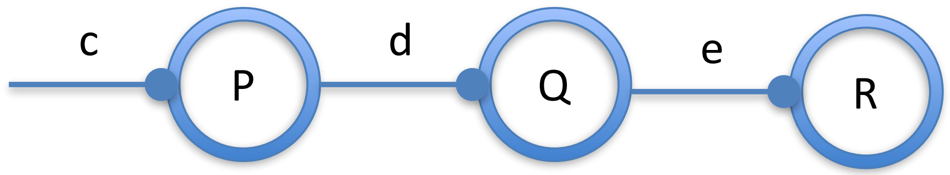
Refinements for Session-typed Concurrency

Josh Acay & Frank Pfenning

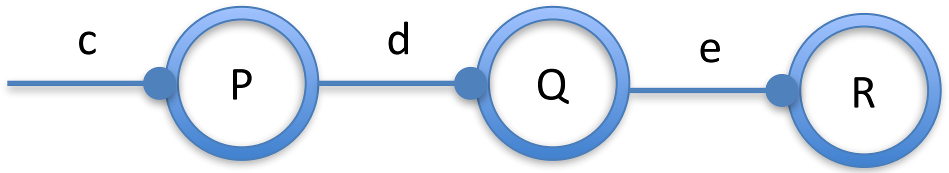
Message-passing Concurrency

- Processes represented as nodes
- Channels between processes as edges
- Each channel is "provided" by a specific process (P provides c, Q provides d etc.)



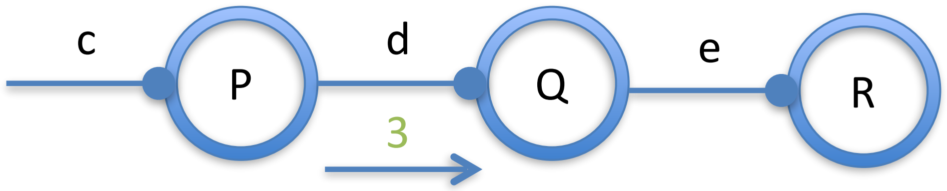
Message-passing Concurrency

- Processes compute internally
- Exchange messages along channels



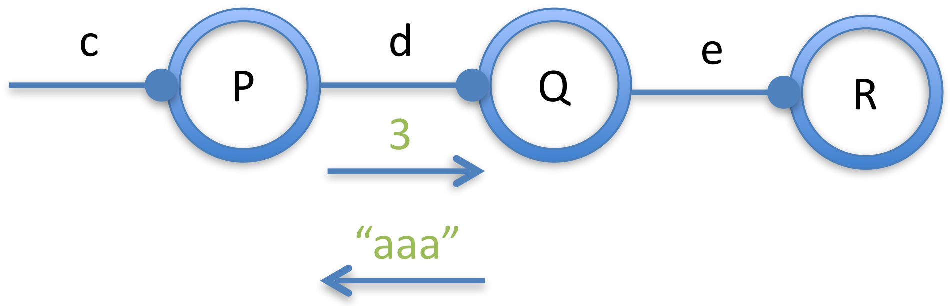
Message-passing Concurrency

- Processes compute internally
- Exchange messages along channels



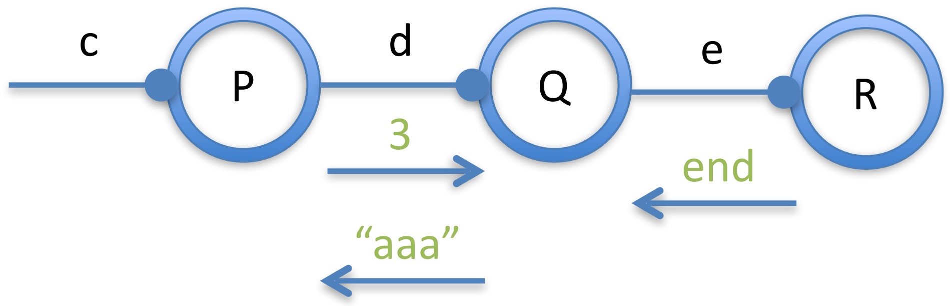
Message-passing Concurrency

- Processes compute internally
- Exchange messages along channels



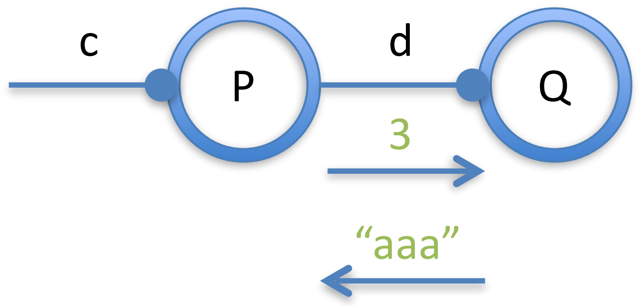
Message-passing Concurrency

- Processes compute internally
- Exchange messages along channels



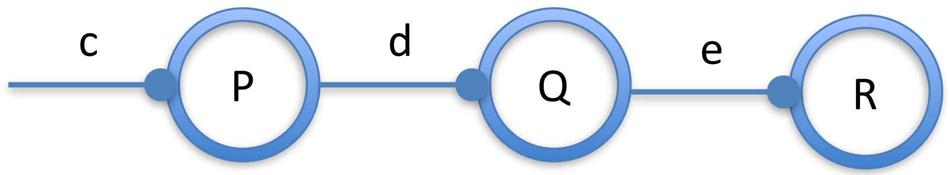
Message-passing Concurrency

- Processes compute internally
- Exchange messages along channels



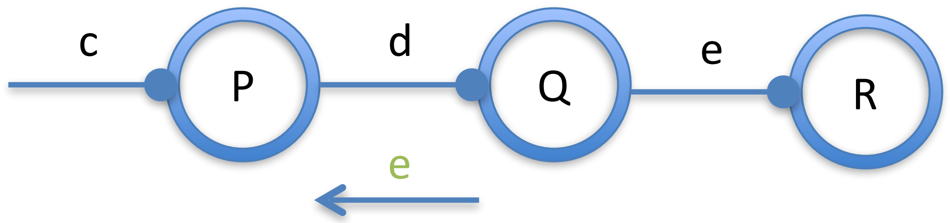
Message-passing Concurrency

- Processes can also send channels they own



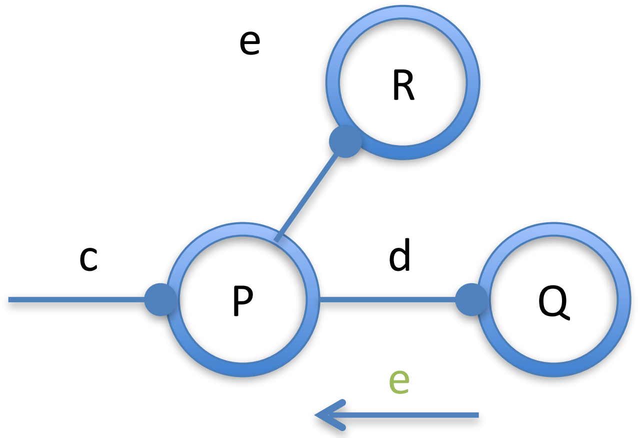
Message-passing Concurrency

- Processes can also send channels they own



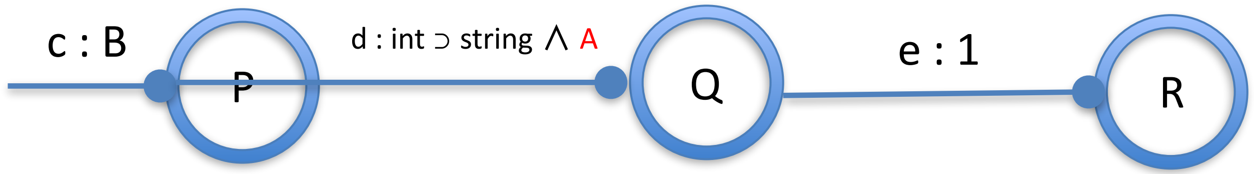
Message-passing Concurrency

- Processes can also send channels they own



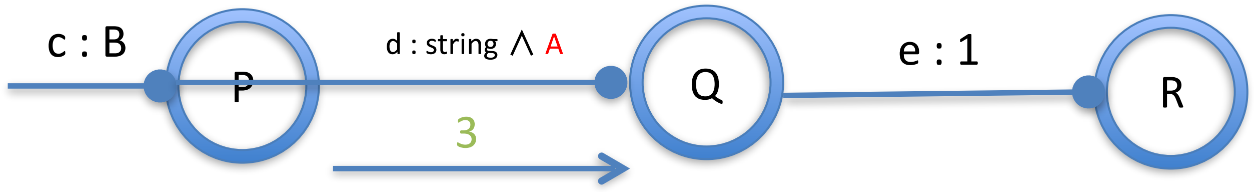
Linear Session-types

- Don't want to send int if expecting string
- Don't try to receive if other process is not sending
- Assign types to each channel from provider's perspective



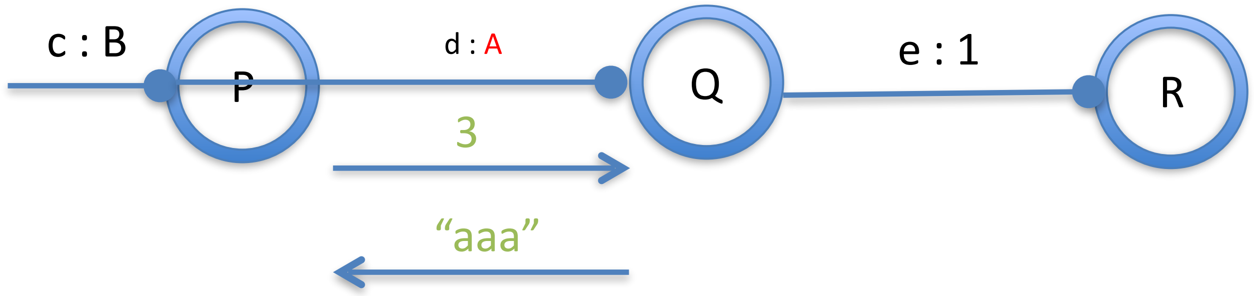
Linear Session-types

- Don't want to send int if expecting string
- Don't try to receive if other process is not sending
- Assign types to each channel from provider's perspective



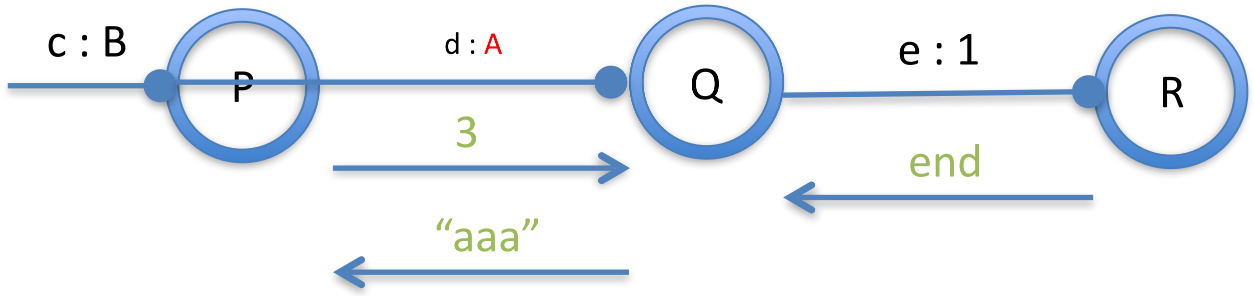
Linear Session-types

- Don’t want to send int if expecting string
- Don’t try to receive if other process is not sending
- Assign types to each channel from provider’s perspective



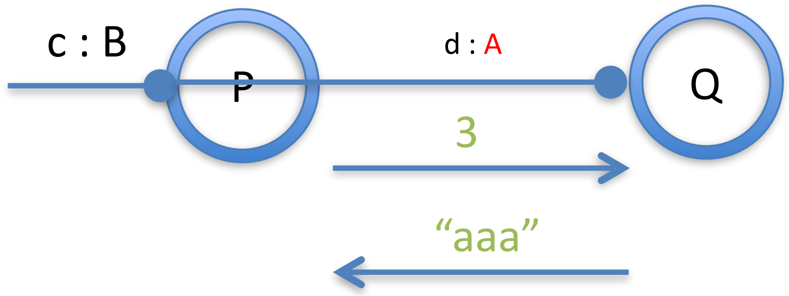
Linear Session-types

- Don’t want to send int if expecting string
- Don’t try to receive if other process is not sending
- Assign types to each channel from provider’s perspective



Linear Session-types

- Don’t want to send int if expecting string
- Don’t try to receive if other process is not sending
- Assign types to each channel from provider’s perspective



Linear Session Types

- Example interface specification:

```
queue = &{enq: A —o queue,
          deq: ⊕{none: 1, some: A ⊗ queue}}

* where A is some predetermined type
```

`1`	Terminate
$\&\{lab_i : A_i\}_i$	External choice (receive) between lab_i, continue as A_i
$A \multimap B$	Receive channel of type A, continue as B
$\tau \supset B$	Receive value of type τ, continue as B
$\oplus\{lab_i : A_i\}_i$	Internal choice (send) between lab_i, continue as A_i
$A \otimes B$	Send channel of type A, continue as B
$\tau \wedge B$	Send value of type τ, continue as B

Implementation of Queues

```
queue = &{enq: A —o queue,
          deq: ⊕{none: 1, some: A ⊗ queue}}

empty : queue
q ← empty =
  case q
    enq → x ← recv q ;
          e ← empty ;
          q ← elem x e
    deq → q.none ; close q

elem : A —o queue —o queue
q ← elem x r =
  case q
    enq → y ← recv q ;
          r.enq ; send r y ;
          q ← elem x r
    deq → q.some ; send q x ;
          q ← r
```

Intersections and Unions

- Allows describing more interesting behavior
- Intersection of two types: $A \sqcap B$
  - $c : A \sqcap B$ if channel c offers both behaviors
- Union of two types: $A \sqcup B$
  - $c : A \sqcup B$ if channel c offers either behavior

Refinement Types

- What if we want to track more properties of queues? Empty, non-empty, even length?
- We can define them in the base system:

```
empty-queue = &{enq: A —o nonempty-queue,
                deq: ⊕{none: 1}}

nonempty-queue = &{enq: A —o nonempty-queue,
                   deq: ⊕{some: A ⊗ queue}}
```

Refinement Types

- But we need intersections and unions to write interesting programs

```
queue A = empty-queue ⊔ nonempty-queue

empty : empty-queue

elem : (A ⊸ queue ⊸ nonempty-queue)

concat : (empty-queue ⊸ empty-queue ⊸ empty-queue)
       ⊓ (queue ⊸ nonempty-queue ⊸ nonempty-queue)
       ⊓ (nonempty-queue ⊸ queue ⊸ nonempty-queue)
```

Decidability of Type-checking

- Algorithmic system that is easy to translate to code
- Prove sound and complete with respect to the original system
- Partial implementation in Haskell

Type Safety

- Progress
  - Deadlock freedom in concurrent setting
  - At least one process can make progress if the configuration is well-typed

- Preservation [*currently in progress*]
  - Session fidelity in concurrent setting
  - Processes obey session-types