Rast: Resource-Aware Session Types with Arithmetic Refinements

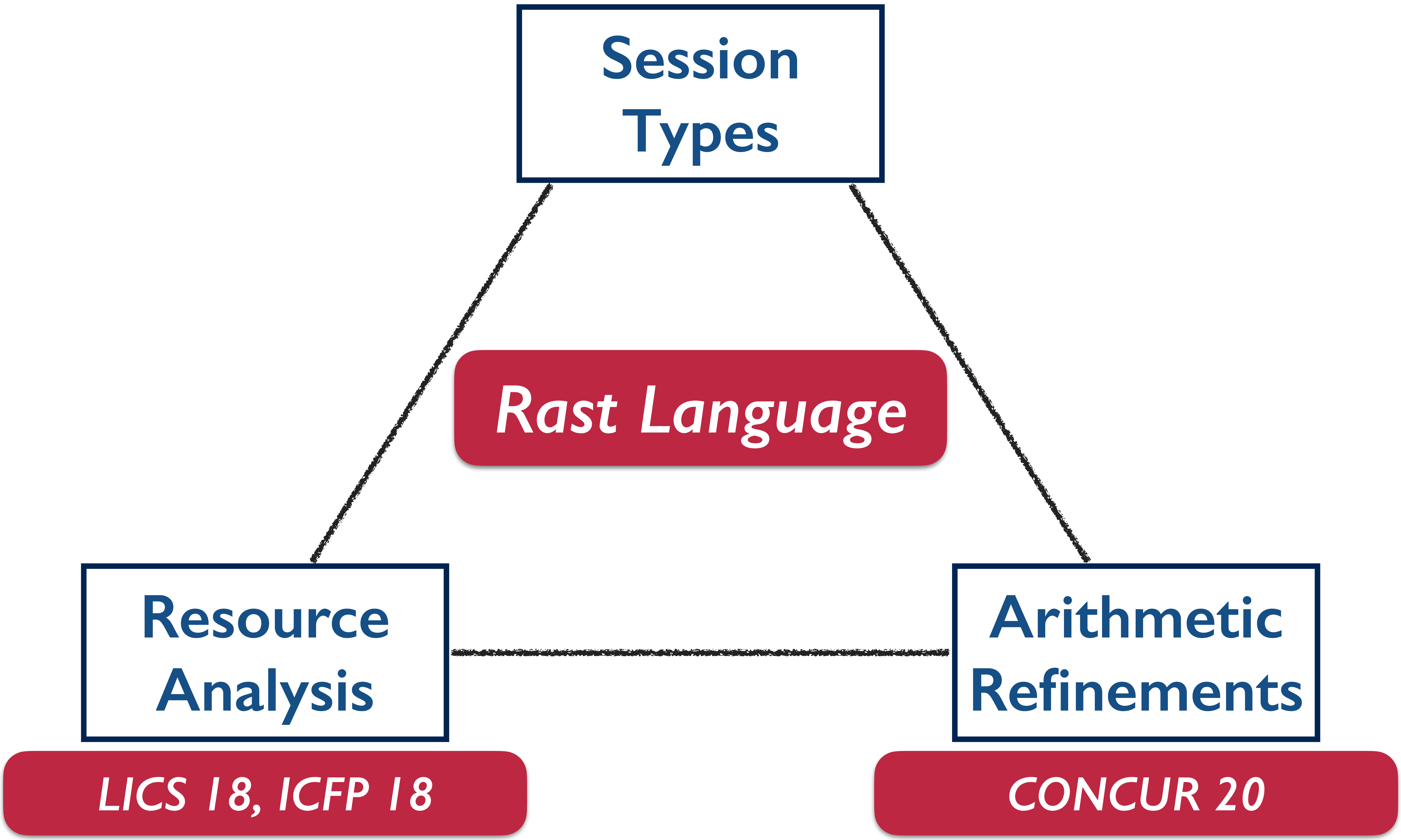
Ankush Das*

Frank Pfenning

Carnegie Mellon University

FSCD 2020

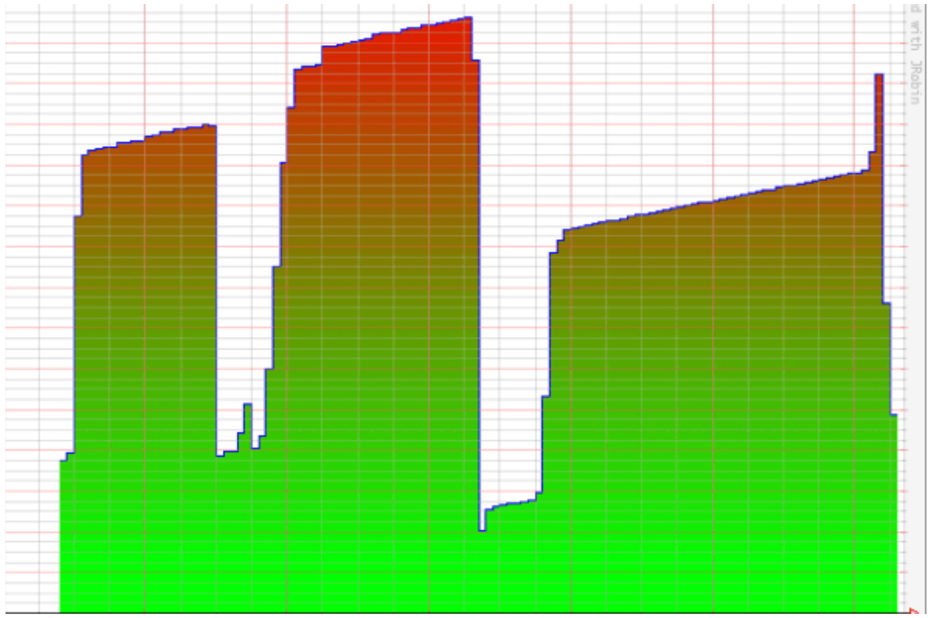
Key Features of Rast



Goal of Rast

Resource Analysis of Concurrent Programs

Execution Time



Memory Usage

Why Resource Analysis?

Why Resource Analysis?

Complexity of Parallel Algorithms

Çiçek et. al. (ESOP ’15)

Why Resource Analysis?

Complexity of Parallel Algorithms

Çiçek et. al. (ESOP ’15)

Design of Optimal Scheduling Policies

Acar et. al. (JFP ’16)

Why Resource Analysis?

Complexity of Parallel Algorithms

Çiçek et. al. (ESOP ’15)

Design of Optimal Scheduling Policies

Acar et. al. (JFP ’16)

Prevention of Side-Channel Attacks

Ngo et. al. (S&P ’17)

Why Resource Analysis?

Complexity of Parallel Algorithms

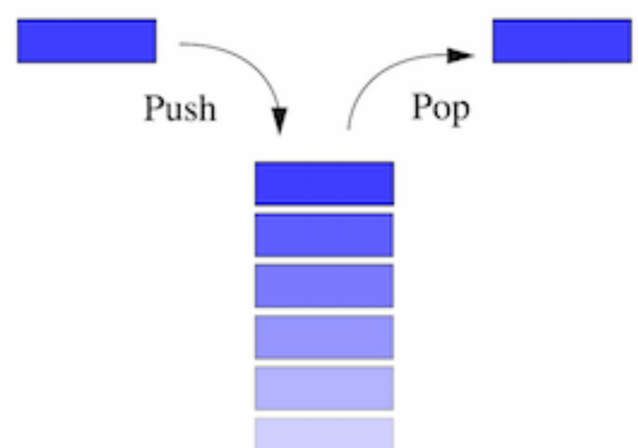
Çiçek et. al. (ESOP ’15)

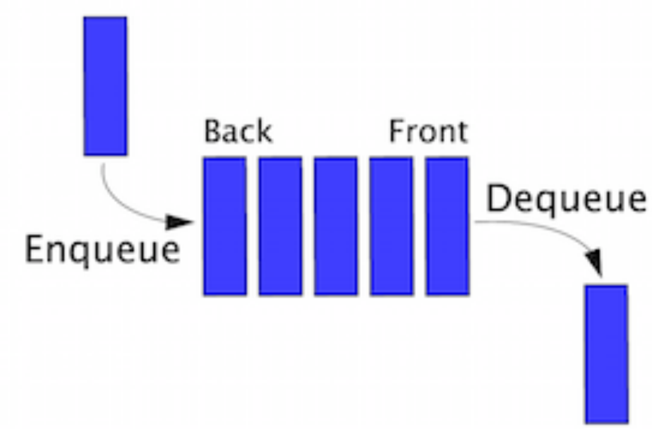
Design of Optimal Scheduling Policies

Acar et. al. (JFP ’16)

Prevention of Side-Channel Attacks

Ngo et. al. (S&P ’17)





Response Time of Concurrent Data Structures

Ellen and Brown (PODC ’16)

Concurrent Programs

Concurrent Programs

Need an appropriate abstraction for representing concurrent programs

Concurrent Programs

Need an appropriate abstraction for representing concurrent programs

Session Types

Why Session Types?

Concurrent programs are hard to analyze!

Why Session Types?

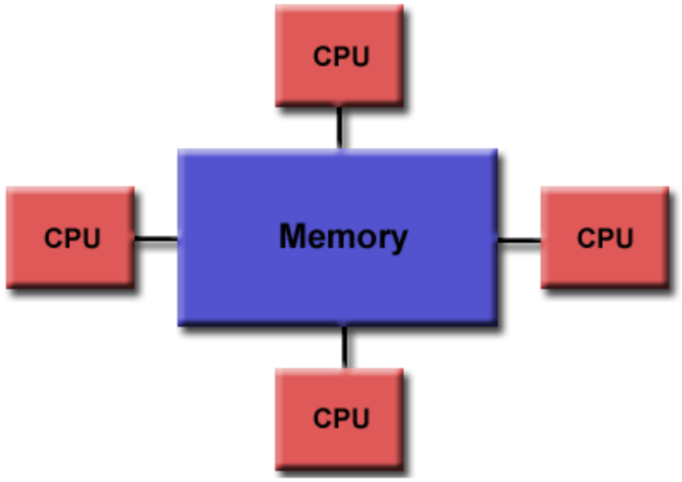
Concurrent programs are hard to analyze!

Communication Overhead

Why Session Types?

Concurrent programs are hard to analyze!

Communication Overhead

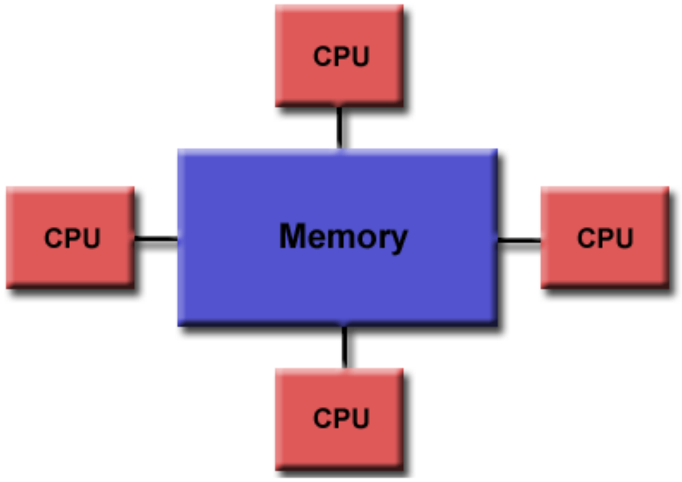


Shared Memory
Read/Write Overhead

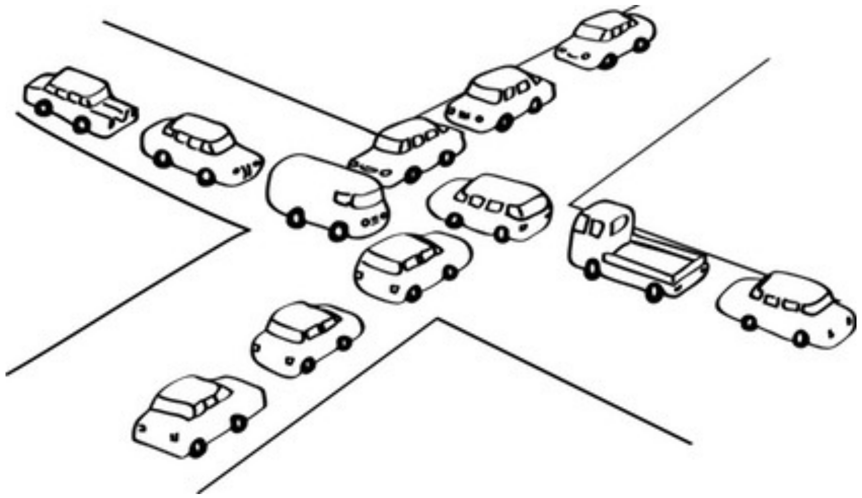
Why Session Types?

Concurrent programs are hard to analyze!

Communication Overhead



Shared Memory
Read/Write Overhead



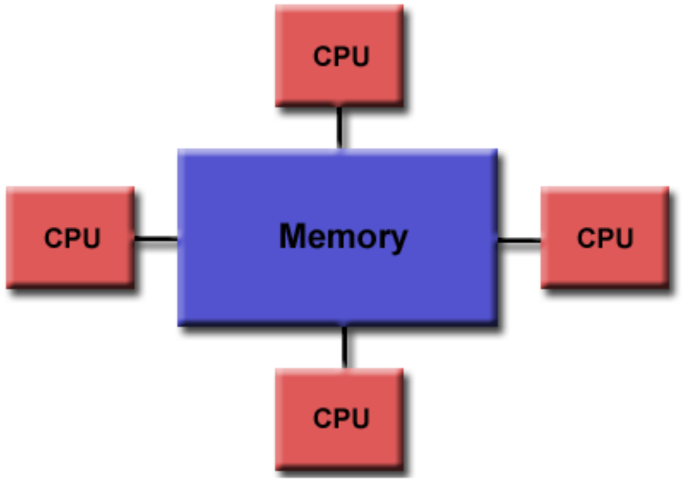
Deadlocks

Why Session Types?

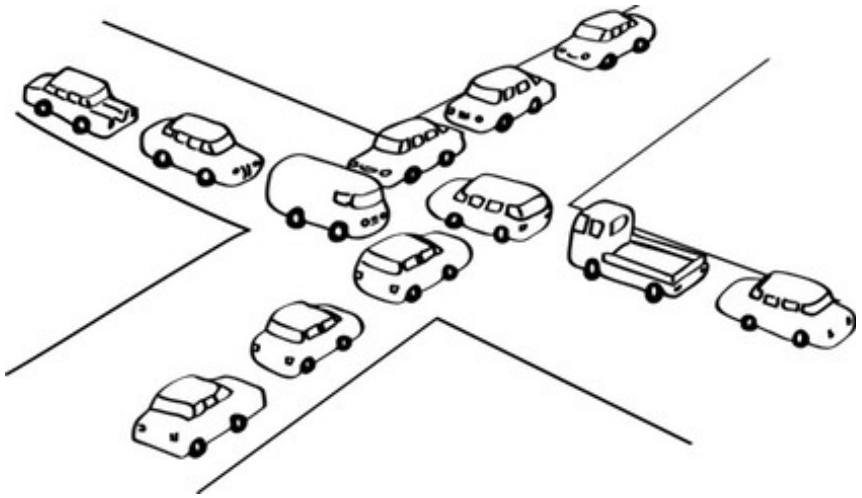
Concurrent programs are hard to analyze!

With Session Types

Communication Overhead



Shared Memory Read/Write Overhead



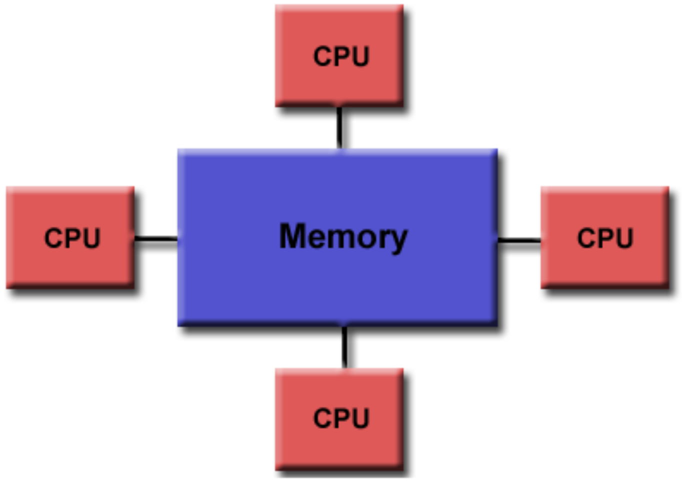
Deadlocks

Why Session Types?

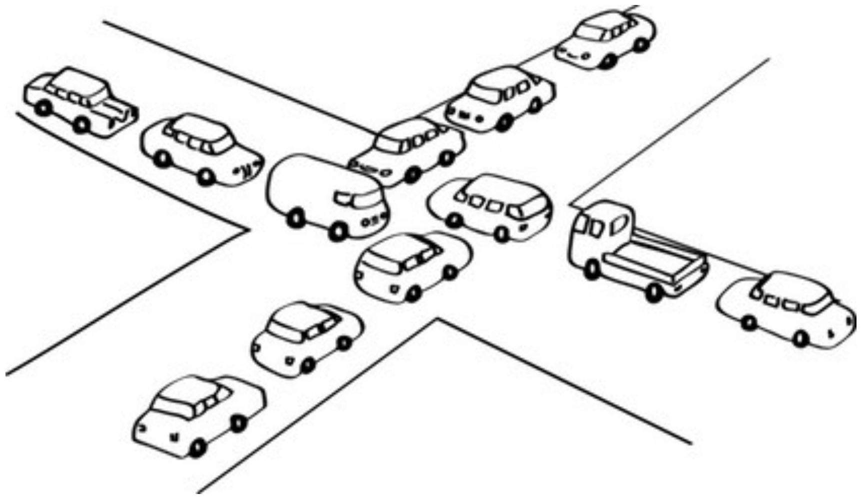
Concurrent programs are hard to analyze!

With Session Types

Types strictly enforce communication protocols



Shared Memory Read/Write Overhead



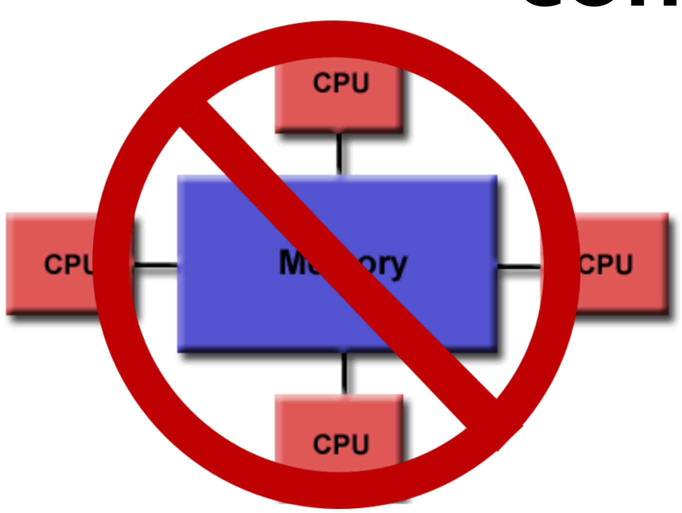
Deadlocks

Why Session Types?

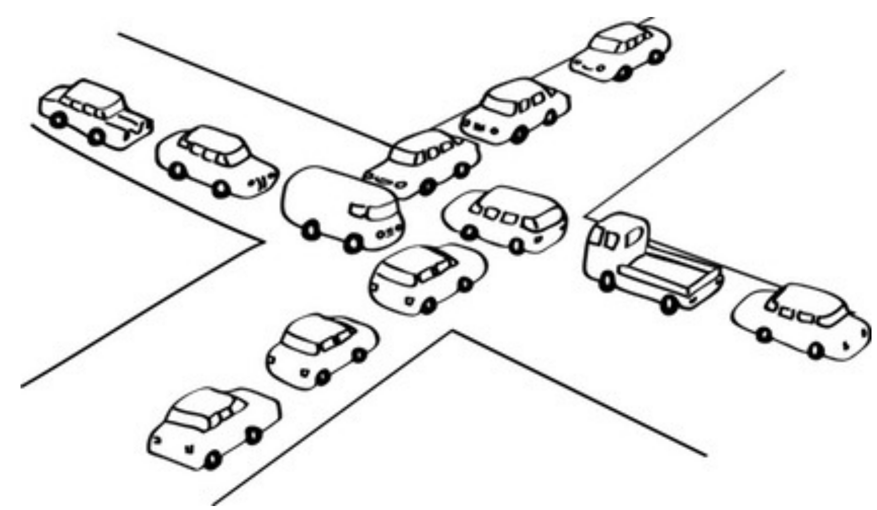
Concurrent programs are hard to analyze!

With Session Types

Types strictly enforce communication protocols



No Shared Memory



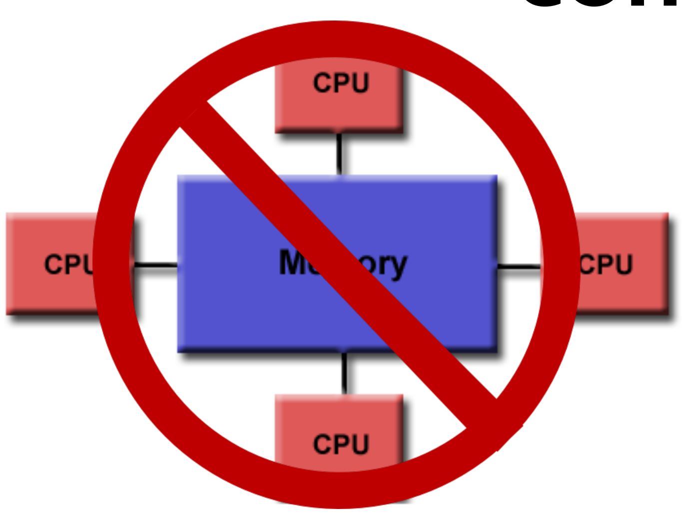
Deadlocks

Why Session Types?

Concurrent programs are hard to analyze!

With Session Types

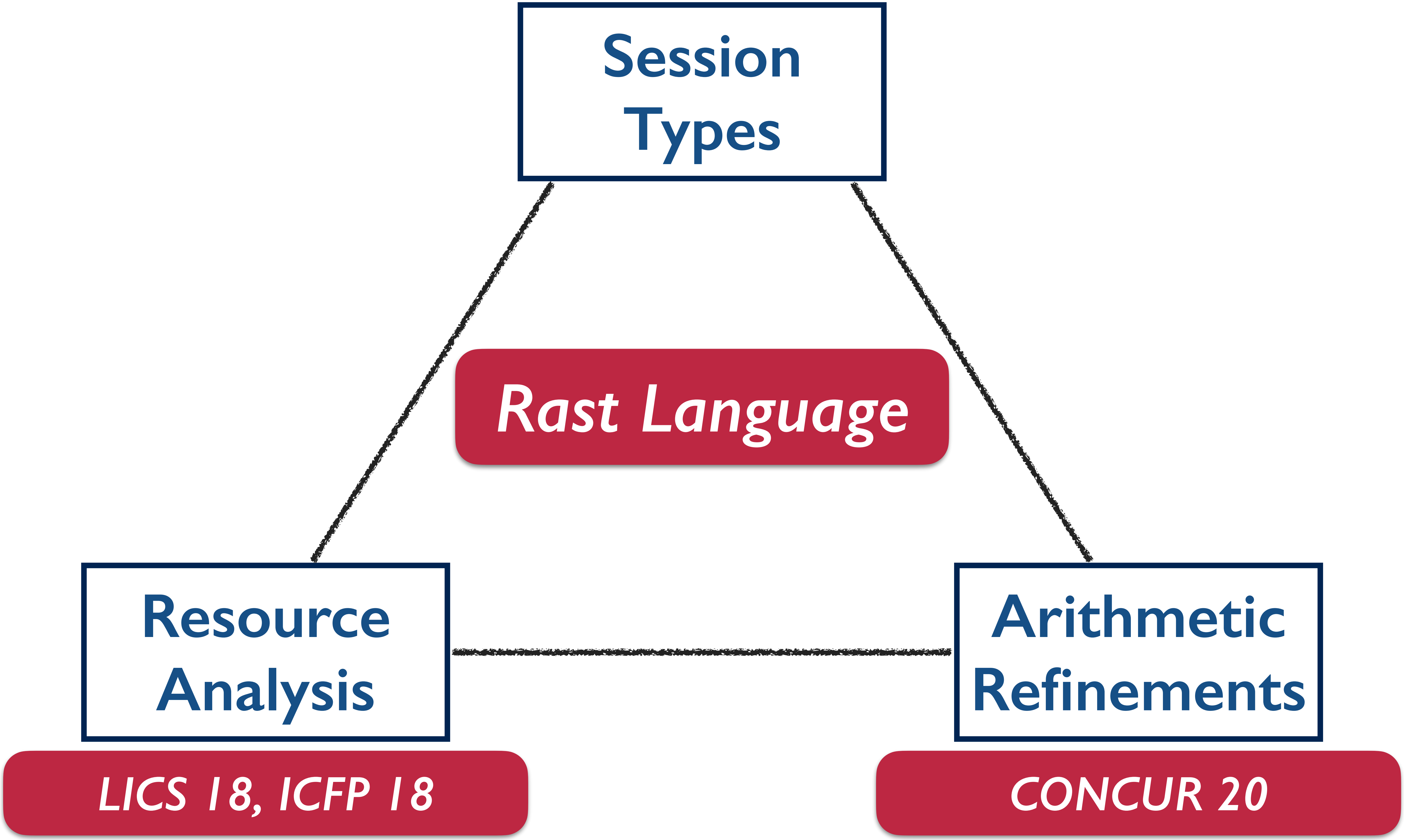
Types strictly enforce communication protocols



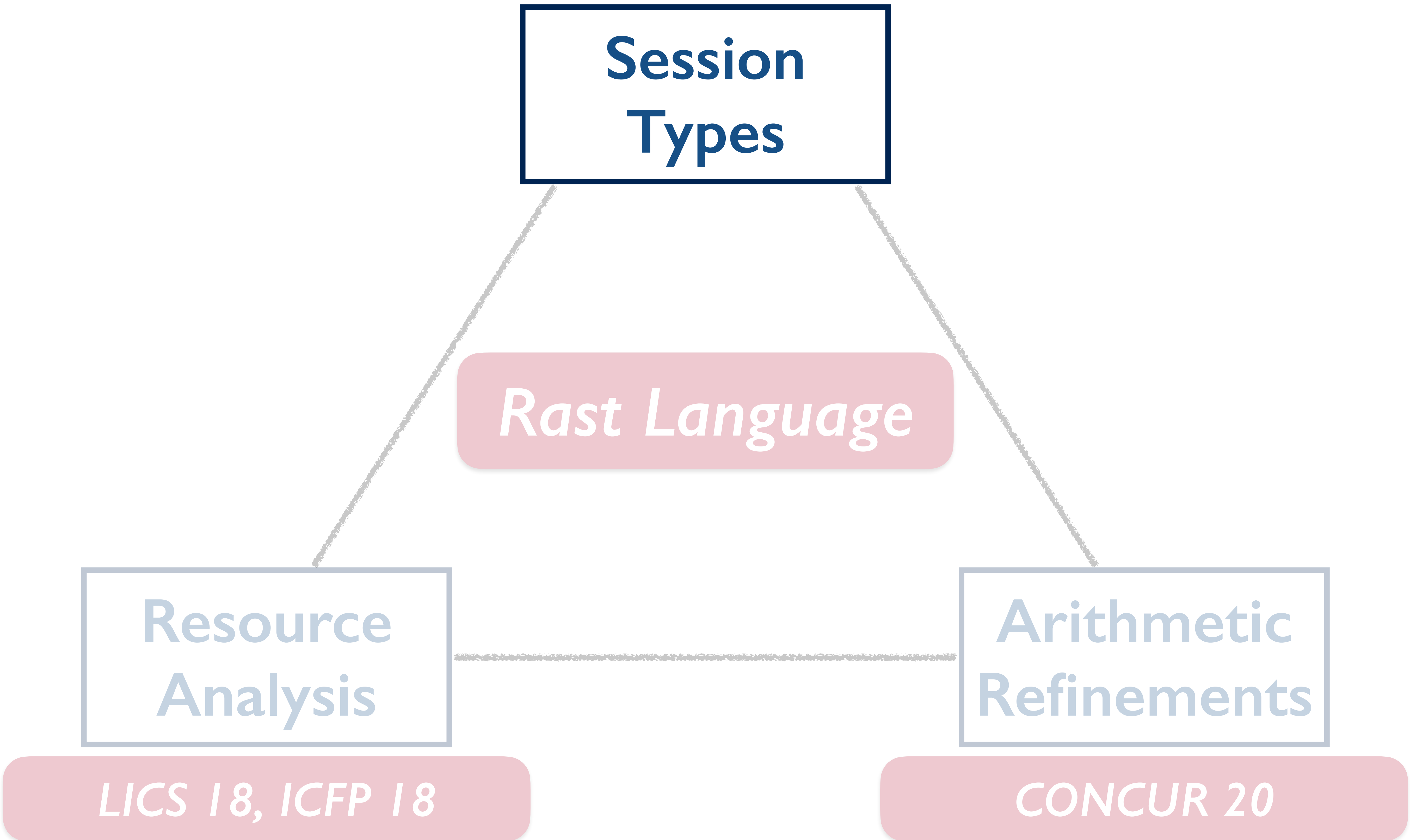
No Shared Memory

Deadlock Freedom

Key Features of Rast



Key Features of Rast



What are Session Types?

- Implement message-passing concurrent programs
- Communication via typed bi-directional channels
- Communication protocol enforced by session types

What are Session Types?

- Implement message-passing concurrent programs
- Communication via typed bi-directional channels
- Communication protocol enforced by session types

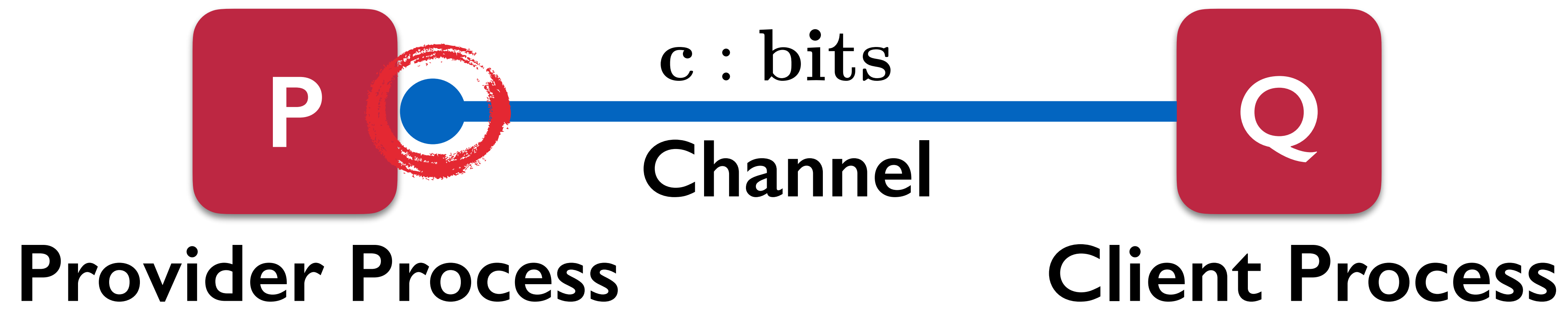
What are Session Types?

- Implement message-passing concurrent programs
- Communication via typed bi-directional channels
- Communication protocol enforced by session types

What are Session Types?

- Implement message-passing concurrent programs
- Communication via typed bi-directional channels
- Communication protocol enforced by session types

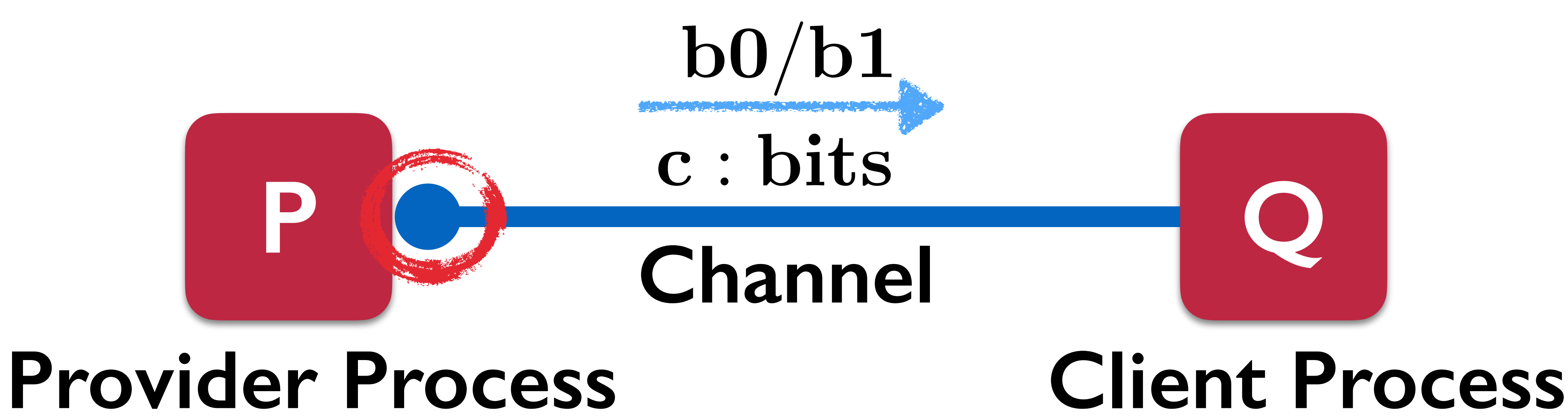
$$\mathrm{bits} = \oplus\{\mathrm{b0} : \mathrm{bits}, \mathrm{b1} : \mathrm{bits}\}$$



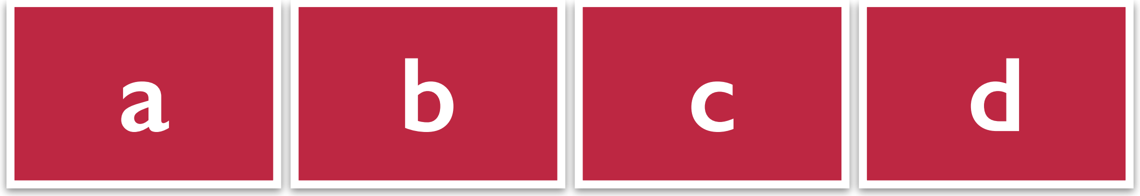
What are Session Types?

- Implement message-passing concurrent programs
- Communication via typed bi-directional channels
- Communication protocol enforced by session types

$$\mathrm{bits} = \oplus\{\mathrm{b0} : \mathrm{bits}, \mathrm{b1} : \mathrm{bits}\}$$

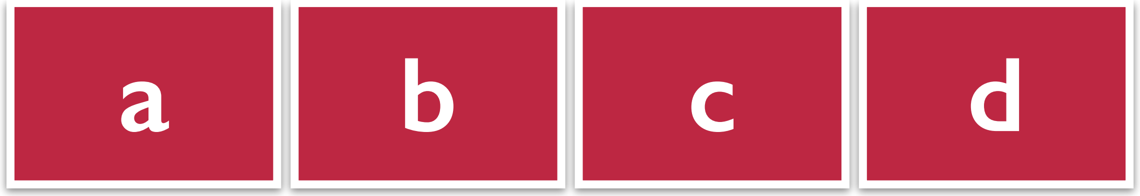


Example: Queues



$$
\begin{aligned}
\mathsf{queue}_{\mathbf{A}} = \&\{&\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},\\
&\mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\\
&\qquad\qquad \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}
\end{aligned}
$$

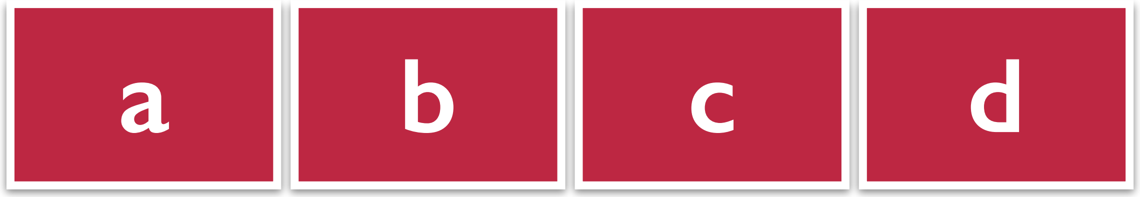
Example: Queues



offers choice of ins/del

$$\begin{aligned}\mathsf{queue}_{\mathbf{A}} = \&\{&\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},\\ &\mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\\ &\qquad\qquad \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}\end{aligned}$$

Example: Queues

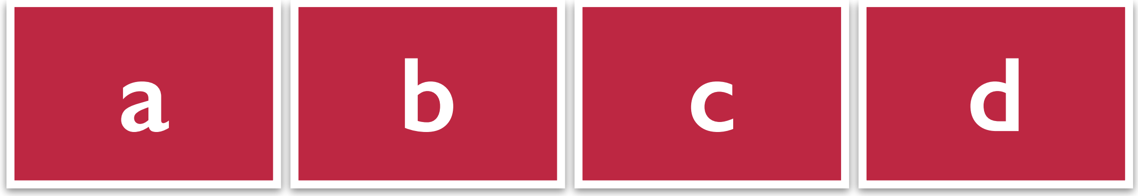


offers choice of ins/del

recv element of type A

$$\mathsf{queue}_{\mathbf{A}} = \&\{\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},\; \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\; \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}$$

Example: Queues



offers choice of ins/del

recv element of type A

behave as queue again

$$
\begin{aligned}
\mathsf{queue}_{\mathbf{A}} = \&\{&\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},\\
&\mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\\
&\qquad\qquad \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}
\end{aligned}
$$

Example: Queues

ins, e

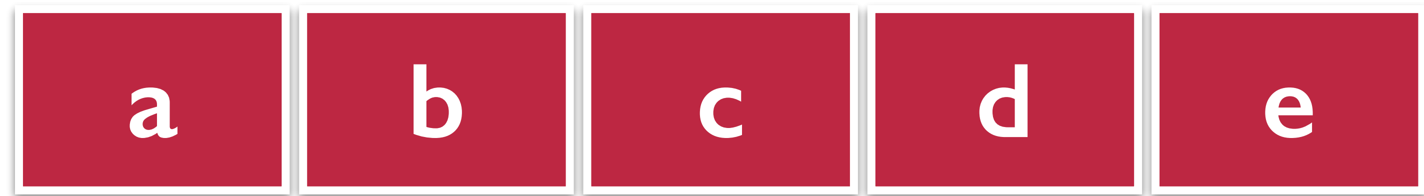
offers choice of ins/del

recv element of type A

behave as queue again

$$\mathsf{queue}_{\mathbf{A}} = \&\{\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},$$
$$\mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},$$
$$\mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}$$

Example: Queues



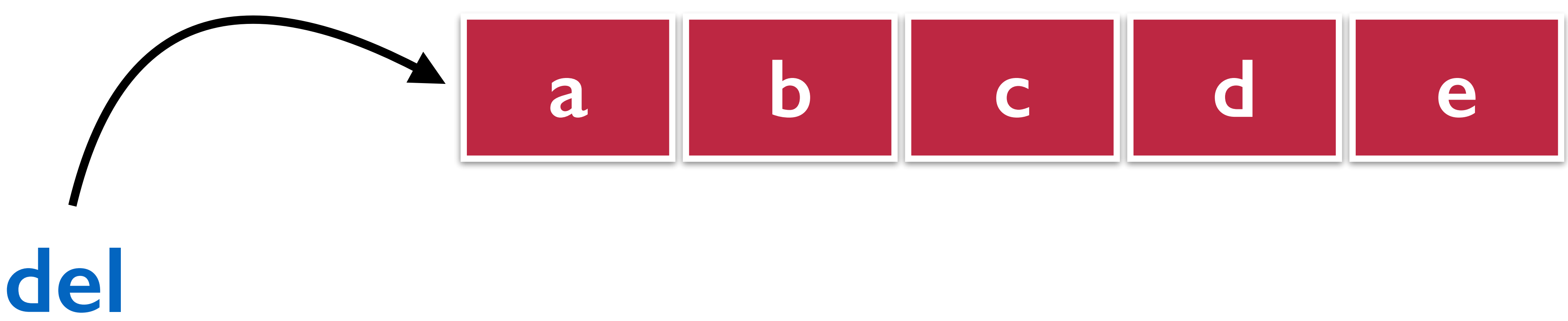
offers choice of ins/del

recv element of type A

behave as queue again

$$\begin{aligned}\mathrm{queue}_{\mathbf{A}} = \&\{\mathrm{ins} : \mathbf{A} \multimap \mathrm{queue}_{\mathbf{A}},\\ \mathrm{del} : \oplus\{\mathrm{none} : \mathbf{1},\\ \mathrm{some} : \mathbf{A} \otimes \mathrm{queue}_{\mathbf{A}}\}\}\end{aligned}$$

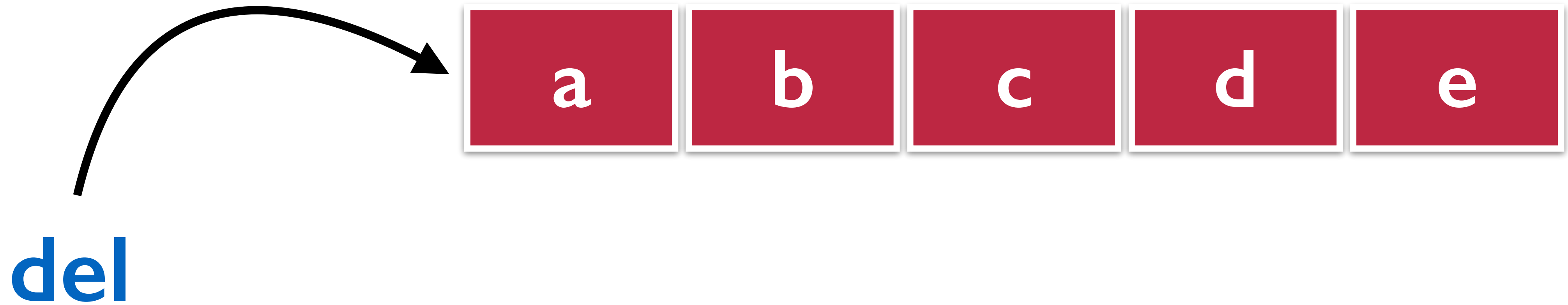
Example: Queues



offers choice of ins/del

$$\begin{aligned}\mathsf{queue}_{\mathbf{A}} = \&\{&\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},\\ &\mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\\ &\qquad\qquad \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}\end{aligned}$$

Example: Queues

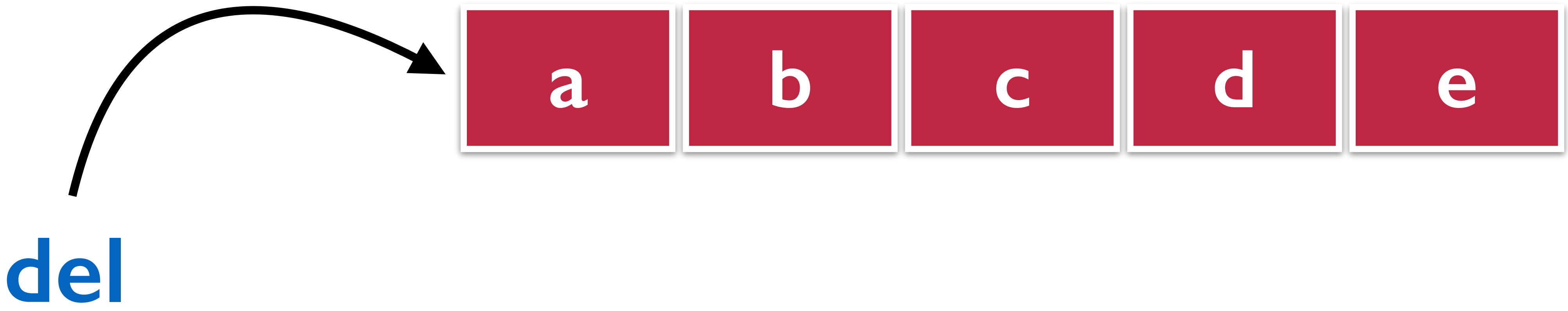


offers choice of ins/del

$$\mathsf{queue}_{\mathbf{A}} = \&\{\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},\ \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\ \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}$$

send none if queue is empty

Example: Queues



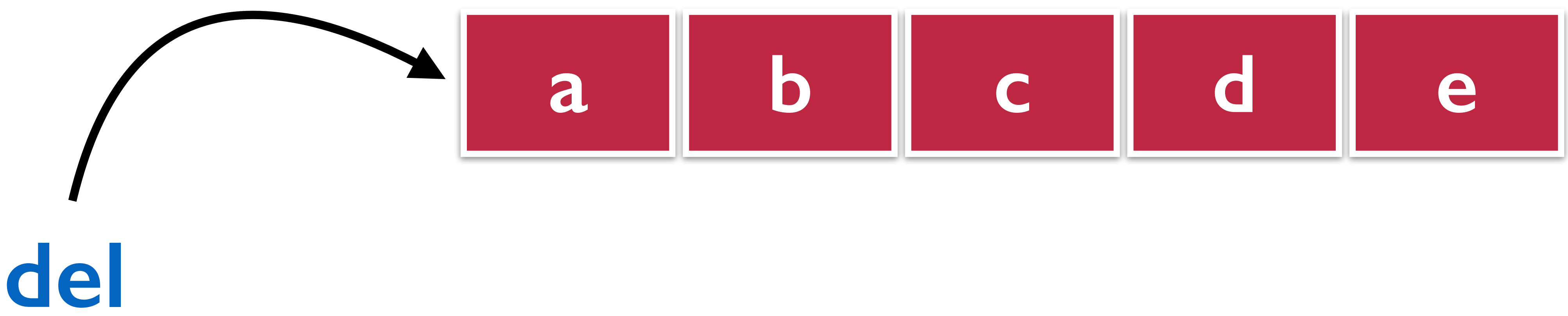
offers choice of ins/del

$$
\begin{aligned}
\mathsf{queue}_{\mathbf{A}} = \&\{ & \mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}}, \\
& \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \\
& \qquad \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}
\end{aligned}
$$

terminate

send none if queue is empty

Example: Queues

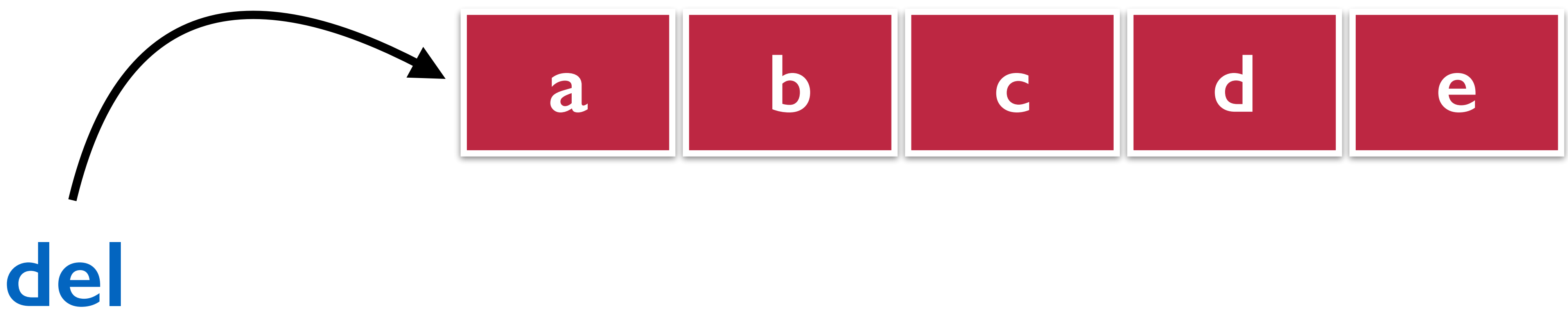


offers choice of ins/del

$$
\begin{aligned}
\mathsf{queue}_{\mathbf{A}} = \&\{&\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},\\
&\mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\\
&\qquad\qquad \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}
\end{aligned}
$$

send some otherwise

Example: Queues



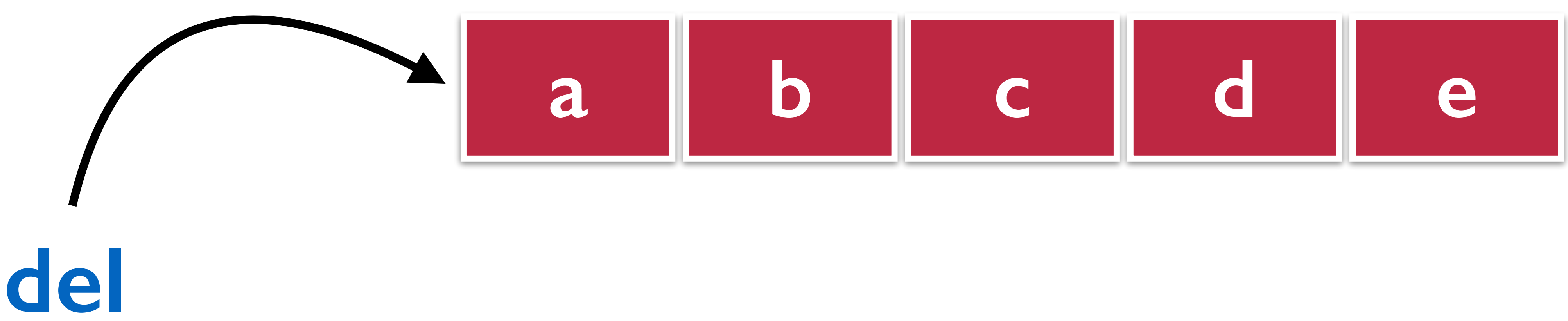
offers choice of ins/del

send element of type A

$$\mathsf{queue}_{\mathbf{A}} = \&\{\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},\ \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\ \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}$$

send some otherwise

Example: Queues



offers choice of ins/del

send element of type A

behave as queue again

$$\mathsf{queue}_{\mathbf{A}} = \&\{\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},\ \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\ \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}$$

send some otherwise

Example: Queues

offers choice of ins/del

send element of type A

behave as queue again

$$\mathsf{queue}_{\mathbf{A}} = \&\{\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},\ \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\ \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}$$

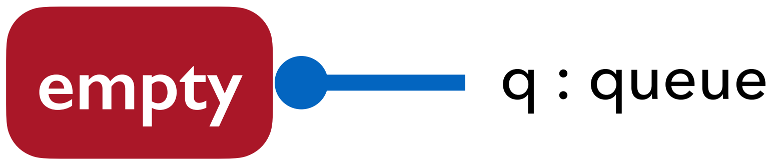
send some otherwise

Queues in Rast

```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

```
proc q <- empty =
  case q (
    ins => x <- recv q ;
           t <- empty ;
           q <- elem x t
  | del => q.none ;
           close q )
```

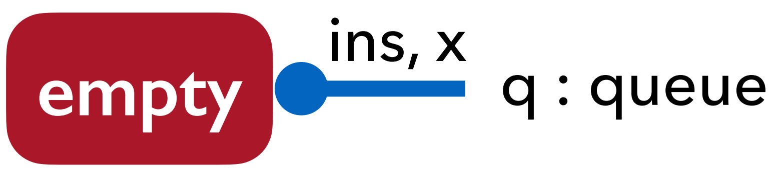


Queues in Rast


```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

```
proc q <- empty =
  case q (
    ins => x <- recv q ;
           t <- empty ;
           q <- elem x t
  | del => q.none ;
           close q )
```



Queues in Rast


```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

```
proc q <- empty =
  case q (
    ins => x <- recv q ;
           t <- empty ;
           q <- elem x t
  | del => q.none ;
           close q )
```

Queues in Rast


```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

```
proc q <- empty =
  case q (
    ins => x <- recv q ;
           t <- empty ;
           q <- elem x t
  | del => q.none ;
           close q )
```

Queues in Rast


```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

```
proc q <- empty =
  case q (
    ins => x <- recv q ;
           t <- empty ;
           q <- elem x t
  | del => q.none ;
           close q )
```

Queues in Rast


```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

```
proc q <- empty =
  case q (
    ins => x <- recv q ;
           t <- empty ;
           q <- elem x t
  | del => q.none ;
           close q )
```

Queues in Rast


```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

```
proc q <- empty =
  case q (
    ins => x <- recv q ;
           t <- empty ;
           q <- elem x t
  | del => q.none ;
           close q )
```

Queues in Rast


```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

```
proc q <- elem x t =
  case q (
    ins => y <- recv q ;
           t.ins ;
           send t y ;
           q <- elem x t
  | del => q.some ;
           send q x ;
           q <-> t )
```

Queues in Rast


```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

```
proc q <- elem x t =
  case q (
    ins => y <- recv q ;
           t.ins ;
           send t y ;
           q <- elem x t
  | del => q.some ;
           send q x ;
           q <-> t )
```

Queues in Rast

```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

```
proc q <- elem x t =
  case q (
    ins => y <- recv q ;
           t.ins ;
           send t y ;
           q <- elem x t
  | del => q.some ;
           send q x ;
           q <-> t )
```

Queues in Rast


```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

```
proc q <- elem x t =
  case q (
    ins => y <- recv q ;
           t.ins ;
           send t y ;
           q <- elem x t
  | del => q.some ;
           send q x ;
           q <-> t )
```

Queues in Rast


```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

```
proc q <- elem x t =
  case q (
    ins => y <- recv q ;
           t.ins ;
           send t y ;
           q <- elem x t
  | del => q.some ;
           send q x ;
           q <-> t )
```

Queues in Rast


```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

```
proc q <- elem x t =
  case q (
    ins => y <- recv q ;
           t.ins ;
           send t y ;
           q <- elem x t
  | del => q.some ;
           send q x ;
           q <-> t )
```

Queues in Rast


```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

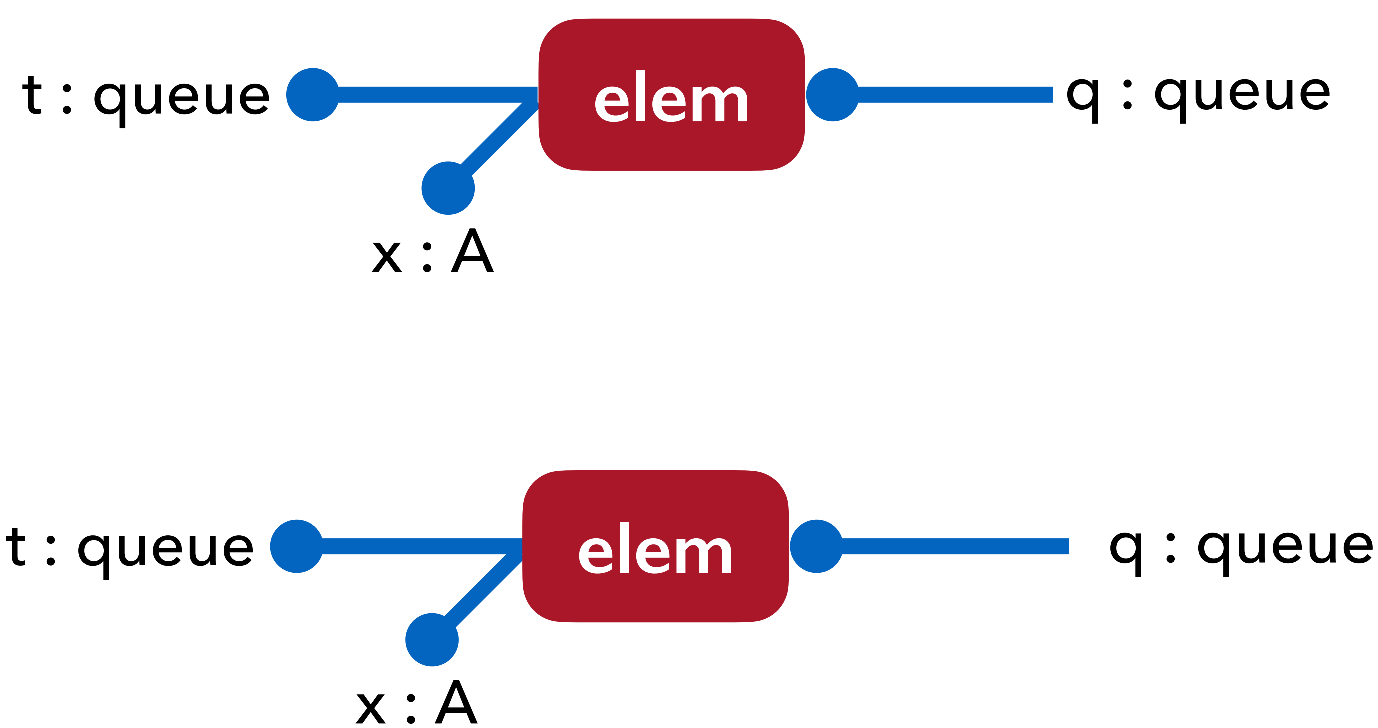
```
proc q <- elem x t =
  case q (
    ins => y <- recv q ;
           t.ins ;
           send t y ;
           q <- elem x t
  | del => q.some ;
           send q x ;
           q <-> t )
```

Queues in Rast


```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

```
proc q <- elem x t =
  case q (
    ins => y <- recv q ;
           t.ins ;
           send t y ;
           q <- elem x t
  | del => q.some ;
           send q x ;
           q <-> t )
```



Queues in Rast

```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

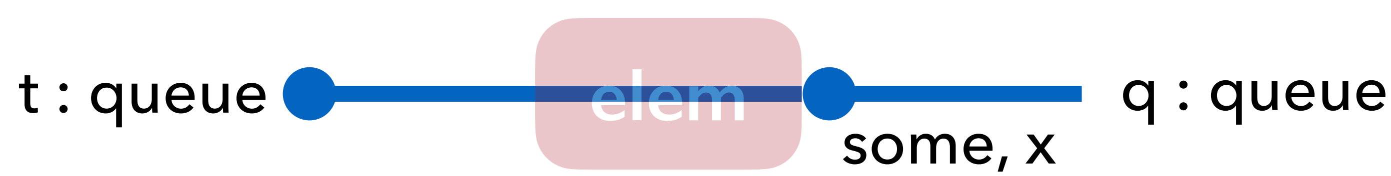
```
proc q <- elem x t =
  case q (
    ins => y <- recv q ;
           t.ins ;
           send t y ;
           q <- elem x t
  | del => q.some ;
           send q x ;
           q <-> t )
```

Queues in Rast

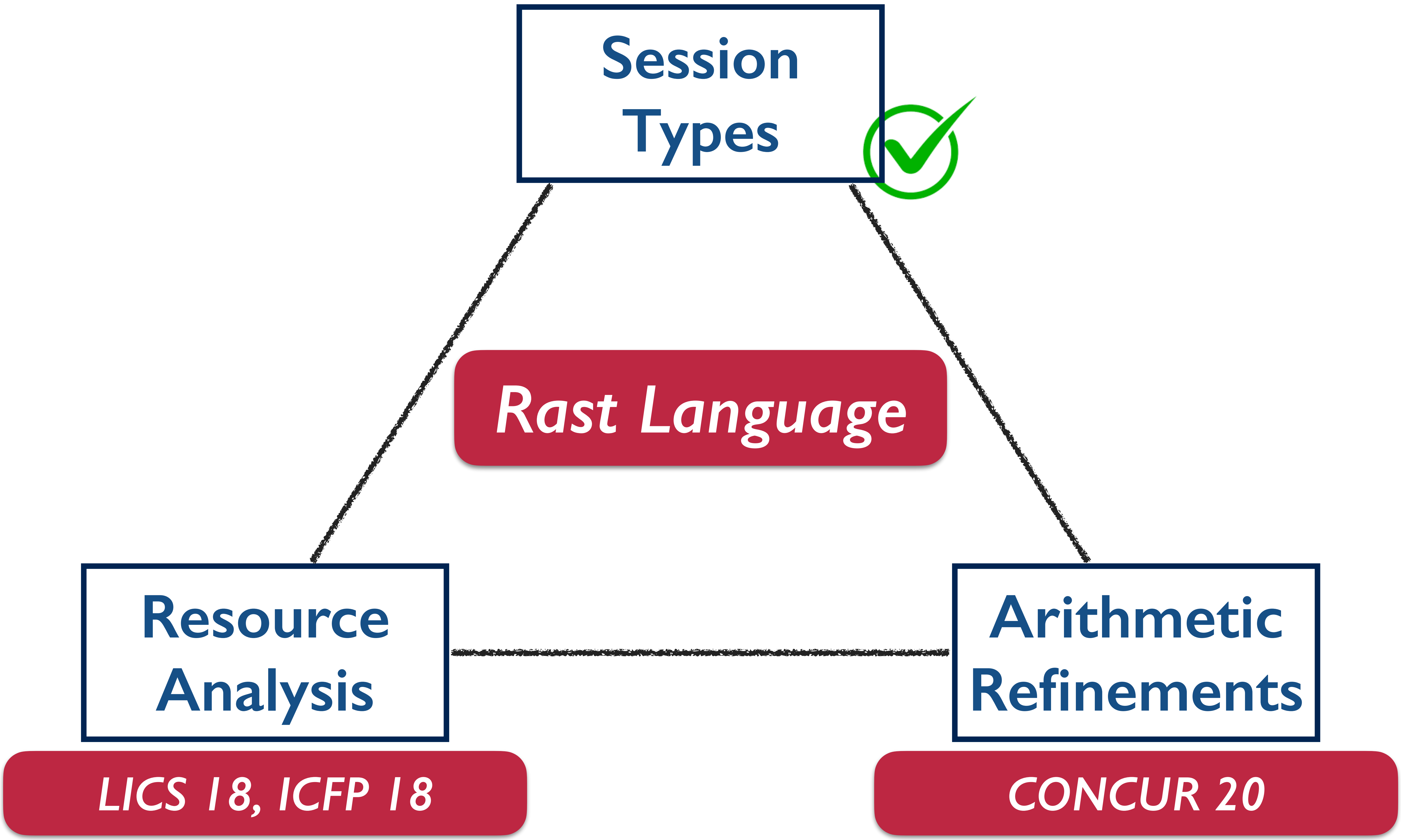

```
type queue = &{ins : A -o queue,
               del : +{none : 1,
                       some : A * queue}}

decl empty : . |- (q : queue)
decl elem : (x : A) (t : queue) |- (q : queue)
```

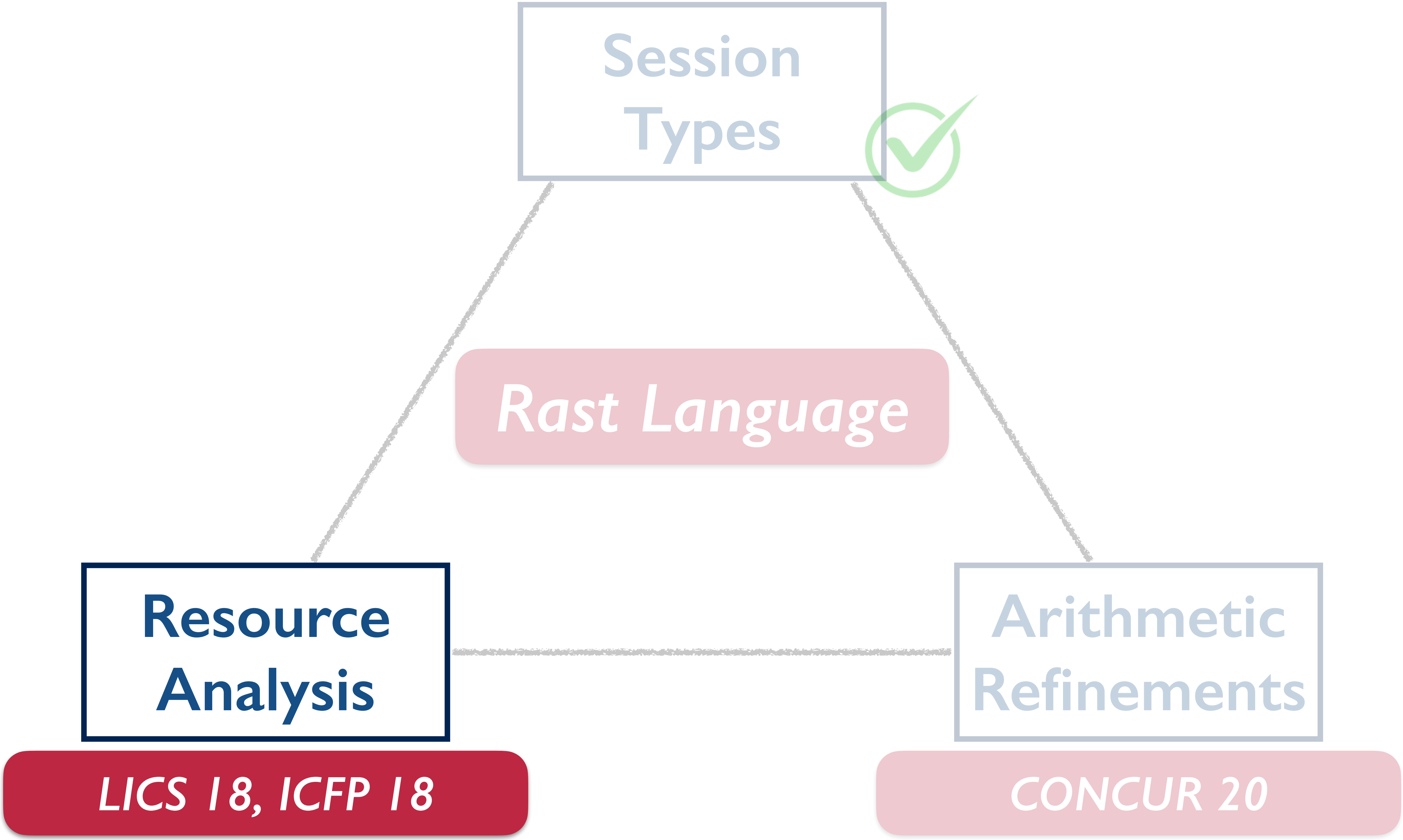
```
proc q <- elem x t =
  case q (
    ins => y <- recv q ;
           t.ins ;
           send t y ;
           q <- elem x t
  | del => q.some ;
           send q x ;
           q <-> t )
```



Key Features of Rast



Key Features of Rast



Complexity Measures

Complexity Measures

Work
Sequential Complexity

Execution time on one processor

LICS 18

Complexity Measures

Work
Sequential Complexity

Execution time on one processor

LICS 18

Span
Parallel Complexity

Execution time on arbitrarily many processors

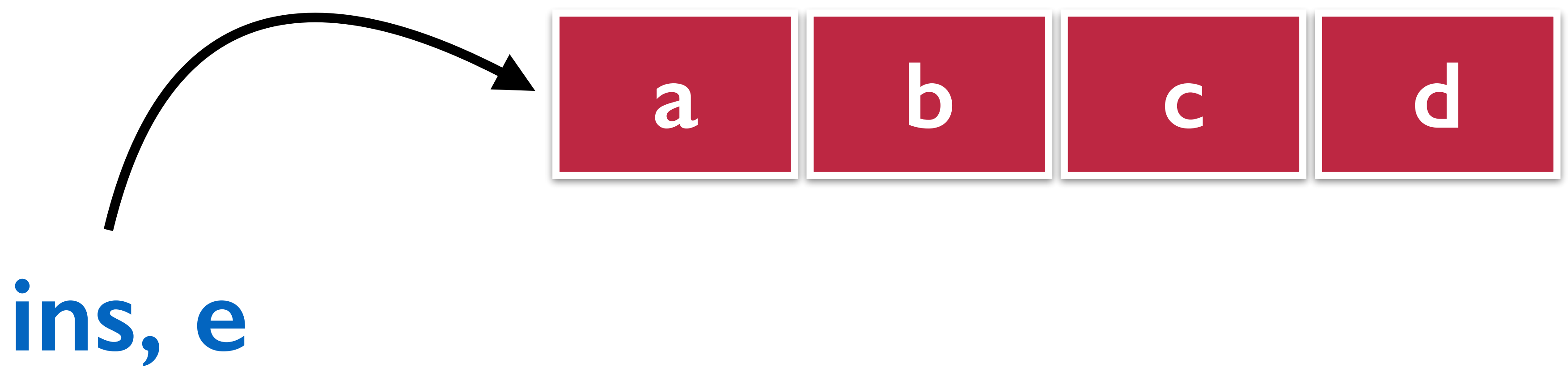
ICFP 18

Work done by Queue

Count the total number of messages!

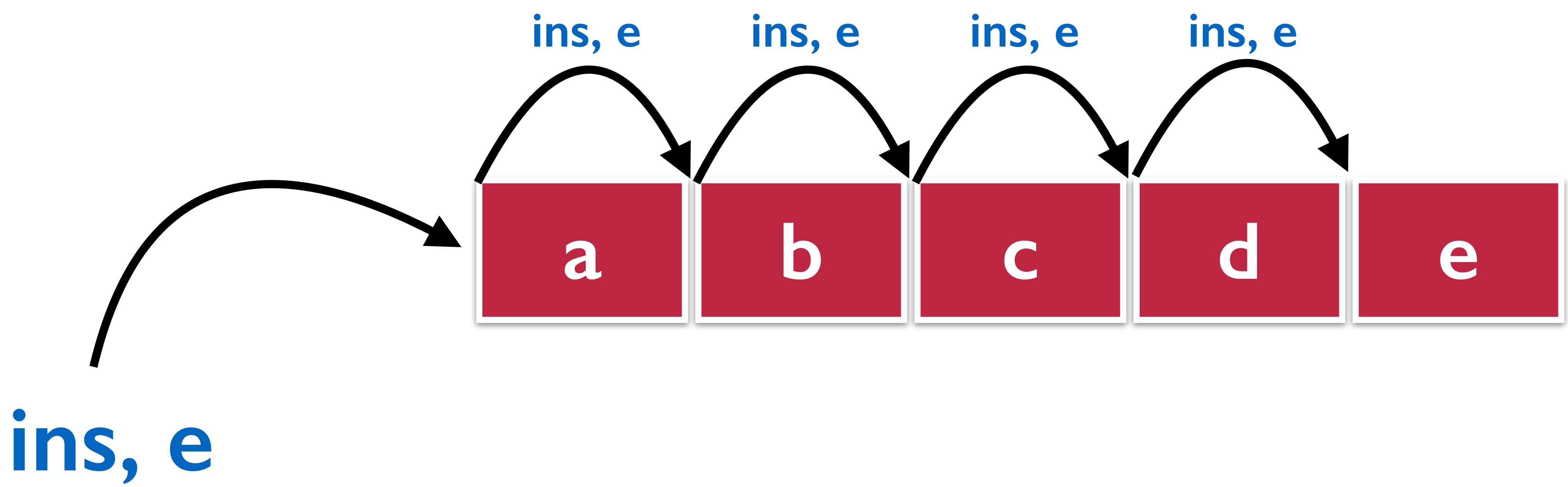
Work done by Queue

Count the total number of messages!



Work done by Queue

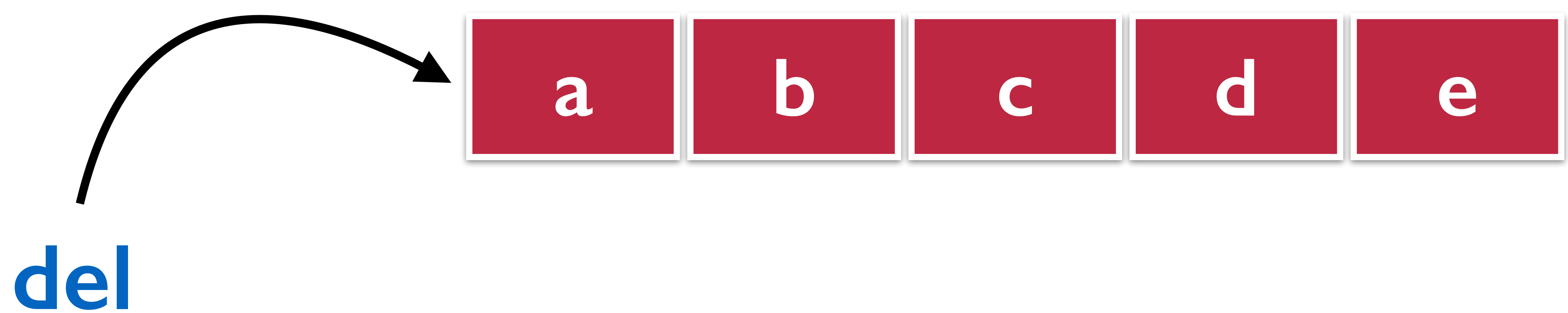
Count the total number of messages!



w_i = Work done to process insertion
= 2n (n is the size of queue)
= 'ins' and 'e' travel to end of queue

Work done by Queue

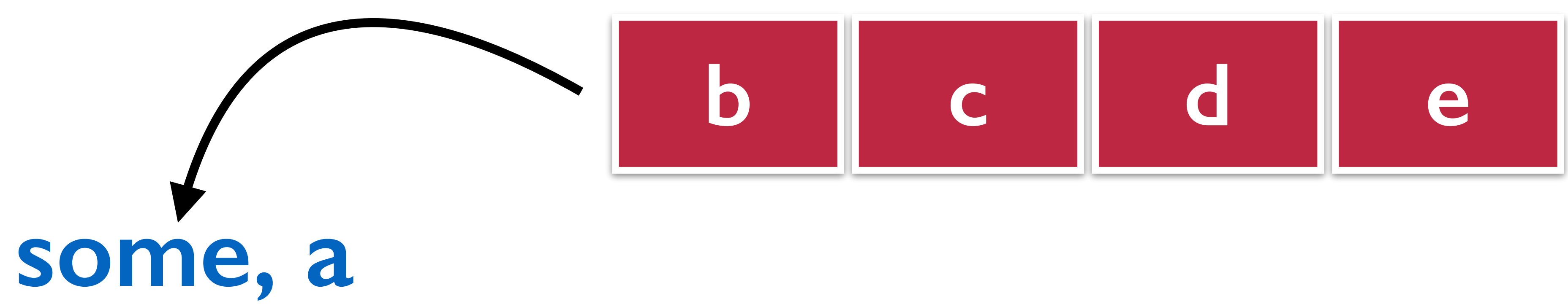
Count the total number of messages!



w_i = Work done to process insertion
= 2n (n is the size of queue)
= 'ins' and 'e' travel to end of queue

Work done by Queue

Count the total number of messages!



w_i = Work done to process insertion
= 2n (n is the size of queue)
= ‘ins’ and ‘e’ travel to end of queue

w_d = Work done to process deletion
= 2 (sends back ‘some’ and ‘a’)

Potential Method

- Processes store potential
- Potential is exchanged via messages
- Potential is consumed to perform 'work'

Potential Method

- Processes store potential
- Potential is exchanged via messages

User defined cost model
This talk: number of messages

- Potential is consumed to perform 'work'

Potential Method

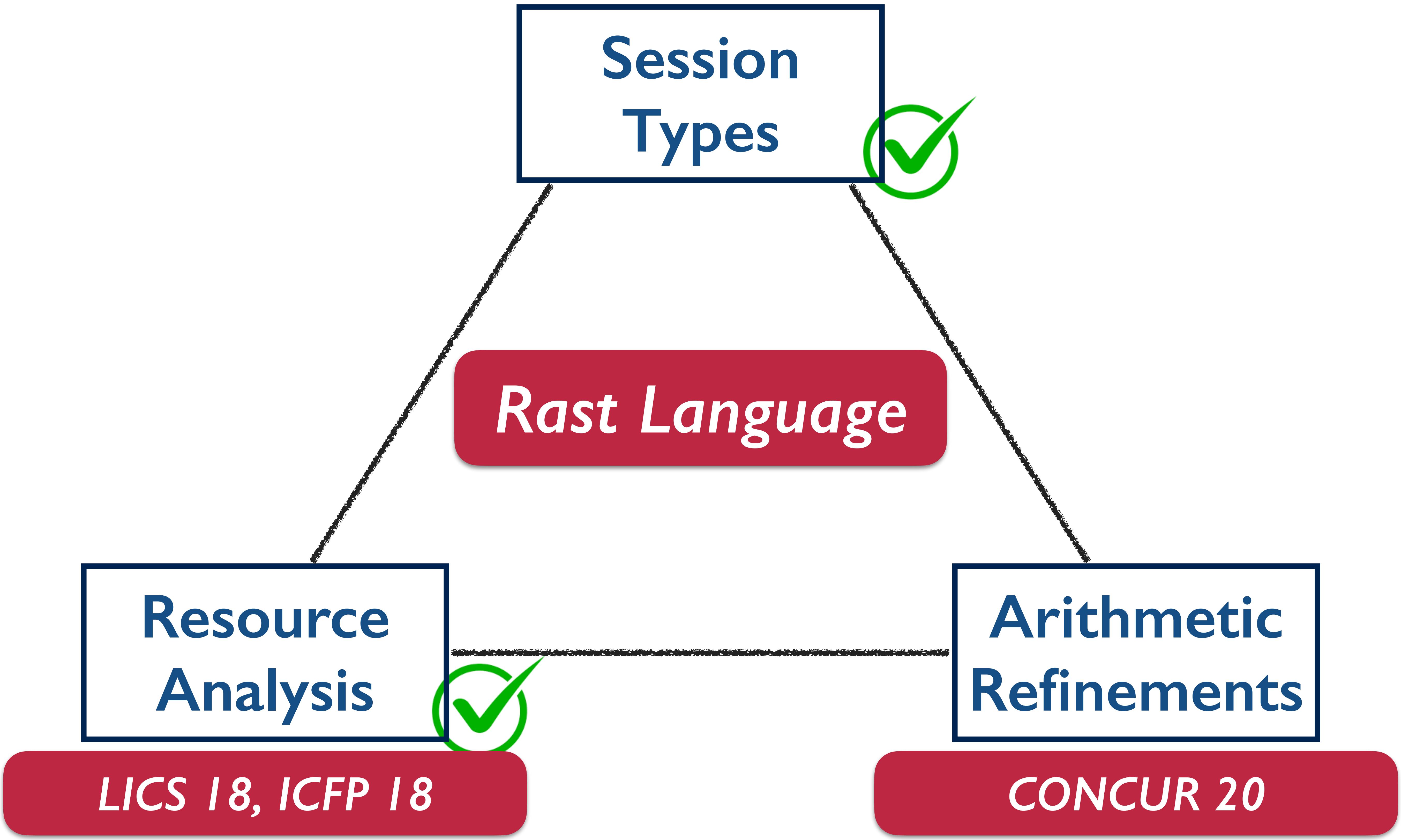
- Processes store potential
- Potential is exchanged via messages

User defined cost model
This talk: number of messages

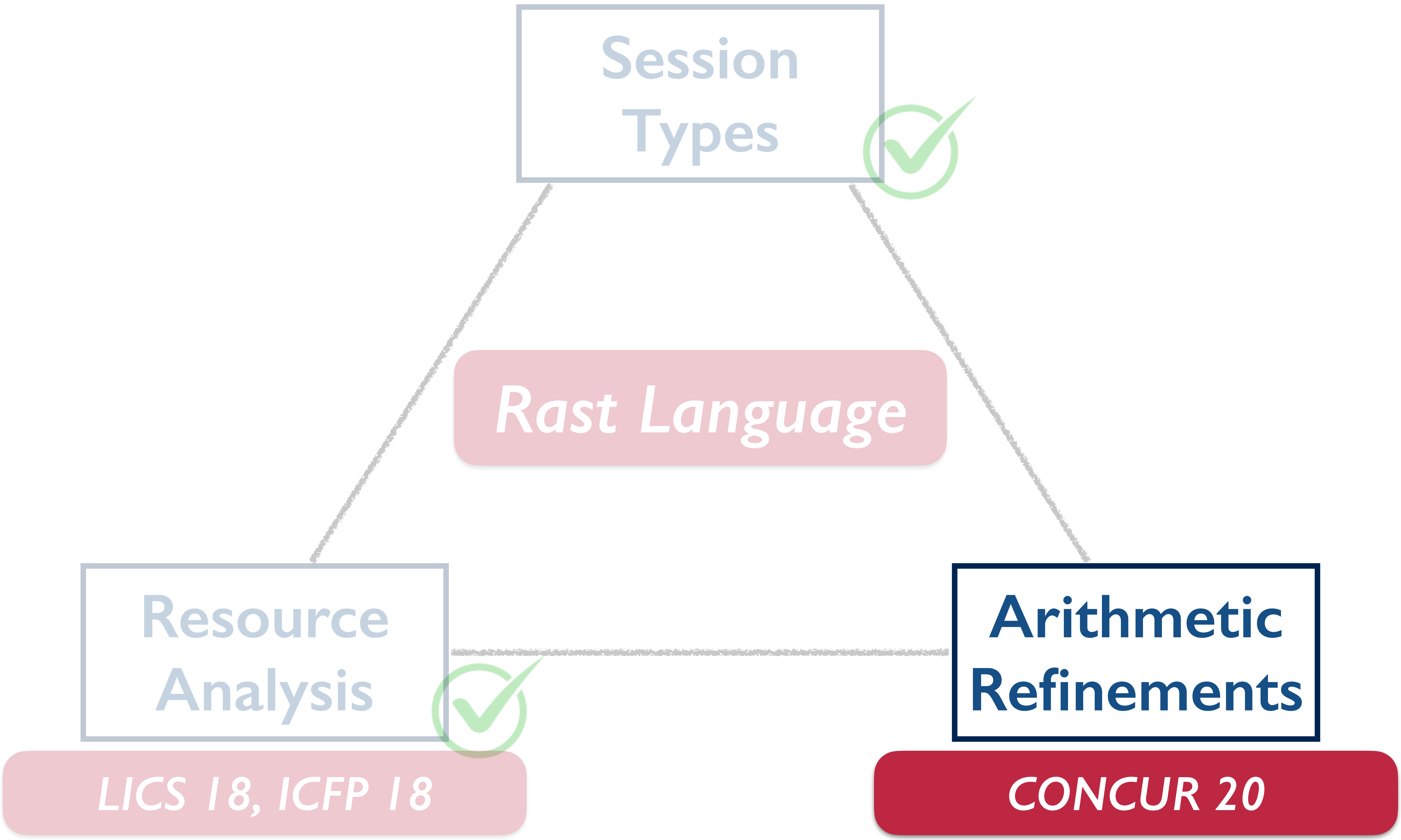
- Potential is consumed to perform 'work'

Insertion: potential needed = 2n
How do you refer to n in the queue type?

Key Features of Rast



Key Features of Rast



Refined Queue Type

$$\begin{aligned}\mathsf{queue}_A[n] = \&\{\mathbf{ins} : A \multimap \mathsf{queue}_A[n+1],\\ \mathbf{del} : \oplus\{\mathbf{none} : ?\{n = 0\}.\, \mathbf{1},\\ \mathbf{some} : ?\{n > 0\}.\, A \otimes \mathsf{queue}_A[n-1]\}\}\end{aligned}$$

Refined Queue Type

$$\begin{aligned}\mathrm{queue}_A[n] = \&\{\mathbf{ins} : A \multimap \mathrm{queue}_A[n+1],\\ \mathbf{del} : \oplus\{\mathbf{none} : ?\{n = 0\}.\,\mathbf{1},\\ \mathbf{some} : ?\{n > 0\}.\,A \otimes \mathrm{queue}_A[n-1]\}\}\end{aligned}$$

Index Refinement
(Size of Queue)

Refined Queue Type

$$\begin{aligned}\mathrm{queue}_A[n] = \&\{\mathbf{ins} : A \multimap \mathrm{queue}_A[n+1],\\ \mathbf{del} : \oplus\{\mathbf{none} : ?\{n = 0\}.\,1,\\ \mathbf{some} : ?\{n > 0\}.\,A \otimes \mathrm{queue}_A[n-1]\}\}\end{aligned}$$

Index Refinement (Size of Queue)

Proof Constraints (Sent by queue)

- 'none' branch: send (proof of) constraint $\{n=0\}$
- 'some' branch: send (proof of) constraint $\{n>0\}$
- Only constraints are exchanged, not proofs

Refined Queues in Rast


```
type queue{n} = &{ins : A -o queue{n+1},
                  del : +{none : ?{n = 0}. 1,
                         some : ?{n > 0}. A * queue{n-1}}}

decl empty : . |- (q : queue{0})
decl elem{n | n > 0} : (x : A) (t : queue{n-1}) |- (q : queue{n})
```

```
proc q <- empty =
  case q (
    ins => x <- recv q ;
           t <- empty ;
           q <- elem{1} x t
  | del => q.none ;
           assert q {0 = 0} ;
           close q )
```

```
proc q <- elem{n} x t =
  case q (
    ins => y <- recv q ;
           t.ins ;
           send t y ;
           q <- elem{n+1} x t
  | del => q.some ;
           assert q {n > 0} ;
           send q x ;
           q <-> t )
```

Refined Queues in Rast


```
type queue{n} = &{ins : A -o queue{n+1},
                  del : +{none : ?{n = 0}. 1,
                         some : ?{n > 0}. A * queue{n-1}}}

decl empty : . |- (q : queue{0})
decl elem{n | n > 0} : (x : A) (t : queue{n-1}) |- (q : queue{n})
```

```
proc q <- empty =
  case q (
    ins => x <- recv q ;
           t <- empty ;
           q <- elem{1} x t
  | del => q.none ;
           assert q {0 = 0} ;
           close q )
```

send constraint

```
proc q <- elem{n} x t =
  case q (
    ins => y <- recv q ;
           t.ins ;
           send t y ;
           q <- elem{n+1} x t
  | del => q.some ;
           assert q {n > 0} ;
           send q x ;
           q <-> t )
```

Refined Queues in Rast


```
type queue{n} = &{ins : A -o queue{n+1},
                  del : +{none : ?{n = 0}. 1,
                         some : ?{n > 0}. A * queue{n-1}}}

decl empty : . |- (q : queue{0})
decl elem{n | n > 0} : (x : A) (t : queue{n-1}) |- (q : queue{n})
```

```
proc q <- empty =
  case q (
    ins => x <- recv q ;
           t <- empty ;
           q <- elem{1} x t
  | del => q.none ;
           assert q {0 = 0} ;
           close q )
```

```
proc q <- elem{n} x t =
  case q (
    ins => y <- recv q ;
           t.ins ;
           send t y ;
           q <- elem{n+1} x t
  | del => q.some ;
           assert q {n > 0} ;
           send q x ;
           q <-> t )
```

send constraint

Ergometric Queue Type

$$
\begin{aligned}
\mathrm{queue}_A[n] = \&\{\mathbf{ins} : {\color{red}\triangleleft^{2n}}(A \multimap \mathrm{queue}_A[n+1]),\\
\mathbf{del} : {\color{red}\triangleleft^{2}} \oplus \{\mathbf{none} : ?\{n = 0\}.\, \mathbf{1},\\
\mathbf{some} : ?\{n > 0\}.\, A \otimes \mathrm{queue}_A[n-1]\}\}
\end{aligned}
$$

Ergometric Queue Type

$$\begin{aligned}\text{queue}_A[n] = \&\{\textbf{ins} : \triangleleft^{2n}(A \multimap \text{queue}_A[n+1]),\\ \textbf{del} : \triangleleft^{2} \oplus \{\textbf{none} : ?\{n = 0\}.\, \mathbf{1},\\ \textbf{some} : ?\{n > 0\}.\, A \otimes \text{queue}_A[n-1]\}\}\end{aligned}$$

Potential Annotations

Ergometric Queue Type

$$\begin{aligned}\mathsf{queue}_A[n] = \&\{\mathbf{ins} : \triangleleft^{2n}(A \multimap \mathsf{queue}_A[n+1]),\\ \mathbf{del} : \triangleleft^{2} \oplus \{\mathbf{none} : ?\{n = 0\}.\, \mathbf{1},\\ \mathbf{some} : ?\{n > 0\}.\, A \otimes \mathsf{queue}_A[n-1]\}\}\end{aligned}$$

Potential Annotations

- receive 2n units of potential after ‘ins’
- receive 2 units of potential after ‘del’
- potential is consumed to exchange messages

Ergometric Queue in Rast


```
type queue{n} = &{ins : <{2*n}| A -o queue{n+1},
                  del : <{2}| +{none : ?{n = 0}. 1,
                               some : ?{n > 0}. A * queue{n-1}}}
```

```
proc q <- elem{n} x t =
  case q (
    ins => get q {2*n} ;
           y <- recv q ;
           t.ins ;
           pay t {2*(n-1)} ;
           send t y ;
           q <- elem{n+1} x t
  | del => get q {2} ;
           q.some ;
           assert q {n > 0} ;
           send q x ;
           q <-> t )
```

Ergometric Queue in Rast


```
type queue{n} = &{ins : <{2*n}| A -o queue{n+1},
                  del : <{2}| +{none : ?{n = 0}. 1,
                               some : ?{n > 0}. A * queue{n-1}}}
```

```
proc q <- elem{n} x t =
  case q (
    ins => get q {2*n} ;
           y <- recv q ;
           t.ins ;
           pay t {2*(n-1)} ;
           send t y ;
           q <- elem{n+1} x t
  | del => get q {2} ;
           q.some ;
           assert q {n > 0} ;
           send q x ;
           q <-> t )
```

unit cost of sending a message

Ergometric Queue in Rast


```
type queue{n} = &{ins : <{2*n}| A -o queue{n+1},
                  del : <{2}| +{none : ?{n = 0}. 1,
                               some : ?{n > 0}. A * queue{n-1}}}
```

```
proc q <- elem{n} x t =
  case q (
    ins => get q {2*n} ;
           y <- recv q ;
           t.ins ;
           pay t {2*(n-1)} ;
           send t y ;
           q <- elem{n+1} x t
  | del => get q {2} ;
           q.some ;
           assert q {n > 0} ;
           send q x ;
           q <-> t )
```

unit cost of sending a message

get 2n units of potential

Ergometric Queue in Rast


```
type queue{n} = &{ins : <{2*n}| A -o queue{n+1},
                  del : <{2}| +{none : ?{n = 0}. 1,
                                some : ?{n > 0}. A * queue{n-1}}}
```

```
proc q <- elem{n} x t =
  case q (
    ins => get q {2*n} ;
           y <- recv q ;
           t.ins ;
           pay t {2*(n-1)} ;
           send t y ;
           q <- elem{n+1} x t
  | del => get q {2} ;
           q.some ;
           assert q {n > 0} ;
           send q x ;
           q <-> t )
```

unit cost of sending a message

get 2n units of potential

pay 2(n-1) units of potential

Ergometric Queue in Rast


```
type queue{n} = &{ins : <{2*n}| A -o queue{n+1},
                  del : <{2}| +{none : ?{n = 0}. 1,
                               some : ?{n > 0}. A * queue{n-1}}}
```

```
proc q <- elem{n} x t =
  case q (
    ins => get q {2*n} ;
           y <- recv q ;
           t.ins ;
           pay t {2*(n-1)} ;
           send t y ;
           q <- elem{n+1} x t
  | del => get q {2} ;
           q.some ;
           assert q {n > 0} ;
           send q x ;
           q <-> t )
```

unit cost of sending a message

get 2n units of potential

pay 2(n-1) units of potential

cost of 2 for sending 2 msgs

Ergometric Queue in Rast


```
type queue{n} = &{ins : <{2*n}| A -o queue{n+1},
                  del : <{2}| +{none : ?{n = 0}. 1,
                              some : ?{n > 0}. A * queue{n-1}}}
```

```
proc q <- elem{n} x t =
  case q (
    ins => get q {2*n} ;
           y <- recv q ;
           t.ins ;
           pay t {2*(n-1)} ;
           send t y ;
           q <- elem{n+1} x t
  | del => get q {2} ;
           q.some ;
           assert q {n > 0} ;
           send q x ;
           q <-> t )
```

unit cost of sending a message

get 2n units of potential

pay 2(n-1) units of potential

cost of 2 for sending 2 msgs

get 2 units of potential

Ergometric Queue in Rast


```
type queue{n} = &{ins : <{2*n}| A -o queue{n+1},
                  del : <{2}| +{none : ?{n = 0}. 1,
                                some : ?{n > 0}. A * queue{n-1}}}
```

unit cost of sending a message

```
proc q <- elem{n} x t =
  case q (
    ins => get q {2*n} ;
           y <- recv q ;
           t.ins ;
           pay t {2*(n-1)} ;
           send t y ;
           q <- elem{n+1} x t
  | del => get q {2} ;
           q.some ;
           assert q {n > 0} ;
           send q x ;
           q <-> t )
```

get 2n units of potential

pay 2(n-1) units of potential

cost of 2 for sending 2 msgs

get 2 units of potential

cost of 2 for sending 2 msgs

Natural Numbers


```
type nat{n} = +{succ : ?{n > 0}. nat{n-1},
                zero : ?{n = 0}. 1}

decl successor{n} : (x : nat{n}) |- (y : nat{n+1})
decl double{n} : (x : nat{n}) |- (y : nat{2*n})
decl add{m}{n} : (x : nat{m}) (y : nat{n}) |- (z : nat{m+n})
decl predecessor{n | n > 0} : (x : nat{n}) |- (y : nat{n-1})
```

```
proc y <- predecessor{n} x =
  case x (
    succ => assume x {n > 0} ;
            y <-> x
  | zero => assume x {n = 0} ;
            impossible )
```

Natural Numbers

```
type nat{n} = +{succ : ?{n > 0}. nat{n-1},
                zero : ?{n = 0}. 1}

decl successor{n} : (x : nat{n}) |- (y : nat{n+1})
decl double{n} : (x : nat{n}) |- (y : nat{2*n})
decl add{m}{n} : (x : nat{m}) (y : nat{n}) |- (z : nat{m+n})
decl predecessor{n | n > 0} : (x : nat{n}) |- (y : nat{n-1})
```

```
proc y <- predecessor{n} x =
  case x (
    succ => assume x {n > 0} ;
            y <-> x
  | zero => assume x {n = 0} ;
            impossible )
```

receive constraint

Natural Numbers


```
type nat{n} = +{succ : ?{n > 0}. nat{n-1},
                zero : ?{n = 0}. 1}

decl successor{n} : (x : nat{n}) |- (y : nat{n+1})
decl double{n} : (x : nat{n}) |- (y : nat{2*n})
decl add{m}{n} : (x : nat{m}) (y : nat{n}) |- (z : nat{m+n})
decl predecessor{n | n > 0} : (x : nat{n}) |- (y : nat{n-1})
```

```
proc y <- predecessor{n} x =
  case x (
    succ => assume x {n > 0} ;
            y <-> x
  | zero => assume x {n = 0} ;
            impossible )
```

receive constraint

impossible branch

Implicit Syntax

- skip assume, assert, impossible, pay, get
- automatically reconstructed using ‘forcing calculus’
- makes the code compact, enables reuse, reduces programming errors

Implicit Syntax

- skip assume, assert, impossible, pay, get
- automatically reconstructed using ‘forcing calculus’
- makes the code compact, enables reuse, reduces programming errors

```
proc y <- predecessor{n} x =
  case x (
    succ => y <-> x )
```

Implicit Syntax

- skip assume, assert, impossible, pay, get
- automatically reconstructed using ‘forcing calculus’
- makes the code compact, enables reuse, reduces programming errors

```
proc y <- predecessor{n} x =
  case x (
    succ => y <-> x )
```

```
proc y <- predecessor{n} x =
  case x (
    succ => assume x {n > 0} ;
            y <-> x
  | zero => assume x {n = 0} ;
            impossible )
```

Evaluation

Module	iLOC	eLOC	#Defs	R (ms)	T (ms)
arithmetic	69	143	8	0.353	1.325
integers	90	114	8	0.200	1.074
linlam	54	67	6	0.734	4.003
list	244	441	29	1.534	3.419
primes	90	118	8	0.196	1.646
segments	48	65	9	0.239	0.195
ternary	156	235	16	0.550	1.967
theorems	79	141	16	0.361	0.894
tries	147	308	9	1.113	5.283
Total	**977**	**1632**	**109**	**5.280**	**19.806**

The Rast Language

- *Resource-Aware Session Types:* refinement session types with support for verifying *sequential* and *parallel* complexity bounds automatically
- *Lightweight verification* using refinements
- *Reconstruction:* constructs pertaining to refinement layer are inserted automatically
- *Evaluation:* implemented standard benchmarks
- *Availability:* implementation open-source on https://bitbucket.org/fpfenning/rast/src/master/rast/