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
• **Non-interference**: Contracts cannot influence correct executions

• **Trace completeness**: 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

Sort library

- `return sort`
- `call sort(lst, cmp)`
- `call cmp(...)`
- `ret cmp(...)`
- `...`

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
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)

- sort library
- Arbitrary code
- client code

- export sort
- call sort(..., cmp)
- call cmp(...)
- return cmp(...)
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(_)

export sort

client library

library

call sort(..., cmp)

client code

call cmp(...)

... call-cmp( x, y) !ret-cmp( r)

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(_)

// export sort
export sort

// call sort
call sort(..., cmp)

// call cmp
call cmp(...)

// return 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

<table>
<thead>
<tr>
<th>Atomic</th>
<th>519</th>
<th>number?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transient</td>
<td>51</td>
<td>map</td>
</tr>
<tr>
<td>Anti-transient</td>
<td>17</td>
<td>curry</td>
</tr>
<tr>
<td>Unconstrained</td>
<td>13</td>
<td>apply</td>
</tr>
</tbody>
</table>
Temporal Contracts for Security

- Implementation of multi-player Tic-Tac-Toe
- Each player implements $\text{turn : Board} \rightarrow \text{Board}$
- 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