1 Elimination Stack

For simplicity, we assume a value $\perp$, which is different from NULL and any other value the stack can store. Since NULL is used as a legit return value, representing the value of POP operation (when exchanging values using the elimination array), NULL can not be used to represent an initialization value, different then any stack value. The same holds for a Node, since a NULL node represent an empty stack, the value $\perp$ is used to distinguish between initialization value and empty stack.

For simplicity, we split the RECOVER routine into sub-routines, based on which operation (PUSH, POP, EXCHANGE) is pending, or needs to be recover. This can be concluded easily by the type of record stored in *Announce*[*pid*] (ExInfo or OpInfo), thus there is no need to explicitly know where exactly in the code the crash took place. Also, the RECOVER routine returns FAIL in case the last pending operation did not took affect (no linearization point), nor it will take in any future run. In such case, the user has the option to either re-invoke the operation, or to skip it, depends on the needs and circumstances of the specific use of the data structure.

The given implementation ignores the log of failures and successes of the exchange routine when recovering. That is, in case of a crash during an EXCHANGE, a process is able to recover the EXCHANGE routine, however, the log of successes and failures is not update, since it might be the process already updated it. In addition, in case of a FAIL response, we do not know whether the time limit (timeout) was reached, or that the process simply crashed earlier in the routine without completing it. The given implementation can be expanded to also consider the log. Nonetheless, for ease of presentation we do not handle the log in case of a crash. Assuming crash events are rare, the log still gives a roughly good approximation to the number of failures and successes, thus our approach might be useful in practice.

1.1 A Lock-Free Exchanger

An exchanger object supports the EXCHANGE procedure, which allows

```
Type Node {
    T value
    int popby
    Node *next
}

Type PushInfo {                    ▷ subtype of Info
    Node *pushNd
}

Type PopInfo {                     ▷ subtype of Info
    Node *popNd
}

Type ExInfo {                      ▷ subtype of Info
    {EMPTY, WAITING, BUSY} state
    T value, result
    ExInfo *partner, *slot
}
```

Figure 1: Type definition

ExInfo $default$ - a global static ExInfo object with state = EMPTY

Algorithm 1: T EXCHANGE (ExInfo $*slot$, T $myitem$, long $timeout$)

```
1  long timeBound := getNanos() + timeout
2  ExInfo myop := new ExInfo(WAITING, myitem, ⊥, ⊥, slot)
3  Announce[pid] := myop
4  while true do
5      if getNanos() > timeBound then
6          myop.result := TIMEOUT                                  // time limit reached
7          return TIMEOUT
8      yourop := slot
9      switch yourop.state do
10         case EMPTY
11             myop.state := WAITING                               // try to replace default
12             myop.partner := ⊥
13             if slot.CAS(yourop, myop) then
14                 while getNanos() < timeBound do
15                     yourop := slot
16                     if yourop ≠ myop then                       // a collision was done
17                         if youop.parnter = myop then            // yourop collide with myop
18                             SWITCHPAIR(myop, yourop)
19                             slot.CAS(yourop, default)           // release slot
20                         return myop.result
21                 end
                   // time limit reached and no process collide with me
22                 if slot.CAS(myop, default) then                 // try to release slot
23                     myop.result := TIMEOUT
24                     return TIMEOUT
25                 else                                            // some process show up
26                     yourop := slot
27                     if yourop.partner = myop then
28                         SWITCHPAIR(myop, yourop)                // complete the collision
29                         slot.CAS(yourop, default)               // release slot
30                     return myop.result
31             end
32             break
33         case WAITING                                            // some process is waiting in slot
34             myop.partner := yourop                              // attempt to replace yourop
35             myop.state := BUSY
36             if slot.CAS(yourop, myop) then                      // try to collide
37                 SWITCHPAIR(myop, yourop)                        // complete the collision
38                 slot.CAS(myop, default)                         // release slot
39                 return myop.result
40             break
41         case BUSY                                               // a collision in progress
42             SWITCHPAIR(yourop, yourop.parnter)                  // help to complete the collision
43             slot.CAS(yourop, default)                           // release slot
44             break
45     endsw
46 end
```

Algorithm 2: void SwitchPair(ExInfo *first*, ExInfo *second*)

```
   // exchange the valus of the two operations
47 first.result := second.value
48 second.result := first.value
```

Algorithm 3: T Visit (T *value*, int *range*, long *duration*)

```
   // invoke Exchange on a random selected entery in the collision array
49 int cell := randomNumber(range)
50 return Exchange(exchanger[cell], value, duration)
```

Algorithm 4: T Exchange-Recover ()

```
51 ExInfo *myop := Announce[pid]                          // read your last operation record,
52 ExInfo *slot := myop.slot                               // and the slot on which it act
53 if myop.state = Waiting then
       // crash while trying to exchange defualt, or waiting for a process to collide
          with me
54     yourop := slot
55     if yourop = myop then                               // still waiting for a collide
56         if slot.CAS(myop, default) then                          // try to release slot
57             return Fail
58         else                                                    // some process show up
59             yourop := slot
60             if yourop.partner = myop then
61                 SwitchPair(myop, yourop)                  // complete the collision
62                 slot.CAS(yourop, default)                                // release slot
63             return myop.result
64     if yourop.partner = myop then                       // yourop collide with myop
65         SwitchPair(myop, yourop)                          // complete the collision
66         slot.CAS(yourop, default)                                    // release slot
67         return myop.result
68 if myop.state = Busy then
       // crash while trying to collide with myop.partner
69     yourop := slot
70     if yourop = myop then                                 // collide was successful
71         SwitchPair(myop, myop.partner)                    // complete the collision
72         slot.CAS(myop, default)                                      // release slot
73         return myop.result
74 if myop.result ≠ ⊥ then
75     return myop.result                              // collide was successfuly completed
76 else
77     return Fail
```

Figure 2: Elimination Array routines

Algorithm 5: boolean TryPush (Node $*new$)

```
78 Node *oldTop := Top
79 new.next := oldTop
80 if Top.CAS(oldTop, new) then
81 |   nd.popby.CAS(⊥, Null)
82 |   return true
83 return false
```

Algorithm 6: boolean Push (T *myitem*)

```
84 Node *nd = new Node (myitem)
85 nd.popby := ⊥
86 PushInfo data := new PushInfo (nd)
87 while true do
88 |   Announce[pid] := data
89 |   if TryPush(nd) then
90 |   |   return true
91 |   range := CalculateRange()
92 |   duration := CalculateDuration()
93 |   othervalue := Visit(myitem, range, duration)
94 |   if othervalue = Null then
95 |   |   RecordSuccess ()
96 |   |   return true
97 |   else if othervalue = Timeout then
98 |   |   RecordFailure ()
99 end
```

Algorithm 7: boolean Push-Roceover ()

```
100 Node *nd := Announce[pid].pushNd
101 if nd.popby ≠ ⊥ then
102 |   return true
103 if Search(nd) || nd.popby ≠ ⊥ then
104 |   nd.popby.CAS(⊥, Null)
105 |   return true
106 return Fail
```

Algorithm 8: boolean Search (Node $*nd$)

```
107 Node *iter := Top
108 while iter ≠ ⊥ do
109 |   if iter = nd then
110 |   |   return true
111 |   iter := iter.next
112 end
113 return false
```

Figure 3: Push routine

Algorithm 9: T TryPop()

```
114 Node *oldTop := Top
115 Node *newTop
116 Announce[pid].popNd := oldTop
117 if oldTop = ⊥ then
118 |   return Empty
119 newTop := oldTop.next
120 oldTop.popby.CAS(⊥, Null)
121 if Top.CAS(oldTop, newTop) then
122 |   if newTop.popby.CAS(Null, pid) then
123 |   |   return oldTop
124 else
125 |   return ⊥
```

Algorithm 10: T Pop ()

```
126 Node *result
127 PopInfo data := new PopInfo (Top)
128 while true do
129 |   Announce[pid] := data
130 |   result := TryPop()
131 |   if result = Empty then
132 |   |   return Empty
133 |   else if result ≠ ⊥ then
134 |   |   return result.value
135 |   range := CalculateRange()
136 |   duration := CalculateDuration()
137 |   othervalue := Visit(Null, range, duration)
138 |   if othervalue = Timeout then
139 |   |   RecordFailure ()
140 |   else if othervalue ≠ Null then
141 |   |   RecordSuccess ()
142 |   |   return othervalue
143 end
```

Algorithm 11: T Pop-Recover()

```
144 Node *nd := Announce[pid].popNd
145 if nd = ⊥ then
146 |   return Empty
147 if Search(nd) then
148 |   return Fail
149 nd.popby.CAS(Null, pid)
150 if nd.popby = pid then
151 |   return nd.value
152 return Fail
```

Figure 4: Pop routine