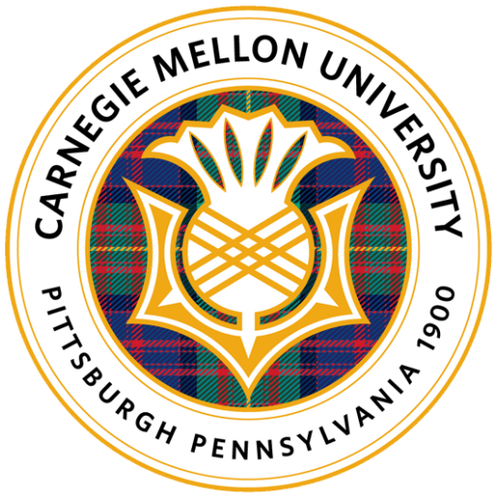
Verified Linear Session-Typed Concurrent Programming

Ankush Das*

Frank Pfenning

Carnegie Mellon University



PPDP 2020

What are Session Types?

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

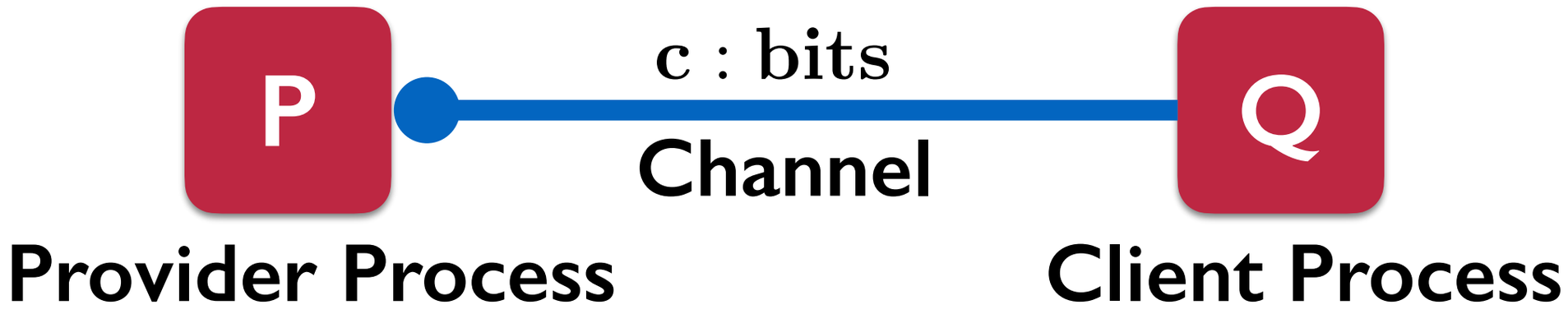
What are Session Types?

- 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

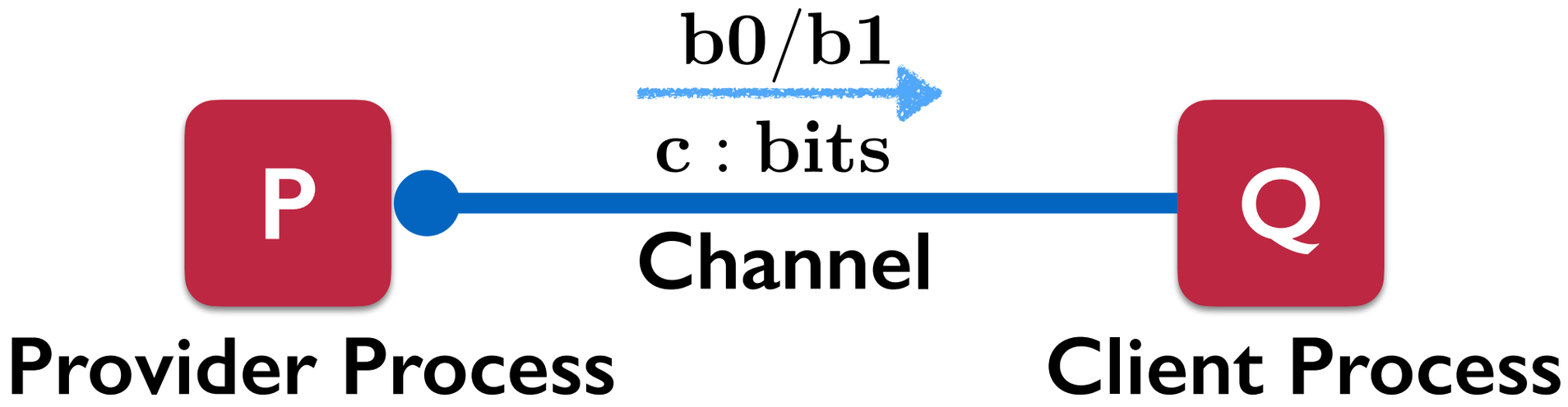
$$\mathbf{bits} = \oplus\{\mathbf{b0} : \mathbf{bits}, \mathbf{b1} : \mathbf{bits}\}$$



What are Session Types?

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

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



Example: Queues

$$\text{queue}_A = \&\{\textbf{ins} : A \multimap \text{queue}_A,$$
$$\textbf{del} : \oplus\{\textbf{none} : \mathbf{1},$$
$$\textbf{some} : A \otimes \text{queue}_A\}\}$$

Example: Queues

offers choice of ins/del

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

Example: Queues

offers choice of ins/del

recv element of type A

$$\text{queue}_A = \&\{\textbf{ins} : A \multimap \text{queue}_A,$$
$$\textbf{del} : \oplus\{\textbf{none} : \mathbf{1},$$
$$\textbf{some} : A \otimes \text{queue}_A\}\}$$

Example: Queues

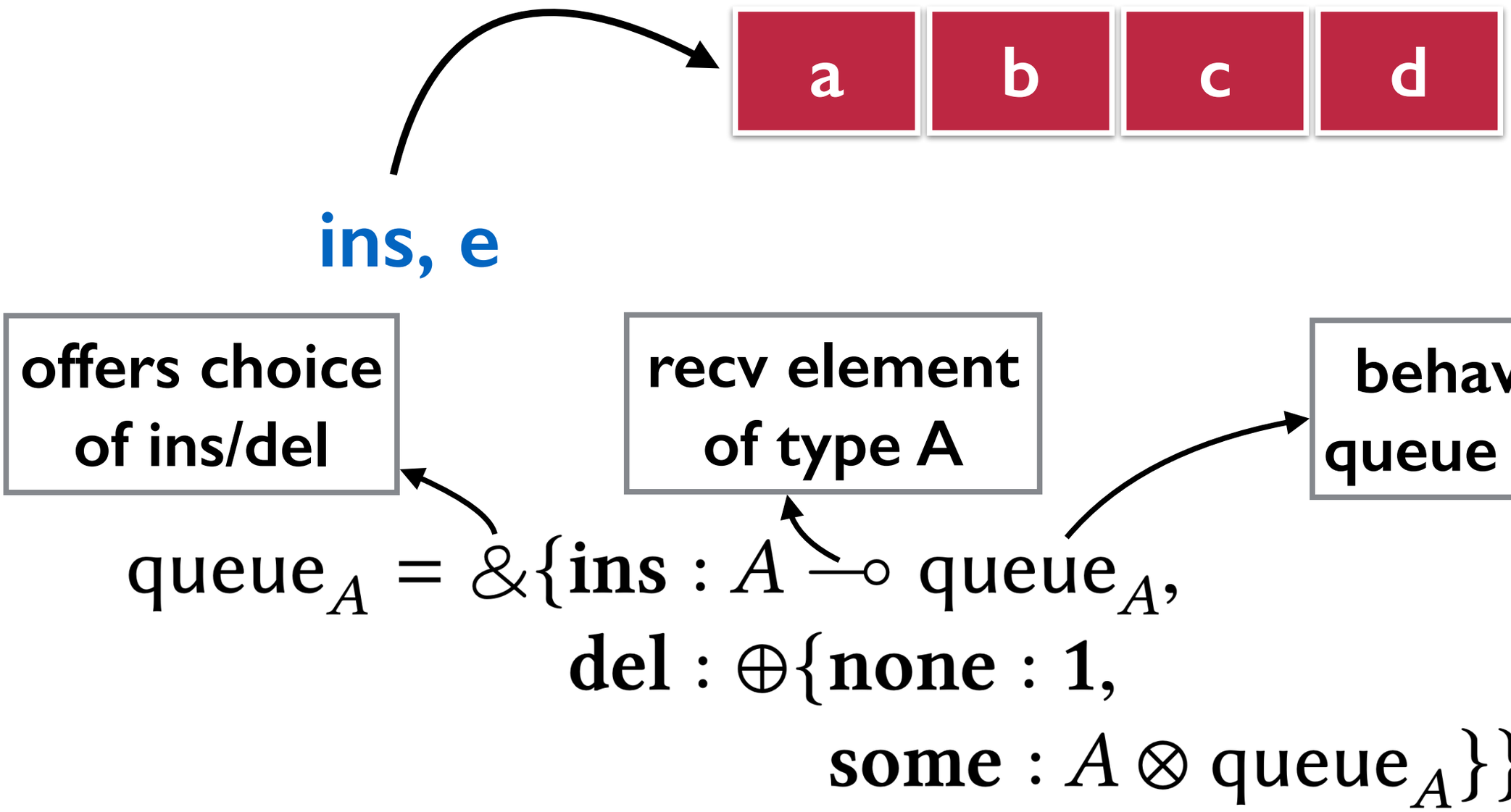
offers choice of ins/del

recv element of type A

behave as queue again

$$\text{queue}_A = \&\{\textbf{ins} : A \multimap \text{queue}_A,$$
$$\textbf{del} : \oplus\{\textbf{none} : \mathbf{1},$$
$$\textbf{some} : A \otimes \text{queue}_A\}\}$$

Example: Queues



$$\text{queue}_A = \&\{\textbf{ins} : A \multimap \text{queue}_A,$$
$$\textbf{del} : \oplus\{\textbf{none} : \mathbf{1},$$
$$\textbf{some} : A \otimes \text{queue}_A\}\}$$

Example: Queues

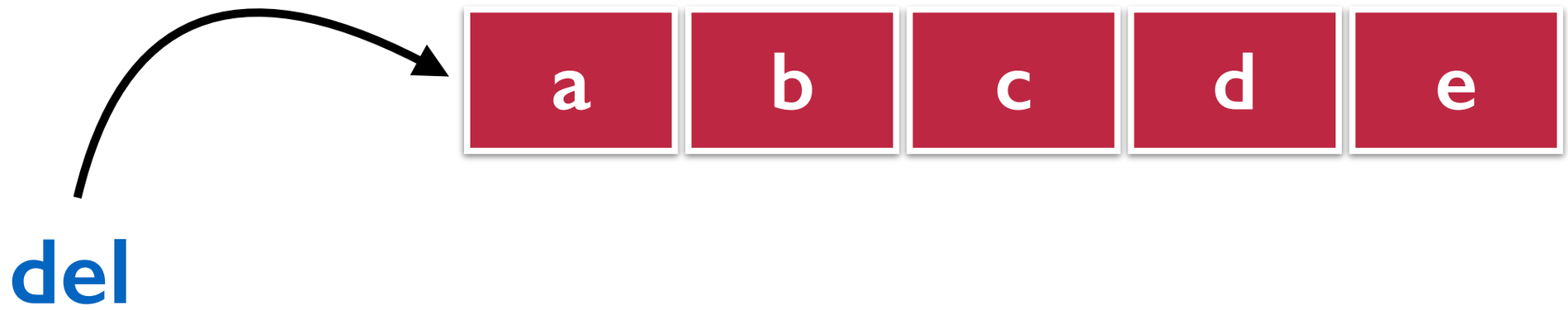
offers choice of ins/del

recv element of type A

behave as queue again

$$\text{queue}_A = \&\{\textbf{ins} : A \multimap \text{queue}_A,$$
$$\textbf{del} : \oplus\{\textbf{none} : 1,$$
$$\textbf{some} : A \otimes \text{queue}_A\}\}$$

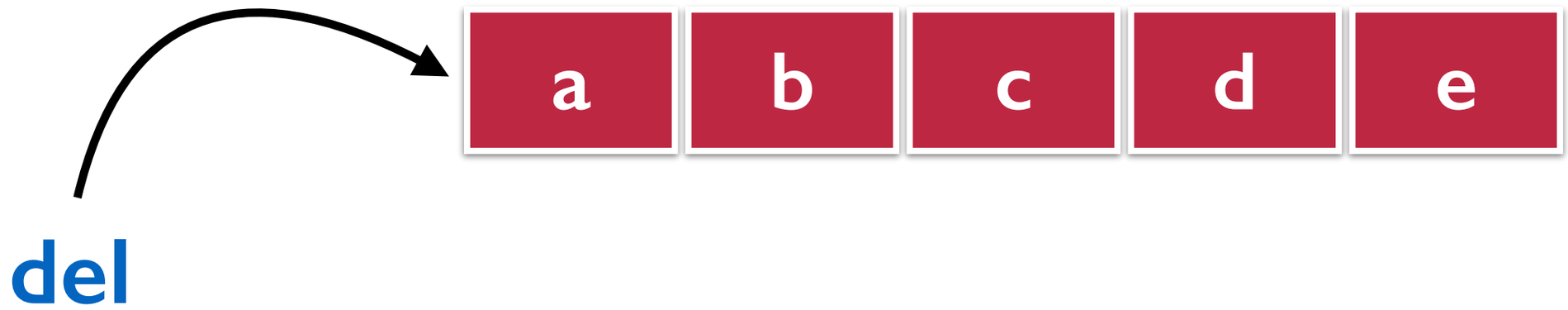
Example: Queues



offers choice of ins/del

$$\text{queue}_A = \&\{\textbf{ins} : A \multimap \text{queue}_A,$$
$$\textbf{del} : \oplus\{\textbf{none} : \mathbf{1},$$
$$\textbf{some} : A \otimes \text{queue}_A\}\}$$

Example: Queues

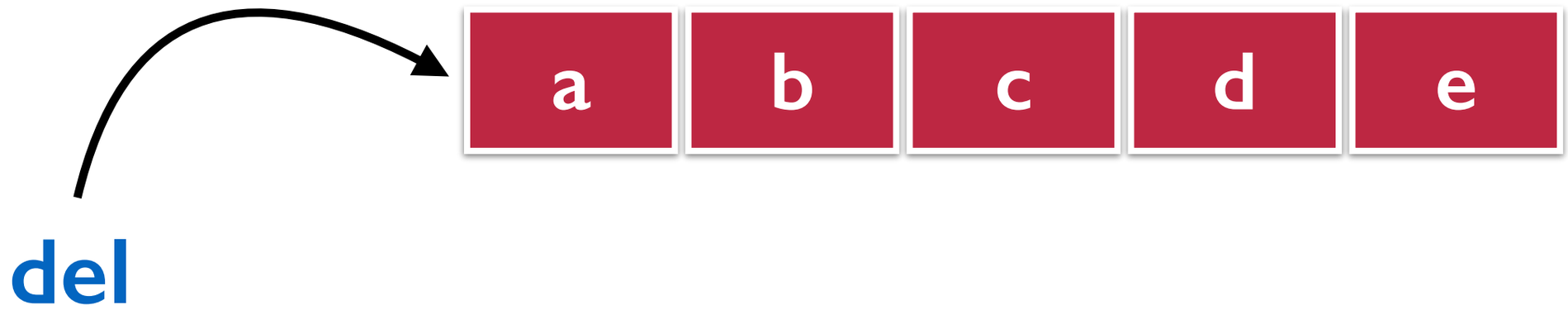


offers choice of ins/del

$$\text{queue}_A = \&\{\textbf{ins} : A \multimap \text{queue}_A,$$
$$\textbf{del} : \oplus\{\textbf{none} : \mathbf{1},$$
$$\textbf{some} : A \otimes \text{queue}_A\}\}$$

send none if queue is empty

Example: Queues



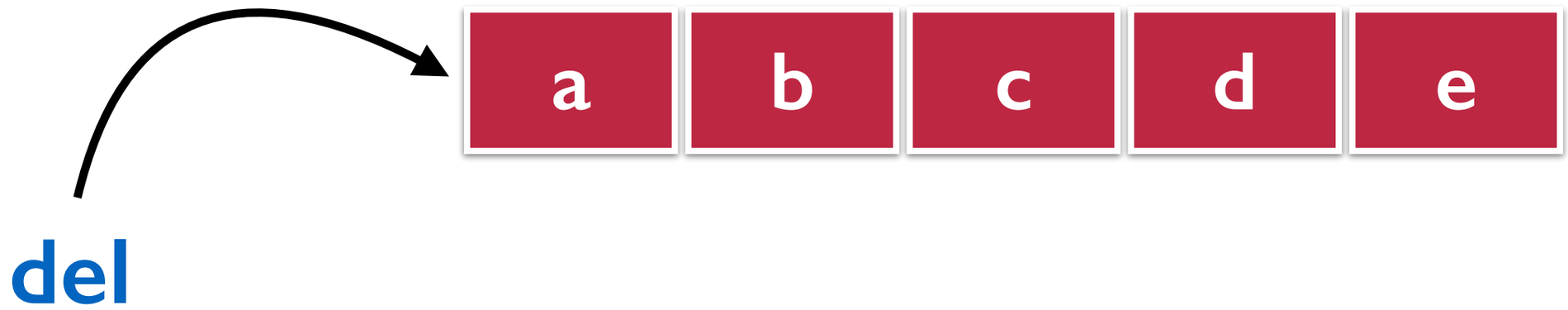
offers choice of ins/del

$$\text{queue}_A = \&\{\textbf{ins} : A \multimap \text{queue}_A,$$
$$\textbf{del} : \oplus\{\textbf{none} : \mathbf{1},$$
$$\textbf{some} : A \otimes \text{queue}_A\}\}$$

terminate

send none if queue is empty

Example: Queues

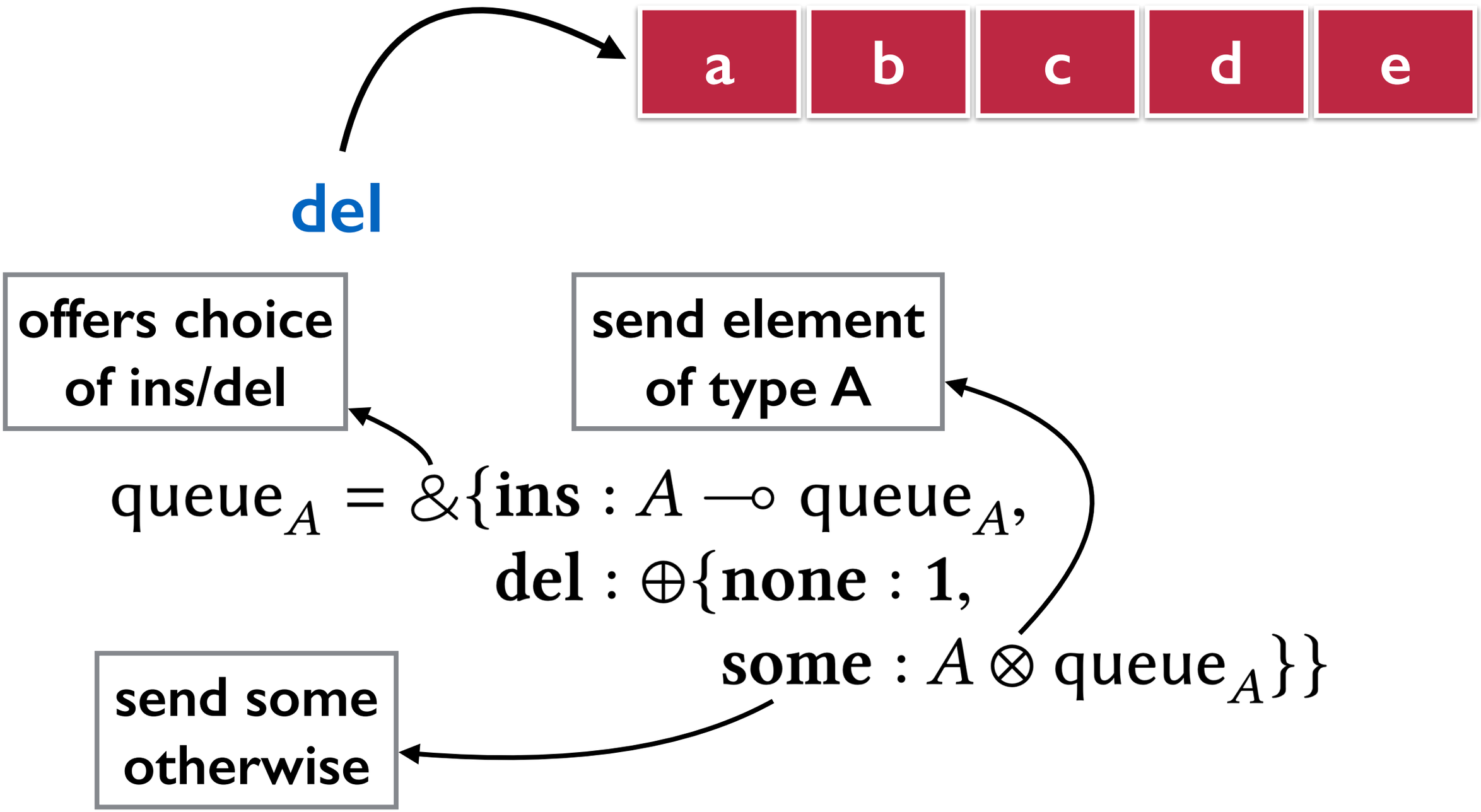


offers choice of ins/del

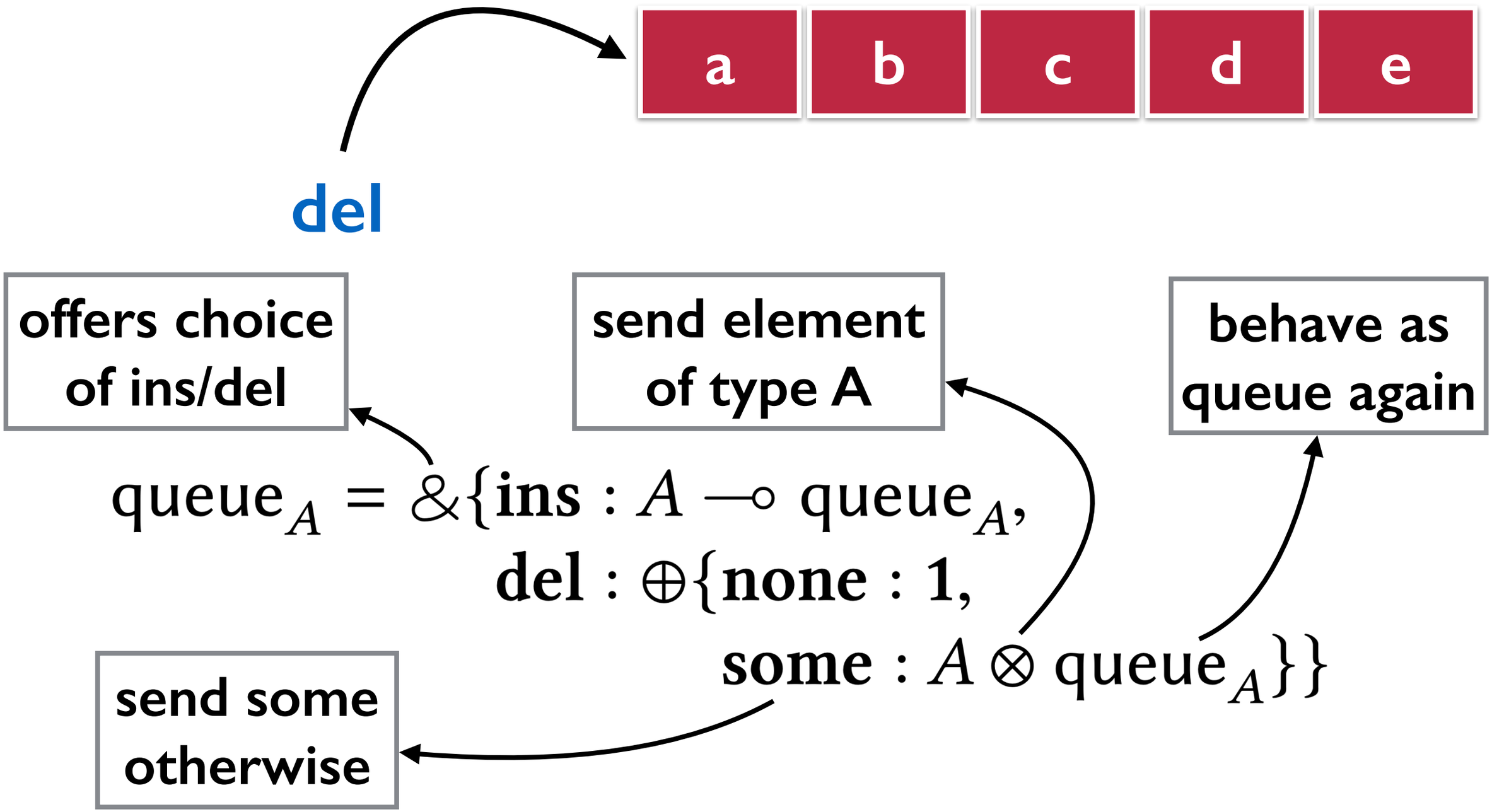
$$\text{queue}_A = \&\{\textbf{ins} : A \multimap \text{queue}_A,$$
$$\textbf{del} : \oplus\{\textbf{none} : \mathbf{1},$$
$$\textbf{some} : A \otimes \text{queue}_A\}\}$$

send some otherwise

Example: Queues



Example: Queues



Example: Queues

some, a

offers choice of ins/del

send element of type A

behave as queue again

$$\text{queue}_A = \&\{\textbf{ins} : A \multimap \text{queue}_A,$$
$$\textbf{del} : \oplus\{\textbf{none} : 1,$$
$$\textbf{some} : A \otimes \text{queue}_A\}\}$$

send some otherwise

Example: Queues

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

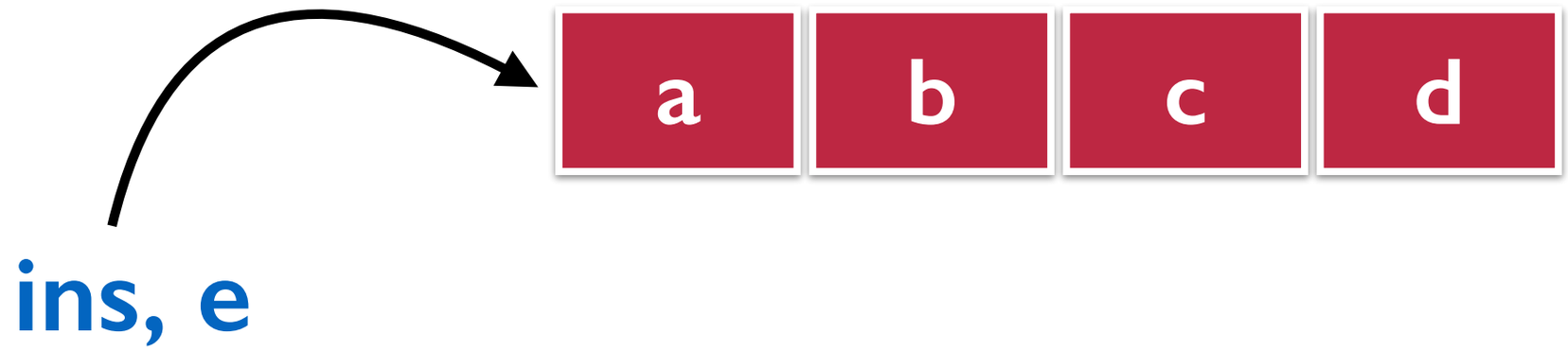
When are the none and some branches taken?
Can the size of queue be encoded in the type?

Work done by Queue

Count the total number of messages!

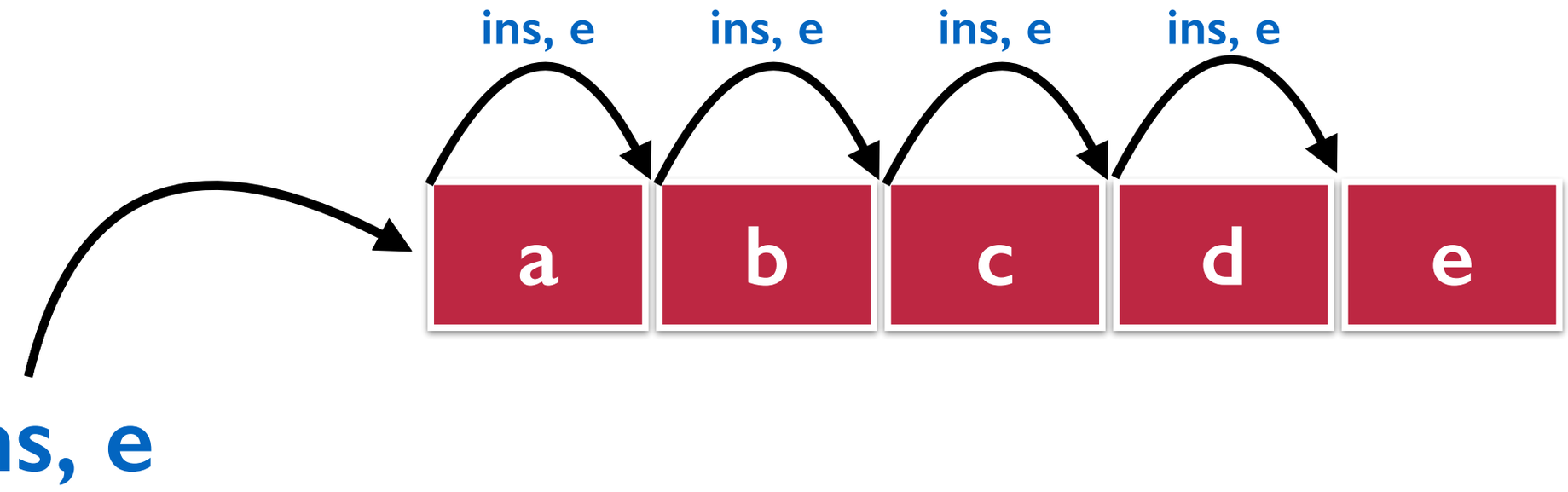
Work done by Queue

Count the total number of messages!



Work done by Queue

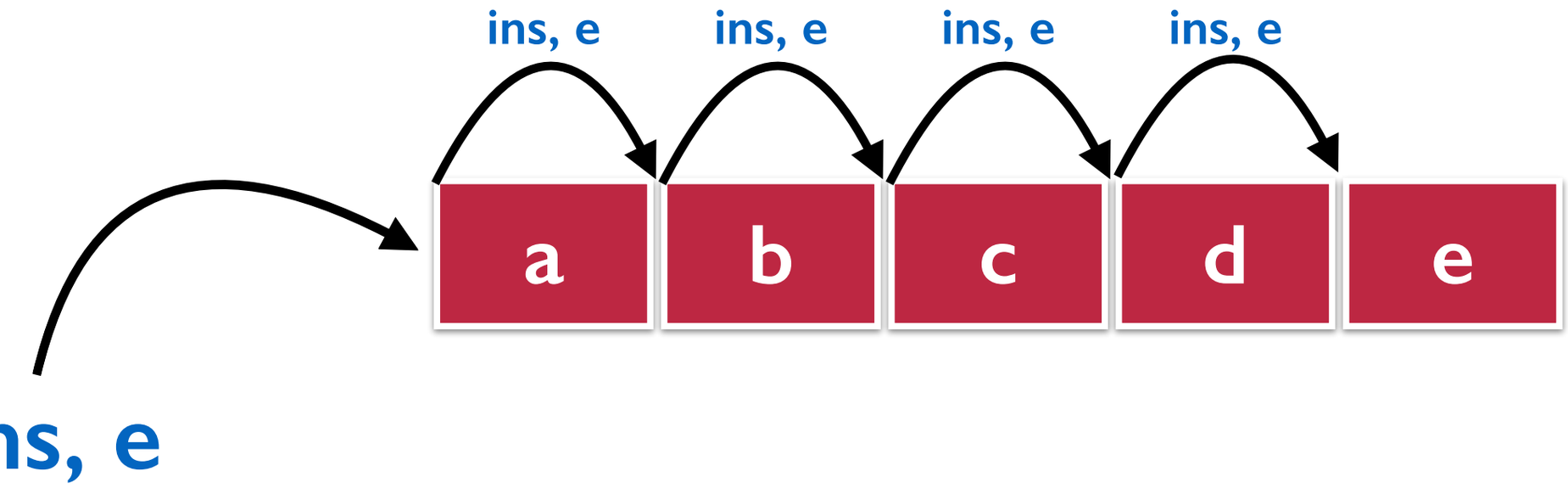
Count the total number of messages!



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

Insertion: How do you refer to n in the queue type?

Refined Queue Type

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

Refined Queue Type

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

Index Refinement (Size of Queue)

Refined Queue Type

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

Index Refinement (Size of Queue)

Proof Constraints (Sent by queue)

- ‘none’ branch: send (proof of) constraint $\{n=0\}$
- ‘some’ branch: send (proof of) constraint $\{n>0\}$
- Domain of constraints: Presburger Arithmetic

Two New Type Operators

Two New Type Operators

?{ϕ}. A

- Provider sends (proof of) ϕ, then continues to provide A
- Client receives (proof of) constraint ϕ

Two New Type Operators

?{ϕ}. A

- Provider sends (proof of) ϕ, then continues to provide A
- Client receives (proof of) constraint ϕ

!{ϕ}. A

- Provider receives (proof of) ϕ, then continues to provide A
- Client sends (proof of) constraint ϕ

Two New Type Operators

?{ϕ}. A

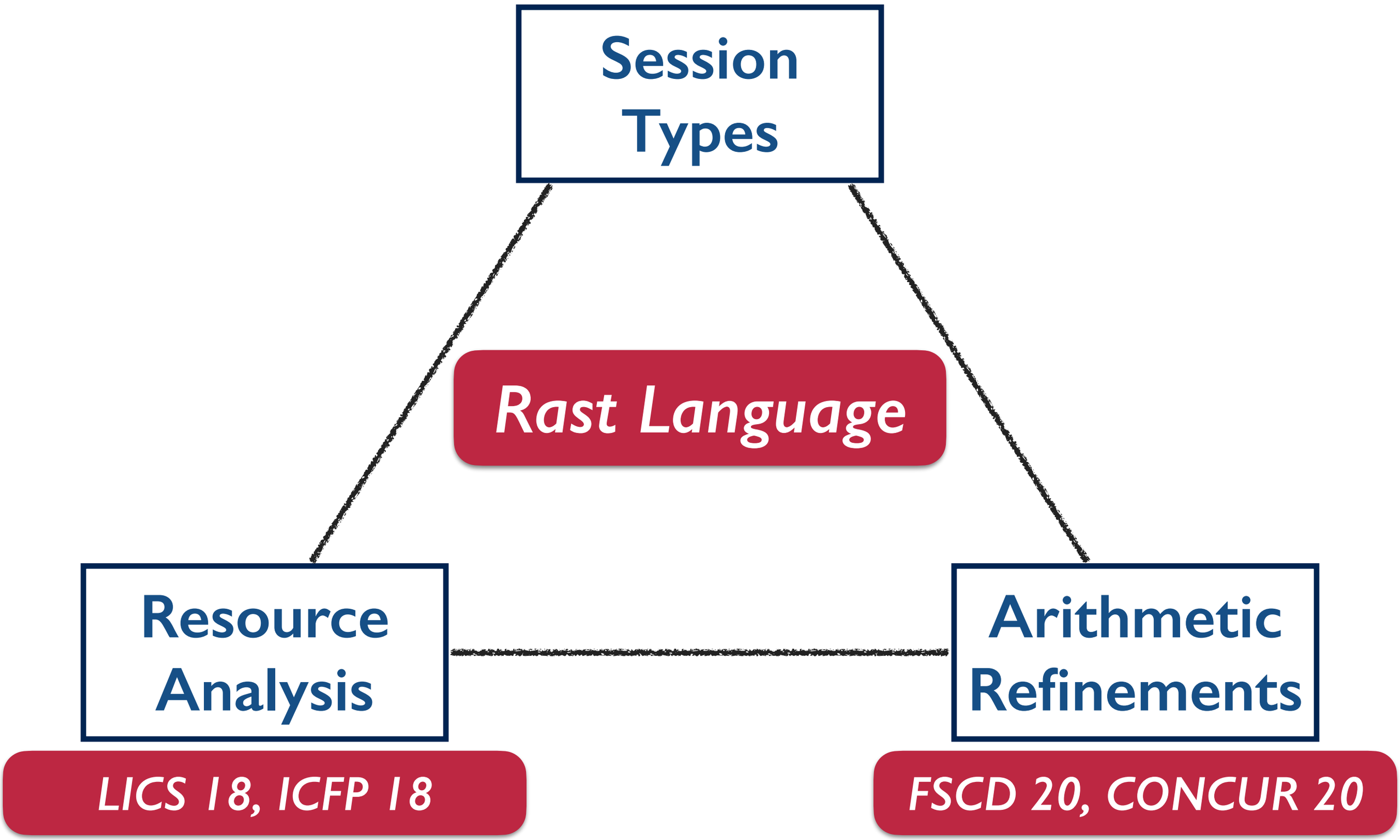
- Provider sends (proof of) ϕ, then continues to provide A
- Client receives (proof of) constraint ϕ

!{ϕ}. A

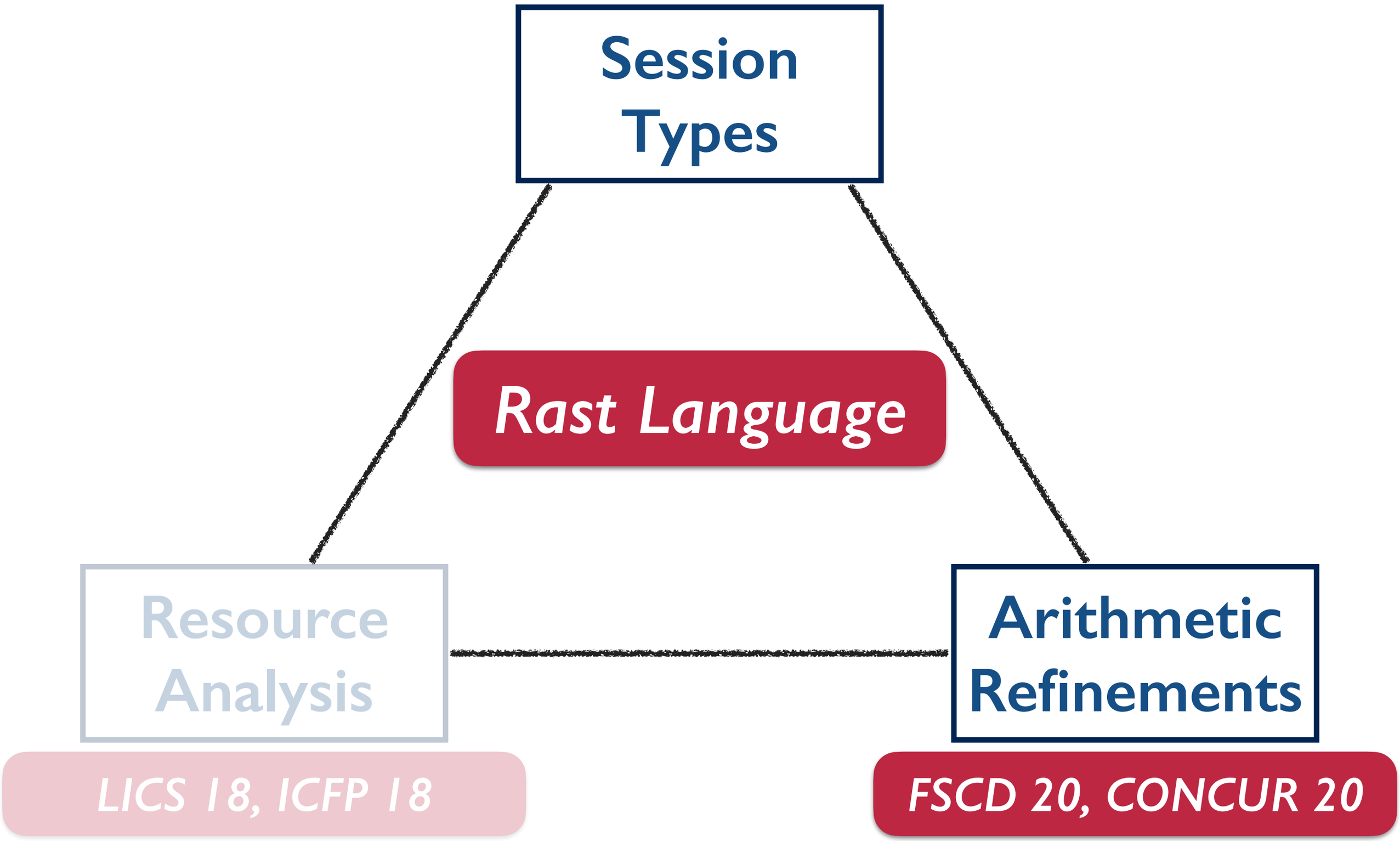
- Provider receives (proof of) ϕ, then continues to provide A
- Client sends (proof of) constraint ϕ

Since Presburger arithmetic is decidable and only closed programs are executed, no need to exchange proofs/constraints at runtime

Contributions



Contributions



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)
```

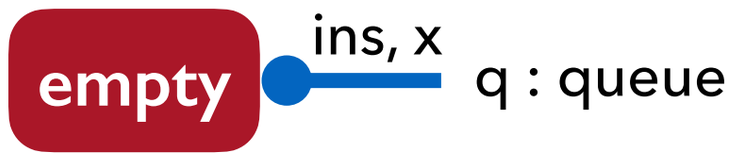
```
proc q <- empty =
  case q (
    ins => x <- recv q ;
           t <- empty ;
           q <- elem x t
  | del => q.none ;
           close q )
```

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 )
```

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 )
```

```
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 )
```

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

Key Observation

sending a proof corresponds to an *assertion*
receiving a proof corresponds to an *assumption*

Two New Constructs

assert x {ϕ}: send constraint ϕ along channel x

assume x {ϕ}: receive constraint ϕ along channel x

Formal Typing Judgment

$$\mathcal{V} \; ; \; C \; ; \; \Delta \vdash_{\Sigma} P :: (x : A)$$

Process P uses channels in Δ and offers channel $x : A$ under free variables $\mathcal{V}$ satisfying constraint C

Formal Typing Judgment

Free variables

$$\mathcal{V} \; ; \; C \; ; \; \Delta \vdash_{\Sigma} P :: (x : A)$$

Process P uses channels in Δ and offers channel $x : A$ under free variables $\mathcal{V}$ satisfying constraint C

Formal Typing Judgment

Constraint satisfied by V

Free variables

$$\mathcal{V} \; ; \; C \; ; \; \Delta \vdash_{\Sigma} P :: (x : A)$$

Process P uses channels in Δ and offers channel $x : A$ under free variables $\mathcal{V}$ satisfying constraint C

Formal Typing Judgment

Constraint satisfied by V

Free variables

Context (Channel Types)

$$\mathcal{V} \; ; \; C \; ; \; \Delta \vdash_{\Sigma} P :: (x : A)$$

Process P uses channels in Δ and offers channel $x : A$ under free variables $\mathcal{V}$ satisfying constraint C

Formal Typing Judgment

Free variables

Constraint satisfied by V

Context (Channel Types)

$$\mathcal{V} \; ; \; C \; ; \; \Delta \vdash_{\Sigma} P :: (x : A)$$

Process Expression

Process P uses channels in Δ and offers channel $x : A$ under free variables $\mathcal{V}$ satisfying constraint C

Formal Typing Judgment

Constraint satisfied by V

Free variables

Context (Channel Types)

$$\mathcal{V} \; ; \; C \; ; \; \Delta \vdash_{\Sigma} P :: (x : A)$$

Process Expression

Offered Channel Type

Process P uses channels in Δ and offers channel $x : A$ under free variables $\mathcal{V}$ satisfying constraint C

Typing Rules

$$\frac{\mathcal{V} \;;\; C \vDash \phi \quad \mathcal{V} \;;\; C \;;\; \Delta \vdash P :: (x : A)}{\mathcal{V} \;;\; C \;;\; \Delta \vdash \mathsf{assert}\ x\ \{\phi\} \;;\; P :: (x : ?\{\phi\}.\, A)}\ ?R$$

$$\frac{\mathcal{V} \;;\; C \wedge \phi \;;\; \Delta, (x : A) \vdash Q :: (z : C)}{\mathcal{V} \;;\; C \;;\; \Delta, (x : ?\{\phi\}.\, A) \vdash \mathsf{assume}\ x\ \{\phi\} \;;\; Q :: (z : C)}\ ?L$$

Typing Rules

$\mathcal{C}$ proves ϕ

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \vDash \phi \quad \mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash P :: (x : A)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta \vdash \mathsf{assert}\ x\ \{\phi\}\ ;\ P :: (x : ?\{\phi\}.\, A)}\ ?R$$

$$\frac{\mathcal{V}\ ;\ \mathcal{C} \wedge \phi\ ;\ \Delta, (x : A) \vdash Q :: (z : C)}{\mathcal{V}\ ;\ \mathcal{C}\ ;\ \Delta, (x : ?\{\phi\}.\, A) \vdash \mathsf{assume}\ x\ \{\phi\}\ ;\ Q :: (z : C)}\ ?L$$

Typing Rules

C proves ϕ

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \vDash \phi \quad \mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta \vdash P :: (x : A)}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta \vdash \text{assert } x \; \{\phi\} \; ; \; P :: (x : ?\{\phi\}.\, A)} \; ?R$$

Add ϕ to C

$$\frac{\mathcal{V} \; ; \; \mathcal{C} \wedge \phi \; ; \; \Delta, (x : A) \vdash Q :: (z : C)}{\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta, (x : ?\{\phi\}.\, A) \vdash \text{assume } x \; \{\phi\} \; ; \; Q :: (z : C)} \; ?L$$

Too Many Asserts/Assumes!

$$\mathrm{nat}[n] = \oplus\{\mathbf{succ} : ?\{n > 0\}.\, \mathrm{nat}[n-1],\ \mathbf{zero} : ?\{n = 0\}.\, \mathbf{1}\}$$

$$\mathit{predecessor}[n] : (x : \mathrm{nat}[n+1]) \vdash (y : \mathrm{nat}[n])$$

$$y \leftarrow \mathit{predecessor}[n]\ x =$$
$$\quad \mathrm{case}\ x\ (\mathbf{succ} \Rightarrow y \leftrightarrow x)$$

Too Many Asserts/Assumes!

$$\mathrm{nat}[n] = \oplus\{\mathbf{succ} : ?\{n > 0\}.\, \mathrm{nat}[n-1],\ \mathbf{zero} : ?\{n = 0\}.\, \mathbf{1}\}$$

$$\mathit{predecessor}[n] : (x : \mathrm{nat}[n+1]) \vdash (y : \mathrm{nat}[n])$$

```
y ← predecessor[n] x =
    case x (
      succ ⇒ y ↔ x)
```

```
y ← predecessor[n] x =
    case x (
      succ ⇒ assume x {n + 1 > 0} ;
              y ↔ x
    | zero ⇒ assume x {n + 1 = 0} ;
              impossible)
```

Too Many Asserts/Assumes!

$$\mathrm{nat}[n] = \oplus\{\mathbf{succ} : ?\{n > 0\}.\, \mathrm{nat}[n-1],\ \mathbf{zero} : ?\{n = 0\}.\, \mathbf{1}\}$$

$$\mathit{predecessor}[n] : (x : \mathrm{nat}[n+1]) \vdash (y : \mathrm{nat}[n])$$

$$y \leftarrow \mathit{predecessor}[n]\ x =$$
$$\quad \mathrm{case}\ x\ ($$
$$\quad\quad \mathbf{succ} \Rightarrow y \leftrightarrow x)$$

$$y \leftarrow \mathit{predecessor}[n]\ x =$$
$$\quad \mathrm{case}\ x\ ($$
$$\quad\quad \mathbf{succ} \Rightarrow \mathrm{assume}\ x\ \{n+1 > 0\}\ ;$$
$$\quad\quad\quad\quad y \leftrightarrow x$$
$$\quad\quad \mid \mathbf{zero} \Rightarrow \mathrm{assume}\ x\ \{n+1 = 0\}\ ;$$
$$\quad\quad\quad\quad \mathrm{impossible})$$

Insert asserts/assumes/impossible automatically?

Program Reconstruction

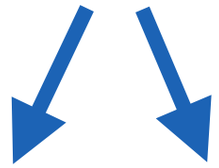
- **Key Idea : Treat constraints as *money* and the reconstruction engine as a *greedy salesperson***
- **Eagerly insert assumes (*get money*) and lazily insert asserts (*pay money*)**
- **Logically: type rules for assumes are invertible, therefore applied eagerly**
- **Formalized using the forcing calculus**

Forcing Calculus

$$\mathcal{V} \; ; \; \mathcal{C} \; ; \; \Delta \vdash_{\Sigma} P :: (x : A)$$

Forcing Calculus

$$\mathcal{V} \; ; \; C \; ; \; \Delta \vdash_{\Sigma} P :: (x : A)$$



$$\mathcal{V} \; ; \; C \; ; \; \Delta \; ; \; \Omega \vdash P :: (x : A)$$

Forcing Calculus

$$\mathcal{V} \; ; \; C \; ; \; \Delta \vdash_{\Sigma} P :: (x : A)$$

$$\mathcal{V} \; ; \; C \; ; \; \Delta \; ; \; \Omega \vdash P :: (x : A)$$

invertible rules applied

invertible rules not applied

- **Δ : linear context, corresponding assumes inserted**
- **Ω : ordered context, assumes to be inserted**

Reconstruction Properties

$$If\, \mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ \Delta \vdash P :: (x : A),\ then\, \mathcal{V}\ ;\ C\ ;\ \Delta\ {}_i\!\vdash P :: (x : A).$$

Soundness

$$If\, \mathcal{V}\ ;\ C\ ;\ \Delta\ {}_i\!\vdash P :: (x : A),\ then\, \mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ \Delta \vdash P :: (x : A).$$

Completeness

Reconstruction Properties

$$If\, \mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ \Delta \vdash P :: (x : A),\ then\, \mathcal{V}\ ;\ C\ ;\ \Delta\ {}_i\!\vdash P :: (x : A).$$

Soundness

$$If\, \mathcal{V}\ ;\ C\ ;\ \Delta\ {}_i\!\vdash P :: (x : A),\ then\, \mathcal{V}\ ;\ C\ ;\ \cdot\ ;\ \Delta \vdash P :: (x : A).$$

Completeness

- **Soundness: the program that the reconstruction engine generates is well-typed!**
- **Completeness: if there is a reconstruction possible, our engine will find it!**

Examples in the Rast Programming Language

List Operations


```
type list{n} = +{cons : ?{n > 0}. A * list{n-1},
                 nil : ?{n = 0}. 1}
```

List Operations


```
type list{n} = +{cons : ?{n > 0}. A * list{n-1},
                 nil : ?{n = 0}. 1}
```

```
decl cons{n} : (x : A) (t : list{n}) |- (l : list{n+1})

decl append{n1}{n2} : (l1 : list{n1}) (l2 : list{n2}) |- (l : list{n1+n2})

decl split{n} : (l : list{2*n}) |- (m : list{n} * list{n})

decl reverse{n} : (l : list{n}) |- (m : list{n})

decl map{n} : (k : list{n}) (m : mapper) |- (l : list{n})

decl filter{n} : (s : selector) (k : list{n}) |- (l : ?m. ?{m <= n}. list{m})
```

Linear λ-Calculus


```
type exp = +{ lam : exp -o exp,
              app : exp * exp }

type val = +{ lam : exp -o exp }
```

Linear λ-Calculus


```
type exp = +{ lam : exp -o exp,
              app : exp * exp }

type val = +{ lam : exp -o exp }
```

expression sizes

```
type exp{n} = +{lam : ?{n > 0}. !n1.exp{n1} -o exp{n1+n-1},
                app : ?n1. ?n2. ?{n = n1+n2+1}. exp{n1} * exp{n2}}

type val{n} = +{lam : ?{n > 0}. !n1.exp{n1} -o exp{n1+n-1}}
```

Linear λ-Calculus


```
type exp = +{ lam : exp -o exp,
              app : exp * exp }

type val = +{ lam : exp -o exp }
```

expression sizes

```
type exp{n} = +{lam : ?{n > 0}. !n1.exp{n1} -o exp{n1+n-1},
                app : ?n1. ?n2. ?{n = n1+n2+1}. exp{n1} * exp{n2}}

type val{n} = +{lam : ?{n > 0}. !n1.exp{n1} -o exp{n1+n-1}}

decl eval{n} : (e : exp{n}) |- (v : ?k. ?{k <= n}. val{k})
```

size of value is smaller

Evaluation

Module	iLOC	eLOC	R (ms)
arithmetic	69	143	0.353
integers	90	114	0.200
linlam	54	67	0.734
list	244	441	1.534
primes	90	118	0.196
segments	48	65	0.239
ternary	156	235	0.550
theorems	79	141	0.361
tries	147	308	1.113
Total	**977**	**1632**	**5.280**

Evaluation

Before Reconstruction

Module	iLOC	eLOC	R (ms)
arithmetic	69	143	0.353
integers	90	114	0.200
linlam	54	67	0.734
list	244	441	1.534
primes	90	118	0.196
segments	48	65	0.239
ternary	156	235	0.550
theorems	79	141	0.361
tries	147	308	1.113
Total	**977**	**1632**	**5.280**

Evaluation

Before Reconstruction | **After Reconstruction**

Module	iLOC	eLOC	R (ms)
arithmetic	69	143	0.353
integers	90	114	0.200
linlam	54	67	0.734
list	244	441	1.534
primes	90	118	0.196
segments	48	65	0.239
ternary	156	235	0.550
theorems	79	141	0.361
tries	147	308	1.113
Total	**977**	**1632**	**5.280**

Evaluation

Before Reconstruction

After Reconstruction

Module	iLOC	eLOC	R (ms)
arithmetic	69	143	0.353
integers	90	114	0.200
linlam	54	67	0.734
list	244	441	1.534
primes	90	118	0.196
segments	48	65	0.239
ternary	156	235	0.550
theorems	79	141	0.361
tries	147	308	1.113
Total	**977**	**1632**	**5.280**

67% increase in lines of code after recon!

Evaluation

Before Reconstruction

After Reconstruction

Module	iLOC	eLOC	R (ms)
arithmetic	69	143	0.353
integers	90	114	0.200
linlam	54	67	0.734
list	244	441	1.534
primes	90	118	0.196
segments	48	65	0.239
ternary	156	235	0.550
theorems	79	141	0.361
tries	147	308	1.113
Total	**977**	**1632**	**5.280**

67% increase in lines of code after recon!

recon is very efficient! <2ms

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/