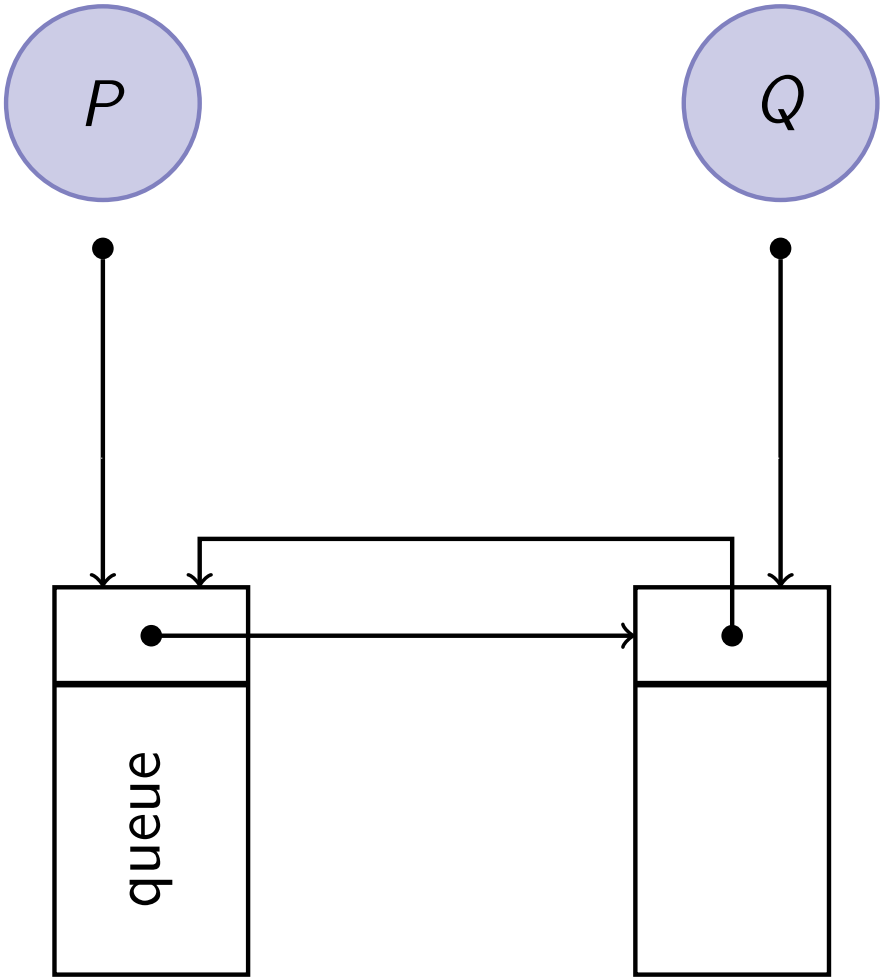
Exception Handling for Copyless Messaging

Svetlana Jakšić Luca Padovani

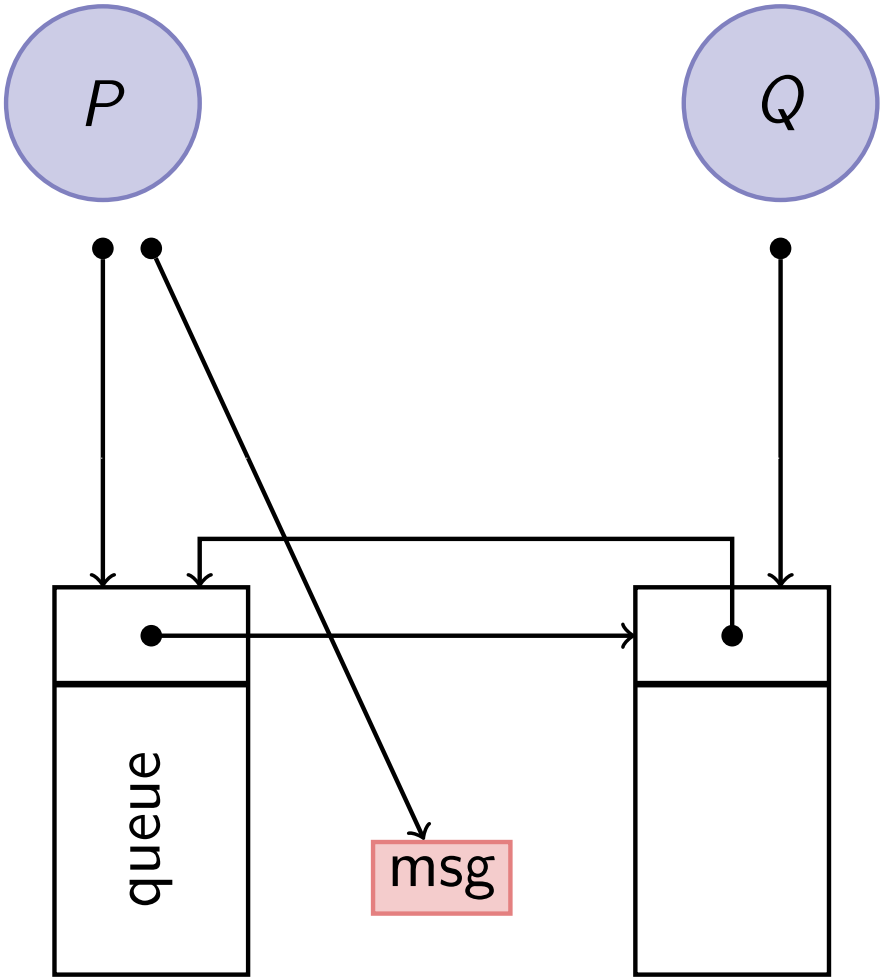
Univerzitet u Novom Sadu, Fakultet tehničkih nauka, Serbia
Università di Torino, Dipartimento di Informatica, Italy

PPDP 2012

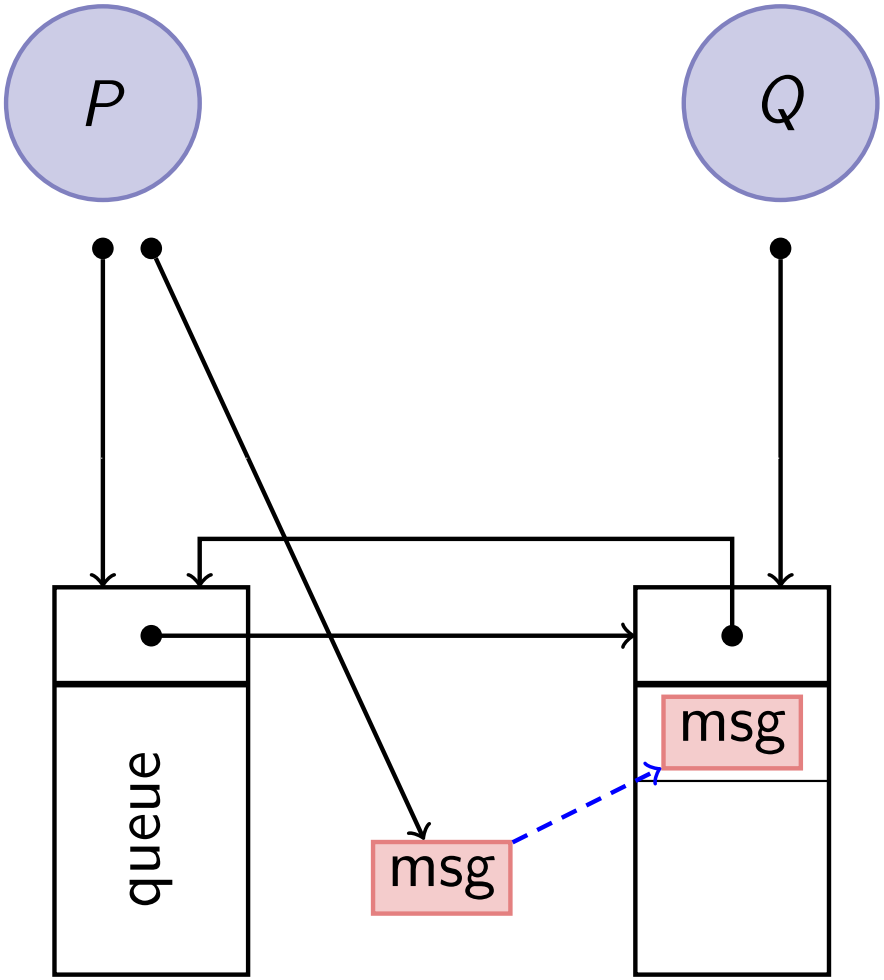
Copyful messaging



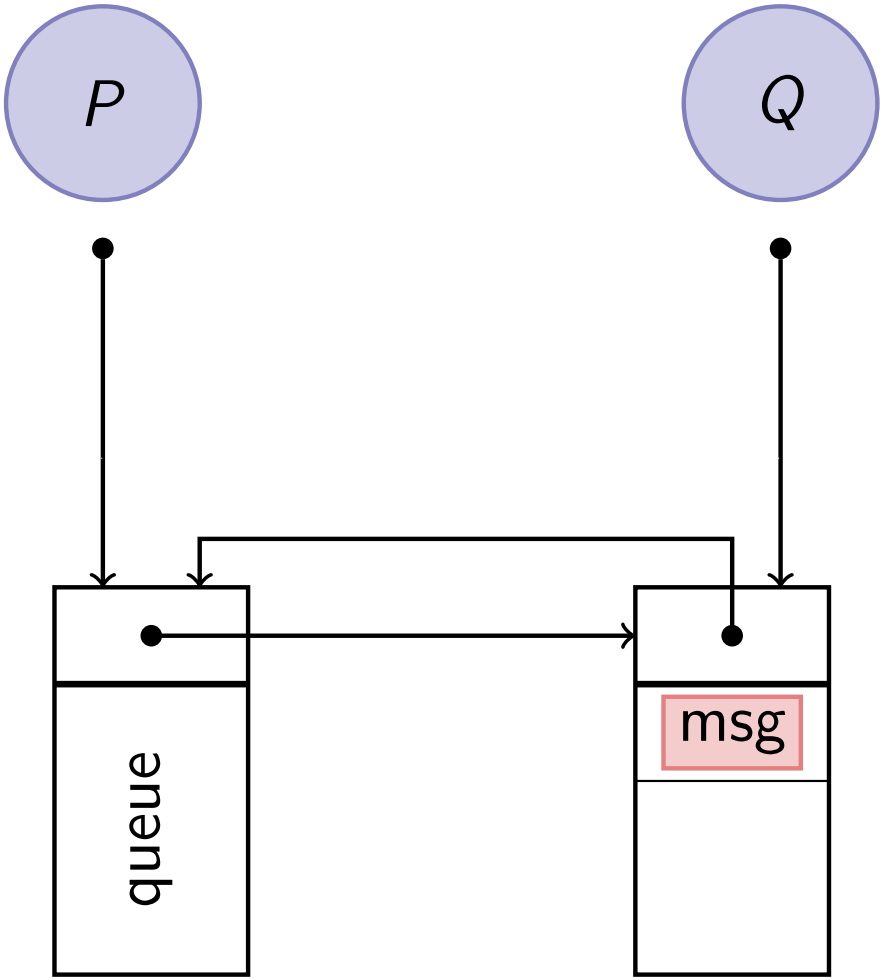
Copyful messaging



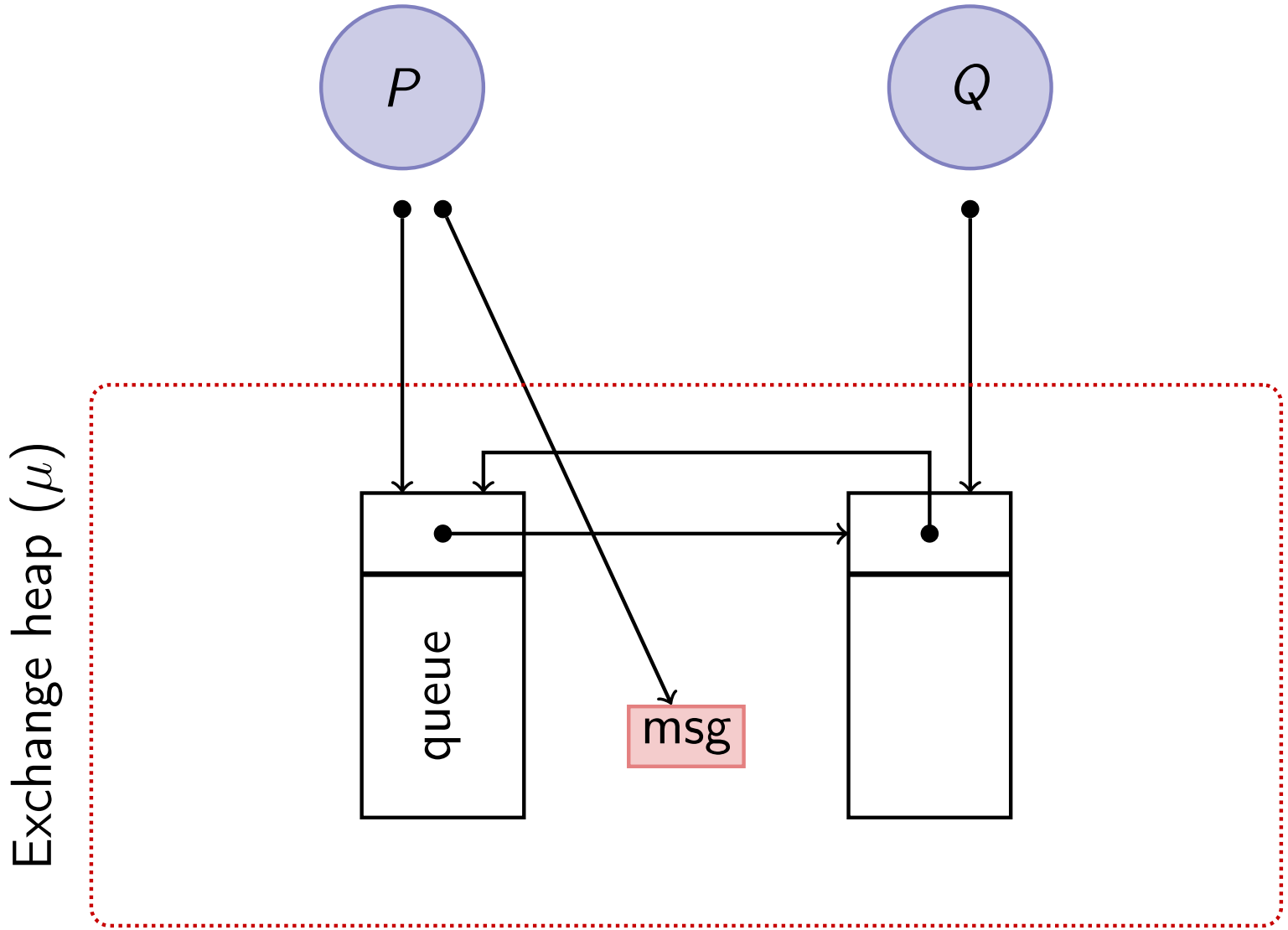
Copyful messaging



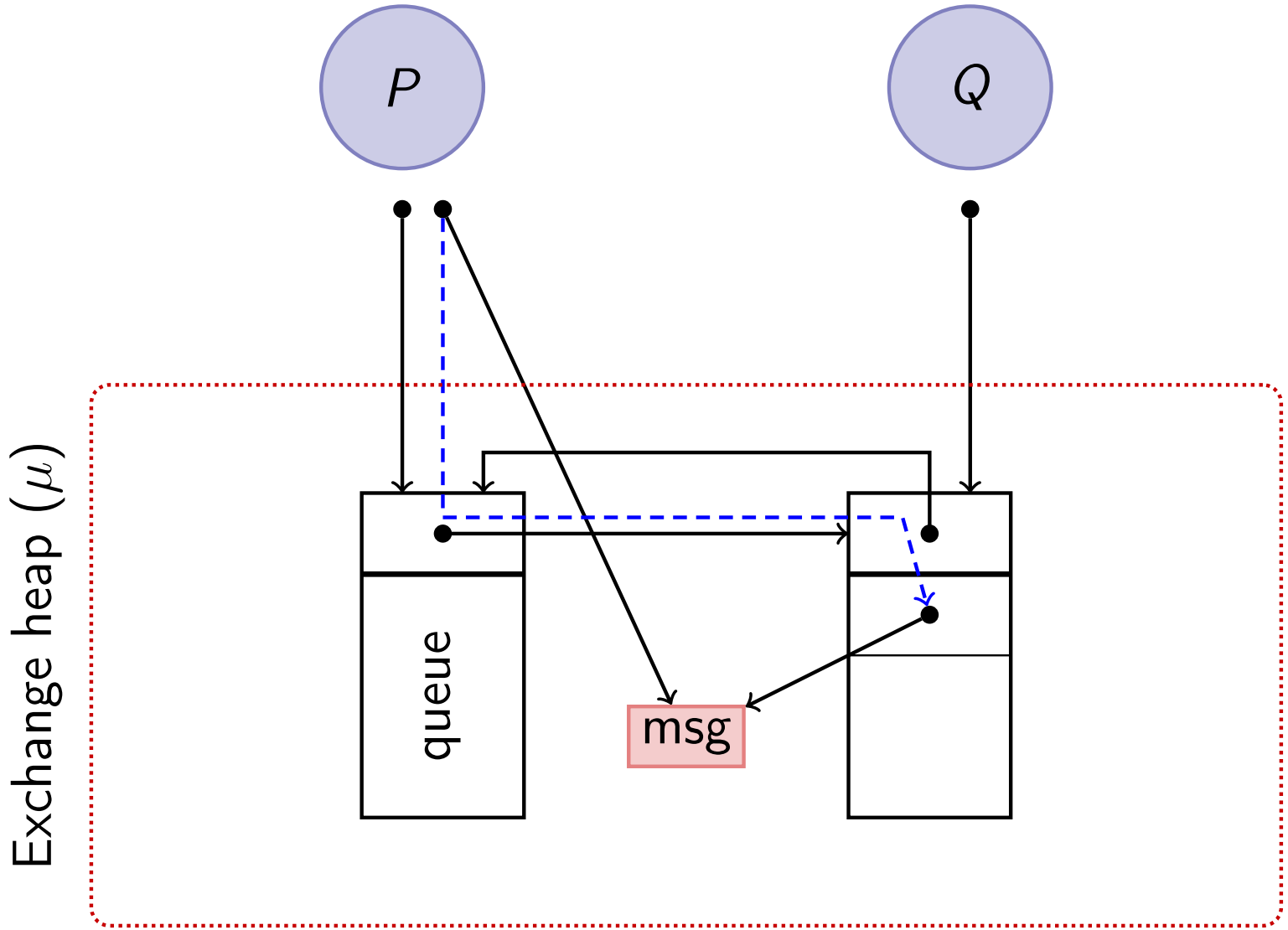
Copyful messaging



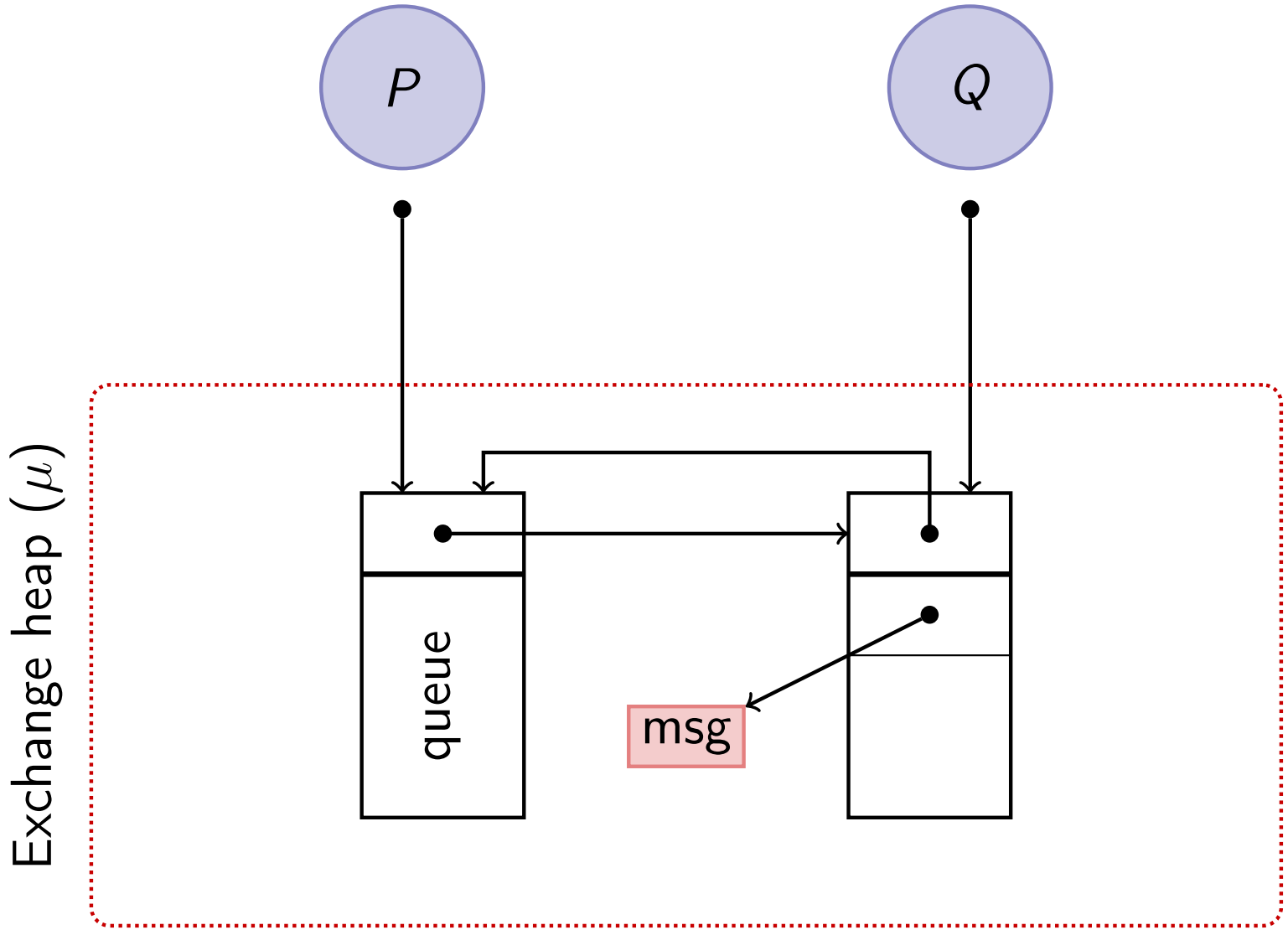
Copyless messaging



Copyless messaging



Copyless messaging



An example from Singularity OS

```
ns = DS.NewClientEndpoint();
try {
  while (true) {
    NewChannel(out imp, out exp);
    ns.SendRegister(imp);
    switch receive {
      case ns.AckRegister():
        return exp;
      case ns.NakRegister(nakImp, error):
        if (error != AlreadyExists)
          throw new Exception();
        delete exp;
        delete nakImp;
    }
  }
} finally { delete ns; }
```

Safety properties

Safety is...

- no communication errors
- no memory faults
- no memory leaks

Recipe for safety

- use channel contracts (aka session types)
- impose linear ownership of pointers (+ a little more)

An example from Singularity OS

```
ns = DS.NewClientEndpoint();
try {
  while (true) {
    NewChannel(out imp, out exp);
    ns.SendRegister(imp);
    switch receive {
      case ns.AckRegister():
        return exp;
      case ns.NakRegister(nakImp, error):
        if (error != AlreadyExists)
          throw new Exception();
        delete exp;
        delete nakImp;
    }
  }
} finally { delete ns; }
```

An example from Singularity OS

```
ns = DS.NewClientEndpoint();
try {
  while (true) {
    NewChannel(out imp, out exp);
    ns.SendRegister(imp);
    switch receive {
      case ns.AckRegister():
        return exp;
      case ns.NakRegister(nakImp, error):
        if (error != AlreadyExists)
          throw new Exception();
        delete exp;
        delete nakImp;
    }
  }
} finally { delete ns; }
```

Preventing communication errors

```
contract DSContract { // default is exporting view
  state Ready {
    Register? → DoRegister;
    CreateDirectory? → ...
    // ...more transitions
  }
  state DoRegister {
    AckRegister! → Stop;
    NakRegister! → Ready;
  }
  state Stop { }
}
```

$$\texttt{DSContract.Exp} : \texttt{Ready} = \texttt{rec}\ \alpha.?\texttt{Register}.(!\texttt{AckRegister.end} \oplus !\texttt{NakRegister}.\alpha)$$

Preventing communication errors

```
ns = DS.NewClientEndpoint(); // DSContract.Imp:Ready
try {
  while (true) {
    NewChannel(out imp, out exp);
    ns.SendRegister(imp);
    switch receive {
      case ns.AckRegister():
        return exp;
      case ns.NakRegister(nakImp, error):
        if (error != AlreadyExists)
          throw new Exception();
        delete exp;
        delete nakImp;
    }
  }
} finally { delete ns; }
```

Preventing communication errors

```
ns = DS.NewClientEndpoint(); // DSContract.Imp:Ready
try {
  while (true) {
    NewChannel(out imp, out exp);
    ns.SendRegister(imp); // DSContract.Imp:DoRegister
    switch receive {
      case ns.AckRegister():
        return exp;
      case ns.NakRegister(nakImp, error):
        if (error != AlreadyExists)
          throw new Exception();
        delete exp;
        delete nakImp;
    }
  }
} finally { delete ns; }
```

Preventing communication errors

```
ns = DS.NewClientEndpoint(); // DSContract.Imp:Ready
try {
  while (true) {
    NewChannel(out imp, out exp);
    ns.SendRegister(imp); // DSContract.Imp:DoRegister
    switch receive {
      case ns.AckRegister(): // DSContract.Imp:Stop
        return exp;
      case ns.NakRegister(nakImp, error):
        if (error != AlreadyExists)
          throw new Exception();
        delete exp;
        delete nakImp;
    }
  }
} finally { delete ns; }
```

Preventing communication errors

```
ns = DS.NewClientEndpoint(); // DSContract.Imp:Ready
try {
  while (true) {
    NewChannel(out imp, out exp);
    ns.SendRegister(imp); // DSContract.Imp:DoRegister
    switch receive {
      case ns.AckRegister(): // DSContract.Imp:Stop
        return exp;
      case ns.NakRegister(nakImp, error):
        if (error != AlreadyExists)
          throw new Exception();
        delete exp;
        delete nakImp;
    }
  }
} finally { delete ns; }
```

Preventing memory faults and leaks

Each pointer is owned by exactly one process

- Process isolation = no faults
- Single ownership = no leaks

Preventing memory faults and leaks

```
ns = DS.NewClientEndpoint();
try {
  while (true) {
    NewChannel(out imp, out exp); { imp, exp }
    ns.SendRegister(imp);
    switch receive {
      case ns.AckRegister():
        return exp;
      case ns.NakRegister(nakImp, error):
        if (error != AlreadyExists)
          throw new Exception();
        delete exp;
        delete nakImp;
    }
  }
} finally { delete ns; }
```

Preventing memory faults and leaks

```
ns = DS.NewClientEndpoint();
try {
  while (true) {
    NewChannel(out imp, out exp); { imp, exp }
    ns.SendRegister(imp); { exp }
    switch receive {
      case ns.AckRegister():
        return exp;
      case ns.NakRegister(nakImp, error):
        if (error != AlreadyExists)
          throw new Exception();
        delete exp;
        delete nakImp;
    }
  }
} finally { delete ns; }
```

Preventing memory faults and leaks

```
ns = DS.NewClientEndpoint();
try {
  while (true) {
    NewChannel(out imp, out exp); { imp, exp }
    ns.SendRegister(imp); { exp }
    switch receive {
      case ns.AckRegister():
        return exp; { }
      case ns.NakRegister(nakImp, error):
        if (error != AlreadyExists)
          throw new Exception();
        delete exp;
        delete nakImp;
    }
  }
} finally { delete ns; }
```

Preventing memory faults and leaks

```
ns = DS.NewClientEndpoint();
try {
  while (true) {
    NewChannel(out imp, out exp); { imp, exp }
    ns.SendRegister(imp); { exp }
    switch receive {
      case ns.AckRegister():
        return exp; { }
      case ns.NakRegister(nakImp, error): { exp, nakImp }
        if (error != AlreadyExists)
          throw new Exception();
        delete exp;
        delete nakImp;
    }
  }
} finally { delete ns; }
```

Preventing memory faults and leaks

```
ns = DS.NewClientEndpoint();
try {
  while (true) {
    NewChannel(out imp, out exp); { imp, exp }
    ns.SendRegister(imp); { exp }
    switch receive {
      case ns.AckRegister():
        return exp; { }
      case ns.NakRegister(nakImp, error): { exp, nakImp }
        if (error != AlreadyExists)
          throw new Exception();
        delete exp; { nakImp }
        delete nakImp;
    }
  }
} finally { delete ns; }
```

Preventing memory faults and leaks

```
ns = DS.NewClientEndpoint();
try {
  while (true) {
    NewChannel(out imp, out exp); { imp, exp }
    ns.SendRegister(imp); { exp }
    switch receive {
      case ns.AckRegister():
        return exp; { }
      case ns.NakRegister(nakImp, error): { exp, nakImp }
        if (error != AlreadyExists)
          throw new Exception();
        delete exp; { nakImp }
        delete nakImp; { }
    }
  }
} finally { delete ns; }
```

Preventing memory faults and leaks

```
ns = DS.NewClientEndpoint();
try {
  while (true) {
    NewChannel(out imp, out exp); { imp, exp }
    ns.SendRegister(imp); { exp }
    switch receive {
      case ns.AckRegister():
        return exp; { }
      case ns.NakRegister(nakImp, error): { exp, nakImp }
        if (error != AlreadyExists)
          throw new Exception(); // leak
        delete exp; { nakImp }
        delete nakImp; { }
    }
  }
} finally { delete ns; }
```

Dealing with exceptions (damage control)

Dynamic type checking

```
if (ns.InState(DSContract.Imp:Ready)) ...
```

- ☹ error prone
- ☹ defeats the purpose of static type checking

Careful placement of `delete`

- ☹ error prone
- ☹ sometimes impossible (scoping rules)

Our proposal: `try` block = transaction

When the transaction starts...

- synchronize affected processes
- save state of the heap

If an exception is thrown...

- undo heap changes
- notify affected processes and run handlers

If the transaction completes...

- discard handlers
- commit heap changes

Our proposal: `try` block = transaction

When the transaction starts...

- synchronize affected processes
- ~~save state of the heap~~ do nothing

If an exception is thrown...

- ~~undo heap changes~~ do some local clean up
- notify affected processes and run handlers

If the transaction completes...

- discard handlers
- ~~commit heap changes~~ do nothing

Modeling processes

P	::=		**Process**
		done	(inaction)
	$\mid$	open$(a, b).P$	(open channel)
	$\mid$	close$(u).P$	(close endpoint)
	$\mid$	$u!\mathtt{m}\langle v\rangle.P$	(send)
	$\mid$	$\sum_{i\in I} u?\mathtt{m}_i\langle x_i\rangle.P_i$	(receive)
	$\mid$	$P \oplus Q$	(conditional)
	$\mid$	$P \mid Q$	(parallel)
	$\mid$	try(u) $\{Q\}P$	(start transaction)
	$\mid$	throw	(exception)
	$\mid$	commit$(u).P$	(commit transaction)

Types and endpoint types

T	$::=$		**Endpoint type**
		$\mathtt{end}$	(to-be-closed)
	$\mid$	$\{!\mathtt{m}_i\langle t_i\rangle.T_i\}_{i\in I}$	(output)
	$\mid$	$\{?\mathtt{m}_i\langle t_i\rangle.T_i\}_{i\in I}$	(input)
	$\mid$	$\{S\}[\![T$	(start transaction)
	$\mid$	$]\!]T$	(commit transaction)
t	$::=$		**Type**
		T	(endpoint type)
	$\mid$	$[t]$	(sealed type)

Typing transactions

$$\frac{[\Delta], u : T \vdash P \qquad \Delta, u : S \vdash Q}{\Delta, u : \{S\}[\![T \vdash \mathsf{try}(u)\ \{Q\}P}$$

$$\Delta \vdash \mathsf{throw} \qquad \text{(guarded by } \texttt{try}\text{)}$$

$$\frac{\mathsf{unsealed}(\Delta_2) \qquad \Delta_1, u : T, \Delta_2 \vdash P}{[\Delta_1], u : [\![T, \Delta_2 \vdash \mathsf{commit}(u).P}$$

Heap properties induced by endpoint types

Well-formedness: $[\![$ and $]\!]$ must be balanced

- read: no `end` within transactions
- prevents deallocation of endpoints in transactions (except for endpoints created within transactions)

Duality: $\overline{\{S\}[\![T}} = \{\overline{S}\}[\![\overline{T}$

- read: no input/output allowed at transaction boundaries
- endpoint queues are empty at transaction boundaries

DSContract revisited

```
contract DSContract {
  state Ready {
    Register? → DoRegister;
    CreateDirectory? → ...
    // ...more transitions
  }
  state DoRegister {
    AckRegister! → Stop;
    NakRegister! → Ready;
  }
  state Stop { }
}
```

$$\texttt{DSContract.Exp} : \texttt{Ready} = \{\texttt{end}\}[\![\texttt{rec}\ \alpha.?\texttt{Register}.(!\texttt{AckRegister}.]\!]\texttt{end} \oplus !\texttt{NakRegister}.\alpha)$$

Transaction semantics

$$\mu; (\mathsf{try}(a)\ \{Q_1\}P_1 \mid \mathsf{try}(b)\ \{Q_2\}P_2) \qquad \mathsf{peers}(a, b)$$
$$\to \mu; \langle \{a, b\}, \emptyset, \{Q_1 \mid Q_2\}(P_1 \mid P_2)\rangle$$

$$\mu; \langle \{a, b\}, B, \{Q\}(\mathsf{commit}(a).P_1 \mid \mathsf{commit}(b).P_2)\rangle \to \mu; P_1 \mid P_2$$

$$\frac{\mu; P \to \mu'; P'}{\mu; \langle A, B, \{Q\}P\rangle \to \mu'; \langle A, B', \{Q\}P'\rangle}$$

$$\mu_1, \{a_i \mapsto [b_i, q_i]\}_{i\in 1,2}, \mu_2; \langle \{a_i\}_{i\in 1,2}, \mathrm{dom}(\mu_2), \{Q\}(\mathsf{throw} \mid P)\rangle$$
$$\to \mu_1, \{a_i \mapsto [b_i, \varepsilon]\}_{i\in 1,2}; Q$$

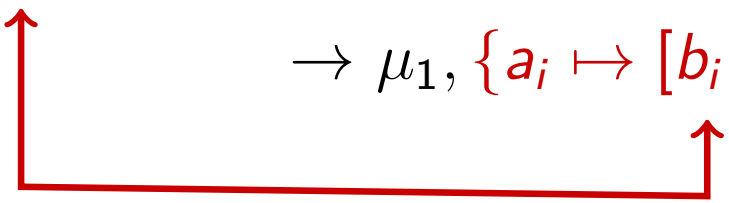
Transaction semantics

$$\mu; (\mathsf{try}(a)\ \{Q_1\}P_1 \mid \mathsf{try}(b)\ \{Q_2\}P_2) \qquad \mathsf{peers}(a, b)$$
$$\rightarrow \mu; \langle \{a, b\}, \emptyset, \{Q_1 \mid Q_2\}(P_1 \mid P_2)\rangle$$

$$\mu; \langle \{a, b\}, B, \{Q\}(\mathsf{commit}(a).P_1 \mid \mathsf{commit}(b).P_2)\rangle \rightarrow \mu; P_1 \mid P_2$$

$$\frac{\mu; P \rightarrow \mu'; P'}{\mu; \langle A, B, \{Q\}P\rangle \rightarrow \mu'; \langle A, B', \{Q\}P'\rangle}$$

$$\mu_1, \{a_i \mapsto [b_i, q_i]\}_{i\in 1,2}, \mu_2; \langle \{a_i\}_{i\in 1,2}, \mathrm{dom}(\mu_2), \{Q\}(\mathsf{throw} \mid P)\rangle$$
$$\rightarrow \mu_1, \{a_i \mapsto [b_i, \varepsilon]\}_{i\in 1,2}; Q$$



Transaction semantics

$$\mu; (\mathsf{try}(a)\ \{Q_1\}P_1 \mid \mathsf{try}(b)\ \{Q_2\}P_2) \qquad \mathsf{peers}(a, b)$$
$$\to \mu; \langle \{a, b\}, \emptyset, \{Q_1 \mid Q_2\}(P_1 \mid P_2)\rangle$$

$$\mu; \langle \{a, b\}, B, \{Q\}(\mathsf{commit}(a).P_1 \mid \mathsf{commit}(b).P_2)\rangle \to \mu; P_1 \mid P_2$$

$$\frac{\mu; P \to \mu'; P'}{\mu; \langle A, B, \{Q\}P\rangle \to \mu'; \langle A, B', \{Q\}P'\rangle}$$

$$\mu_1, \{a_i \mapsto [b_i, q_i]\}_{i\in 1,2}, \mu_2; \langle \{a_i\}_{i\in 1,2}, \mathrm{dom}(\mu_2), \{Q\}(\mathsf{throw} \mid P)\rangle$$
$$\to \mu_1, \{a_i \mapsto [b_i, \varepsilon]\}_{i\in 1,2}; Q$$

Well-typed processes are well behaved

P is well behaved if $\emptyset; P \Rightarrow \mu; Q$ implies:

1. $Q \equiv Q_1 \mid Q_2$ and $\mu; Q_1 \not\rightarrow$ where Q_1 is an input from a, then the queue of a is empty
2. $\mathrm{dom}(\mu) = \mu\text{-reach}(\mathrm{fn}(Q))$

Theorem (Subject reduction)

If $\Delta \vdash P$ and $\mu; P \rightarrow \mu'; P'$, then $\Delta' \vdash P'$ for some Δ'.

Theorem (Soundness)

If $\vdash P$, then P is well behaved.

Concluding remarks

Model of copyless messaging with exceptions

- programs $\Rightarrow$ π-like processes
- pointers $\Rightarrow$ names
- `try` blocks $\Rightarrow$ transactions

Endpoint types

- prevent communication errors
- identify transactions in communications
- enable lightweight heap restoration

Related work

On copyless messaging (without exceptions)

- Villard, Lozes, Calcagno, **Proving Copyless Message Passing**, APLAS 2009
- Bono, Padovani, **Typing Copyless Message Passing**, LMCS 2012

On exceptions within sessions (copyful messaging with leaks)

- Carbone, Honda, Yoshida, **Structured interactional exceptions in session types**, CONCUR 2008
- Capecchi, Giachino, Yoshida, **Global escape in multiparty sessions**, FSTTCS 2010

Ongoing work

$$\frac{\Delta, u : T \vdash P}{\Delta, u : !\mathtt{m}\langle S\rangle.T, v : S \vdash u!\mathtt{m}\langle v\rangle.P}$$

- sealed values should be safe to send, but...
- ...this is hard to prove (for us)
- temporary ownership invariant violation

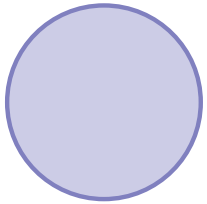
Ongoing work

$$\frac{\Delta, u : T \vdash P}{\Delta, u : !\mathtt{m}\langle s \rangle.T, v : s \vdash u!\mathtt{m}\langle v \rangle.P}$$

- sealed values should be safe to send, but...
- ...this is hard to prove (for us)
- temporary ownership invariant violation

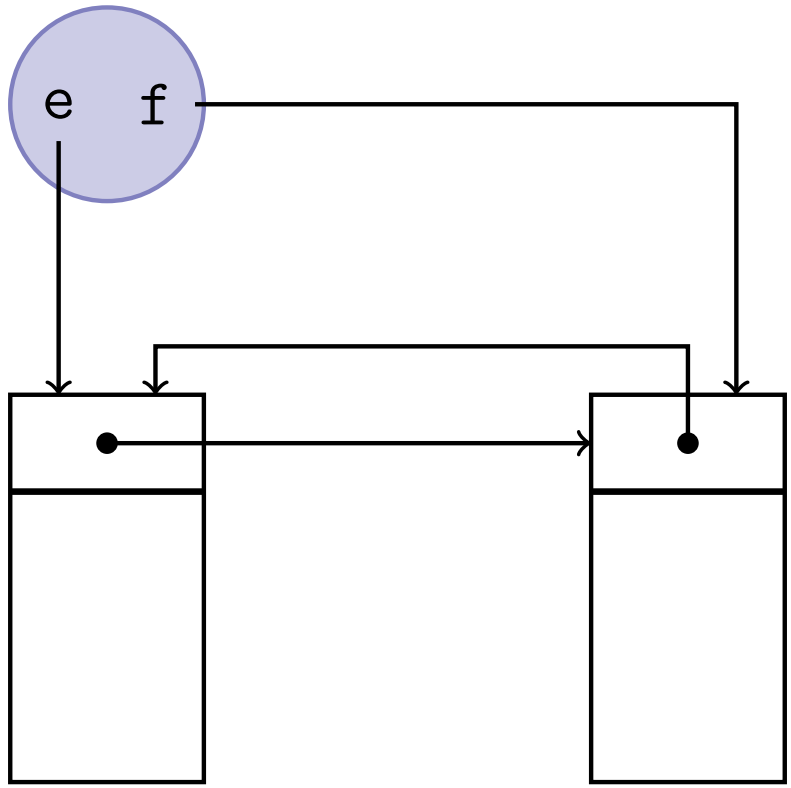
Linear ownership is too permissive

```
NewChannel(e, f);
e.Send(f);
delete e;
```



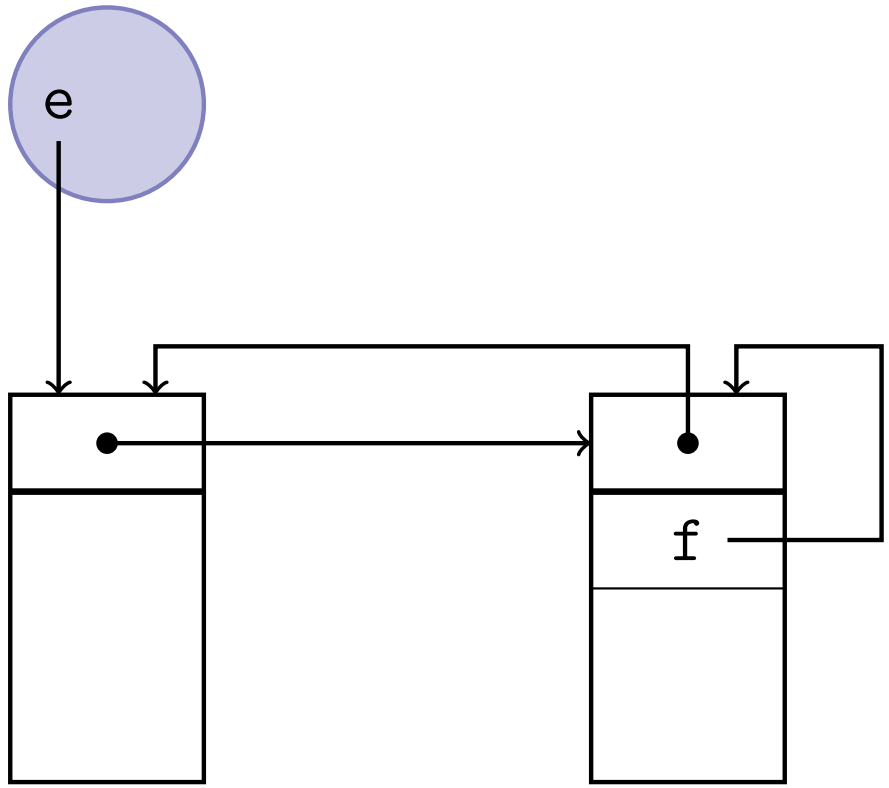
Linear ownership is too permissive

```
NewChannel(e, f);
e.Send(f);
delete e;
```



Linear ownership is too permissive

```
NewChannel(e, f);
e.Send(f);
delete e;
```



Linear ownership is too permissive

```
NewChannel(e, f);
e.Send(f);
delete e;
```