THE TYPE DISCIPLINE OF BEHAVIORAL SEPARATION

Luís Caires
(joint work with João C. Seco)
Universidade Nova de Lisboa CITI@DI

Set 2012 IFIP WG 2.2 Meeting CWI - Amsterdam

OVERVIEW

OVERVIEW

- **Statically** checking that ho imperative programs don't go wrong in the presence of interference is notoriously hard

OVERVIEW

- **Statically** checking that ho imperative programs don't go wrong in the presence of interference is notoriously hard
- Key problem: rule out "bad" interference, allow "good" interference, ensuring program correctness invariants

OVERVIEW

- **Statically** checking that ho imperative programs don’t go wrong in the presence of interference is notoriously hard
- Key problem: rule out "bad" interference, allow "good" interference, ensuring program correctness invariants
- Recent progress: separation logics, substructural types. Extending these approaches to general ho imperative concurrency is promising but still very challenging

OVERVIEW

- **Statically** checking that ho imperative programs don't go wrong in the presence of interference is notoriously hard
- Key problem: rule out "bad" interference, allow "good" interference, ensuring program correctness invariants
- Recent progress: separation logics, substructural types. Extending these approaches to general ho imperative concurrency is promising but still very challenging
- We intro ***behavioral separation*** as a general principle for disciplining interference in higher-order imperative concurrent programs (and illustrate with a type system)

KEY IDEAS

KEY IDEAS

- Depart from a state-based view towards a behavioral ("process description") view of assertions

KEY IDEAS

- Depart from a state-based view towards a behavioral ("process description") view of assertions
- Take inspiration on sep logics and beh types but shifting the focus from the separation of (static) state properties to the separation of (dynamic) usage behaviors of individual values

KEY IDEAS

- Depart from a state-based view towards a behavioral ("process description") view of assertions
- Take inspiration on sep logics and beh types but shifting the focus from the separation of (static) state properties to the separation of (dynamic) usage behaviors of individual values
- classical "structural" operators (usage) + "temporal" operators (traces) + "spatial" operators (aliasing / sharing)

KEY IDEAS

- Depart from a state-based view towards a behavioral ("process description") view of assertions
- Take inspiration on sep logics and beh types but shifting the focus from the separation of (static) state properties to the separation of (dynamic) usage behaviors of individual values
- classical "structural" operators (usage) + "temporal" operators (traces) + "spatial" operators (aliasing / sharing)
- We carry out our development within a clean substructural type theory based on a lambda calculus with references and concurrency constructs

PROGRAMMING LANGUAGE

$$
\begin{array}{rcll}
e, f & ::= & x & (\textit{Variable}) \\
 & | & \lambda x.e & (\textit{Abstraction}) \\
 & | & e_1 e_2 & (\textit{Application}) \\
 & | & \textbf{let } x = e_1 \textbf{ in } e_2 & (\textit{Definition}) \\
 & | & \textbf{var } a \textbf{ in } e & (\textit{Heap variable decl}) \\
 & | & a := v & (\textit{Assignment}) \\
 & | & a & (\textit{Dereference}) \\
 & | & [l_1 = e_1, \ldots] & (\textit{Tupling}) \\
 & | & e.l & (\textit{Selection}) \\
 & | & l(e) & (\textit{Variant}) \\
 & | & \textbf{case } e \textbf{ of } l_i(x_i) \rightarrow e_i & (\textit{Conditional}) \\
 & | & \textbf{rec}(X)e & (\textit{Recursion}) \\
 & | & X & (\textit{Recursion variable}) \\
 & | & \textbf{fork } e & (\textit{New thread}) \\
 & | & \textbf{wait } e & (\textit{Wait}) \\
 & | & \textbf{sync}(a)e & (\textit{Synchronized block})
\end{array}
$$

A COLLECTION ADT

$$
\begin{aligned}
&\textbf{let } newNode = \lambda[].\textbf{var } next, elt \textbf{ in}\\
&\qquad\qquad\qquad\qquad [\; setElt = \lambda e.(elt := e),\\
&\qquad\qquad\qquad\qquad\;\; getElt = elt,\\
&\qquad\qquad\qquad\qquad\;\; setNext = \lambda p.(next := p),\\
&\qquad\qquad\qquad\qquad\;\; getNext = next \;] \textbf{ in}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{let } newColl =\\
&\quad \lambda[].\textbf{var } hd, id \textbf{ in}\\
&\qquad [\; init = \lambda i.(hd := \texttt{NULL}; id := i)\\
&\qquad\;\; getId = id,\\
&\qquad\;\; add = \lambda e.\textbf{let } n = (newNode \textbf{ nil}) \textbf{ in}\\
&\qquad\qquad\qquad ((n.setElt\ e); (n.setNext\ hd); hd := \texttt{NODE}(n)),\\
&\qquad\;\; scan = \textbf{var } s \textbf{ in } (\\
&\qquad\qquad\qquad s := hd;\\
&\qquad\qquad\qquad \textbf{rec } L.\textbf{case } s \textbf{ of}\\
&\qquad\qquad\qquad\qquad \texttt{NULL} \to \textbf{nil}\\
&\qquad\qquad\qquad\qquad \texttt{NODE}(n) \to (s := n.getNext; L))]
\end{aligned}
$$

A COLLECTION ADT

$$
\begin{array}{l}
\textbf{let } newNode = \lambda[].\textbf{var } next, elt \textbf{ in} \\
\qquad\qquad\qquad [\ setElt = \lambda e.(elt := e), \\
\qquad\qquad\qquad\ \ getElt = elt, \\
\qquad\qquad\qquad\ \ setNext = \lambda p.(next := p), \\
\qquad\qquad\qquad\ \ getNext = next\ ] \textbf{ in}
\end{array}
$$

$$
\begin{array}{l}
\textbf{let } newColl = \\
\quad \lambda[].\textbf{var } hd, id \textbf{ in} \\
\qquad [\ init = \lambda i.(hd := \mathtt{NULL}; id := i) \\
\qquad\ \ getId = id, \\
\qquad\ \ add = \lambda e.\textbf{let } n = (newNode\ \textbf{nil}) \textbf{ in} \\
\qquad\qquad\qquad ((n.setElt\ e); (n.setNext\ hd); hd{:=}\mathtt{NODE}(n)), \\
\qquad\ \ scan = \textbf{var } s \textbf{ in } (\\
\qquad\qquad\qquad s := hd; \\
\qquad\qquad\qquad \textbf{rec } L.\textbf{case } s \textbf{ of} \\
\qquad\qquad\qquad\qquad \mathtt{NULL} \rightarrow \textbf{nil} \\
\qquad\qquad\qquad\qquad \mathtt{NODE}(n) \rightarrow (s := n.getNext; L))]
\end{array}
$$

$$SC \triangleq init{:}\mathtt{str}{\mapsto}0\ ;(getId{:}\mathtt{str}\ \&\ add{:}\mathtt{nat}{\mapsto}0\ \&\ scan{:}0)^*$$

A COLLECTION ADT

$$
\begin{array}{l}
\textbf{let } newNode = \lambda[].\textbf{var } next, elt \textbf{ in} \\
\qquad\qquad\qquad [\ \ setElt = \lambda e.(elt := e), \\
\qquad\qquad\qquad\quad getElt = elt, \\
\qquad\qquad\qquad\quad setNext = \lambda p.(next := p), \\
\qquad\qquad\qquad\quad getNext = next \ ] \textbf{ in}
\end{array}
$$

$$
\begin{array}{l}
\textbf{let } newColl = \\
\quad \lambda[].\textbf{var } hd, id \textbf{ in} \\
\qquad [\ \ init = \lambda i.(hd := \mathtt{NULL}; id := i) \\
\qquad\ \ getId = id, \\
\qquad\ \ add = \lambda e.\textbf{let } n = (newNode \textbf{ nil}) \textbf{ in} \\
\qquad\qquad\qquad ((n.setElt\ e); (n.setNext\ hd); hd{:=}\mathtt{NODE}(n)), \\
\qquad\ \ scan = \textbf{var } s \textbf{ in } (\\
\qquad\qquad\qquad s := hd; \\
\qquad\qquad\qquad \textbf{rec } L.\textbf{case } s \textbf{ of} \\
\qquad\qquad\qquad\qquad \mathtt{NULL} \to \textbf{nil} \\
\qquad\qquad\qquad\qquad \mathtt{NODE}(n) \to (s := n.getNext; L))]
\end{array}
$$

$$CC \triangleq (init{:}\mathtt{str}{\mapsto}0)\,;(!getId{:}\mathtt{str}\,|(!scan{:}0\,;add{:}\mathtt{nat}{\mapsto}0)^*)$$

TYPING THE COLLECTION

$$CC \triangleq (init{:}\mathtt{str}{\mapsto}\mathtt{0})\,;(!getId{:}\mathtt{str}\,|(!scan{:}\mathtt{0}\,;add{:}\mathtt{nat}{\mapsto}0)^*)$$

$$newColl : \mathtt{0} \mapsto \circ CC$$

USING THE COLLECTION

USING THE COLLECTION

$\textbf{let } c = newColl\ [] \textbf{ in } (c.init\ \text{``}my\text{''}); c.scan; (c.add\ 1)$

$\textbf{let } c = newColl\ [] \textbf{ in } (c.init\ \text{``}my\text{''}); (c.add\ 1); c.getId; c.scan$

$CC \triangleq (init{:}\mathtt{str}{\mapsto}0)\,;(!getId{:}\mathtt{str}\,|(!scan{:}0\,;add{:}\mathtt{nat}{\mapsto}0)^*)$

BORROWING

$$\textbf{let } c = newColl\ [] \textbf{ in}$$
$$\quad \textbf{let } f = \lambda x.(x.init \text{ ``}your\text{''}) \textbf{ in } (f\ c); (c.add\ 2)$$

$$\textbf{let } c = newColl\ [] \textbf{ in}$$
$$\quad \textbf{let } g = \lambda x.(x.scan) \textbf{ in}$$
$$\qquad (c.init \text{ ``}my\text{''}); (g\ c); c.scan; (c.add\ 2); (g\ c)$$

$$\textbf{let } c = newColl\ [] \textbf{ in}$$
$$\quad \textbf{let } h = \lambda x.(x.init \text{ ``}your\text{''}) \textbf{ in } (c.add\ 2); (h\ c)$$

BORROWING THROUGH THE STORE

$\textbf{let } c = newColl\ [] \textbf{ in var } a \textbf{ in}$
$\quad (a := c; (a.init \text{ ``}my\text{''}); (a.add\ 1); (a.add\ 1); c.scan))$

$\textbf{let } c = newColl\ [] \textbf{ in } ((c.init \text{ ``}my\text{''});$
$\quad \textbf{var } a \textbf{ in } (a := c.add; (a\ 1); c.scan))$

FRAMING

$\textbf{let}\ c = newColl\ []\ \textbf{in let}\ m = c.init\ \textbf{in}\ c.scan$

$\textbf{var}\ s\ \textbf{in}\ (s := \text{“}hi\text{”};$
$\quad \textbf{let}\ F = \lambda x.(\textbf{let}\ c = newColl\ []\ \textbf{in}\ (c.init\ x; c))\ \textbf{in}$
$\quad (\textbf{let}\ u = (F\ s)\ \textbf{in}\ (s := \text{“}ok\text{”}; u.add\ 1)))$

$\textbf{let}\ c = newColl\ []\ \textbf{in}$
$\quad \textbf{var}\ a\ \textbf{in let}\ f = \lambda x.a := x\ \textbf{in}\ ((f\ c); (a.init\ \text{“}y\text{”}))$

CONCURRENCY

$$\textbf{let } c = newColl\ [\,]\ \textbf{in}$$
$$((c.init\ \text{“}my\text{”}); (c.add\ 1); (c.scan \| c.scan))$$

$$\textbf{let } c = newColl\ [\,]\ \textbf{in let } f = \lambda x.(x.scan)\ \textbf{in}$$
$$((c.init\ \text{“}my\text{”}); ((f\ c) \| c.scan); (c.getId \| (c.add\ 2)); (f\ c))$$

$$\textbf{let } c = newColl\ [\,]\ \textbf{in } ((c.init\ \text{“}my\text{”}); ((c.add\ 1) \| (c.scan))$$

$$\textbf{let } c = newColl\ [\,]\ \textbf{in}$$
$$\textbf{let } f = \lambda x.((x.add\ 0) \| (c.add\ 1))\ \textbf{in } ((c.init\ \text{“}my\text{”}); (f\ c))$$

CONCURRENCY

$$\textbf{let } c = newColl\ [] \textbf{ in}$$
$$\quad \textbf{let } f = \lambda x.(x.scan \| c.scan) \textbf{ in } ((c.init \text{ ``}my\text{''}); (f\ c))$$

$$\textbf{let } c = newColl\ [] \textbf{ in } ((c.init \text{ ``}my\text{''});$$
$$\quad \textbf{var } a \textbf{ in } (a := \textbf{fork}(c.scan); c.scan; \textbf{wait}(a); (c.add\ 1)))$$

$$\textbf{let } c = newColl\ [] \textbf{ in } ((c.init \text{ ``}my\text{''});$$
$$\quad \textbf{var } a \textbf{ in } (a := \textbf{fork}(c.scan); (c.add\ 1); \textbf{wait}(a)))$$

INVARIANT-BASED SEPARATION

$$
\begin{array}{l}
\textbf{let } newColl = \\
\quad \lambda[].\textbf{var } hd, id, inv \textbf{ in} \\
\qquad [\ \ init = \lambda i.(hd := \texttt{NULL}; id := i) \\
\qquad\quad getId = id, \\
\qquad\quad add = \lambda e.\textbf{sync}(inv)(\textbf{let } n = (newNode\ \textbf{nil})\ \textbf{in} \\
\qquad\qquad\qquad ((n.setElt\ e); (n.setNext\ hd); hd{:=}\texttt{NODE}(n))), \\
\qquad\quad scan = \textbf{sync}(inv)(\textbf{var } s \textbf{ in } (\\
\qquad\qquad\qquad s := hd; \\
\qquad\qquad\qquad \textbf{rec } L.\textbf{case } s \textbf{ of} \\
\qquad\qquad\qquad\qquad \texttt{NULL} \rightarrow \textbf{nil} \\
\qquad\qquad\qquad\qquad \texttt{NODE}(n) \rightarrow (s := n.getNext; L)))]
\end{array}
$$

$$C \triangleq (init{:}\texttt{str}{\mid\!\rightarrow}0)\ ;(!getId{:}\texttt{str} \mid !scan{:}0 \mid !add{:}\texttt{nat}{\mid\!\rightarrow}0)$$

BEHAVIORAL SEPARATION TYPES

BEHAVIORAL SEPARATION TYPES

$$
\begin{array}{lclrclr}
T, U & ::= & \mathtt{0} & (\textit{stop}) & \mid & T \mapsto V & (\textit{function}) \\
 & \mid & T \,\mathtt{;}\, U & (\textit{sequential}) & \mid & T \mid U & (\textit{parallel}) \\
 & \mid & T \,\&\, U & (\textit{intersection}) & \mid & l{:}T & (\textit{qualification}) \\
 & \mid & \bigoplus_{l \in I} l{:}T_l & (\textit{sum}) & \mid & !T & (\textit{shared}) \\
 & \mid & \circ T & (\textit{isolated}) & \mid & \tau(T) & (\textit{thread}) \\
 & \mid & \mathtt{rec}(X)T & (\textit{recursion}) & \mid & X & (\textit{recursion var})
\end{array}
$$

SEQUENTIAL AND PARALLEL

$$U ; (V ; T) <:> (U ; V) ; T \quad U ; 0 <:> U \quad 0 ; U <:> U$$

$$U \mid (V \mid T) <:> (U \mid V) \mid T \quad U \mid V <:> V \mid U \quad U \mid 0 <:> U$$

$$(A ; C) \mid (B ; D) <: (A \mid B) ; (C \mid D)$$

INTERSECTION

$$U \mathbin{\&} V <: U$$

$$U \mathbin{\&} V <: V$$

$$U <: U \mathbin{\&} U$$

SHARED

$$!U <: U$$

$$!U <: !!U$$

$$0 <: !0$$

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

$$!U <: 0$$

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

ISOLATED

$$0 <: \circ 0$$

$$\circ A \,|\, \circ B <: \circ(A \,|\, B)$$

$$\circ A <: A$$

$$\circ A <: \circ\circ A$$

$$\circ A <: 0$$

$$!\circ A <: \circ !A$$

$$(\circ A \,|\, B)\,;C <: \circ A \,|(B\,;C)$$

TYPES FOR HEAP REFERENCES

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

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

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

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

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

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

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

KEY ALGEBRAIC STRUCTURE

- symmetric monoidal closed

$$(T, 0, (- \mid -), \mapsto)$$

- concurrent Kleene algebra

$$(T, (- \,\&\, -), (- \mid -), (- \,;\, -), 0)$$

- monoidal co-monads

$$\circ(-)$$

$$!(-)$$

REMARKS

- interleaving

$$U \mid V \;<:\; V \,;\, U$$

- isolation

$$(\circ U)\,;\,V \;<:\; (\circ U) \mid V \qquad\qquad (\circ A)\,;\,B \;<:\; B\,;(\circ A)$$

- shared isolated types

let $T = !\circ U$. Then T <:> $T \mid T$ and T <:> $\circ T$.

include “pure” basic types, such as nat, bool, etc

REMARKS

- arrow types

shared $!(U \mapsto V)$

iterated $(U \mapsto V)^*$

pure $!\circ(T \mapsto T)$

...

TYPE SYSTEM

TYPE ASSERTIONS

$$A, B ::= x{:}T \mid A\,\mathbf{;}\,B \mid A|B \mid A \mathbin{\&} B \mid !A \mid \circ A \mid X \mid \mathtt{rec}(X)A$$

TYPE ASSERTIONS

$$A, B ::= x{:}T \mid A\,\mathbf{;}\,B \mid A|B \mid A \mathbin{\&} B \mid\, !A \mid \circ A \mid X \mid \mathtt{rec}(X)A$$

$(f{:}U \mapsto V\ \mathbf{;}\ y{:}U) \mid z{:}U$ $\quad$ $(f\ \ z)$ $\quad$ $y{:}U$

$(f{:}U \mapsto V\ \mathbf{;}\ y{:}U) \mid z{:}U$ $\quad$ $(f\ \ y)$ $\quad$ invalid for precondition

TYPING JUDGMENTS

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

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

TYPING JUDGMENTS

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

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

example

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

STRUCTURAL

$$x{:}U \vdash_z x :: z{:}U \ (Id)$$

$$\frac{A \vdash_x e_1 :: B \quad B \vdash_y e_2 :: C}{A \vdash_y \textbf{let } x = e_1 \textbf{ in } e_2 :: C} \ (Let)$$

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

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

ARROW TYPE

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

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

TUPLE TYPE

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

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

INTERSECTION TYPE

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

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

BEHAVIORAL SEPARATION TYPES

$$\mathsf{0} \vdash_y v :: \mathsf{0}\ (VStop) \qquad \frac{A \vdash_y v :: C \quad B \vdash_y v :: D}{A\,\mathbf{;}\,B \vdash_y v :: C\,\mathbf{;}\,D}(VSeq)$$

$$\frac{!A_1 \mid \ldots \mid !A_n \vdash_x v{::}B}{!A_1 \mid \ldots \mid !A_n \vdash_x v{::}!B}\,(VShr) \qquad \frac{A \vdash_y v{::}C \quad B \vdash_y v{::}D}{A \mid B \vdash_y v :: C \mid D}(VPar)$$

BEHAVIORAL SEPARATION TYPES

$$A \vdash_y v :: A \; (VId) \qquad \frac{B \vdash_y v :: C}{A\,\mathbf{;}\,B \vdash_y v :: A\,\mathbf{;}\,C} \; (VLPar)$$

ISOLATED TYPE

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

SUM TYPE

$$\frac{A \vdash_y e_c :: y : \bigoplus_{l \in I} l{:}T_l \quad x_i{:}T_i \,|\, B \vdash_z e_i :: C}{A \,|\, B \vdash_z \textbf{case } e_c \textbf{ of } l(x) \rightarrow e :: C} \; (Case)$$

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

HEAP REFERENCES

$$\frac{a{:}\mathtt{var} \mid A \vdash_x e :: C}{A \vdash_x \textbf{var}\ a\ \textbf{in}\ e :: C}\ (Var)$$

HEAP REFERENCES (DEREF)

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

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

HEAP REFERENCES (ASSIGN)

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

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

EXAMPLE (WRONG)

$$r{:}U \mid a{:}\mathtt{wr}(U) \vdash_x r :: x{:}U \mid a{:}\mathtt{wr}(U)$$
$$r{:}U \mid a{:}\mathtt{wr}(U) \vdash_x a := r :: 0$$
$$r{:}U \,;\, V \mid a{:}\mathtt{wr}(U) \vdash_x a := r :: r{:}V$$
$$r{:}U \,;\, V \mid a{:}\mathtt{wr}(U); \mathtt{rd}(U) \vdash_x a := r :: r{:}V \,;\, a{:}\mathtt{rd}(U)$$
$$r{:}U \,;\, V \mid a{:}\mathtt{use} \vdash_x a := r :: r{:}V \,;\, a{:}\mathtt{rd}(U)$$

EXAMPLE (RIGHT)

$$r{:}U \mid a{:}\texttt{use} \vdash_x r :: x{:}U \mid a{:}\texttt{use}$$
$$r{:}U \mid a{:}\texttt{use} \vdash_x a := r :: a{:}\texttt{rd}(U)$$
$$r{:}U \,;\, V \mid a{:}\texttt{use} \vdash_x a := r :: a{:}\texttt{rd}(U) \,;\, r{:}V$$

A COLLECTION ADT

$$
\begin{aligned}
&\textbf{let } newNode = \lambda[].\textbf{var } next, elt \textbf{ in} \\
&\qquad\qquad\qquad [\ setElt = \lambda e.(elt := e), \\
&\qquad\qquad\qquad\ \ getElt = elt, \\
&\qquad\qquad\qquad\ \ setNext = \lambda p.(next := p), \\
&\qquad\qquad\qquad\ \ getNext = next\ ] \textbf{ in} \\
&\textbf{let } newColl = \\
&\quad \lambda[].\textbf{var } hd, id \textbf{ in} \\
&\qquad [\ init = \lambda i.(hd := \mathtt{NULL}; id := i) \\
&\qquad\ \ getId = id, \\
&\qquad\ \ add = \lambda e.\textbf{let } n = (newNode\ \textbf{nil}) \textbf{ in} \\
&\qquad\qquad\qquad ((n.setElt\ e); (n.setNext\ hd); hd{:=}\mathtt{NODE}(n)), \\
&\qquad\ \ scan = \textbf{var } s \textbf{ in } (\\
&\qquad\qquad\qquad s := hd; \\
&\qquad\qquad\qquad \textbf{rec } L.\textbf{case } s \textbf{ of} \\
&\qquad\qquad\qquad\qquad \mathtt{NULL} \rightarrow \textbf{nil} \\
&\qquad\qquad\qquad\qquad \mathtt{NODE}(n) \rightarrow (s := n.getNext; L))]
\end{aligned}
$$

TYPING THE COLLECTION ADT

$$InitNode \triangleq setElt{:}(\mathtt{nat} \mapsto 0)\,;\, setNext{:}(!{\circ}PNode \mapsto 0)$$

$$INode \triangleq\, !getNext{:}PNode \mid !getElt{:}\mathtt{nat}$$

$$Node \triangleq InitNode\,;\, !{\circ}INode$$

$$PNode \triangleq\, !\mathtt{Opt}(INode)$$

$$CC \triangleq (init{:}\mathtt{str}{\mapsto}\mathtt{0})\,;(!getId{:}\mathtt{str} \mid (!scan{:}\mathtt{0}\,;add{:}\mathtt{nat}{\mapsto}0)^{*})$$

INVARIANT-BASED SEPARATION

INVARIANT-BASED SEPARATION

- “external usage” view of $(- \mid -)$ naturally enables behavioral separation to conceal (abstract) “good” interference
- relax “internal physical” separation (disjointness) to “external observable” safe usage separation
- typed atomicity construct ($\textbf{sync}(inv)e$) to force behavioral separation (cf. the Hoare monitor principle)
- typed atomicity construct already useful in non-concurrent
- serialization invariant $\iota(inv)$ must be an isolated assertion $\circ R$

INVARIANT-BASED SEPARATION

$\boxed{A \vdash^{\iota}_{z} e :: B}$ $\quad$ (e types from A to z in B under ι)

- ι invariant mapping
- assigns a "lock" invariant to each heap variable (cf. Java)
- a lock invariant is any assertion R s.t. $R <: \circ R$

INVARIANT-BASED SEPARATION

$$\frac{A <: B \,|\, R \quad a{:}\mathtt{var} \,|\; B \vdash_x^{\iota\{R/a\}} e :: C}{A \vdash_x^{\iota} \textbf{var}\ a\ \textbf{in}\ e :: C}\ (\mathit{Var})$$

$$\frac{\iota(a) \,|\, A \vdash_x^{\iota \backslash a} e :: \iota(a) \,|\, B}{A \vdash_x^{\iota} \textbf{sync}(a)e :: B}\ (\mathit{Sync})$$

“ATOMIC” MEMORY CELL

$$
\begin{aligned}
&\textbf{let } atomic = \lambda v.\\
&\quad \textbf{var } s \textbf{ in } s := v;\\
&\qquad \textbf{var } lock \textbf{ in } [\ set = \lambda x.\textbf{sync}(lock)s := x,\\
&\qquad\qquad\qquad\quad get = \textbf{sync}(lock)s\ ] \textbf{ in } \ldots
\end{aligned}
$$

$$atomic{:}U \mid\!\!\rightarrow (!set{:}(U \mapsto 0) \mid !get{:}U)$$

FIFO QUEUE ON LINKED LIST

$$
\begin{array}{l}
\textbf{let}\ new = \\
\quad \lambda[].\textbf{var}\ next\ \textbf{in} \\
\qquad next := \texttt{NULL}; \\
\qquad \textbf{var}\ shr\ \textbf{in} \\
\qquad\quad [\ \ unLink = \textbf{sync}(shr)\textbf{let}\ x = next \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ (next := \texttt{NULL}; x) \\
\qquad\qquad link = \lambda x.\textbf{sync}(shr) next := x\ ] \\
\textbf{in}\ \textbf{var}\ head, tail\ \textbf{in}\ (\\
\qquad head := \texttt{NULL}; tail := \texttt{NULL}; \\
\qquad [\ \ enq = \textbf{let}\ n = (new\ \textbf{nil})\ \textbf{in} \\
\qquad\qquad\qquad \textbf{case}\ tail\ \textbf{of} \\
\qquad\qquad\qquad \texttt{NULL} \rightarrow (head := \texttt{NODE}(n); \\
\qquad\qquad\qquad\qquad\quad tail := \texttt{NODE}(n)) \\
\qquad\qquad\qquad \texttt{NODE}(y) \rightarrow (y.link\ \texttt{NODE}(n)); \\
\qquad\qquad\qquad\qquad\quad tail := \texttt{NODE}(n)), \\
\qquad\quad deq = \textbf{case}\ head\ \textbf{of} \\
\qquad\qquad\quad \texttt{NULL} \rightarrow head := \texttt{NULL} \\
\qquad\qquad\quad \texttt{NODE}(y) \rightarrow (head := y.unLink; \\
\qquad\qquad\qquad\quad \textbf{case}\ head\ \textbf{of} \\
\qquad\qquad\qquad\quad \texttt{NULL} \rightarrow tail := \texttt{NULL}; head := \texttt{NULL} \\
\qquad\qquad\qquad\quad \texttt{NODE}(y) \rightarrow head := \texttt{NODE}(y))\ ]
\end{array}
$$

$$
\begin{array}{l}
Node \triangleq HeadT \mid TailT \\
SHeadT \triangleq \mathtt{Opt}(HeadT) \\
HeadT \triangleq \circ unlink{:}\circ SHeadT \\
TailT \triangleq \circ link{:}\circ SHeadT \mapsto \mathsf{0}
\end{array}
$$

$$\vdash_q SQueue :: (q{:}enq{:}\mathsf{0} \,\&\, q{:}deq{:}\mathsf{0})^*$$

FIFO QUEUE ON LINKED LIST

$$
\begin{array}{l}
\textbf{let } new = \\
\quad \lambda[].\textbf{var } next \textbf{ in} \\
\qquad next := \mathtt{NULL}; \\
\\
\qquad\quad [\ \ unLink = \textbf{let } x = next \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } (next := \mathtt{NULL}; x) \\
\qquad\qquad link = \lambda x.next := x\ ] \\
\textbf{in var } head, tail \textbf{ in } (\\
\qquad head := \mathtt{NULL}; tail := \mathtt{NULL}; \\
\qquad [\ \ enq = \textbf{let } n = (new\ \textbf{nil})\ \textbf{in} \\
\qquad\qquad\qquad \textbf{case } tail \textbf{ of} \\
\qquad\qquad\qquad \mathtt{NULL} \rightarrow (head := \mathtt{NODE}(n); \\
\qquad\qquad\qquad\qquad\quad tail := \mathtt{NODE}(n)) \\
\qquad\qquad\qquad \mathtt{NODE}(y) \rightarrow (y.link\ \mathtt{NODE}(n)); \\
\qquad\qquad\qquad\qquad\quad tail := \mathtt{NODE}(n)), \\
\qquad\quad deq = \textbf{case } head \textbf{ of} \\
\qquad\qquad\quad \mathtt{NULL} \rightarrow head := \mathtt{NULL} \\
\qquad\qquad\quad \mathtt{NODE}(y) \rightarrow (head := y.unLink; \\
\qquad\qquad\qquad\quad \textbf{case } head \textbf{ of} \\
\qquad\qquad\qquad\quad \mathtt{NULL} \rightarrow tail := \mathtt{NULL}; head := \mathtt{NULL} \\
\qquad\qquad\qquad\quad \mathtt{NODE}(y) \rightarrow head := \mathtt{NODE}(y))\ ]
\end{array}
$$

$$
\begin{array}{l}
Node \triangleq HeadT \mid TailT \\
SHeadT \triangleq \mathtt{Opt}(HeadT) \\
HeadT \triangleq \circ unlink{:}\circ SHeadT \\
TailT \triangleq \circ link{:}\circ SHeadT \mapsto 0
\end{array}
$$

$$\vdash_q SQueue :: (q{:}enq{:}0 \,\&\, q{:}deq{:}0)^*$$

CONCURRENT FIFO QUEUE

$\textbf{var}\ head, tail\ \textbf{in}\ ($

$$head := \texttt{NULL}; tail := \texttt{NULL};$$

$$\textbf{var}\ qinv\ \textbf{in}$$

$$[\quad enq = \textbf{sync}(qinv)(\ldots)$$
$$deq = \textbf{sync}(qinv)(\ldots)\quad]$$

- invariant $\iota(qinv)$

$$head{:}\texttt{rd}(\circ SHeadT)\,;\texttt{var}\mid tail{:}\texttt{rd}(\circ STailT)\,;\texttt{var}$$

- concurrent FIFO Queue interface and client code

$$\vdash_q CQueue :: !enq{:}0 \mid !q{:}deq{:}0$$

$$\textbf{let}\ q = CQueue\ \textbf{in}\ (q.enq; q.enq \| q.deq; q.deq)$$

LANDIN'S KNOT

var a **in** ($a := \lambda x.x;$
var $linv$ **in let** $f = \lambda y.($**sync**$(linv)(a)\ y)$
in (**sync**$(linv)(a := f); (f$ **nil**$))$)

- invariant $\iota(linv)$

$$a{:}\mathtt{rd}(!\circ(0 \mapsto 0))\,;\mathtt{var}$$

- type for f : $!\circ(0 \mapsto 0)$

SUM UP

- We introduce the concept of ***behavioral separation*** as a general principle for disciplining interference in higher-order imperative concurrent programs
- We develop the concept within a clean substructural typed lambda calculus, combining ideas from separation logic and behavioral type systems for process algebras
- Expressiveness of our approach extends current static verification of aliasing and concurrency (fine-grained state manipulation, ho store, first-class threads, seq-par frame dependency, and synchronization (atomicity) constructs)
- We are investigating algorithmic properties of the system