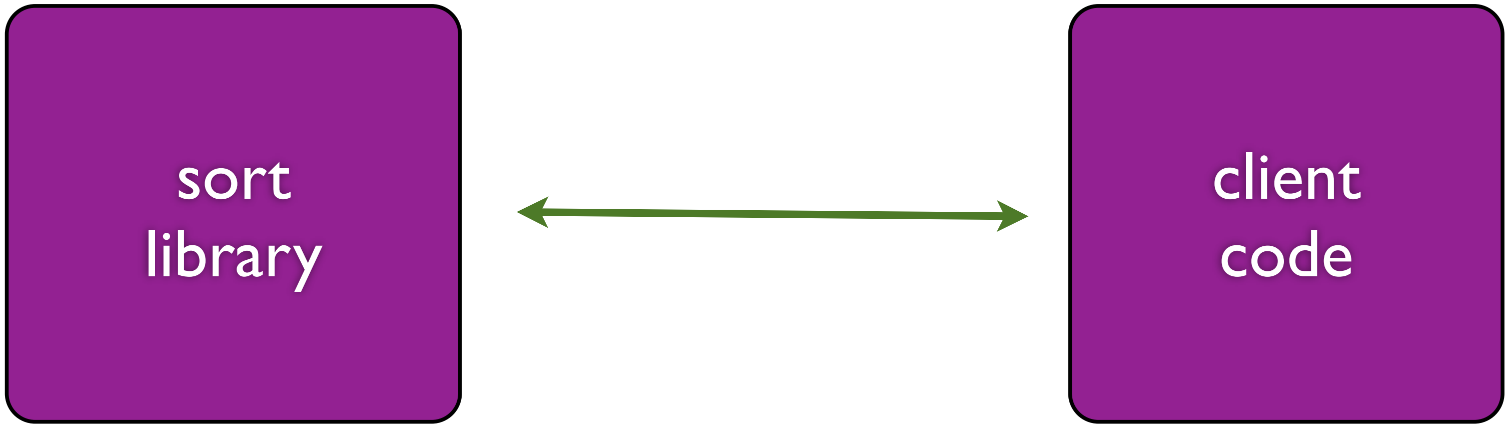
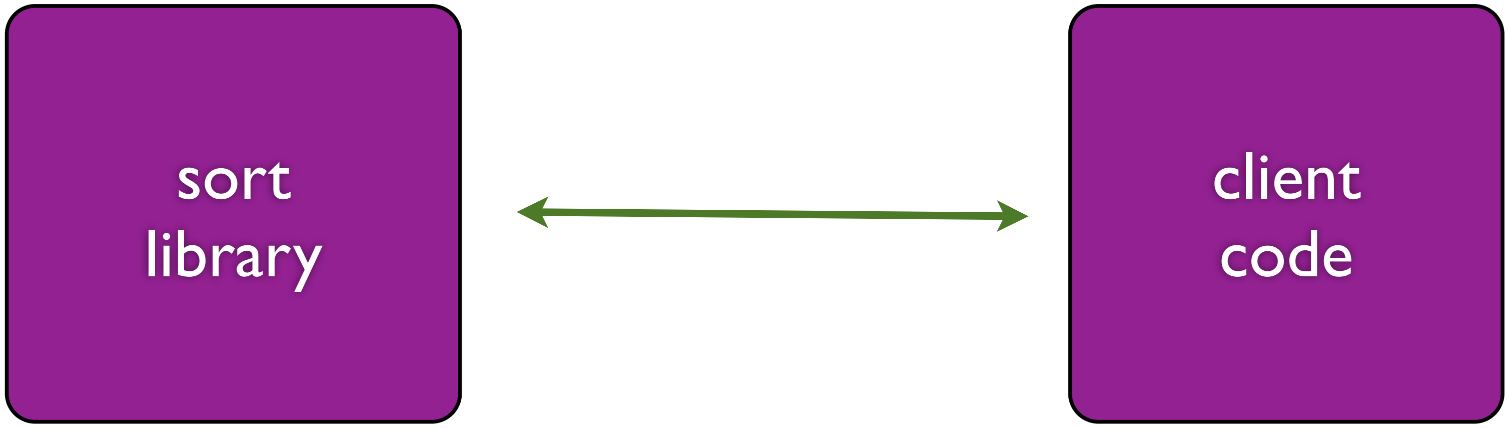
Temporal Higher-Order Contracts

Tim Disney   Jay McCarthy   Cormac Flanagan

ICFP '11



```
SortContract =
  sort : (List Int)
         (cmp : Int → Int → Bool)
       → (List Int)
```

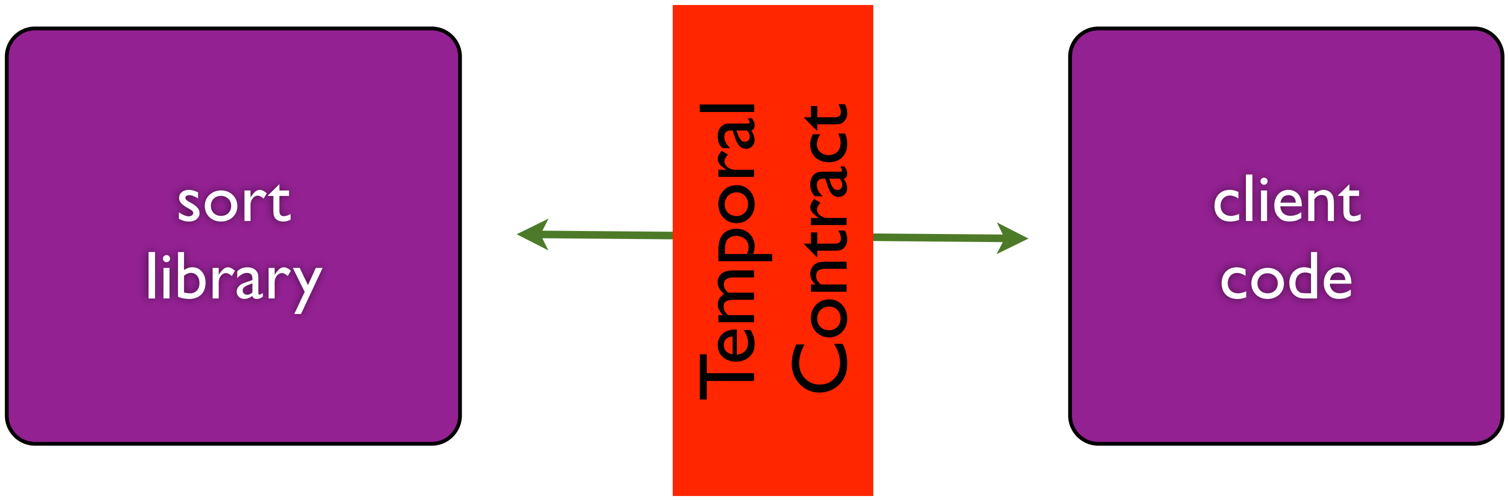


```
SortContract =
  sort : (List Int)
         (cmp : Int → Int → Bool)
       → (List Int)
```

sort is not re-entrant

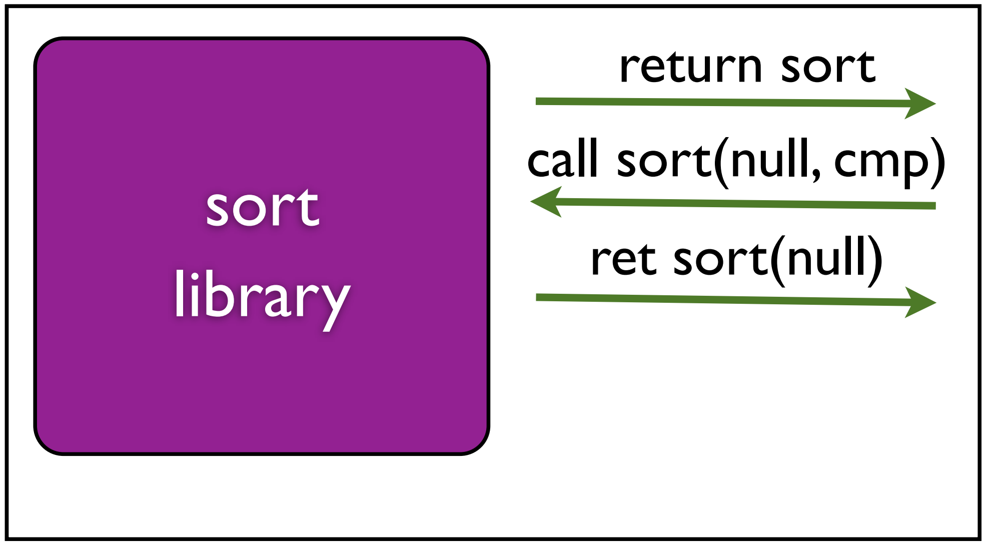
cmp is atomic

cmp is consistent



- <u>Non-interference</u>: Contracts cannot influence correct executions
- <u>Trace completeness</u>: Contracts can enforce any decidable restriction on module behavior

CSI Abstract Machine



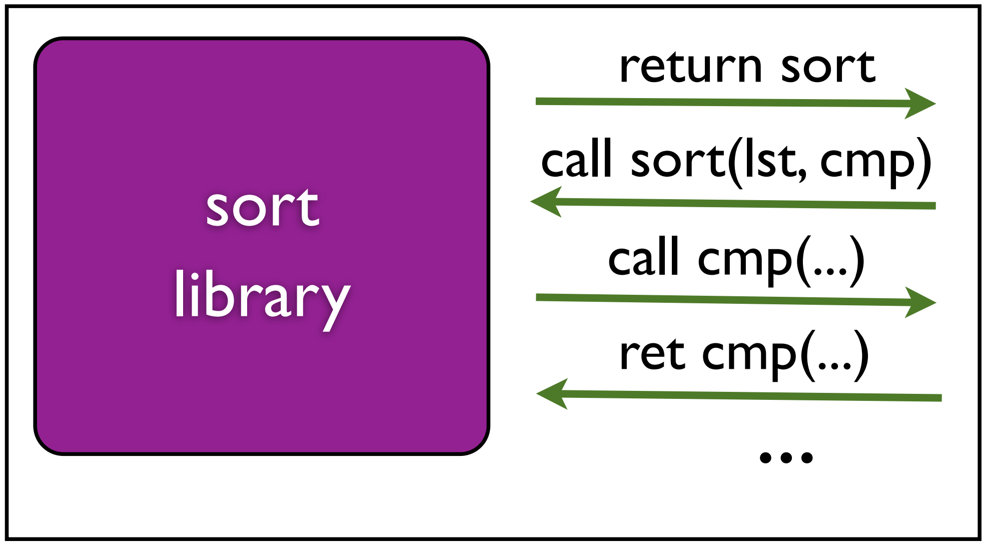
Properties of traces

- alternating send, receive
- stack-like calls & returns
- only send constants, vars
- notion of variable scope
- sends deterministic
- receive non-deterministic

CSI machine extends Control-Store machine with RPC

Semantics of module =
set of traces generated under CSI machine

CSI Abstract Machine



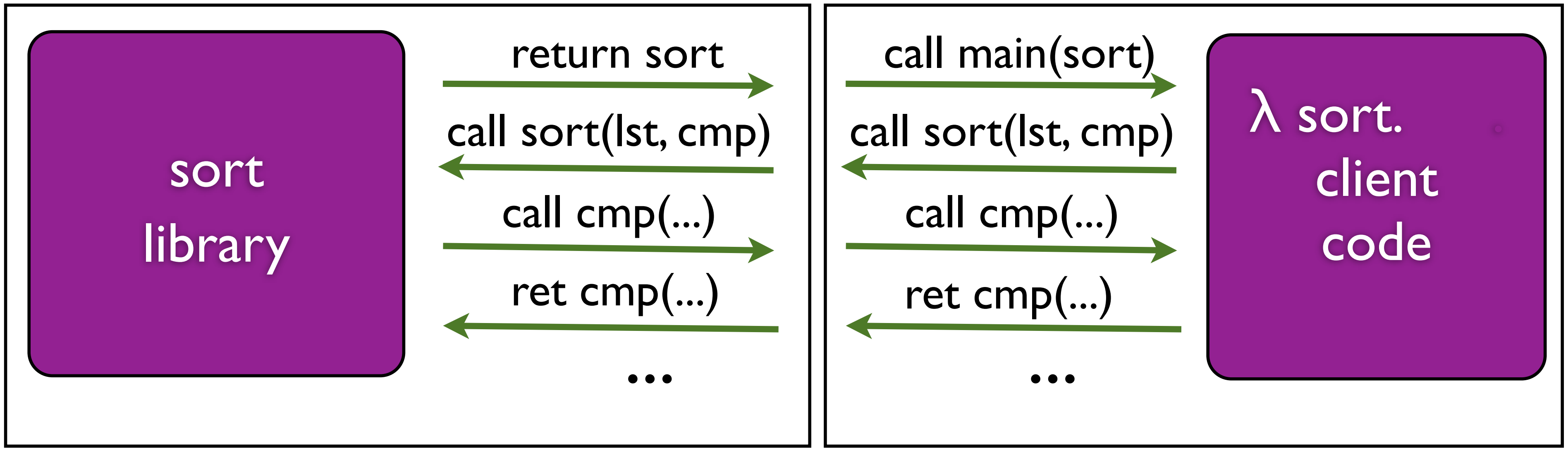
Properties of traces

- alternating send, receive
- stack-like calls & returns
- only send constants, vars
- notion of variable scope
- sends deterministic
- receive non-deterministic

CSI machine extends Control-Store machine with RPC

Semantics of module =
set of traces generated under CSI machine

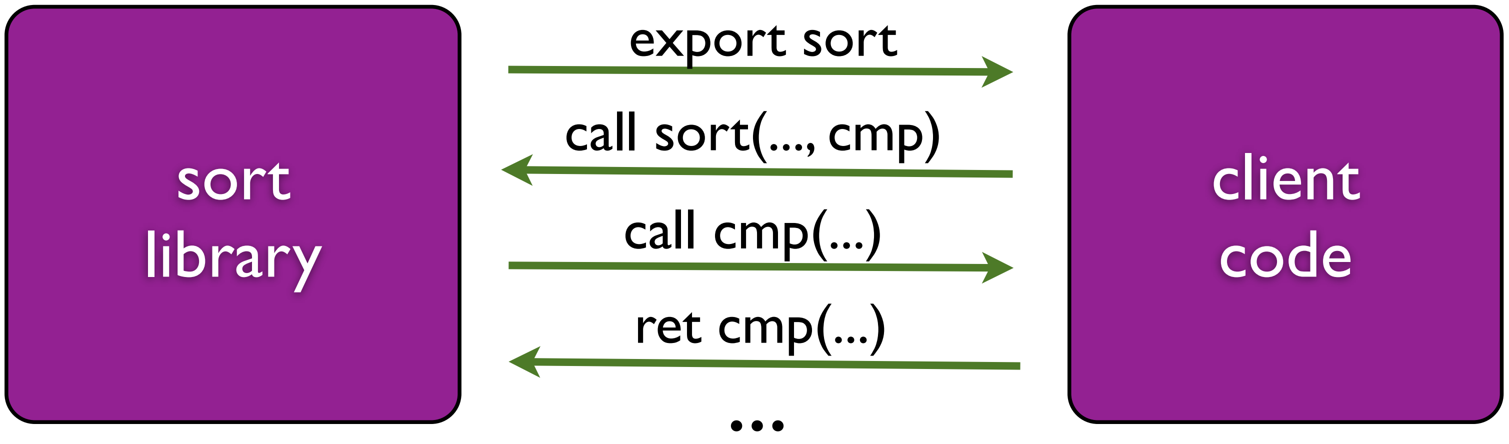
Linking CSI Machines



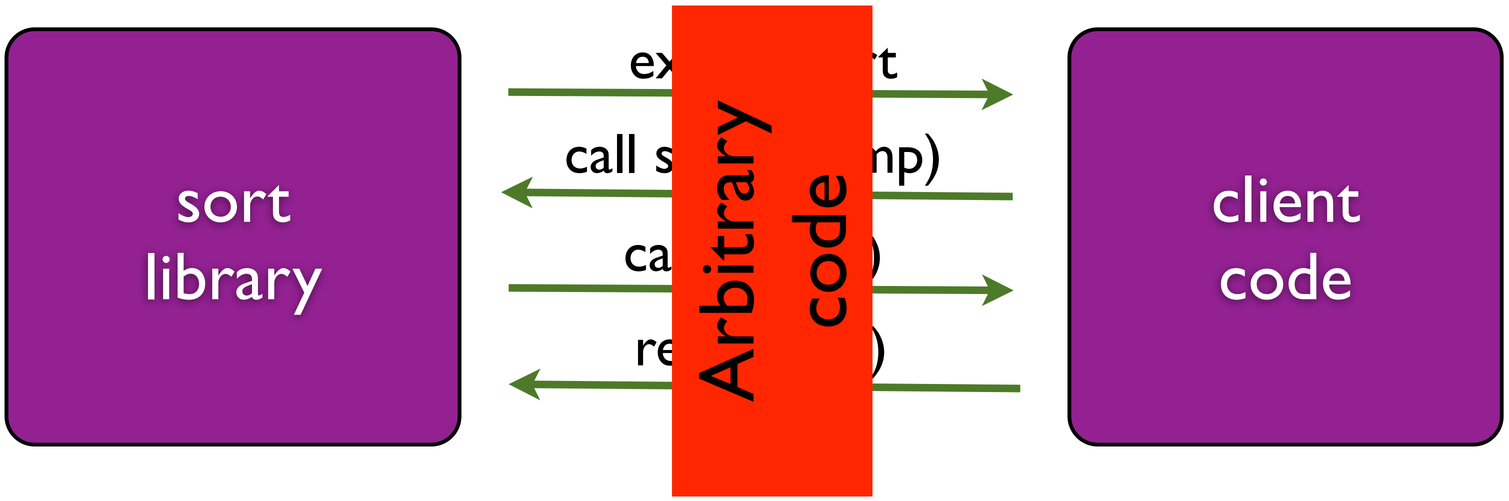
Module composition by linking CSI machines (matching sends with receives, etc)

Equivalent to running (client sort) on single machine

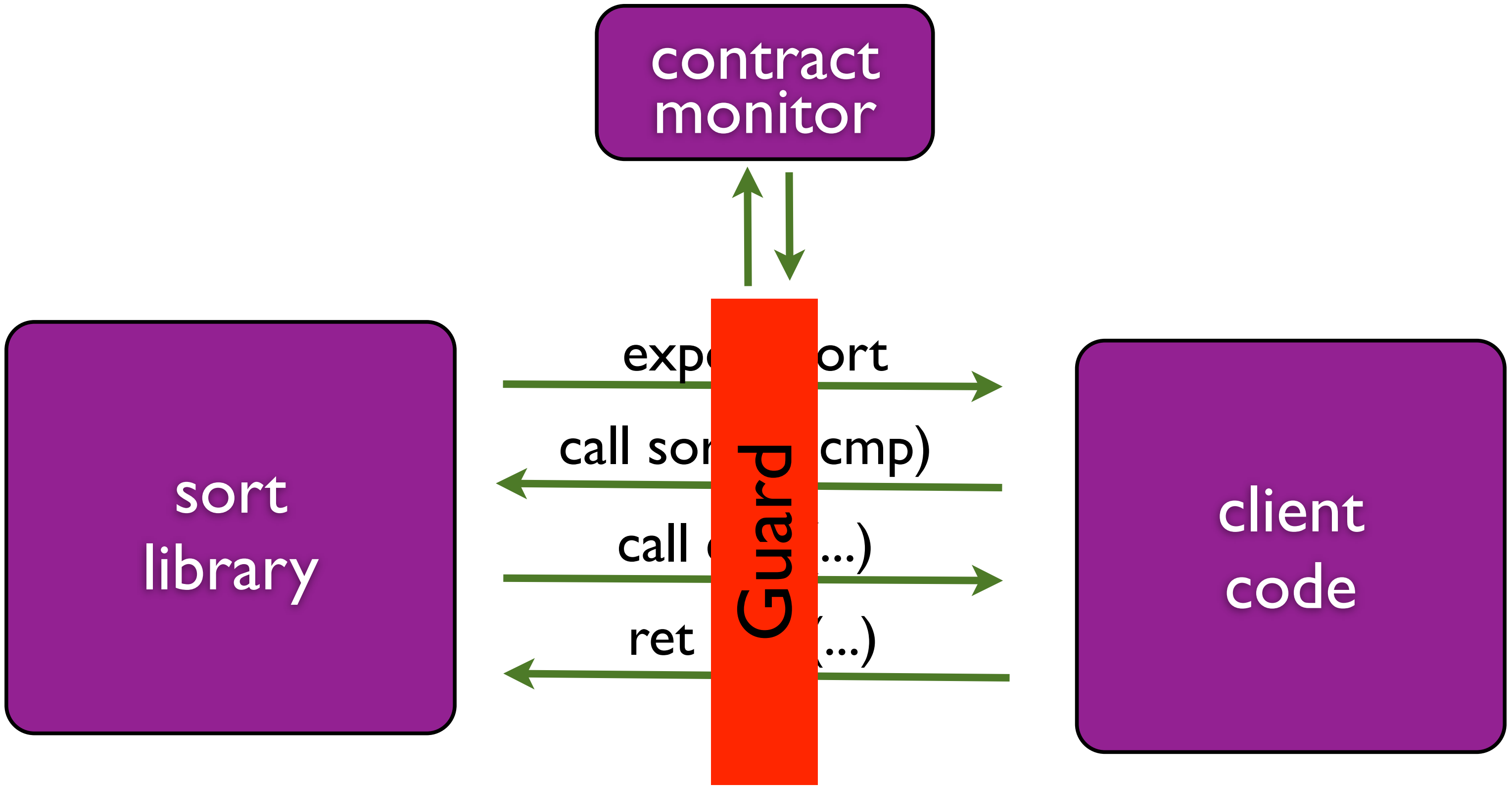
Trace completeness (without non-interference)



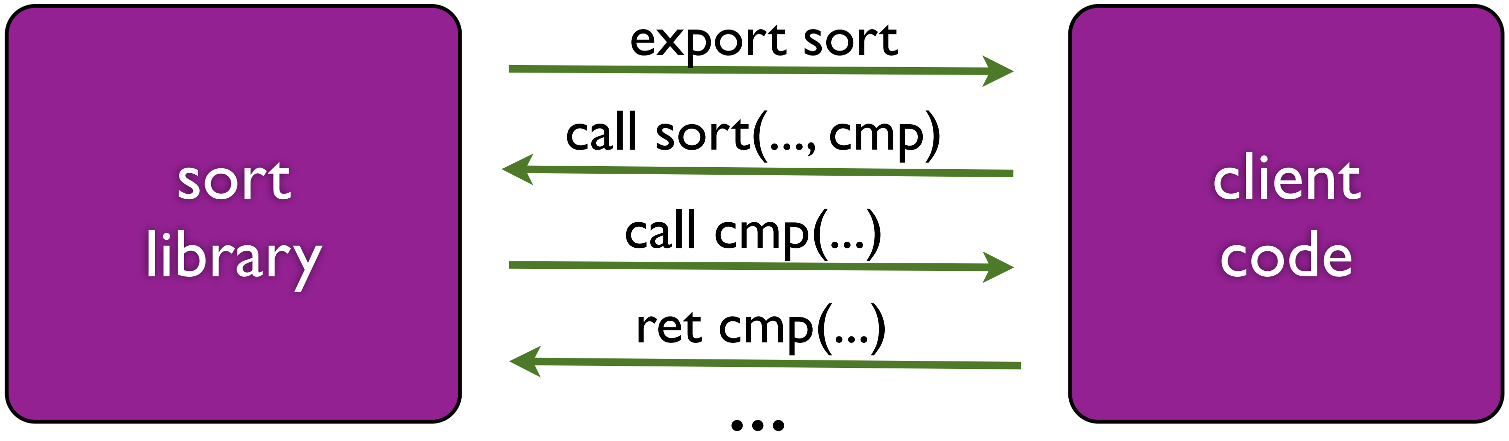
Trace completeness (without non-interference)



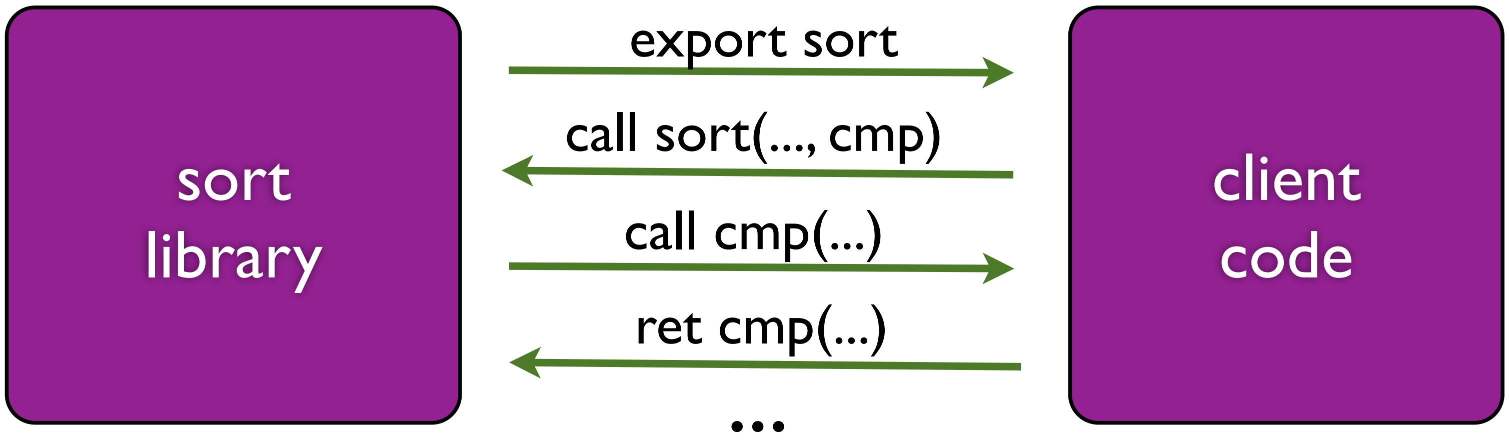
Trace completeness with non-interference



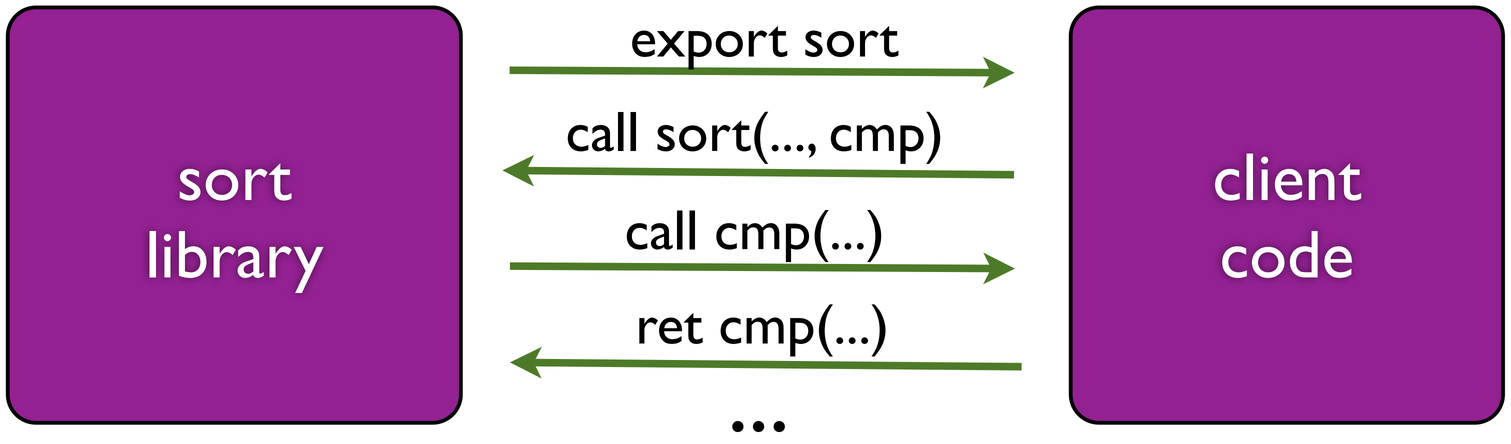
- Guard enforces non-interference without losing trace completeness
- Monitor code never sees function or refs, so cannot influence behavior except via errors



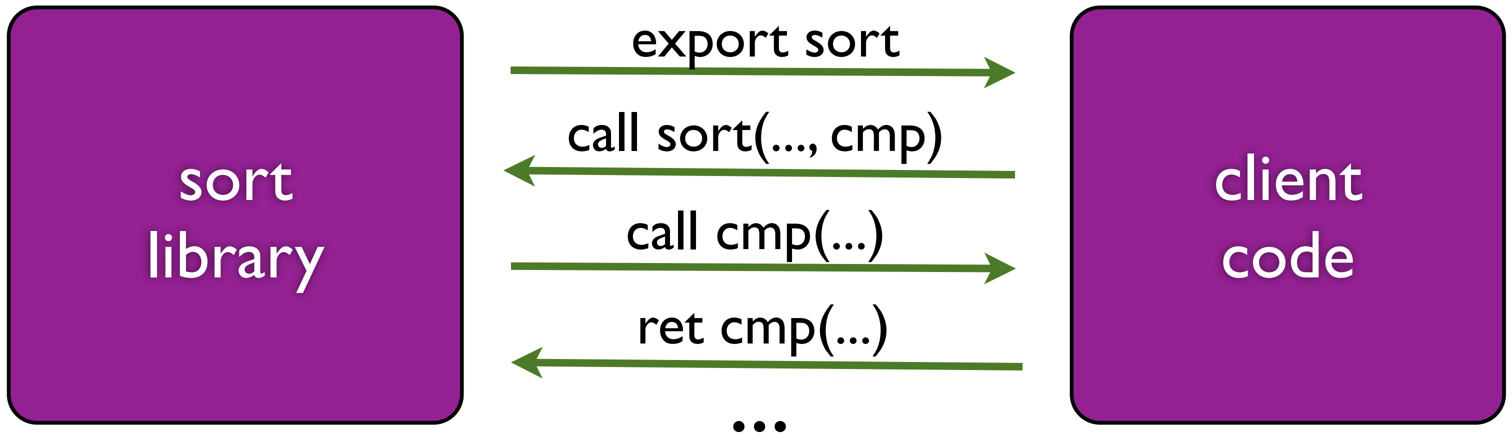
```
SortContract =
  sort : (List Int)
         (cmp : Int → Int → Bool)
       → (List Int)

  // sort is not re-entrant
  where not ... call-sort(_) !ret-sort(_)* call-sort(_)
```



```
SortContract =
  sort : (List Int)
         (cmp : Int → Int → Bool)
       → (List Int)

  // sort is not re-entrant
  where not ... call-sort(_) !ret-sort(_)* call-sort(_)
  // cmp is atomic
    and not ... call-cmp(_) !ret-cmp(_)
```

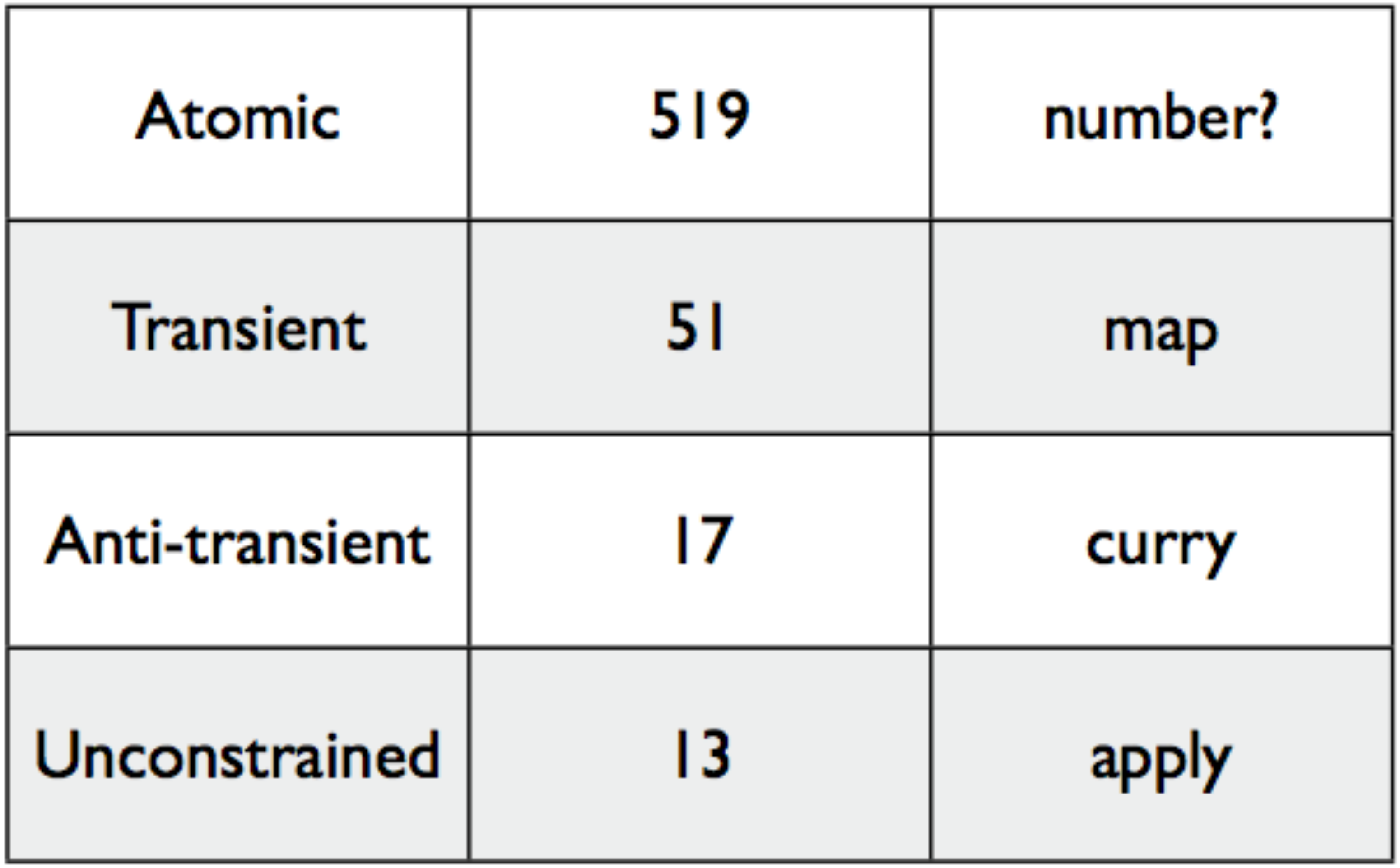


```
SortContract =
  sort : (List Int)
         (cmp : Int → Int → Bool)
       → (List Int)

  // sort is not re-entrant
  where not ... call-sort(_) !ret-sort(_)* call-sort(_)
  // cmp is atomic
    and not ... call-cmp(_) !ret-cmp(_)
  // cmp is consistent
    and not ... call-cmp(?x,?y)  ret-cmp(?r)
            ... call-cmp( x, y) !ret-cmp( r)
```



```
SortContract =
  sort : (List Int)
         (cmp : Int → Int → Bool
            // cmp is consistent
            where not ... call-cmp(?x,?y)  ret-cmp(?r)
                      ... call-cmp( x, y) !ret-cmp( r) )
       → (List Int)

  // sort is not re-entrant
  where not ... call-sort(_) !ret-sort(_)* call-sort(_)
  // cmp is atomic
    and not ... call-cmp(_) !ret-cmp(_)
```

Temporal properties in the Racket Standard Library



Atomic	519	number?
Transient	51	map
Anti-transient	17	curry
Unconstrained	13	apply

Temporal Contracts for Security

- Implementation of multi-player Tic-Tac-Toe
- Each player implements `turn : Board -> Board`
- Both interactive and AI player implementations
- Players may try to cheat!
  - update board multiple times during a turn
  - overwrite previous contents on the board
- Restricted using temporal contracts
- Caught cheaters, both human and AI