in \Xi$ and a \ $\mathcal{L} \neq \emptyset$

- session types

send : a $\multimap$!a;s $\multimap$ s
recv : ?a;s $\multimap$ a $\otimes$ s

- Looks similar?

OPM

- common generalization of sets of languages and session types
- **Ordered Partial Monoids**
- also known as resource algebras for Kripke models of BI-Logic

OPM instance: Languages

- pick an *envelope language* $\mathcal{L} \subseteq \Xi^*$
- elements: (regular) subsets of prefix languages of $\mathcal{L}$
- unit: $\{\varepsilon\}$
- multiplication: concatenation of languages (if result $\subseteq \mathcal{L}$)
- order: $\subseteq$ on sets of strings of symbols

OPM instance: Context-free Session Types

- elements

```
S   ::= skip | S;S   | !A | ?A | ...
S⊥  ::= end | S;S⊥ | ...
```

- unit: skip
- multiplication: sequence operator ;
- order: type conversion (or subtyping)

Ordered Logic

Recall:

```
let (x1; x2) = split B x in
  f (x1); ... x2 ...
```

"must consume x_1 before x_2"

Features of ordered logic

- linear
- order of assumptions matters
- related work: Frank Pfenning, Jeff Polakow, Robert Simmons, Henry DeYoung
- mostly on proof theory / term language with full reduction
- but here: for a sequential call-by-value lambda calculus

Typing

Typing contexts inspired by BI

```
Γ,Δ,Θ ::= ∅
  | x:T
  | Γ;Δ            -- ordered
  | Γ‖Δ        -- unordered
```

subject to relation _≈_

- _‖_ and ∅ commutative monoid
- _;_ and ∅ monoid

Selected typing rules

Variables

$x{:}T \vdash x{:}T$

Splitting

assume OPM $(M, \odot, \mathbb{1}, \leq)$

just a family of constants

$$\frac{\mathtt{m}_1 \odot \mathtt{m}_2 \leq \mathtt{m}}{\Gamma \vdash \mathtt{split}\ \mathtt{m}_1\ \mathtt{m}_2 : [\ \mathtt{m}\ ] \multimap [\ \mathtt{m}_1\ ] \otimes\!\!> [\ \mathtt{m}_2\ ]}$$

Unordered Product

Intro

$$\frac{\Gamma_1 \vdash \mathtt{M}_1 : \mathtt{T}_1 \qquad \Gamma_2 \vdash \mathtt{M}_2 : \mathtt{T}_2}{\Gamma_1 \parallel \Gamma_2 \vdash (\mathtt{M}_1 \otimes \mathtt{M}_2) : \mathtt{T}_1 \otimes \mathtt{T}_2}$$

Elimination

$$\frac{\Gamma \vdash \mathtt{M} : \mathtt{T}_1 \otimes \mathtt{T}_2 \qquad \mathcal{G}[\ \mathtt{x}_1 \parallel \mathtt{x}_2\ ] \vdash \mathtt{N} : \mathtt{T}}{\mathcal{G}[\ \Gamma\ ] \vdash \mathtt{let}\ \mathtt{x}_1 \otimes \mathtt{x}_2 = \mathtt{M}\ \mathtt{in}\ \mathtt{N} : \mathtt{T}}$$

Ordered Product (left-to-right)

Intro

- requires effects: $\mathtt{e} \in \{0, 1\}$ with $0 < 1$
- $\Gamma \vdash M : T \,|\, 0$ — no operations in M
- $\Gamma \vdash M : T \,|\, 1$ — M may contain operations

$$\frac{\Gamma_1 \vdash \mathtt{M_1} : \mathtt{T_1} \mid \mathtt{e_1} \qquad \Gamma_2 \vdash \mathtt{M_2} : \mathtt{T_2} \mid \mathtt{e_2} \qquad \mathtt{e_2} = \mathtt{1} \to \mathtt{unr}\ (\Gamma_1)}{\Gamma_1 \ ; \ \Gamma_2 \vdash (\mathtt{M_1} \ ; \ \mathtt{M_2}) : \mathtt{T_1} \odot \mathtt{T_2} \mid \mathtt{e_1} \sqcup \mathtt{e_2}}$$

Elimination

$$\frac{\Gamma \vdash \mathtt{M} : \mathtt{T_1} \odot \mathtt{T_2} \qquad \mathcal{LG}[\ \mathtt{x_1} \ ; \ \mathtt{x_2}\ ] \vdash \mathtt{N} : \mathtt{T}}{\mathcal{LG}[\ \Gamma\ ] \vdash \mathtt{let}\ (\mathtt{x_1} \ ; \ \mathtt{x_2}) = \mathtt{M}\ \mathtt{in}\ \mathtt{N} : \mathtt{T}}$$

Functions

- four different kinds of function arrows: unrestricted, linear, left-right, right-left
- four different lambdas
- four different applications

Status

- (logical) type system
- meaningful operational semantics
- syntactic type soundness (manual proof)
  - no run-time errors
  - operations adhere to object protocols
- algorithmic version of type system (bidirectional)
  - sound wrt logical version
  - decidable type checking (Agda proof)
- implemented in an interpreter

Conclusion

- novel transition-based foundation for typestate that admits borrowing
- interesting applications (e.g., session types)
- new perspective on ordered types using inspiration from BI
- simpler control of aliasing (compared to classical typestate)
- avoids resource identities and names (compared to alias types)

  Coblenz et al. [2020, Sec. 4]: “[A language] including typestate could result in a design that was hard to use, since typical typestate languages require users to understand a complex permission model”. In our design, users just have to understand the resource API and traces.

Questions