idioms

Daniel Jackson
MIT Lab for Computer Science
6898: Advanced Topics in Software Design
February 13, 2002

issues

state change

- how do you describe change in a logic?

mutation

- how do you modularize the change?

frame conditions

- how do you control unwanted changes?

"the relational model"

ideas

- view behaviour as a relation on states
- represent as a simple formula
- distinct variables for pre- and post-state

example

- a procedure that returns the index of an element in an array

```
fun Find (t: Index ->? Elt, e: Elt): Index {
    t[result] = e
    }
```

this procedure is

- partial: some pre-states have no post-states
- non-deterministic: some pre-states have >1 post-state

preconditions

what does partiality mean?

› guard

  a control: won't allow invocation in bad state

› precondition

  a disclaimer: havoc if invoked in bad state

the VDM/Larch approach

make precondition explicit

› op Find (t: Index ->? Elt, e: Elt): Index

    pre   e in Index.(t.map)

    post   t[result] = e

do 'implementability check'

› all s: State | pre(s) => some s': State | post(s,s')

the Z approach

precondition is implicit

- Find = [t: Index ->? Elt, e? : Elt, i ! : Index | t[i !] = e?]

compute precondition

- Pre (Find) = some s’: State | Find (s,s’)

and check properties

- Pre (Find) = true ??

the Alloy approach

roll your own!

- can split pre/post into distinct functions
- can combine in different ways
- can interpret as guard or precondition

```
fun FindPre (t: Index ->? Elt, e: Elt) { e in Index.t }

fun FindPost (t: Index ->? Elt, e: Elt): Index { t[result] = e }

fun Find (t: Index ->? Elt, e: Elt): Index {
   FindPre (t, e) => FindPost (t, result, e)
   }
```

frame conditions

procedure that removes dupls

```
› fun RemoveDupls (t, t’: Index ->? Elt) {
    Index.t = Index.t’
    #Index.t’ = #t’
    }
```

procedure that sorts array without dupls

```
› fun Sort (t, t’: Index ->? Elt) {
    Index.t = Index.t’
    all i, j: t.Elt | i->j in Index$lt => t[i]->t[j] in Elt$lt
    }
```

are these constrained enough?

summary so far

declarative style is powerful

- spec by conjunction
- separation of concerns

declarative style is dangerous

- implicit preconditions
- underconstraint

frame conditions

- subtle for non-deterministic operations
- subtle in OO programs
- important stylistic issue, even when not subtle

modularizing change

3 approaches

- global state machine
- local state machines
- object oriented

a static model of router tables

```
module routing
sig IP {}
sig Link {from, to: Router}
sig Router {ip: IP, table: IP ->? Link, nexts: set Router}
   {
   table[IP].from in this
   nexts = table[IP].to
   no table[ip]
   }
fact {inj (Router$ip)}
fun Consistent () {
   all r: Router, i: IP | r.table[i].to in i.~ip.*~nexts
   }
fun inj [t,t'] (r: t->t') {all x: t' | sole r.x}
```

a dynamic model

```
sig State {}
sig IP {}
sig Link {from, to: State ->! Router}
sig Router {ip: IP, table: State -> IP ->? Link,
   nexts: State -> Router}
   {
   all s: State {
      (table[s][IP].from) [s] in this
      nexts[s] = (table[s][IP].to)[s]
      no table[s] [ip]
      }}
fun Consistent (s: State) {
   let rnexts = {r,r': Router | r->s->r' in Router$nexts} |
      all r: Router, i: IP |  (r.table[s][i].to) [s] in i.~ip.*~rnexts}
```

propagation

```
fun Propagate (s, s': State) {
   let rnexts = {r,r': Router | r->s->r' in Router$nexts} |
     all r: Router | r.table[s'] in r.table[s] + r.~rnexts.table[s] }

assert PropagationOK {
   all s, s': State |
     Consistent (s) && Propagate (s,s') => Consistent (s') }
```

can you write NoTopologyChange?

```
fun NoTopologyChange (s,s': State) {
   all x: Link {
     x.from[s] = x.from[s'] && x.to[s] = x.to[s']
     }
   }
```

global state version

```
sig IP {} sig Link {} sig Router {}

sig LinkState {
   from, to: Link -> Router }

sig NetworkState extends LinkState {
   ip: Router -> IP,
   table: Router -> IP ->? Link,
   nexts: Router -> Router } {
     all r: Router {
        table[r][IP].from in r
        nexts[r] = table[r][IP].to
        no table[r][r.ip] }
     inj (ip) }
```

operations for global version

```
fun Consistent (s: NetworkState) {
  all r: Router, i: IP |
    s.table[r][i].to[s] in i.~(ip[s]).*~(nexts[s]) }

fun Propagate (s, s': NetworkState) {
  all r: Router {
    s'.table[r] in s.table[r] + r.~(nexts[s]).(table[s])
    s.ip[r] = s'.ip[r] } }
```

can you write NoTopologyChange?

```
fun NoTopologyChange (s,s': NetworkState) {
  s.from = s'.from
  s.to = s'.to
  }
```

object oriented version

```
sig IP {}
sig Link {from, to: RouterRef}
sig Router {ip: IP, table: IP ->? LinkRef, nexts: set RouterRef} {
   no table[ip] }
fact {inj (Router$ip)}
sig LinkRef {}
sig RouterRef {}

sig State {
   robj: RouterRef ->! Router,
   lobj: LinkRef ->! Link
   }
```

invariants

router invariant is now recognized as ‘cross object’

```
fact {
   all s: State, r: RouterRef {
      r.(s.robj).table[IP].(s.lobj).from in r
      r.(s.robj).nexts = r.(s.robj).table[IP].(s.lobj).to
      }
   }
```

operations in OO version

```
fun Consistent (s: State) {
   all r: RouterRef, i: IP |
      r.(s.robj).table[i].(s.lobj).to in
         i.~ip.~(s.robj).*~(s.robj.nexts) }

fun Propagate (s, s': State) {
   let rnexts = {r,r': Router | r->s->r' in Router$nexts} |
   all r: RouterRef |
      r.(s'.robj).table in r.(s.robj).table
         + r.~(s.robj.nexts).(s.robj).table }
```

can you write NoTopologyChange?

```
fun NoTopologyChange (s,s': State) {
   s.lobj = s'.lobj }
```

comparison

how do the approaches compare?

- ease of expression
- degree of modularity
- how systematic
- frame conditions

didn't get round to discussing this