1 General Info

$\text{array}(a, [x_1, x_2, \ldots x_n]) \triangleq a \mapsto x_1 * (a+4) a \mapsto x_2 * \ldots * (a+4n) \mapsto x_n$

- For stacks:

$$a \rightsquigarrow nil \triangleq a \mapsto \text{NULL}$$
$$a \rightsquigarrow [x_1 \ldots, x_n] \triangleq a \mapsto (x_1, a_1) * \ldots a_{n-1} \mapsto (x_n, a_n) * a_n \mapsto \text{NULL}$$

- For queues:

$$(hd, tl) \rightsquigarrow nil \triangleq hd \mapsto \text{NULL}\ tl \mapsto \text{NULL}$$
$$(hd, tl) \rightsquigarrow [x_n \ldots, x_1] \triangleq hd \mapsto (x_1, a_1) * \ldots a_{n-1} \mapsto tl = (\text{NULL}\ , x_n)$$

$l \bigcirc\!\!\rightarrow R \triangleq l \underset{0}{\square\!\!\rightarrow} R$ and R is constant (i.e. doesn't depend on its variable)

1.1 Join Spawn rules

$$\frac{\{P\} f \{Q\} \quad l \text{ fresh in } F}{\{F * P\} \text{Spawn f } \{F * l \bigcirc\!\!\rightarrow Q\}}\ \text{spwn}$$

$$\frac{}{\{l \bigcirc\!\!\rightarrow Q\} \text{Join(l) } \{Q\}}\ \text{join}$$

1.2 Histories

$$Hist \triangleq \mathbb{N} \rightharpoonup \text{list}\ * \text{list}$$
$$t \hookrightarrow_h (l_1, l_2) \triangleq h[t] = (l_1, l_2)$$

- bounded $h \triangleq \exists t. \forall t' > t, t' \notin h$
- last $h \triangleq \min\{t | \forall t' > t, t' \notin h\}$
- listof $h \triangleq \pi_2(h[\text{last } h])$ (i.e. $(\text{last } h) \hookrightarrow (\_, \text{listof } h)$)
- continuous $h \triangleq \forall t. t \in h \wedge (t+1) \in h \rightarrow \exists l. t \hookrightarrow (\_, l) \wedge (t+1) \hookrightarrow (l, \_)$
- gapless $h \triangleq \forall t \in h \rightarrow \forall t' < t, t' \in h$
- init $h \triangleq 0 \hookrightarrow (\epsilon, \_)$
- stacklike $h \triangleq \forall t \in h \rightarrow \exists l, x, t \hookrightarrow (x :: l, l) \vee t \hookrightarrow (l, x :: l)$
- queuelike $h \triangleq \forall t \in h \rightarrow \exists l, x, t \hookrightarrow (l, x :: l) \vee t \hookrightarrow (l :: x, l)$
- stack_history $h \triangleq$ continuous $h \wedge$ gapless $h \wedge$ bounded $v \wedge$ init $h \wedge$ stacklike h
- queue_history $h \triangleq$ continuous $h \wedge$ gapless $h \wedge$ bounded $v \wedge$ init $h \wedge$ queuelike h

2 Multiple-thread counter.

```
int main() {
  { emp}
  a = malloc (n);
  { array(a, [_, _, ..., _]_n}
  (l, c) = malloc (LOCK_SIZE);
  { l ↦ _ * c ↦ _ * array(a, [_, _, ..., _]_n}
  c = 0;
  { l ↦ _ * c ↦ 0 * array(a, [_, _, ..., _]_n}
  MakeLock(l);
  { l□→^1_0 R * c ↦ 0 * Hold l, R, 0 * array(a, [_, _, ..., _]_n}   //R = λv.c ↦ v
  Release(l);
  { l□→^1_0 R * array(a, [_, _, ..., _]_n}
  { ★_{j=0}^{n-1} l□→^{1/n}_0 R * array(a, [_, _, ..., _]_n}
  for (i = 0; i < n; i ++) {
  { ★_{j=0}^{i} l_j○→ R_j * ★_{j=i+1}^{n-1} l□→^{1/n}_0 R * array(a, [l_1, ..., l_i, _, ...]_n}
    a[i] = Spawn(incr, (l,c));
  { ★_{j=0}^{i+1} l_j○→ R_j * ★_{j=i+2}^{n-1} l□→^{1/n}_0 R * array(a, [l_1, ..., l_i, l_{i+1}, _, ...]_n}
  }
  { ★_{j=0}^{n} l_j○→ R_j * array(a, [l_1, ..., l_n]}
  for (i = 0; i < n; i ++)
  {
  { ★_{j=0}^{i} R_j * ★_{j=i}^{n} l_j○→ R_j * array(a, [l_1, ..., l_n]}
    Join(a[i]);
  { ★_{j=0}^{i+1} R_j * ★_{j=i+1}^{n} l_j○→ R_j * array(a, [l_1, ..., l_n]}
  }
  { ★_{j=0}^{n} R_j * array(a, [l_1, ..., l_n]}   //R_j = l□→^{1/n}_1 R
  { l□→_n R * array(a, [l_1, ..., l_n]}
  free(a);
  { l□→_n R}
  Acquire(l);
  { l□→_n R * ∃v_o, c ↦ (n + v_o) * Hold l, R, (n + v_o)}
  { l□→_n R * c ↦ n * Hold l, R, n}
  ret = c;
  { ret ↦ n * l□→_n R * c ↦ n * Hold l, R, n}
  FreeLock (l);
  { ret ↦ n * l ↦ 0 * c ↦ n}
```

```
  free(l,c);
  { ret ↦ n}
  return ret }


void incr(l,c) {
  { l□-(1/n)->(0) R }
  Acquire(l);
  { ∃v_o, c ↦ v_o * Hold l, R, v_o * l□-(1/n)->(0) R}
  (*c)++;
  { ∃v_o, c ↦ (v_o + 1) * Hold l, R, v_o * l□-(1/n)->(0) R}
  Release(l);
  { l□-(1/n)->(1) R}
}
```

3 Single Initialize / concurrent read

$\{\ l\underset{\bot}{\overset{\pi}{\square\!\!\rightarrow}} R\ \}$ \\ $R = \lambda v.init \mapsto 0 \wedge v = \bot \vee init \mapsto 1 * d \overset{\top - v}{\mapsto} \text{data} \wedge [\top > v]$

```
data *first_access(l) {
```

$\{\ l\underset{\bot}{\overset{\pi}{\square\!\!\rightarrow}} R\ \}$

```
  Acquire(l);
```

$\{\ \exists v_o, init \mapsto 0 \wedge v_o = \bot \vee init \mapsto 1 * d \overset{s_o}{\mapsto} \text{data} * \text{Hold}\ l, R, v_o * l\underset{\bot}{\overset{\pi}{\square\!\!\rightarrow}} R\ \}$

\\ where $s_o = \top - v_o$

```
  if(init) {
```

$\{\ init \mapsto 1 * d \overset{s_o}{\mapsto} \text{data} * \text{Hold}\ l, R, v_o * l\underset{\bot}{\overset{\pi}{\square\!\!\rightarrow}} R\ \}$

$\{\ d \overset{\frac{s_o}{2}}{\mapsto} \text{data} * \left(init \mapsto 1 * d \overset{\top - (v_o + \frac{s_o}{2})}{\mapsto} \text{data}\right) * \text{Hold}\ l, R, v_o * l\underset{\bot}{\overset{\pi}{\square\!\!\rightarrow}} R\ \}$

```
    Release(l);
```

$\{\ d \overset{\frac{s_o}{2}}{\mapsto} \text{data} * l\underset{\frac{s_o}{2}}{\overset{\pi}{\square\!\!\rightarrow}} R\ \}$

```
    return d;
```

$\{\ d \overset{\frac{s_o}{2}}{\mapsto} \text{data} * l\underset{\frac{s_o}{2}}{\overset{\pi}{\square\!\!\rightarrow}} R \wedge ret = d\ \}$

```
  }
  else {
```

$\{\ init \mapsto 0 * \text{Hold}\ l, R, \bot * l\underset{\bot}{\overset{\pi}{\square\!\!\rightarrow}} R\ \}$

```
    InitializeData (d);
```

$\{\ d \mapsto \text{data} * init \mapsto 0 * \text{Hold}\ l, R, \bot * l\underset{\bot}{\overset{\pi}{\square\!\!\rightarrow}} R\ \}$

```
    init = 1;
```

$\{\ d \mapsto \text{data} * init \mapsto 1 * \text{Hold}\ l, R, \bot * l\underset{\bot}{\overset{\pi}{\square\!\!\rightarrow}} R\ \}$

$\{\ d \overset{\frac{1}{2}}{\mapsto} \text{data} * \left(d \overset{\frac{1}{2}}{\mapsto} \text{data} * init \mapsto 1\right) * \text{Hold}\ l, R, \bot * l\underset{\bot}{\overset{\pi}{\square\!\!\rightarrow}} R\ \}$

```
    Release(l)
```

$\{\ d \overset{\frac{1}{2}}{\mapsto} \text{data} * l\underset{\bot}{\overset{\pi}{\square\!\!\rightarrow}} R\ \}$

```
    return d;
```

$\{\ d \overset{\frac{1}{2}}{\mapsto} \text{data} * l\underset{\bot}{\overset{\pi}{\square\!\!\rightarrow}} R \wedge ret = d\ \}$

```
  }
}
```

$\{\ \exists \pi_s, d \overset{\pi_s}{\mapsto} \text{data} * l\underset{\bot}{\overset{\pi}{\square\!\!\rightarrow}} R \wedge ret = d\ \}$

4 Stack Producer/consumer

$\{\ \text{emp}\ \}$
void create();
$\{\ \text{hd} \rightsquigarrow \epsilon\ \}$

$\{\ \text{hd} \rightsquigarrow \epsilon\ \}$
void delete();
$\{\ \text{emp}\ \}$

$\{\ \text{hd} \rightsquigarrow ls\ \}$
void isemp();
$\{\ \text{hd} \rightsquigarrow ls \wedge$
$\quad ls = \epsilon \wedge ret = \text{true} \vee$
$\quad \exists x, l'. l = x :: l \wedge ret = \text{false}\ \}$

$\{\ \text{hd} \rightsquigarrow ls\ \}$
void enq(**int** x);
$\{\ \text{hd} \rightsquigarrow x :: ls\ \}$

$\{\ \text{hd} \rightsquigarrow x :: ls\ \}$
void deq();
$\{\ \text{hd} \rightsquigarrow ls \wedge ret = x\ \}$

/ Producer */*
$\{\ l \square\xrightarrow[\bot]{\pi} R\ \}$ \\ $R = \lambda h.\ \text{hd} \rightsquigarrow (\text{listof}(h)) \wedge \text{history\_stack}\ h$
void produce(x, l){
$\quad \{\ l \square\xrightarrow[\bot]{\pi} R\ \}$
$\quad$ Acquire(l);
$\quad \{\ \exists h_o,\ \text{hd} \rightsquigarrow l \wedge \text{history\_stack}\ h * \text{Hold}\ l, R, h_o * l \square\xrightarrow[\bot]{\pi} R\ \}$ \\ $l = \text{listof}(h_o)$
$\quad$ enq(x);
$\quad \{\ \text{hd} \rightsquigarrow x :: l \wedge \text{history\_stack}\ h * \text{Hold}\ l, R, h_o * l \square\xrightarrow[\bot]{\pi} R\ \}$
$\quad \{\ (\ \text{hd} \rightsquigarrow (\text{listof}(h_o + t \hookrightarrow (l, x :: l))) \wedge \text{history\_stack}\ (h_o + t \hookrightarrow (l, x :: l)))$
$\qquad * \text{Hold}\ l, R, h_o * l \square\xrightarrow[\bot]{\pi} R\ \}$ \\ $t = \text{last}\ h_o + 1$
$\quad$ Release(l);
$\quad \{\ l \square\xrightarrow[t \hookrightarrow (l, x::l)]{\pi} R\ \}$
} $\{\ l \square\xrightarrow[t \hookrightarrow (l, x::l)]{\pi} R\ \}$

/ Consumer */*
$\{\ l \square\xrightarrow[\bot]{\pi} R\ \}$ \\ $R = \lambda h.\ \text{hd} \rightsquigarrow (\text{listof}(h)) \wedge \text{history\_stack}\ h$
void consume(l){
$\quad \{\ l \square\xrightarrow[\bot]{\pi} R\ \}$
$\quad$ bool cont = true;

$\{\ cont = true \wedge l\square\xrightarrow[\bot]{\pi} R\ \}$

while (cont) {

Acquire(l);

$\{\ cont = true \wedge \exists h_o,\ \mathrm{hd} \rightsquigarrow l \wedge \mathrm{history\_stack}\ h$
$* \mathrm{Hold}\ l, R, h_o * l\square\xrightarrow[\bot]{\pi} R\ \}\quad \backslash\backslash l = \mathrm{listof}(h_o)$

if (isemp()) {

Release(l);

$\{\ cont = true \wedge l\square\xrightarrow[\bot]{\pi} R\ \}$

} **else** {

$\{\ \exists x, l'.l = x :: l \wedge cont = true \wedge$
$\mathrm{hd} \rightsquigarrow l \wedge \mathrm{history\_stack}\ h * \mathrm{Hold}\ l, R, h_o * l\square\xrightarrow[\bot]{\pi} R\ \}$

ret = deq();

$\{\ ret = x \wedge cont = true \wedge$
$\mathrm{hd} \rightsquigarrow l' \wedge \mathrm{history\_stack}\ h * \mathrm{Hold}\ l, R, h_o * l\square\xrightarrow[\bot]{\pi} R\ \}$

$\{\ ret = x \wedge cont = true \wedge$
$(\ \mathrm{hd} \rightsquigarrow (\mathrm{listof}(h_o + t \hookrightarrow (l, l'))) \wedge \mathrm{history\_stack}\ h)$
$* \mathrm{Hold}\ l, R, h_o * l\square\xrightarrow[\bot]{\pi} R\ \}\qquad \backslash\backslash\quad t = \mathrm{last}\ h_o + 1$

Release(l);

$\{\ ret = x \wedge cont = true \wedge l\square\xrightarrow[t\hookrightarrow(l,l')]{\pi} R\ \}$

cont = false;

$\{\ ret = x \wedge cont = false \wedge l\square\xrightarrow[t\hookrightarrow(l,l')]{\pi} R\ \}$

}

$\{\ cont = true \wedge l\square\xrightarrow[\bot]{\pi} R \vee cont = false \wedge ret = x \wedge l\square\xrightarrow[t\hookrightarrow(l,l')]{\pi} R\ \}$

}

$\{\ cont = false \wedge ret = x \wedge l\square\xrightarrow[t\hookrightarrow(l,l')]{\pi} R\ \}$

return ret;

}

$\{\ ret = x \wedge l\square\xrightarrow[t\hookrightarrow(x::l',l')]{\pi} R\ \}$

```
/* Organizer */
{ l□ --π--> (⊥) R * a ↦ _ * b ↦ _ }   \\   R = λh. hd ⇝ (listof(h)) ∧ history_stack h
void organize1(l, a, b){
  { l□ --π--> (⊥) R * a ↦ _ * b ↦ _ }
  (t1, v1) = consume(l);
  { (t1, v1) = x ∧ l□ --π, t1↪(x::l,l)--> R * a ↦ _ * b ↦ _ }
  if (t1) {
    { t1 ↦ 1 * (t, v1) = x ∧ l□ --π, t↪(x::l,l)--> R * a ↦ _ * b ↦ _ }
    a = v1;
    { t1 ↦ 1 * (t, v1) = x ∧ l□ --π, t↪(x::l,l)--> R * a ↦ v1 * b ↦ _ }
  } else {
    { t1 ↦ 0 * (t, v1) = x ∧ l□ --π, t↪(x::l,l)--> R * a ↦ _ * b ↦ _ }
    b = v1;
    { t1 ↦ 0 * (t, v1) = x ∧ l□ --π, t↪(x::l,l)--> R * a ↦ _ * b ↦ v1 }
  }
}
{ (t1, v1) = x ∧ l□ --π, t↪(x::l,l)--> R *
    t1 ↦ 1 * a ↦ v1 * b ↦ _ ∨
    t1 ↦ 0 * a ↦ _ * b ↦ v1 }

{ l□ --π--> (⊥) R * a ↦ _ * b ↦ _ }
void organize2(l,a,b){
  { l□ --π--> (⊥) R * a ↦ _ * b ↦ _ }
  organize1(l,a,b);
  { (t1, v1) = x ∧ l□ --π, t↪(x::l,l)--> R *
    t1 ↦ 1 * a ↦ v1 * b ↦ _ ∨
    t1 ↦ 0 * a ↦ _ * b ↦ v1 }
  organize1(l,a,b);
}
{ (t2, v2) = x' ∧ (t1, v1) = x ∧ l□ --π, t↪(x::l,l)*t'↪(x'::l',l')--> R *
    t1 ↦ 1 * t2 ↦ 1 * a ↦ v2 * b ↦ _ ∨
    t1 ↦ 1 * t2 ↦ 0 * a ↦ v1 * b ↦ v2 ∨
    t1 ↦ 0 * t2 ↦ 1 * a ↦ v2 * b ↦ v1 ∨
    t1 ↦ 0 * t2 ↦ 0 * a ↦ _ * b ↦ _ }
```

```
void main(){
  { emp }
  l, x, y,z,a,b, = malloc (LOCK, LOCK, LOCK, LOCK, int, int, );
  hd = create();
  { hd ⇝ ϵ ∗ l ↦ _ ∗ a ↦ _ ∗ b ↦ _ }
  MakeLock(l); \\ R = λh. hd ⇝ (listof(h)) ∧ history_stack h
  { hd ⇝ ϵ ∗ l□ ⟶[⊤, emp] R ∗ Hold l, R, emp ∗ a ↦ _ ∗ b ↦ _ }
  Release(l);
  { l□ ⟶[⊤, emp] R ∗ a ↦ _ ∗ b ↦ _ }
  x = Spawn(produce, ((0,0), l);
  { x○⟶ R_x ∗ l□ ⟶[2/3, emp] R ∗ a ↦ _ ∗ b ↦ _ }  \\ R_x = l□ ⟶[1/3, t↪(l,(0,0))::l] R
  y = Spawn(produce, ((1,1), l);
  { y○⟶ R_y ∗ x○⟶ R_x ∗ l□ ⟶[1/3, emp] R ∗ a ↦ _ ∗ b ↦ _ }  \\ R_y = l□ ⟶[1/3, t↪(l,(1,1)::l)] R
  z = Spawn(organize2, (l, a, b));
  { z○⟶ R_z ∗ y○⟶ R_y ∗ x○⟶ R_x }
  Join(x);
  { h_1 = t_x ↪ (l_x,(0,0) :: l_x) ∧ z○⟶ R_z ∗ y○⟶ R_y ∗ l□ ⟶[1/3, h_1] R }
  Join(y);
  { h_2 = t_x ↪ (l_x,(0,0) :: l_x) ∗ t_y ↪ (l_y,(1,1) :: l_y) ∧ z○⟶ R_z ∗ l□ ⟶[2/3, h_2] R }
  Join(z);
  { h_3 = h_2 ∗ t_z1 ↪ (x :: l_z1, l_z1) ∗ t_z2 ↪ (y :: l_z2, l_z2)) ∧ (k1, v1) = x ∧ (k2, v2) = y ∧
    l□ ⟶[⊤, h_3] R ∗
    k1 ↦ 1 ∗ k2 ↦ 1 ∗ a ↦ v2 ∗ b ↦ _ ∨
    k1 ↦ 1 ∗ k2 ↦ 0 ∗ a ↦ v1 ∗ b ↦ v2 ∨
    k1 ↦ 0 ∗ k2 ↦ 1 ∗ a ↦ v2 ∗ b ↦ v1 ∨
    k1 ↦ 0 ∗ k2 ↦ 0 ∗ a ↦ _ ∗ b ↦ _ }
  Acquire(l);
  { (k1, v1) = x ∧ (k2, v2) = y ∧
    l□ ⟶[⊤, h_3] R ∗ Hold l, R, h_o ∗ hd ⇝ (listof(h_3)) ∧ history_stack (h_3) ∗
    k1 ↦ 1 ∗ k2 ↦ 1 ∗ a ↦ v2 ∗ b ↦ _ ∨
    k1 ↦ 1 ∗ k2 ↦ 0 ∗ a ↦ v1 ∗ b ↦ v2 ∨
    k1 ↦ 0 ∗ k2 ↦ 1 ∗ a ↦ v2 ∗ b ↦ v1 ∨
    k1 ↦ 0 ∗ k2 ↦ 0 ∗ a ↦ _ ∗ b ↦ _ }   \\ Case analysis on h3
  { l□ ⟶[⊤, h_3] R ∗ Hold l, R, h_o ∗ hd ⇝ (ϵ) ∗
    (1,1) = x ∧ (0,0) = y ∧ k1 ↦ 1 ∗ k2 ↦ 0 ∗ a ↦ 1 ∗ b ↦ 0 ∨
    (0,0) = x ∧ (1,1) = y ∧ k1 ↦ 0 ∗ k2 ↦ 1 ∗ a ↦ 1 ∗ b ↦ 0 }
  Free(l); free(k1,k2);
  { hd ⇝ ϵ ∗ a ↦ 1 ∗ b ↦ 0 }
  delete();
}
{ a ↦ 1 ∗ b ↦ 0 }
```

5 Queue Producer/consumer

```
struct elem {
  struct elem *next;
  struct elem *data;
};

struct fifo {
  struct elem *hd;
  struct elem *tl;
};

{ emp }
fifo *create(){
  Q = malloc(sizeof(fifo));
  { Q.hd ↦ _ * Q.hd ↦ _ }
  hd, tl = NULL;
  { Q ⇝ ϵ }
  return Q;
}
{ Q ⇝ ϵ }


{ Q ⇝ ϵ }
void delete(Q){
  free(Q);
}
{ emp }

{ Q ⇝ ls }
void isemp(){
  return (Q.hd == NULL)
  { Q ⇝ ls ∧ ret = (Q.hd == NULL) }
}
{ Q ⇝ ls ∧
    ls = ϵ ∧ ret = true ∨
    ∃x, l'.l = l :: x ∧ ret = false }

{ Q ⇝ ls }
void enq(fifo Q, type x){
  { Q ⇝ ls }
  if (hd==NULL) {
    { Q ⇝ ϵ ∧ hd = NULL }
    Q→hd=(NULL, x);
    Q→tl=(NULL, x);
    { Q ⇝ x :: ϵ }
  }
    else {
    { Q ⇝ [x1, ..., xn] ∧ hd ≠ NULL }
```

```
      tl→next = (NULL, x);
      { Q.hd ↦ (x_1, a_1) * ... a_{n-1} ↦ tl = (a_n, x_n) * a_n ↦ (NULL , x) }
      Q→tl=(\NULL, x);
      { Q.hd ↦ (x_1, a_1) * ... a_{n-1} ↦ (a_n, x_n) * a_n ↦ tl = (NULL , x) }
      { Q ⇝ [x, x_n, ..., x_1] }
    }
  }
{ Q ⇝ x :: ls }

{ Q ⇝ ls :: x }
void deq(fifo Q){
  h=Q→hd→data;
  { h = x ∧ Q ⇝ ls :: x }
  n=Q→hd→next;
  { h = x ∧ n = a_1 ∧ Q.hd ↦ (x, a_1) * a_1 ↦ (x_2, a_2) * ... a_{n-1} ↦ tl = (NULL , x_n) }
  Q→head=n;
  { h = x ∧ n = a_1 ∧ Q.hd ↦ (x_2, a_2) * ... a_{n-1} ↦ tl = (NULL , x_n) }
  return h;
}{ Q ⇝ ls ∧ ret = x }

/* Producer */
{ l□ --π--> R }   \\   R = λh. Q ⇝ (listof(h)) ∧ history_queue h
void produce(fifo Q, type x, lock l){
  { l□ --π--> R }
  Acquire(l);
  { ∃h_o, Q ⇝ l ∧ history_queue h * Hold l, R, h_o * l□ --π--> R }   \\   l = listof(h_o)
  enq(x);
  { Q ⇝ x :: l ∧ history_queue h_o * Hold l, R, h_o * l□ --π--> R }
  { ( Q ⇝ (listof(h_o ⊕ t ↪ (l, x :: l))) ∧ history_queue (h_o ⊕ t ↪ (l, x :: l)))
    * Hold l, R, h_o * l□ --π--> R }   \\   t = last h_o + 1
  Release(l);
  { l□ --π, t↪(l,x::l)--> R }
} { l□ --π, t↪(l,x::l)--> R }

/* Consumer */
{ l□ --π--> R }   \\   R = λh. Q ⇝ (listof(h_o)) ∧ history_queue h
void consume(l){
  { l□ --π--> R }
  bool cont = true;
  { cont = true ∧ l□ --π--> R }
  while (cont) {
    Acquire(l);
```

$\{\ cont = true \wedge \exists h_o,\ \mathrm{Q} \rightsquigarrow l \wedge \mathrm{history\_queue}\ h_o$
$* \mathrm{Hold}\ l, R, h_o * l \square \xrightarrow[\epsilon]{\pi} R\ \}\ \ \backslash\backslash l = \mathrm{listof}(h_o)$
if (isemp()) {
Release(l);
$\{\ cont = true \wedge l \square \xrightarrow[\epsilon]{\pi} R\ \}$
} **else** {
$\{\ \exists x, l'.l = l :: x \wedge cont = true \wedge$
$\mathrm{Q} \rightsquigarrow l \wedge \mathrm{history\_queue}\ h_o * \mathrm{Hold}\ l, R, h_o * l \square \xrightarrow[\epsilon]{\pi} R\ \}$
ret = deq();
$\{\ ret = x \wedge cont = true \wedge$
$\mathrm{Q} \rightsquigarrow l' \wedge \mathrm{history\_queue}\ h_o * \mathrm{Hold}\ l, R, h_o * l \square \xrightarrow[\epsilon]{\pi} R\ \}$
$\{\ ret = x \wedge cont = true \wedge$
$(\ \mathrm{Q} \rightsquigarrow (\mathrm{listof}(h_o \oplus t \hookrightarrow (l, l'))) \wedge \mathrm{history\_queue}\ h_o)$
$* \mathrm{Hold}\ l, R, h_o * l \square \xrightarrow[\epsilon]{\pi} R\ \}\ \ \backslash\backslash\ \ t = \mathrm{last}\ h_o + 1$
Release(l);
$\{\ ret = x \wedge cont = true \wedge l \square \xrightarrow[t \hookrightarrow (l,l')]{\pi} R\ \}$
cont = false;
$\{\ ret = x \wedge cont = false \wedge l \square \xrightarrow[t \hookrightarrow (l,l')]{\pi} R\ \}$
}
$\{\ cont = true \wedge l \square \xrightarrow[\epsilon]{\pi} R \vee cont = false \wedge ret = x \wedge l \square \xrightarrow[t \hookrightarrow (l,l')]{\pi} R\ \}$
}
$\{\ cont = false \wedge ret = x \wedge l \square \xrightarrow[t \hookrightarrow (l,l')]{\pi} R\ \}$
return ret;
}
$\{\ ret = x \wedge l \square \xrightarrow[t \hookrightarrow (l' :: x, l')]{\pi} R\ \}$

```
/* Organizer */
{ l□ --π/ϵ--> R * a ↦ _ * b ↦ _ }   \\   R = λh. Q ⇝ (listof(h)) ∧ history_queue h_o
void organize1(l, a, b){
  { l□ --π/ϵ--> R * a ↦ _ * b ↦ _ }
  (t1, v1) = consume(l);
  { (t1, v1) = x ∧ l□ --π, t1↪(l::x,l)--> R * a ↦ _ * b ↦ _ }
  if (t1) {
    { t1 ↦ 1 * (t, v1) = x ∧ l□ --π, t↪(l::x,l)--> R * a ↦ _ * b ↦ _ }
    a = v1;
    { t1 ↦ 1 * (t, v1) = x ∧ l□ --π, t↪(l::x,l)--> R * a ↦ v1 * b ↦ _ }
  } else {
    { t1 ↦ 0 * (t, v1) = x ∧ l□ --π, t↪(l::x,l)--> R * a ↦ _ * b ↦ _ }
    b = v1;
    { t1 ↦ 0 * (t, v1) = x ∧ l□ --π, t↪(l::x,l)--> R * a ↦ _ * b ↦ v1 }
  }
}
{ (t1, v1) = x ∧ l□ --π, t↪(l::x,l)--> R *
    t1 ↦ 1 * a ↦ v1 * b ↦ _ ∨
    t1 ↦ 0 * a ↦ _ * b ↦ v1 }

{ l□ --π/ϵ--> R * a ↦ _ * b ↦ _ }
void organize2(l,a,b){
  { l□ --π/ϵ--> R * a ↦ _ * b ↦ _ }
  organize1(l,a,b);
  { (t1, v1) = x ∧ l□ --π, t↪(l::x,l)--> R *
    t1 ↦ 1 * a ↦ v1 * b ↦ _ ∨
    t1 ↦ 0 * a ↦ _ * b ↦ v1 }
  organize1(l,a,b);
}
{ R_z = (t2, v2) = x' ∧ (t1, v1) = x ∧ l□ --π, t↪(l::x,l)⊕t'↪(l'::x',l')--> R *
    t1 ↦ 1 * t2 ↦ 1 * a ↦ v2 * b ↦ _ ∨
    t1 ↦ 1 * t2 ↦ 0 * a ↦ v1 * b ↦ v2 ∨
    t1 ↦ 0 * t2 ↦ 1 * a ↦ v2 * b ↦ v1 ∨
    t1 ↦ 0 * t2 ↦ 0 * a ↦ _ * b ↦ _ }
```

void main(){

{ emp }

l, x, y,z,a,b, = malloc (LOCK, LOCK, LOCK, LOCK, **int**, **int**,);

hd = create();

{ $\mathrm{Q} \rightsquigarrow \epsilon * l \mapsto \_ * a \mapsto \_ * b \mapsto \_$ }

MakeLock(l); \\ $R = \lambda h.\ \mathrm{Q} \rightsquigarrow (\mathrm{listof}(h)) \wedge \mathrm{history\_queue}\ h$

{ $\mathrm{Q} \rightsquigarrow \epsilon * l \square \xrightarrow[emp]{\top} R * \mathrm{Hold}\ l, R, emp * a \mapsto \_ * b \mapsto \_$ }

Release(l);

{ $l \square \xrightarrow[emp]{\top} R * a \mapsto \_ * b \mapsto \_$ }

x = Spawn(produce, ((0,0), l);

{ $x \bigcirc\!\!\rightarrow R_x * l \square \xrightarrow[emp]{\frac{2}{3}} R * a \mapsto \_ * b \mapsto \_$ } \\ $R_x = l \square \xrightarrow[t \hookrightarrow (l,(0,0))::l)]{\frac{1}{3}} R$

y = Spawn(produce, ((1,1), l);

{ $y \bigcirc\!\!\rightarrow R_y * x \bigcirc\!\!\rightarrow R_x * l \square \xrightarrow[emp]{\frac{1}{3}} R * a \mapsto \_ * b \mapsto \_$ } \\ $R_y = l \square \xrightarrow[t \hookrightarrow (l,(1,1)::l)]{\frac{1}{3}} R$

z = Spawn(organize2, (l, a, b));

{ $z \bigcirc\!\!\rightarrow R_z * y \bigcirc\!\!\rightarrow R_y * x \bigcirc\!\!\rightarrow R_x$ }

Join(x);

{ $h_1 = t_x \hookrightarrow (l_x, (0,0) :: l_x) \wedge z \bigcirc\!\!\rightarrow R_z * y \bigcirc\!\!\rightarrow R_y * l \square \xrightarrow[h_1]{\frac{1}{3}} R$ }

Join(y);

{ $h_2 = t_x \hookrightarrow (l_x, (0,0) :: l_x) \oplus t_y \hookrightarrow (l_y, (1,1) :: l_y) \wedge z \bigcirc\!\!\rightarrow R_z * l \square \xrightarrow[h_2]{\frac{2}{3}} R$ }

Join(z);

{ $h_3 = h_2 \oplus t_z 1 \hookrightarrow (l_z 1 :: x, l_z 1) \oplus t_z 2 \hookrightarrow (l_z 2 :: x', l_z 2)) \wedge (k1, v1) = x \wedge (k2, v2) = x' \wedge$

$l \square \xrightarrow[h_3]{\top} R *$

$k1 \mapsto 1 * k2 \mapsto 1 * a \mapsto v2 * b \mapsto \_ \vee$

$k1 \mapsto 1 * k2 \mapsto 0 * a \mapsto v1 * b \mapsto v2 \vee$

$k1 \mapsto 0 * k2 \mapsto 1 * a \mapsto v2 * b \mapsto v1 \vee$

$k1 \mapsto 0 * k2 \mapsto 0 * a \mapsto \_ * b \mapsto \_$ }

Acquire(l);

{ $(k1, v1) = x \wedge (k2, v2) = y \wedge$

$l \square \xrightarrow[h_3]{\top} R * \mathrm{Hold}\ l, R, h_o * \ \mathrm{Q} \rightsquigarrow (\mathrm{listof}(h_3)) \wedge \mathrm{history\_queue}\ (h_3) *$

$k1 \mapsto 1 * k2 \mapsto 1 * a \mapsto v2 * b \mapsto \_ \vee$

$k1 \mapsto 1 * k2 \mapsto 0 * a \mapsto v1 * b \mapsto v2 \vee$

$k1 \mapsto 0 * k2 \mapsto 1 * a \mapsto v2 * b \mapsto v1 \vee$

$k1 \mapsto 0 * k2 \mapsto 0 * a \mapsto \_ * b \mapsto \_$ } \\ Case analysis on h3

{ $l \square \xrightarrow[h_3]{\top} R * \mathrm{Hold}\ l, R, h_o * \ \mathrm{Q} \rightsquigarrow (\epsilon) *$

$(1,1) = x \wedge (0,0) = y \wedge k1 \mapsto 1 * k2 \mapsto 0 * a \mapsto 1 * b \mapsto 0 \vee$

$(0,0) = x \wedge (1,1) = y \wedge k1 \mapsto 0 * k2 \mapsto 1 * a \mapsto 1 * b \mapsto 0$ }

Free(l); free(k1,k2);

{ $\mathrm{Q} \rightsquigarrow \epsilon * a \mapsto 1 * b \mapsto 0$ }

delete();

}

{ $a \mapsto 1 * b \mapsto 0$ }

6 Tree add

```
struct node
{
  int k;           //key_value
  struct node *l; //left subtree
  struct node *r; //right subtree
};

void AddTree(struct node * t, int *res){
  { t ⤚ tree * res ↦ _ }
  if (empty(t)){
    { t ⤚ ϵ * res ↦ _ }
    res = 0;
    { t ⤚ ϵ * res ↦ 0 }
  } else {
    { t ⤚ (k, ltree, rtree) * res ↦ _ }
    int *lres, rres;
    thread lth, rth;
    { t ⤚ (k, ltree, rtree) * (res, lres, rres, lth, rth) ↦ _ }
    { t ↦ (k, l, r) * l ⤚ ltree * r ⤚ rtree * (res, lres, rres, lth, rth) ↦ _ }
    lthread = spawn (AddTree, (left, t→l, lies));
    { lth⃝→ R_l * t ↦ (k, l, r) * r ⤚ rtree * (res, rres, rth) ↦ _ }
    rthread = spawn (AddTree, (right, t→r));
    { rth⃝→ R_r * lth⃝→ R_l * t ↦ (k, l, r) * res ↦ _ }
    join (lth);
    { (add_tree(ltree) = k_l) ∧ l ⤚ ltree * lres ↦ k_l) *
    rth⃝→ R_r * t ↦ (k, l, r) * res ↦ _ }
    join (rth);
    { (add_tree(ltree) = k_l) ∧ l ⤚ ltree * lres ↦ k_l) *
    (add_tree(rtree) = k_r) ∧ r ⤚ rtree * rres ↦ k_r) *
    t ↦ (k, l, r) * (res) ↦ _ }
    res = lres + rres + t.k;
    { (add_tree(ltree) = k_l) ∧ lres ↦ k_l) *
    (add_tree(rtree) = k_r) ∧ rres ↦ k_r) *
    t ⤚ (k, ltree, rtree) * (res) ↦ (k_l + k_r + k) }
  }
  { (add_tree(tree)) = k' ∧ t ⤚ tree * (res) ↦ (k') }
}
```

7 Tree add with reporting

```
struct node
{
  lock l;
  int *k;          //sum_value
  int k;           //key_value
  struct node *l; //left subtree
  struct node *r; //right subtree
};
```

$\{\ node.l \square\xrightarrow[\bot]{\pi} R\ \}$ // $R = \lambda v.STUFF$

```
void AddTreeRep(struct node * t, int *RL){
```

$\{\ t \rightarrowtail tree * RL\square\xrightarrow[0]{\pi} R\ \}$

```
  if (empty(t)){
    \\This branch is useless in practice.
```

$\{\ \text{add\_tree}(\epsilon) = 0 \wedge t \rightarrowtail \epsilon * RL\square\xrightarrow[0]{\pi} R\ \}$

```
  } else {
```

$\{\ t \rightarrowtail (k, ltree, rtree) * RL\square\xrightarrow[0]{\pi} R\ \}$

$\{\ t \mapsto (k, l, r) * l \rightarrowtail ltree * r \rightarrowtail rtree * RL\square\xrightarrow[0]{\pi} R\ \}$

```
    lthread = spawn (AddTreeRep, (left, t->l, lies));
```

$\{\ lth\bigcirc\!\!\rightarrow R_l * t \mapsto (k, l, r) * r \rightarrowtail rtree * RL\square\xrightarrow[0]{\frac{2\pi}{3}} R\ \}$

```
    rthread = spawn (AddTreeRep, (right, t->r));
```

$\{\ rth\bigcirc\!\!\rightarrow R_r * lth\bigcirc\!\!\rightarrow R_l * t \mapsto (k, l, r) * RL\square\xrightarrow[0]{\frac{\pi}{3}} R\ \}$

```
    Acquire(RL);
```

$\{\ \exists v_o.result \mapsto (v_o + 0) * \text{Hold } RL, R, v_o *$

$rth\bigcirc\!\!\rightarrow R_r * lth\bigcirc\!\!\rightarrow R_l * t \mapsto (k, l, r) * RL\square\xrightarrow[0]{\frac{\pi}{3}} R\ \}$

```
    result = result + (t->k);
```

$\{\ \exists v_o.result \mapsto (v_o + k) * \text{Hold } RL, R, v_o *$

$rth\bigcirc\!\!\rightarrow R_r * lth\bigcirc\!\!\rightarrow R_l * t \mapsto (k, l, r) * RL\square\xrightarrow[0]{\frac{\pi}{3}} R\ \}$

```
    Release(RL);
```

$\{\ rth\bigcirc\!\!\rightarrow R_r * lth\bigcirc\!\!\rightarrow R_l * t \mapsto (k, l, r) * RL\square\xrightarrow[k]{\frac{\pi}{3}} R\ \}$

```
    join (lth);
```

$\{\ (\text{add\_tree}(ltree) = k_l \wedge l \rightarrowtail ltree * RL\square\xrightarrow[k_l]{\frac{\pi}{3}} R) *$

$rth\bigcirc\!\!\rightarrow R_r * t \mapsto (k, l, r) * RL\square\xrightarrow[k]{\frac{\pi}{3}} R\ \}$

```
    join (rth);
```

$\{\ (\text{add\_tree}(rtree) = k_r \wedge r \rightarrowtail rtree * RL\square\xrightarrow[k_r]{\frac{\pi}{3}} R) *$

$(\text{add\_tree}(ltree) = k_l \wedge l \rightarrowtail ltree * RL\square\xrightarrow[k_l]{\frac{\pi}{3}} R) *$

$t \mapsto (k, l, r) * RL\square\xrightarrow[k]{\frac{\pi}{3}} R\ \}$

$\{\ \text{add\_tree}(rtree) = k_r \wedge \text{add\_tree}(ltree) = k_l \wedge$

$r \rightarrowtail rtree * l \rightarrowtail ltree * t \mapsto (k, l, r) * RL\square\xrightarrow[k+k_l+k_r]{\pi} R\ \}$

}

$\{\ \text{add\_tree}(tree) = k' \wedge t \rightarrowtail tree * RL\square\xrightarrow[k']{\pi} R\ \}$

}

$\{\ \text{add\_tree}(tree) = k' \wedge t \rightarrowtail tree * RL\square\xrightarrow[k']{\pi} R\ \}$

8 Adding a Directed Acyclic Graph with repetitions

$$\text{dag} := \epsilon$$
$$|\forall sum, k, (l, r : \text{dag})(\pi_l, \pi_r : \text{shares})(sum, k, \pi_l, l, \pi_r, r)$$

$$g \overset{\pi}{\dagger} \epsilon \triangleq g = \text{NULL}$$
$$g \overset{\bot}{\underset{\pi}{\dagger}} \epsilon \triangleq g = \text{NULL}$$
$$g \overset{\pi}{\underset{sum}{\dagger}} (sum, k, \pi_l, d_l, \pi_r, d_r) \triangleq g.lock \square \xrightarrow[(sum,k,\pi_l,l,\pi_r,r)]{\pi} R$$

WHERE

$$R \triangleq \lambda(sum, k, \pi_l, d_l, \pi_r, d_r).\exists l, r, k, sum_l, sum_r$$
$$g.k \to k*$$
$$g.l \to l*$$
$$g.r \to r*$$
$$\text{if } g.sum = \text{NULL} \text{ then}$$
$$sum = sum_l = sum_r = \bot$$
$$g.sum = \text{NULL} *$$
$$l \overset{\pi_l}{\underset{\bot}{\dagger}} d_l * r \overset{\pi_r}{\underset{\bot}{\dagger}} d_r$$
$$\text{else}$$
$$sum = k + sum_l + sum_r \wedge$$
$$g.sum \to sum*$$
$$l \overset{\pi_l}{\underset{sum_l}{\dagger}} d_l * r \overset{\pi_r}{\underset{sum_r}{\dagger}} d_r$$

$$\epsilon \oplus \epsilon \triangleq \epsilon$$
$$(sum_1, k, \pi_l, l_1, \pi_r, r_1) \oplus (sum_2, k, \pi_l, l_2, \pi_r, r_2) \triangleq (sum_1 \oplus sum_2, k, \pi_l, l_1 \oplus l_2, \pi_r, r_1 \oplus r_2)$$

```
struct node
{
  lock l;          //lock
  int * sum;         //Partial sum
  int k;           //key_value
  struct node *l; //left subtree
  struct node *r; //right subtree
};
```

\

$\{\ g \overset{\pi}{\underset{\perp}{\dagger}} d\ \}$

```
void AddDag(struct node * g, int *ret){
  if (g = NULL){
```
$\{\ g \overset{\pi}{\underset{\perp}{\dagger}} \epsilon * ret \mapsto \_\ \}$
```
    ret = 0;
```
$\{\ g \overset{\pi}{\underset{0}{\dagger}} \epsilon * ret \mapsto 0\ \}$
```
    return;
  } else {
```
$\{\ g \overset{\pi}{\underset{\perp}{\dagger}} (\perp, k, \pi_l, l_s, \pi_r, r_s) * ret \mapsto \_\ \}$
```
    Acquire(g);
```
$\{\ \exists d_o, R(\perp, k, \pi_l, d_l, \pi_r, d_r) \oplus d_o * \text{Hold}\ g, R, d_o * g \overset{\pi}{\underset{\perp}{\dagger}} (\perp, k, \pi_l, d_l, \pi_r, d_r) * ret \mapsto\ \}$

$\{\ \exists v_o, d_l, d_r, R(\perp \oplus v_o, k, \pi_l, l_s \oplus d_l, \pi_r, r_s \oplus d_r) *$
$\text{Hold}\ g, R, d_o * g \overset{\pi}{\underset{\perp}{\dagger}} (\perp, k, \pi_l, d_l, \pi_r, d_r) * ret \mapsto \_\ \}$
```
    if (g.sum != NULL) {
```
$\{\ R(v_o, k, \pi_l, d_l, \pi_r, d_r) * \text{Hold}\ g, R, d_o * g \overset{\pi}{\underset{\perp}{\dagger}} (\perp, k, \pi_l, d_l, \pi_r, d_r) * ret \mapsto \_\ \}$
```
      ret = g.sum;
```
$\{\ R(v_o, k, \pi_l, d_l, \pi_r, d_r) * \text{Hold}\ g, R, d_o * g \overset{\pi}{\underset{\perp}{\dagger}} (\perp, k, \pi_l, d_l, \pi_r, d_r) * ret \mapsto g.sum\ \}$
```
      Release (g)
```
$\{\ g \overset{\pi}{\underset{v_o}{\dagger}} (v_o, k, \pi_l, d_l, \pi_r, d_r) * ret \mapsto g.sum\ \}$
```
    } else {
```
$\{\ R(\perp, k, \pi_l, d_l, \pi_r, d_r) * \text{Hold}\ g, R, d_o * g \overset{\pi}{\underset{\perp}{\dagger}} (\perp, k, \pi_l, d_l, \pi_r, d_r) * ret \mapsto \_\ \}$

$\{\ \exists l, r, k, sum_l, sum_r, g.k \mapsto k * g.l \mapsto l * g.r \mapsto r * g.sum = \text{NULL}\ *$
$sum = sum_l = sum_r = \perp *$
$l \overset{\pi_l}{\underset{\perp}{\dagger}} d_l * r \overset{\pi_r}{\underset{\perp}{\dagger}} d_r *$
$\text{Hold}\ g, R, d_o * g \overset{\pi}{\underset{\perp}{\dagger}} (\perp, k, \pi_l, d_l, \pi_r, d_r) * ret \mapsto \_\ \}$
```
      int *lret, rret;
      thr = Spawn (AddDag, (g.r, rret));
```
$\{\ thr \bigcirc\!\!\rightarrow R_r * g.k \mapsto k * g.l \mapsto l * g.r \mapsto r * g.sum = \text{NULL}\ *$
$sum = sum_l = \perp * l \overset{\pi_l}{\underset{\perp}{\dagger}} d_l *$
$\text{Hold}\ g, R, d_o * g \overset{\pi}{\underset{\perp}{\dagger}} (\perp, k, \pi_l, d_l, \pi_r, d_r) * ret \mapsto \_\ \}$
```
      AddDag (g.l, lret);
```
$\{\ thr \bigcirc\!\!\rightarrow R_r * g.k \mapsto k * g.l \mapsto l * g.r \mapsto r * g.sum = \text{NULL}\ *$
$l \overset{\pi_l}{\underset{sum_l}{\dagger}} d_l * lret \mapsto sum_l *$

```
      Hold g, R, d_o * g ↾^π_⊥ (⊥, k, π_l, d_l, π_r, d_r) * ret ↦ _ }
    Join(the);
    { g.k ↦ k * g.l ↦ l * g.r ↦ r * g.sum = NULL *
      l ↾^{π_l}_{sum_l} d_l * lret ↦ sum_l *
      l ↾^{π_r}_{sum_r} d_r * rret ↦ sum_r *
      Hold g, R, d_o * g ↾^π_⊥ (⊥, k, π_l, d_l, π_r, d_r) * ret ↦ _ }
    ret = (g.sum = k + sum_l + sum_r);
    { g.k ↦ k * g.l ↦ l * g.r ↦ r *
      g.sum ↦ k + sum_l + sum_r *
      l ↾^{π_l}_{sum_l} d_l * lret ↦ sum_l *
      l ↾^{π_r}_{sum_r} d_r * rret ↦ sum_r *
      Hold g, R, d_o * g ↾^π_⊥ (⊥, k, π_l, d_l, π_r, d_r) * ret ↦ k + sum_l + sum_r }
    { R(k + sum_l + sum_r, k, π_l, d_l, π_r, d_r) *
      Hold g, R, d_o * g ↾^π_⊥ (⊥, k, π_l, d_l, π_r, d_r) * ret ↦ sum' }   \\ sum'=k+sum_l+sum_r
    Release (g);
    { g ↾^π_{sum'} (sum', k, π_l, d_l, π_r, d_r) * ret ↦ sum' }
  }
 }
}
{ ∃sum', d, g ↾^π_{sum'} d * ret ↦ sum' }
```