# CONCURRENCY AND TYPES IN PROGRAMMING

Luís Caires
Universidade Nova de Lisboa
(based on work with Seco, Militão, Aldrich)


NOVA Laboratory for
Computer Science and Informatics

OPCT 2014 Bertinoro Italy

# types in programming

- Types at the heart of standard PLs (OCaml, Java,C#,Scala)
- Foundations in logic (a type system = a specialized logic)
- Highly modular, based on a "lego" of canonic constructions
- Still, "standard" types easy to use by common programmers
- Big impact on software quality ("WTP do not go wrong")
- Big impact on programming discipline ("tame programmers")
- Clearly a success story for theory / foundational research
- But what about programming with concurrency ?

# typing concurrent programming

- A program in a *typed* concurrent programming language should not go wrong !
- Unfortunately, it is still very hard to ensure it does not.
- Many relevant contributions from the community:

  Many specific type systems for abstract models (pi, ambients) and properties (deadlock freedom, race absence, fidelity)

  Emphasis on message passing (communication, sessions)

  Often, introduced concepts are neither easy to transfer to "useful" programming languages or to (the engineers) minds

  But even harder with general logics (so we stick to types)

# a challenge

- May one do for concurrent programming something like “classical type theory" did for general programming?
- What could be the core ingredients of a scalable and reasonably general type theory for concurrency?

Insights from process algebra and (sub)structural logics, suggest that notions of *behavioral types* may help to provide a uniform foundation for typing concurrent programs

“the essence of concurrency is interference” (also in aliasing)

In this talk, I discuss a bit this (not so recent) view, illustrating with some recent [popl’13,ecoop’14] and ongoing work

# a challenge

- May one do for concurrent programming something like “classical type theory" did for general programming?
- What could be the core ingredients of a scalable and reasonably general type theory for concurrency?
- Insights from process algebra and (sub)structural logics, suggest that notions of *behavioral types* may help to provide a uniform foundation for typing concurrent programs
- “the essence of concurrency is interference” (also in aliasing)
- In this talk, I discuss a bit this (not so recent) view, illustrating with some recent [popl’13,ecoop’14] and ongoing work

# programming language

| | | | |
|---|---|---|---|
| $e, f$ | $::=$ | $x$ | *(Variable)* |
| | $\mid$ | $\lambda x.e$ | *(Abstraction)* |
| | $\mid$ | $e_1 e_2$ | *(Application)* |
| | $\mid$ | **let** $x = e_1$ **in** $e_2$ | *(Definition)* |
| | $\mid$ | **ref** | *(Heap cell alloc)* |
| | $\mid$ | **free** $e$ | *(Heap cell free)* |
| | $\mid$ | $a := e$ | *(strong Update )* |
| | $\mid$ | $!a$ | *(Dereference)* |
| | $\mid$ | $[l_1 = e_1 \mid \ldots]$ | *(Tuple)* |
| | $\mid$ | $e.l$ | *(Selection)* |
| | $\mid$ | **if** $(e_1 == e_2)$ **as** $x$ $e_3$ **else** $e_4$ | *(Match)* |
| | $\mid$ | $\#l(e)$ | *(Variant)* |
| | $\mid$ | **case** $e$ **of** $\#l_i(x_i) \rightarrow e_i$ | *(Conditional)* |
| | $\mid$ | **fork** $e$ | *(New thread)* |
| | $\mid$ | **wait** $e$ | *(Wait)* |
| | $\mid$ | **res** $a$ **in** $e$ | *(resource bundle)* |
| | $\mid$ | **open**$(a)$ | *(enter bundle)* |
| | $\mid$ | **close**$(a)$ | *(leave bundle)* |

# a queue ADT

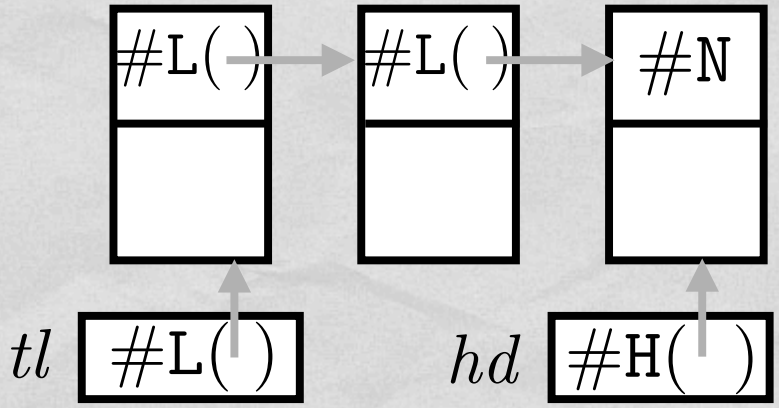
$$
\begin{array}{l}
\textbf{let } newQueue = \\
\quad \lambda[].\textbf{var } hd, tl \textbf{ in } ( \\
\qquad hd := \#\mathtt{F}(newNode\ \textbf{nil}); tl := \#\mathtt{N}; \\
\qquad \textbf{res } s \textbf{ in } ( \\
\qquad\quad [ \\
\qquad\quad\ enq = \cdots \\
\qquad\quad\ | \\
\qquad\quad\ deq = \cdots \\
\qquad\quad ]
\end{array}
$$

# typical queue configurations

$$
\begin{array}{l}
\textbf{let } newQueue = \\
\quad \lambda[].\textbf{var } hd, tl \textbf{ in } ( \\
\qquad hd := \#\mathtt{F}(newNode\ \textbf{nil}); tl := \#\mathtt{N}; \\
\qquad \textbf{res } s \textbf{ in } ( \\
\quad [ \\
\qquad enq = \cdots \\
\qquad | \\
\qquad deq = \cdots \\
\quad ]
\end{array}
$$

# typical queue configurations

$$
\begin{array}{l}
\textbf{let}\ newQueue = \\
\quad \lambda[].\textbf{var}\ hd, tl\ \textbf{in}\ ( \\
\qquad hd := \#\mathtt{F}(newNode\ \textbf{nil});\ tl := \#\mathtt{N}; \\
\qquad \textbf{res}\ s\ \textbf{in}\ ( \\
\qquad\quad [ \\
\qquad\qquad enq = \cdots \\
\qquad\qquad | \\
\qquad\qquad deq = \cdots \\
\qquad\quad ]
\end{array}
$$



# queue Node code

$$
\begin{array}{l}
\textbf{let } newNode = \lambda[].\textbf{var } nxt := \#\mathtt{N} \textbf{ in} \\
\qquad\qquad\qquad\qquad \textbf{res } n \textbf{ in} \\
\qquad\qquad\qquad\quad [\ \ setNxt = \lambda p.(\textbf{open } n; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad nx := \#\mathtt{L}(p); \textbf{close } n) \mid \\
\qquad\qquad\qquad\qquad\ getNxt = \textbf{open } n; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let } p = !nxt \textbf{ in } (\textbf{close } n; p) \mid \\
\qquad\qquad\qquad\qquad\ disp = (\textbf{open } n; \textbf{free } nxt; \textbf{close } n)\ ]
\end{array}
$$

# enqueue operation code

$$
\begin{array}{l}
enq = \textbf{let } n = (newNode\ \textbf{nil})\ \textbf{in} \\
\quad (\\
\quad\quad \textbf{open } s; \\
\quad\quad \textbf{case } hd \textbf{ of} \\
\quad\quad \#\mathtt{H}(hv) \rightarrow (hv.setNxt(n); hd := \#\mathtt{H}(n)) \\
\quad\quad \#\mathtt{F}(fn) \rightarrow (fn.setNxt(n); hd := \#\mathtt{H}(n); tl := fn); \\
\quad\quad \textbf{close } s \\
\quad )
\end{array}
$$

# dequeue operation code

$$
\begin{array}{l}
deq = \ \textbf{open}\ s; \\
\quad \textbf{case}\ hd\ \textbf{of} \\
\quad \#\mathtt{F}(fn) \rightarrow hd := \#\mathtt{F}(fn) \\
\quad \#\mathtt{H}(hv) \rightarrow ( \\
\qquad \textbf{case}\ tl.getNxt\ \textbf{of} \\
\qquad \#\mathtt{N} \rightarrow \textbf{nil} \\
\qquad \#\mathtt{L}(tv) \rightarrow (tl.disp; \\
\qquad\qquad \mathtt{if}\ (hv == tv)\ \mathtt{as}\ u \\
\qquad\qquad\ (hd := \#\mathtt{F}(u)) \\
\qquad\qquad \mathtt{else}\ (hd := \#\mathtt{H}(hv); tl := tv)); \\
\quad \textbf{close}\ s
\end{array}
$$

# client code

$$\begin{array}{l}\textbf{let } q = newQueue()\textbf{in} \\ \quad \textbf{let } t_1 = \textbf{fork}(\textbf{rec}(X).q.enq; X)\textbf{in} \\ \quad\quad \textbf{let } t_2 = \textbf{fork}(\textbf{rec}(X).q.dec; X)\textbf{in} \\ \quad\quad\quad (\textbf{wait } t_1; \textbf{wait } t_2)\end{array}$$

# structural types

$$
\begin{array}{l}
\textbf{let } newQueue = \\
\quad \lambda[].\textbf{var } hd, tl \textbf{ in } ( \\
\qquad hd := \#\mathtt{F}(newNode\ \textbf{nil}); tl := \#\mathtt{N}; \\
\qquad \textbf{res } s \textbf{ in } ( \\
\qquad\quad [ \\
\qquad\quad\ enq = \cdots \\
\qquad\quad\ | \\
\qquad\quad\ deq = \cdots \\
\qquad\quad ]
\end{array}
$$

$$newQueue : 0 \to [enq = 0 | dec = 0]$$

# behavioral types

$$
\begin{array}{l}
\textbf{let } newQueue = \\
\quad \lambda[].\textbf{var } hd, tl \textbf{ in } ( \\
\qquad hd := \#\mathtt{F}(newNode\ \textbf{nil}); tl := \#\mathtt{N}; \\
\qquad \textbf{res } s \textbf{ in } ( \\
\qquad\quad [ \\
\qquad\quad\ enq = \cdots \\
\qquad\quad\ | \\
\qquad\quad\ deq = \cdots \\
\qquad\quad ]
\end{array}
$$

$$newQueue : !(0 \mapsto \mathtt{rec}(X).(enc \,\&\, deq; X))$$

# behavioral types

$$
\begin{array}{l}
\textbf{let } newQueue = \\
\quad \lambda[].\textbf{var } hd, tl \textbf{ in } ( \\
\qquad hd := \#\mathtt{F}(newNode\ \textbf{nil}); tl := \#\mathtt{N}; \\
\qquad \textbf{res } s \textbf{ in } ( \\
\qquad\quad [ \\
\qquad\quad\ enq = \cdots \\
\qquad\quad\ | \\
\qquad\quad\ deq = \cdots \\
\qquad\quad ]
\end{array}
$$

$$newQueue : 0 \to \mathtt{rec}(X).(enc \,\&\, deq; X)$$

# behavioral types

$$
\begin{array}{l}
\textbf{let } newQueue = \\
\quad \lambda[].\textbf{var } hd, tl \textbf{ in } ( \\
\qquad hd := \#\mathtt{F}(newNode\ \textbf{nil}); tl := \#\mathtt{N}; \\
\qquad \textbf{res } s \textbf{ in } ( \\
\qquad\quad [ \\
\qquad\quad enq = \cdots \\
\qquad\quad | \\
\qquad\quad deq = \cdots \\
\qquad\quad ]
\end{array}
$$

$$newQueue : 0 \to \mathtt{rec}(X).(enc; X) \,|\, \mathtt{rec}(X).(deq; X)$$

# behavioral types

$$
\begin{array}{l}
\textbf{let } newQueue = \\
\quad \lambda[].\textbf{var } hd, tl \textbf{ in } ( \\
\qquad hd := \#\mathtt{F}(newNode\ \textbf{nil}); tl := \#\mathtt{N}; \\
\qquad \textbf{res } s \textbf{ in } ( \\
\qquad\quad [ \\
\qquad\quad\ enq = \cdots \\
\qquad\quad\ | \\
\qquad\quad\ deq = \cdots \\
\qquad\quad ] \\
\qquad\quad\ newQueue : 0 \to (!enc \,|\, !deq)
\end{array}
$$

# behavioral types

$$
\begin{array}{l}
\textbf{let } newQueue = \\
\quad \lambda[].\textbf{var } hd, tl \textbf{ in } ( \\
\qquad hd := \#\mathtt{F}(newNode\ \textbf{nil}); tl := \#\mathtt{N}; \\
\qquad \textbf{res } s \textbf{ in } ( \\
\qquad\quad [ \\
\qquad\quad\ enq = \cdots \\
\qquad\quad\ | \\
\qquad\quad\ deq = \cdots \\
\qquad\quad ]
\end{array}
$$

$$Single = hd{:}\mathtt{rd}(\#\mathtt{F}(Node)) \,|\, tl{:}\mathtt{rd}(\#\mathtt{N})$$

$$Many = hd{:}\mathtt{rd}(\#\mathtt{H}(Hv)) \,|\, tl{:}\mathtt{rd}(Tv)$$

$$A = (Single \vee Many)\,;(hd{:}\mathtt{var} \,|\, tl{:}\mathtt{var})$$

$$s : \rho(inv(A))$$

# type structure

# behavioral separation types

$$
\begin{array}{rcrlrl}
T, U & ::= & \mathtt{0} & (stop) & | & T \mapsto V \quad (function) \\
 & | & T\,;U & (sequential) & | & T \,|\, U \quad (parallel) \\
 & | & T \,\&\, U & (intersection) & | & !T \quad (shared) \\
 & | & T \vee U & (union) & | & l{:}T \quad (field) \\
 & | & \oplus_{l \in I} \#l{:}T_l & (alternative) & | & \rho(A) \quad (bundle) \\
 & | & \circ T & (isolated) & | & \tau(T) \quad (thread) \\
 & | & \mathtt{rec}(X)T & (recursion) & | & X \quad (recursion\ var)
\end{array}
$$

- types express safe *usage* capabilities from *client perspective*
- enforces abstraction / information hiding

# linear λ-calculus

$$\frac{A \,|\, x{:}U \vdash e : T}{A \vdash \lambda x.e : U \multimap T}\ (VAbs)$$

$$\frac{A \vdash e_1 : U \multimap T \quad B \vdash e_2 : U}{A \,|\, B \vdash e_1 e_2 : T}\ (App)$$

$0 \vdash \textbf{nil} : 0$

$A, B ::= \mathbf{0} \mid x{:}T, A$

$\boxed{A <: B}$ ($A$ is a subtype of $B$)

$\boxed{A \vdash e : U}$ ($e$ yields value of type $U$ under $B$)

# adding dynamics (cf. Hoare types)

$\boxed{A <: B}$ $(A \text{ is a subtype of } B)$

$\boxed{A \vdash_z e :: B}$ $(e \text{ types from } A \text{ to } z \text{ in } B)$

type assertions (cf. type environments as "global types"):

$$A, B ::= \mathbf{0} \mid x{:}T \mid A\,;B \mid A|B \mid A \,\&\, B \mid A \vee B \mid !A \mid \circ A$$

# (linear) arrow type

$$\frac{A \,|\, x{:}U \vdash_y e :: y{:}T}{A \vdash_z \lambda x.e :: z{:}U \!\mid\!\!\mapsto T}\ (VAbs)$$

$$\frac{A \vdash_z e_1 :: z{:}U \!\mid\!\!\mapsto T \quad B \vdash_x e_2 :: x{:}U}{A \,|\, B \vdash_y e_1 e_2 :: y{:}T}\ (App)$$

- symmetric monoidal closed: $(T, 0, (- \,|\, -), \mid\!\!\mapsto)$

# structural rules

$$x{:}U \vdash_z x :: z{:}U \ (Id) \qquad \frac{A \vdash_x e_1 :: B \quad B \vdash_y e_2 :: C}{A \vdash_y \textbf{let}\ x\ =\ e_1\ \textbf{in}\ e_2 :: C}\ (Let/Cut)$$

$$\frac{A <: A' \quad A' \vdash_x e :: B' \quad B' <: B}{A \vdash_x e :: B}\ (Sub)$$

$$\frac{A \vdash_x e :: B}{A \,|\, C \vdash_x e :: B \,|\, C}\ (Par) \qquad \frac{A \vdash_x e :: B}{A \,;\, C \vdash_x e :: B \,;\, C}\ (Seq)$$

# laws for parallel and sequence

$$U\,;(V\,;T) \mathrel{<:>} (U\,;V)\,;T \quad U\,;0 \mathrel{<:>} U \quad 0\,;U \mathrel{<:>} U$$

$$U\,|(V\,|T) \mathrel{<:>} (U\,|V)\,|T \quad U\,|V \mathrel{<:>} V\,|U \quad U\,|0 \mathrel{<:>} U$$

$$(A\,;C)\,|\,(B\,;D) \mathrel{<:} (A\,|\,B)\;;(C\,|\,D)$$

( special case ) $A\,|\,D \mathrel{<:} A\;;\;D$

# sample typing judgments

$\boxed{A <: B}$ ($A$ is a subtype of $B$)

$\boxed{A \vdash_z e :: B}$ ($e$ types from $A$ to $z$ in $B$)

$(f{:}U \mapsto V \,;\, y{:}U) \mid x{:}U \vdash_z (f\ x) :: z{:}V \,;\, y{:}U$ ✔

$(f{:}U \mapsto V \,;\, y{:}U) \mid x{:}U \vdash_z (f\ y) ::$ ✘

$a{:}\mathtt{use} \vdash_z (\lambda x.a := x) :: z{:}\circ U \mapsto \mathtt{0} \,;\, a{:}\mathtt{rd}(\circ U)$ ✔

# field type

$$\frac{A \vdash_x e :: x{:}U}{A \vdash_z [\ldots l = e \ldots] :: z{:}l{:}U}(Field)$$

$$\frac{A \vdash_z e :: z{:}l{:}T}{A \vdash_x e.l :: x{:}T}(Sel)$$

# (linear) intersection type

$$\frac{A \vdash_y e :: B \quad A \vdash_y e :: C}{A \vdash_y e :: B \ \& \ C}(And)$$

$$\frac{A \vdash_y e :: B_1 \ \& \ B_2}{A \vdash_y e :: B_i}(AndE)$$

$$U \,\&\, V <: U$$

$$U \,\&\, V <: V$$

$$U <: U \,\&\, U$$

- unlabeled choice (e.g., exporting multiple interfaces)
- examples: $A \vdash [up = e_1 | dn = e_2] :: x{:}(up{:}\mathbf{0} \,\&\, dn{:}\mathbf{0})$

# separation types

$$\mathsf{0} \vdash_y v :: \mathsf{0}\ (VStop)$$

$$\frac{A \vdash_y v :: C \quad B \vdash_y v :: D}{A \,;\, B \vdash_y v :: C \,;\, D}(VSeq)$$

$$\frac{A \vdash_y v :: C \quad B \vdash_y v :: D}{A \mid B \vdash_y v :: C \mid D}(VPar)$$

- concurrent Kleene algebra:

$$(T, (- \,\&\, -), (- \mid -), (- \,;\, -), \mathsf{0})$$

# (linear) labeled sum type

$$\frac{A \vdash_y e_c :: y : \oplus_{l \in I} \#l{:}T_l \quad x_i{:}T_i \mid B \vdash_z e_i :: C}{A \mid B \vdash_z \textbf{case}\ e_c\ \textbf{of}\ \#l(x) \to e :: C}\ (Case)$$

$$\frac{A \vdash_z e :: z{:}T_i}{A \vdash_z \#l_i(e) :: z{:} \oplus_{l \in I} \#l{:}T_l}\ (Option)$$

- labeled choice (cf. variant type)

# (linear) union type

$$\frac{A \vdash_z e :: C \quad B \vdash e :: C}{A \vee B \vdash_z e :: C}\ (UnionCase)$$

$$\frac{A \vdash_z e :: z{:}T_i}{A \vdash_z e :: z{:} \vee_{l \in I} T_l}\ (InUnion)$$

- unlabeled union (e.g., exporting some interface)
- examples: $s : \texttt{rec}(X).(empty?{:}\#\texttt{T}\,;\,push \vee empty?{:}\#\texttt{F}\,;(pop \,\&\, push)\,;\,X)$

# types for sharing and isolation

$$\frac{!A_1 \,|\, \ldots \,|\, !A_n \vdash_x v :: B}{!A_1 \,|\, \ldots \,|\, !A_n \vdash_x v :: !B}\ (VShr)$$

$$\frac{\circ A_1 \,|\, \ldots \,|\, \circ A_n \vdash_x e :: B}{\circ A_1 \,|\, \ldots \,|\, \circ A_n \vdash_x e :: \circ B}\ (Iso)$$

- monoidal co-monads:
  - $\circ(-)$ isolated
  - $!(-)$ shared

# laws for sharing (cf. linear logic !)

$$!U <: U$$

$$!U <: !!U$$

$$0 <: !0$$

$$!U \mid !V <: !(U \mid V)$$

$$!U <: 0$$

$$!U <: !U \mid !U$$

# reference type

$$\vdash_x \textbf{ref} :: x{:}\mathtt{var} \quad (Ref)$$

$$\frac{A \vdash_z e :: x{:}\mathtt{var}}{A \vdash_x \textbf{free}\ e :: x{:}\mathtt{0}} \quad (Free)$$

$$\mathtt{var} <: \mathtt{use};\mathtt{var}$$

$$\mathtt{use} <: \mathtt{use};\mathtt{use}$$

$$\mathtt{use} <: \mathtt{wr}(U);\mathtt{rd}(U)$$

$$\mathtt{wr}(\mathtt{0}) <: \mathtt{0} \quad \mathtt{rd}(\mathtt{0}) <: \mathtt{0}$$

$$\mathtt{rd}(U;V) <: \mathtt{rd}(U);\mathtt{rd}(V)$$

$$\mathtt{rd}(U \mid V) <: \mathtt{rd}(U) \mid \mathtt{rd}(V)$$

$$\mathtt{rd}(!U) <: !\mathtt{rd}(!U)$$

$$\mathtt{rd}(\circ U);\mathtt{var} <: \circ(\mathtt{rd}(\circ U);\mathtt{var})$$

# reference type

$$a{:}\mathtt{rd}(U) \vdash_x \,!a :: x{:}U \;(RdVB)$$

$$a{:}\mathtt{rd}(U); \mathtt{use} \vdash_x \,!a :: x{:}U \mid a{:}\mathtt{use} \;(RdVF)$$

$$\frac{A \vdash_z v :: z{:}\circ U \mid a{:}\mathtt{wr}(\circ U)}{A \vdash_z a := v :: \mathtt{0}} \;(WrVF)$$

$$\frac{A \vdash_z v :: z{:}U \mid a{:}\mathtt{use}}{A \vdash_z a := v :: a{:}\mathtt{rd}(U)} \;(WrVB)$$

# resource bundle types

- region type assigns bundle a *rely-guarantee* protocol $R$

  $r : \rho(R)$

- a rely-guarantee protocol is given by:

  $$R ::= 0 \mid \{A\}\{B\}; R \mid \{B\}; R \mid R \vee R \mid \mathtt{rec}(X)R \mid X$$

- let sub-typing laws:

  binary projection op: $\bowtie$

  $$\rho(A \vee B) <: \rho(A) \vee \rho(B)$$

  $$\rho(R) <: \rho(R_1) \mid \rho(R_2) \qquad \text{if } R \bowtie (R_1 \mid R_2)$$

- *projection* splits $R$ into *compatible* protocols $R_1, R_2, \ldots$ ensuring coordinated progress of several views / aliases

# resource bundle types

$$\frac{r{:}\rho(\{A\}\{\mathtt{0}\}) \,|\, C \vdash_x e :: r{:}\rho(\{B\}\{\mathtt{0}\}) \,|\, D}{A \,|\, C \vdash_x \textbf{res}\ r\ \textbf{in}\ e :: B \,|\, D}$$

$$r{:}\rho(\{A\}\{B\}; R) \vdash_x \textbf{open}\ r :: A \,|\, r{:}\rho(\{B\}; R)$$

$$B \,|\, r{:}\rho(\{B\}; R) \vdash_x \textbf{close}\ r :: r{:}\rho(R)$$

- expressing a simpler invariant:

$$inv(A) \triangleq \mathtt{rec}(X).\{A\}\{A\}; X$$

# queue typing

$$
\begin{array}{l}
\textbf{let } newQueue = \\
\quad \lambda[].\textbf{var } hd, tl \textbf{ in } ( \\
\qquad hd := \#\mathtt{F}(newNode\ \textbf{nil}); tl := \#\mathtt{N}; \\
\qquad \textbf{res } s \textbf{ in } ( \\
\quad [ \\
\qquad enq = \cdots \\
\qquad | \\
\qquad deq = \cdots \\
\quad ]
\end{array}
$$

$$newQueue : 0 \to (!enc \,|\, !deq)$$

type checking ensures concurrent safety

# region type for queue

$$
\begin{array}{l}
\textbf{let}\ newQueue = \\
\quad \lambda[].\textbf{var}\ hd, tl\ \textbf{in}\ ( \\
\qquad hd := \#\mathtt{F}(newNode\ \textbf{nil}); tl := \#\mathtt{N}; \\
\qquad \textbf{res}\ s\ \textbf{in}\ ( \\
\qquad\quad [ \\
\qquad\quad\ enq = \cdots \\
\qquad\quad\ | \\
\qquad\quad\ deq = \cdots \\
\qquad\quad ]
\end{array}
$$

$$Single = hd{:}\mathtt{rd}(\#\mathtt{F}(Node)) \mid tl{:}\mathtt{rd}(\#\mathtt{N})$$

$$Many = hd{:}\mathtt{rd}(\#\mathtt{H}(Hv)) \mid tl{:}\mathtt{rd}(Tv)$$

$$A = (Single \vee Many)\,;(hd{:}\mathtt{var} \mid tl{:}\mathtt{var})$$

$$s : \rho(inv(A))$$

# typing queue nodes

$$\begin{array}{l}\textbf{let } newNode = \lambda[].\textbf{var } nxt := \#\mathtt{N} \textbf{ in} \\ \qquad\qquad\qquad\qquad \textbf{res } n \textbf{ in} \\ \qquad\qquad\qquad\qquad [\ \ setNxt = \lambda p.(\textbf{open } n; \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad nx := \#\mathtt{L}(p); \textbf{close } n)\ | \\ \qquad\qquad\qquad\qquad\ \ getNxt = \textbf{open } n; \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let } p = !nxt \textbf{ in } (\textbf{close } n; p)\ | \\ \qquad\qquad\qquad\qquad\ \ disp = (\textbf{open } n; \textbf{free } nxt; \textbf{close } n)\ ]\end{array}$$

$newNode \triangleq 0 \to \circ Node$

$Node \triangleq Hv \,|\, Tv$

$Tv \triangleq \mathtt{rec}(X).getNxt{:}\#\mathtt{L}(Tv)\,;\, disp{:}0 \vee getNxt{:}\#\mathtt{N}\,;\, X$

$Hv \triangleq setNxt{:}(Tv \mapsto 0)$

# typing node bundle

$$
\begin{array}{l}
\textbf{let } newNode = \lambda[].\textbf{var } nxt := \#\mathtt{N} \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{res } n \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad [\ \ setNxt = \lambda p.(\textbf{open } n; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad nx := \#\mathtt{L}(p); \textbf{close } n)\ | \\
\qquad\qquad\qquad\qquad\qquad\ \ getNxt = \textbf{open } n; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let } p = !nxt \textbf{ in } (\textbf{close } n; p)\ | \\
\qquad\qquad\qquad\qquad\qquad\ \ disp = (\textbf{open } n; \textbf{free } nxt; \textbf{close } n)\ ]
\end{array}
$$

$$
\begin{array}{l}
n : \rho\{nxt : \mathtt{rd}(\#\mathtt{N}); \mathtt{var}\}\{0\} \\
\ <: \\
n : \rho\, \mathtt{rec}(X).(\ \{nxt{:}\mathtt{rd}(\#\mathtt{L}(Tv)); \mathtt{var}\}\{nxt{:}\mathtt{var}\}; \{nxt{:}\mathtt{var}\}\{0\} \\
\qquad\qquad\qquad \vee \{nxt{:}\mathtt{rd}(\#\mathtt{N}); \mathtt{var}\}\{nxt{:}\mathtt{rd}(\#\mathtt{N}); \mathtt{var}\}; X\ ) \\
| \\
n : \rho\{nxt{:}\mathtt{rd}(\#\mathtt{N}); \mathtt{var}\}; \{nxt{:}\mathtt{rd}(\#\mathtt{L}(Tv)); \mathtt{var}\}
\end{array}
$$

# typing node bundle

$$
\begin{array}{l}
\textbf{let } newNode = \lambda[].\textbf{var } nxt := \#\mathtt{N} \textbf{ in} \\
\qquad\qquad\qquad\qquad \textbf{res } n \textbf{ in} \\
\qquad\qquad\qquad\quad [\ \ setNxt = \lambda p.(\textbf{open } n; \\
\qquad\qquad\qquad\qquad\qquad\qquad nx := \#\mathtt{L}(p); \textbf{close } n)\ | \\
\qquad\qquad\qquad\quad getNxt = \textbf{open } n; \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{let } p = !nxt \textbf{ in } (\textbf{close } n; p)\ | \\
\qquad\qquad\qquad\quad disp = (\textbf{open } n; \textbf{free } nxt; \textbf{close } n)\ ]
\end{array}
$$

$$
\begin{array}{l}
\{nxt : \mathtt{rd}(\#\mathtt{N}); \mathtt{var}\}\{0\} \\
= \\
\mathtt{rec}(X).(\ \{nxt{:}\mathtt{rd}(\#\mathtt{L}(Tv)); \mathtt{var}\}\{nxt{:}\mathtt{var}\}; \{nxt{:}\mathtt{var}\}\{0\} \\
\qquad\quad \vee \{nxt{:}\mathtt{rd}(\#\mathtt{N}); \mathtt{var}\}\{nxt{:}\mathtt{rd}(\#\mathtt{N}); \mathtt{var}\}; X\ ) \\
\bowtie \\
\{nxt{:}\mathtt{rd}(\#\mathtt{N}); \mathtt{var}\}; \{nxt{:}\mathtt{rd}(\#\mathtt{L}(Tv)); \mathtt{var}\}
\end{array}
$$

(projection)

# credits

- separation logics
- spatial logics for concurrency
- session and conversation types
- linear logic ( + connections with session / behavioral types )
- behavioral separation types
- rely-guarantee protocols

# closing thoughts

- A program in a typed concurrent programming language should not go wrong ! But it does (concurrency "bugs") !
- An open challenge for the CTC: how to ensure safety by typing of concurrency in realistic programming ?
- A broader understanding of the notion of behavioral type may contribute to better foundations for typing general concurrency, while preserving a healthy relationship with key core type theoretical (and logical) concepts.
- Behavioral specs in our sense (cf. regular expressions) are likely to facilitate adoption, as they seem to match software engineer (naive) reasoning about how "the program" works.