A Rehabilitation of Message-Passing Concurrency

Frank Pfenning
Carnegie Mellon University

PWLConf 2018, St. Louis

A Paper I Love

Types for Dyadic Interaction*

Kohei Honda

kohei@mt.cs.keio.ac.jp

Department of Computer Science, Keio University

3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223, Japan

Abstract

We formulate a typed formalism for concurrency where types denote freely composable structure of dyadic interaction in the symmetric scheme. The resulting calculus is a typed reconstruction of name passing process calculi. Systems with both the explicit and implicit typing disciplines, where types form a simple hierarchy of types, are presented, which are proved to be in accordance with each other. A typed variant of bisimilarity is formulated and it is shown that typed β-equality has a clean embedding in the bisimilarity. Name reference structure induced by the simple hierarchy of types is studied, which fully characterises the typable terms in the set of untyped terms. It turns out that the name reference structure results in the deadlock-free property for a subset of terms with a certain regular structure, showing behavioural significance of the simple type discipline.

A Paper I Love

- *Types for Dyadic Interaction*, Kohei Honda, CONCUR 1993

Types for Dyadic Interaction*

Kohei Honda
kohei@mt.cs.keio.ac.jp

Department of Computer Science, Keio University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223, Japan

Abstract

We formulate a typed formalism for concurrency where types denote freely composable structure of dyadic interaction in the symmetric scheme. The resulting calculus is a typed reconstruction of name passing process calculi. Systems with both the explicit and implicit typing disciplines, where types form a simple hierarchy of types, are presented, which are proved to be in accordance with each other. A typed variant of bisimilarity is formulated and it is shown that typed β-equality has a clean embedding in the bisimilarity. Name reference structure induced by the simple hierarchy of types is studied, which fully characterises the typable terms in the set of untyped terms. It turns out that the name reference structure results in the deadlock-free property for a subset of terms with a certain regular structure, showing behavioural significance of the simple type discipline.

A Paper I Love

- *Types for Dyadic Interaction*, Kohei Honda, CONCUR 1993
- With some newer developments
  - *Session Types as Intuitionistic Linear Propositions*, Luís Caires & Pf., CONCUR 2010
  - *Manifest Sharing with Session Types*, Stephanie Balzer & Pf., ICFP 2017

Types for Dyadic Interaction*

Kohei Honda

kohei@mt.cs.keio.ac.jp

Department of Computer Science, Keio University

3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223, Japan

Abstract

We formulate a typed formalism for concurrency where types denote freely composable structure of dyadic interaction in the symmetric scheme. The resulting calculus is a typed reconstruction of name passing process calculi. Systems with both the explicit and implicit typing disciplines, where types form a simple hierarchy of types, are presented, which are proved to be in accordance with each other. A typed variant of bisimilarity is formulated and it is shown that typed β-equality has a clean embedding in the bisimilarity. Name reference structure induced by the simple hierarchy of types is studied, which fully characterises the typable terms in the set of untyped terms. It turns out that the name reference structure results in the deadlock-free property for a subset of terms with a certain regular structure, showing behavioural significance of the simple type discipline.

The Activity of Programming

The Activity of Programming

```
sort(A);
x = A[0];
```

The Activity of Programming

```
sort(A);
x = A[0];
```

```
hd(sort(A))
```

The Activity of Programming

- Every programmer, all the time, reasons

```
sort(A);
x = A[0];
```

```
hd(sort(A))
```

The Activity of Programming

- Every programmer, all the time, reasons
  - Operationally (how)

```
sort(A);
x = A[0];
```

```
hd(sort(A))
```

The Activity of Programming

- Every programmer, all the time, reasons
  - Operationally (how)
  - Logically (what)

```
sort(A);
x = A[0];
```

```
hd(sort(A))
```

The Activity of Programming

- Every programmer, all the time, reasons
  - Operationally (how)
  - Logically (what)
- The effectiveness of a programming language depends critically on

```
sort(A);
x = A[0];
```

```
hd(sort(A))
```

The Activity of Programming

- Every programmer, all the time, reasons
  - Operationally (how)
  - Logically (what)
- The effectiveness of a programming language depends critically on
  - How programs execute

```
sort(A);
x = A[0];
```

```
hd(sort(A))
```

The Activity of Programming

- Every programmer, all the time, reasons
  - Operationally (how)
  - Logically (what)
- The effectiveness of a programming language depends critically on
  - How programs execute
  - What they achieve

```
sort(A);
x = A[0];
```

```
hd(sort(A))
```

The Activity of Programming

- Every programmer, all the time, reasons
  - Operationally (how)
  - Logically (what)
- The effectiveness of a programming language depends critically on
  - How programs execute
  - What they achieve
  - Which reasoning principles connect the operational and logical meaning of a program

```
sort(A);
x = A[0];
```

```
hd(sort(A))
```

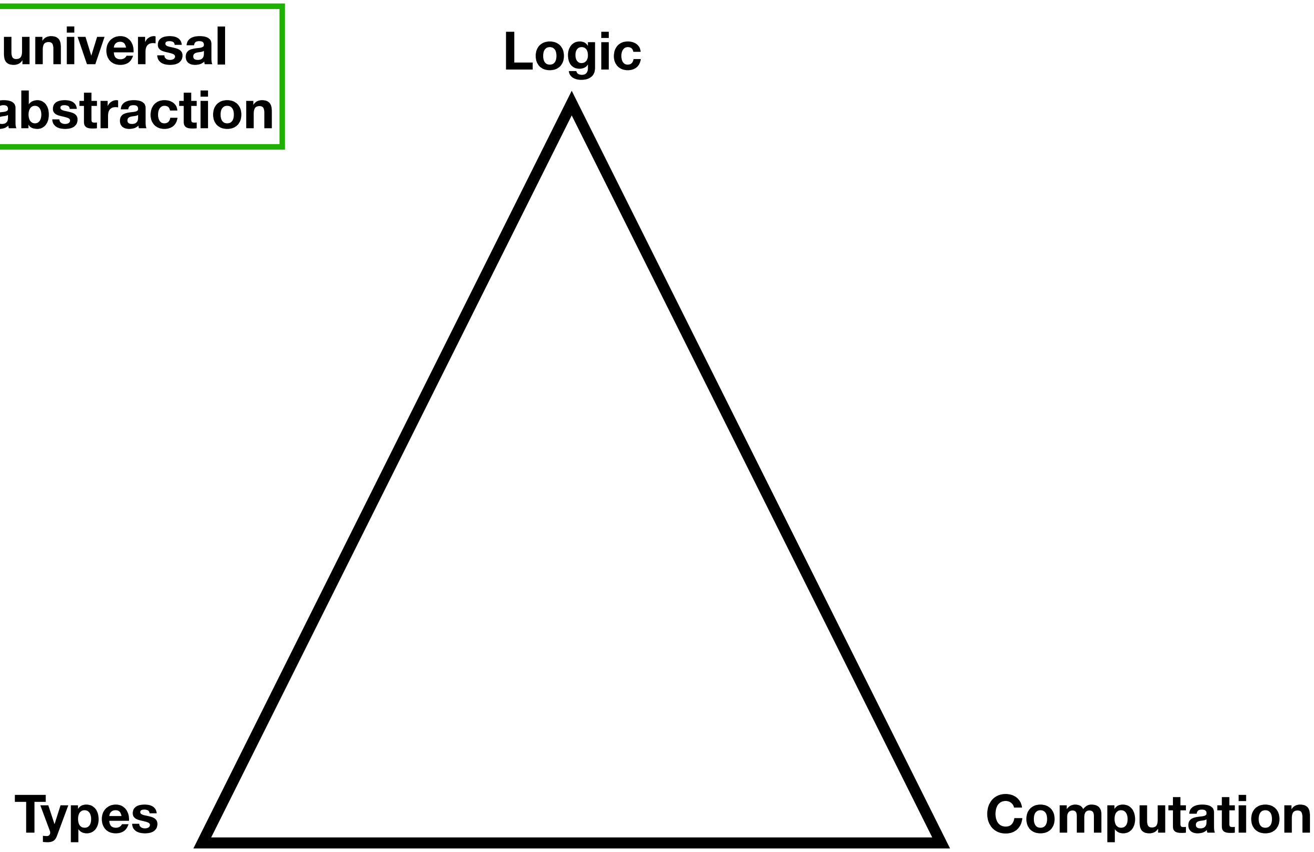
Why is Functional Programming So Effective?

Why is Functional Programming So Effective?

Functions are a universal and fundamental abstraction

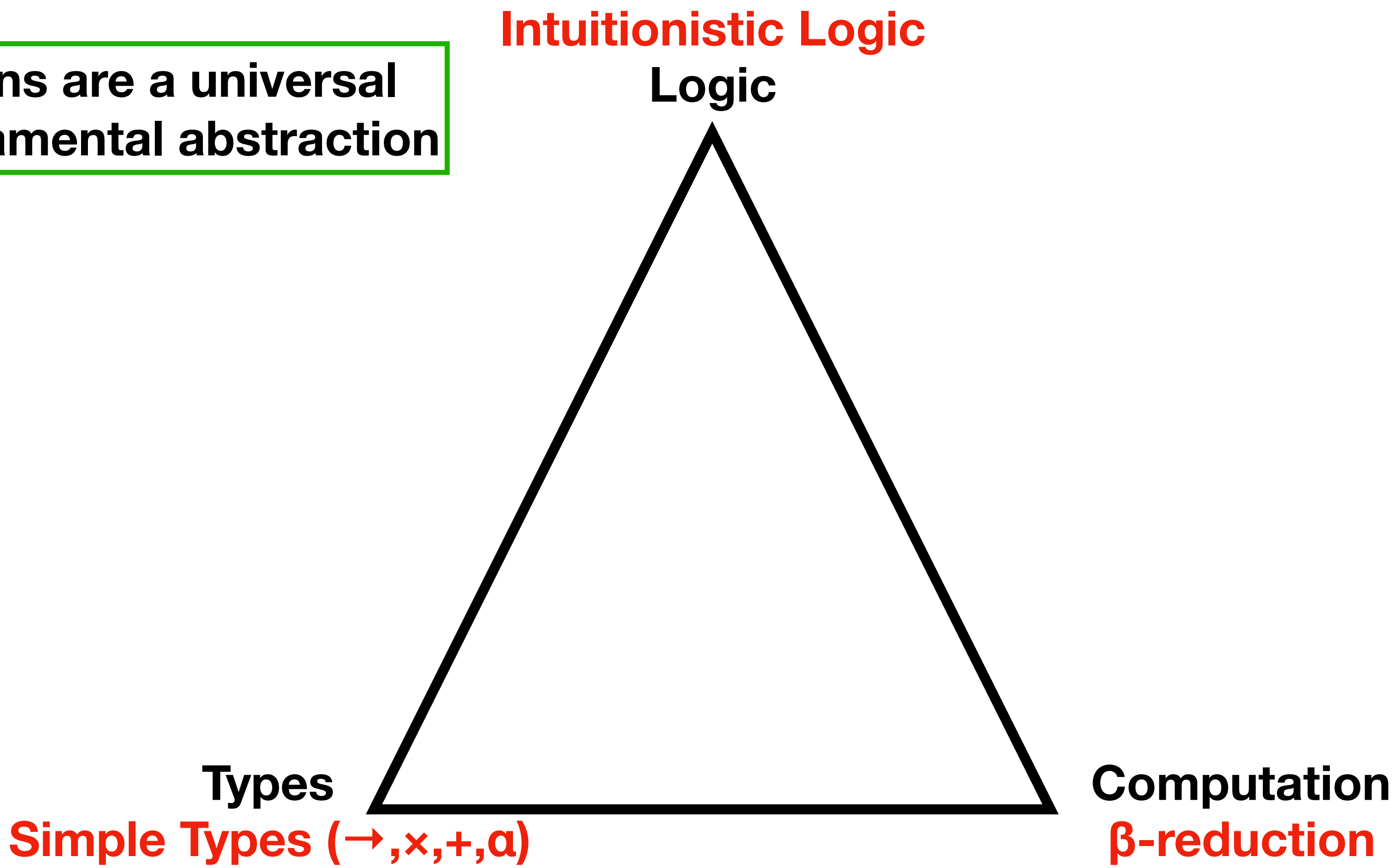
Why is Functional Programming So Effective?

Functions are a universal and fundamental abstraction



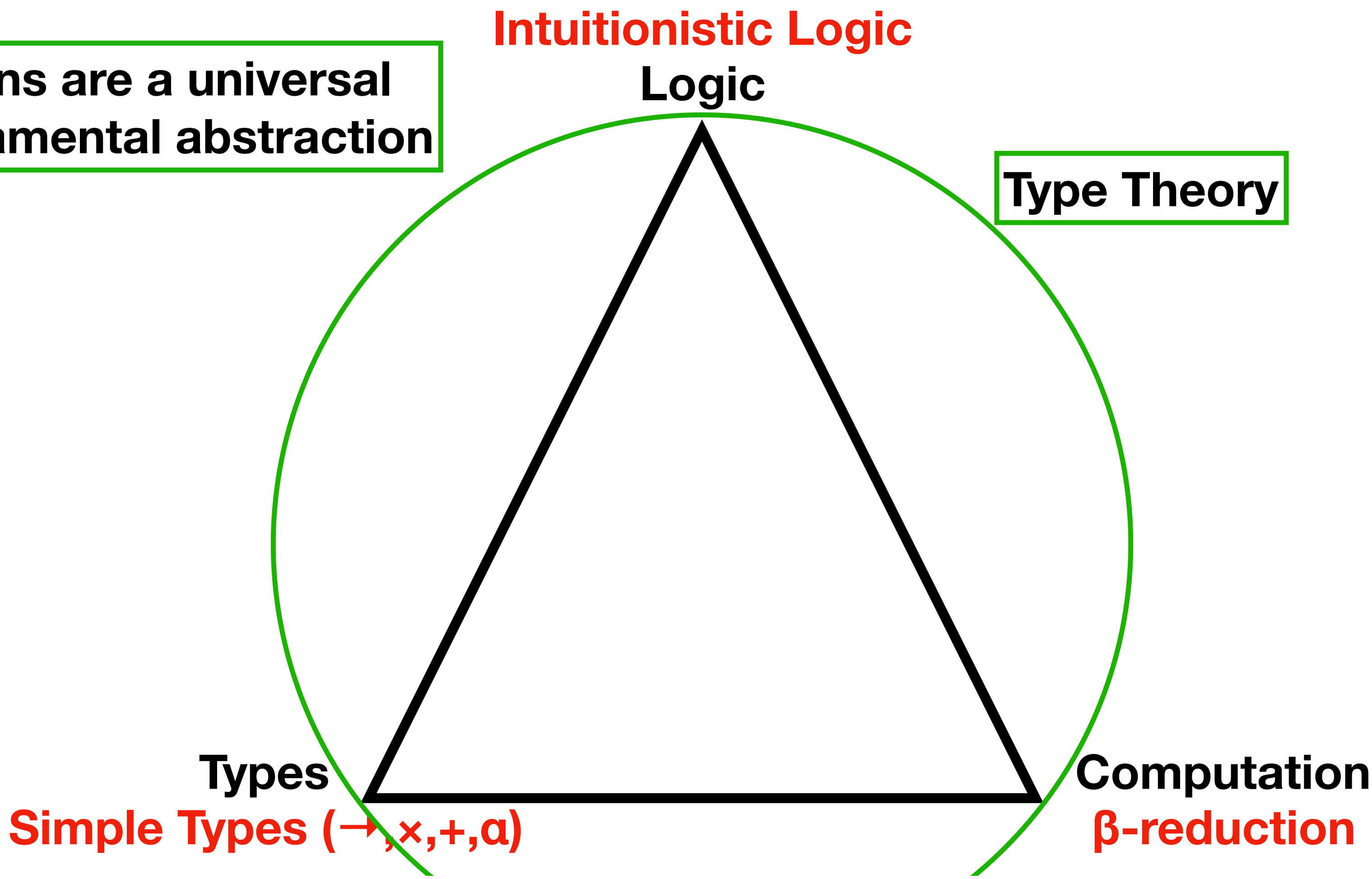
Why is Functional Programming So Effective?

Functions are a universal and fundamental abstraction



Why is Functional Programming So Effective?

Functions are a universal and fundamental abstraction



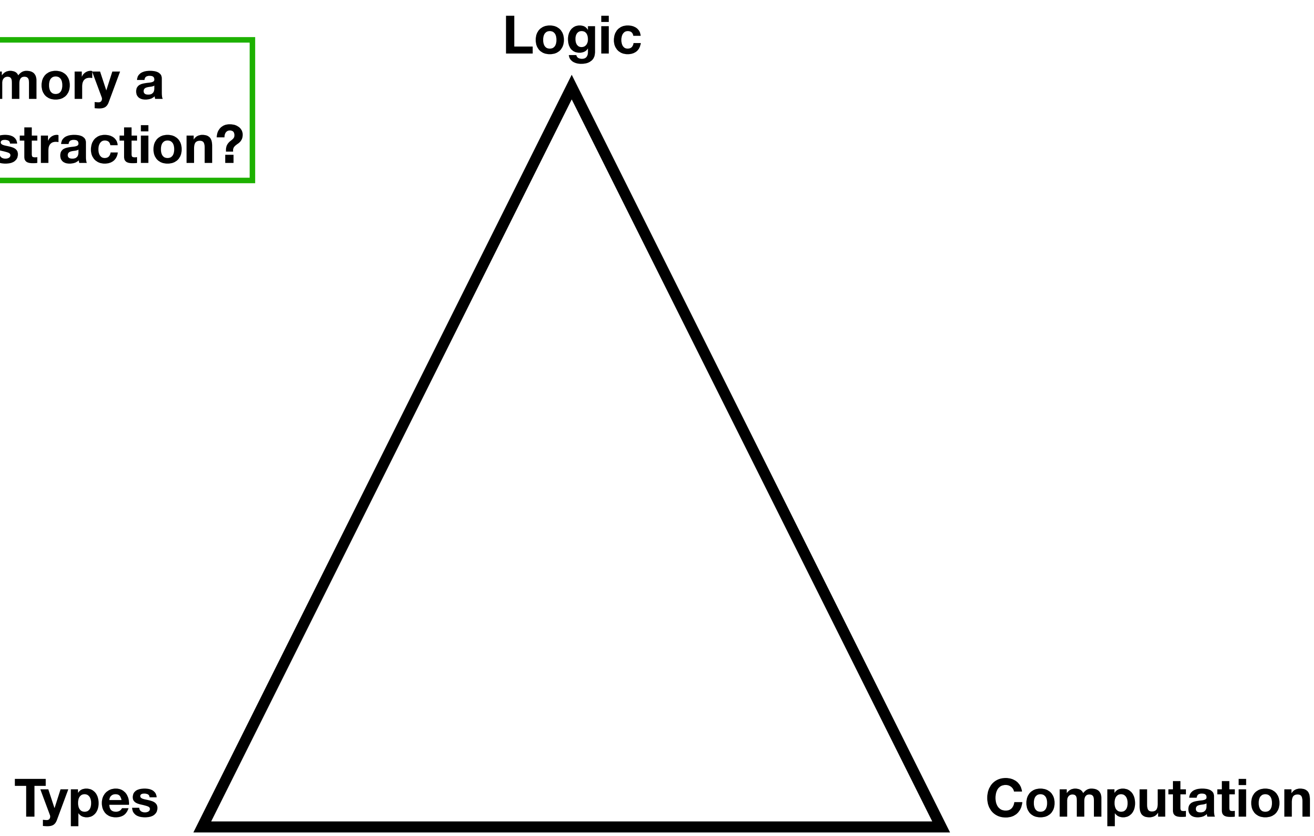
What about Concurrency?

What about Concurrency?

Is Shared Memory a Fundamental Abstraction?

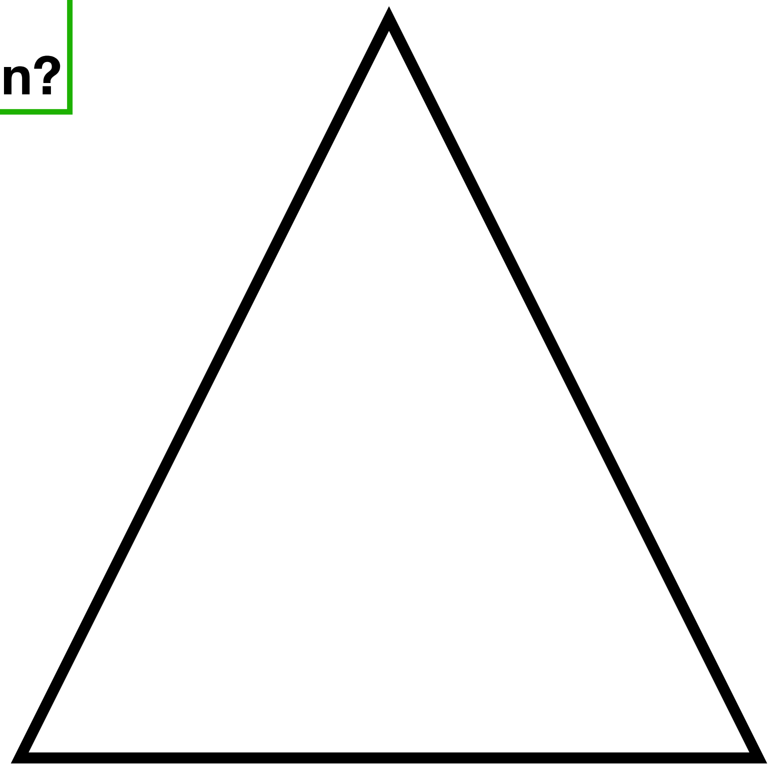
What about Concurrency?

Is Shared Memory a Fundamental Abstraction?



What about Concurrency?

Is Shared Memory a Fundamental Abstraction?



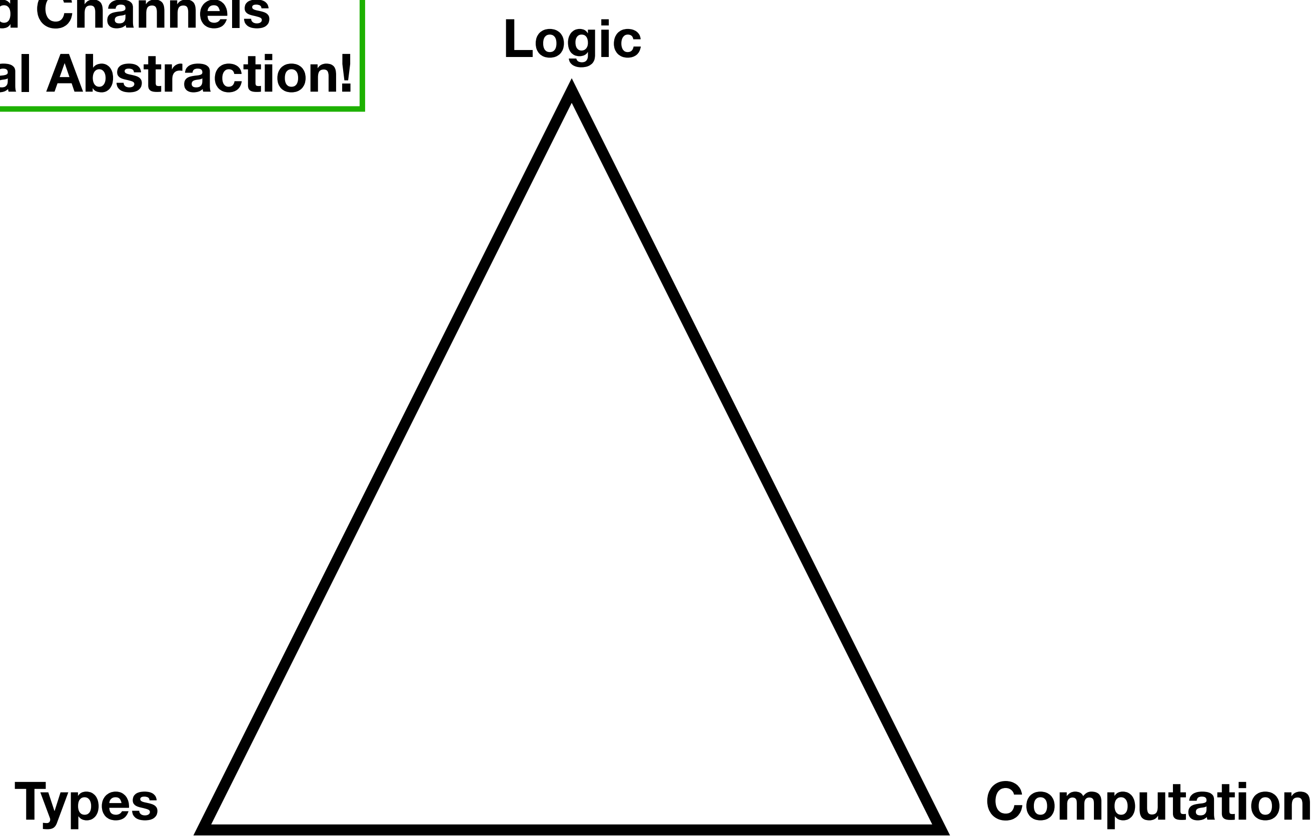
What about Concurrency?

What about Concurrency?

Processes and Channels
are a Fundamental Abstraction!

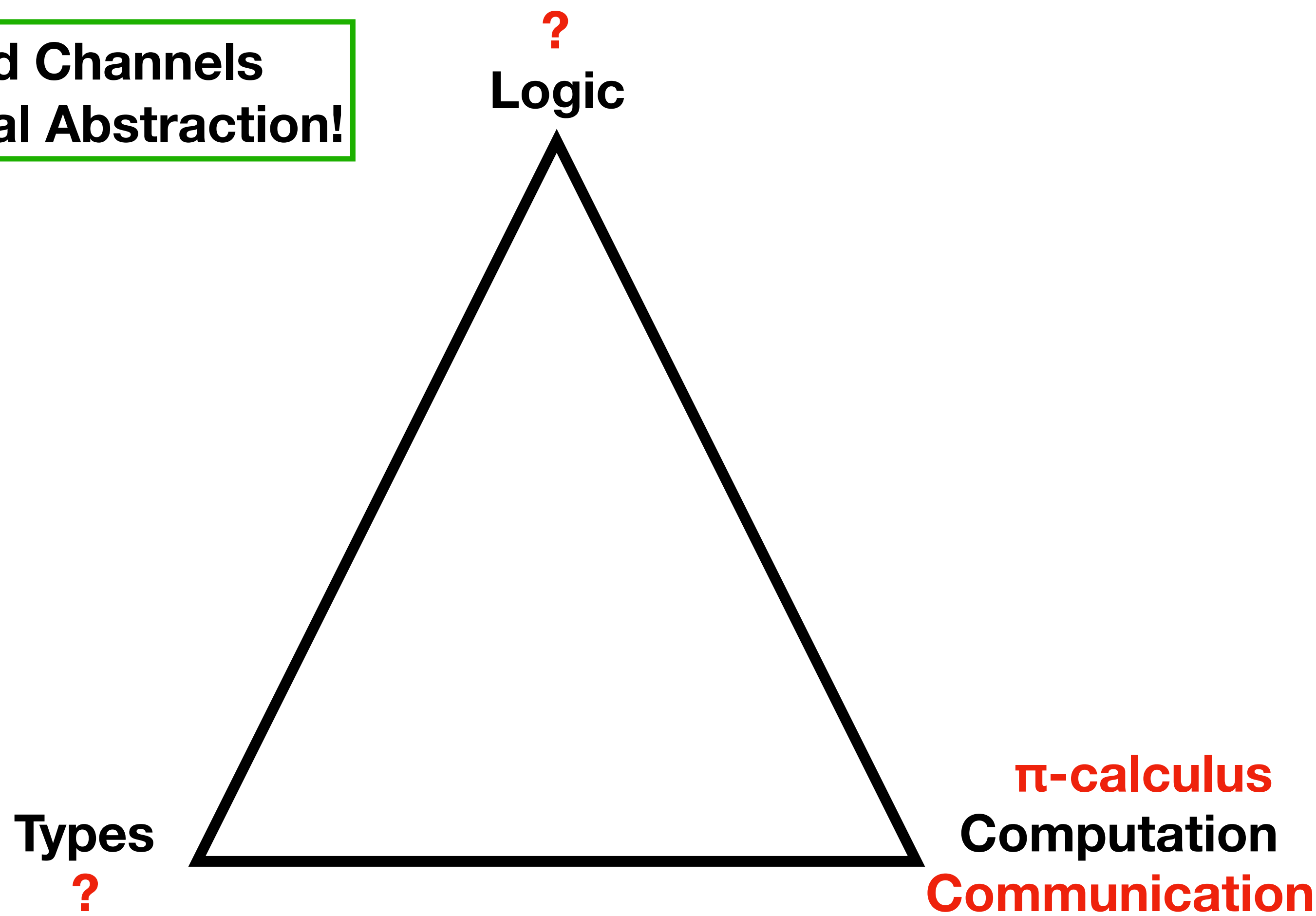
What about Concurrency?

Processes and Channels are a Fundamental Abstraction!



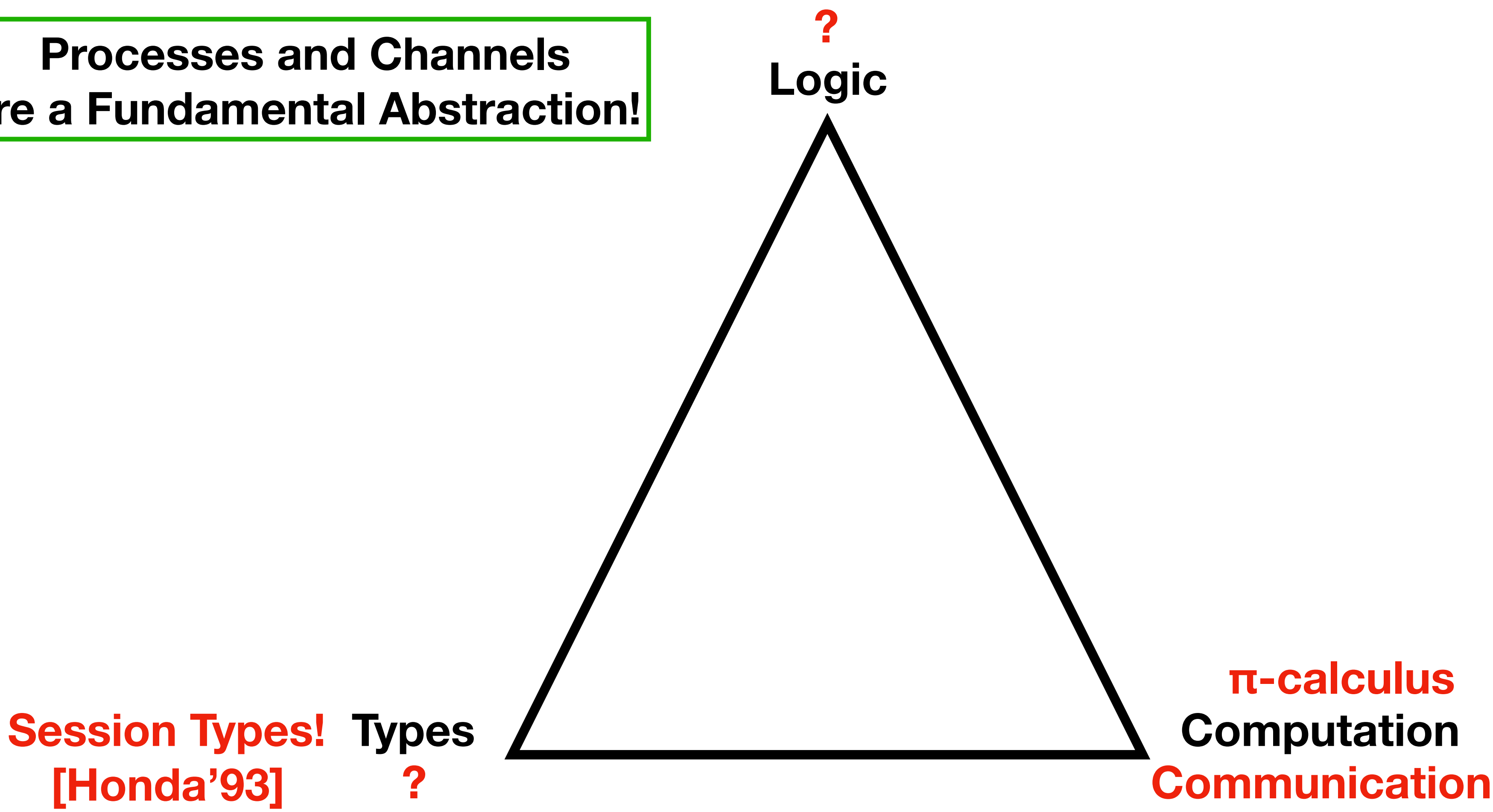
What about Concurrency?

Processes and Channels are a Fundamental Abstraction!



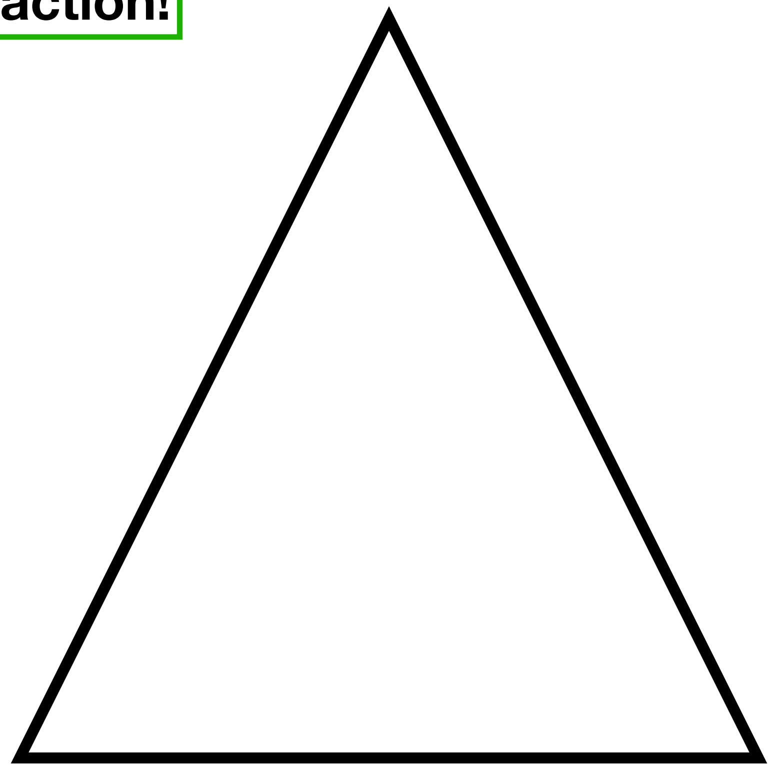
What about Concurrency?

Processes and Channels are a Fundamental Abstraction!

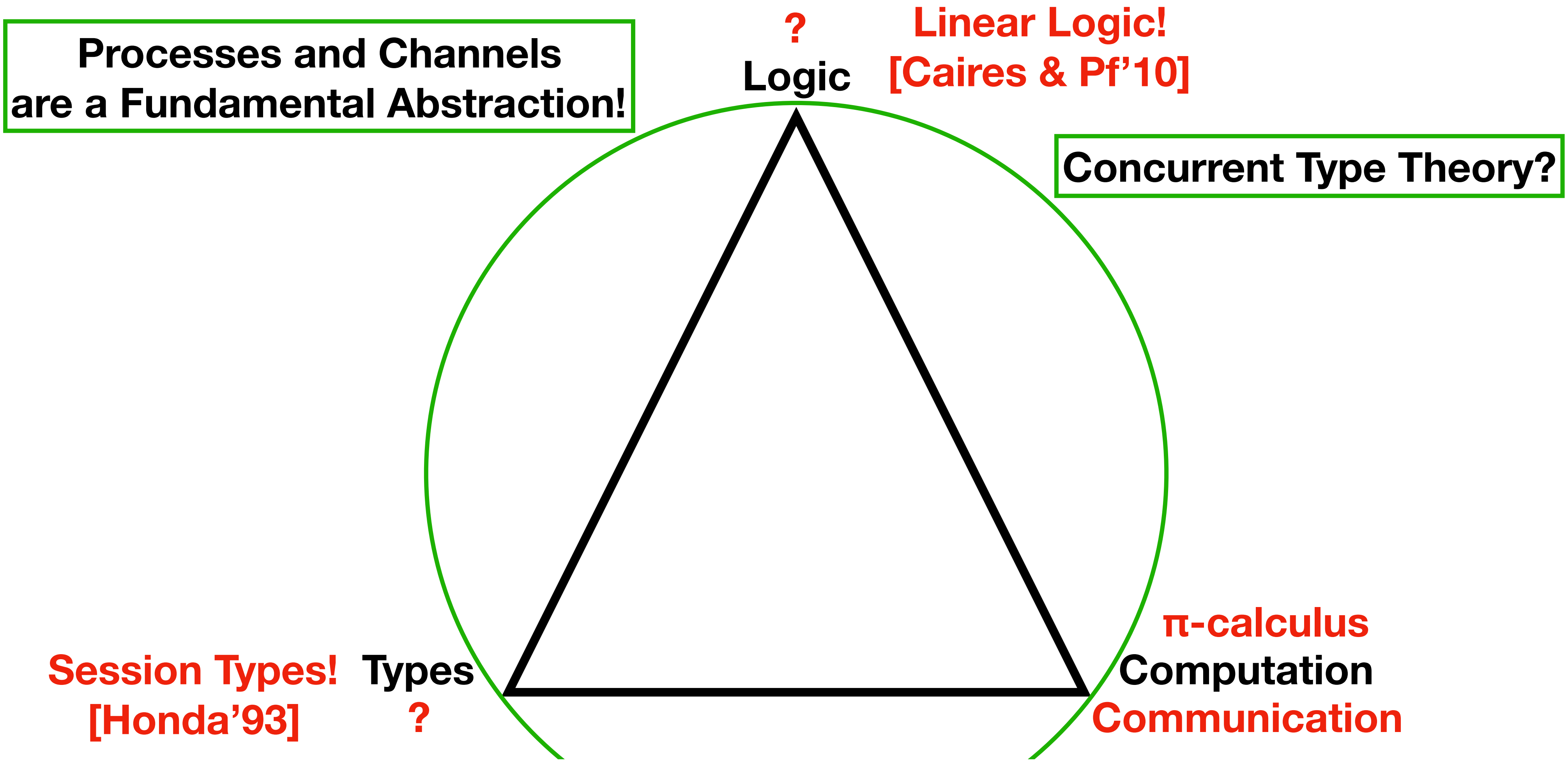


What about Concurrency?

Processes and Channels are a Fundamental Abstraction!



What about Concurrency?



There is Hope

There is Hope

- Previous talk!

There is Hope

- Previous talk!
- From the language point of view: Go

There is Hope

- Previous talk!
- From the language point of view: Go
  - Goroutines (threads/processes) as a fundamental abstraction
  - Channels (chan T) as a fundamental abstraction

There is Hope

- Previous talk!
- From the language point of view: Go
  - Goroutines (threads/processes) as a fundamental abstraction
  - Channels (`chan T`) as a fundamental abstraction
- Connection to logic is missing

There is Hope

- Previous talk!
- From the language point of view: Go
  - Goroutines (threads/processes) as a fundamental abstraction
  - Channels (`chan τ`) as a fundamental abstraction
- Connection to logic is missing
- Types are not expressive enough

There is Hope

Do not communicate by sharing memory; instead, share memory by communicating.
—Effective Go

- Previous talk!
- From the language point of view: Go
    - Goroutines (threads/processes) as a fundamental abstraction
    - Channels (chan τ) as a fundamental abstraction
- Connection to logic is missing
- Types are not expressive enough

Example: A Store (Stack or Queue)

Example: A Store (Stack or Queue)

Example: A Store (Stack or Queue)

- Protocol

Example: A Store (Stack or Queue)

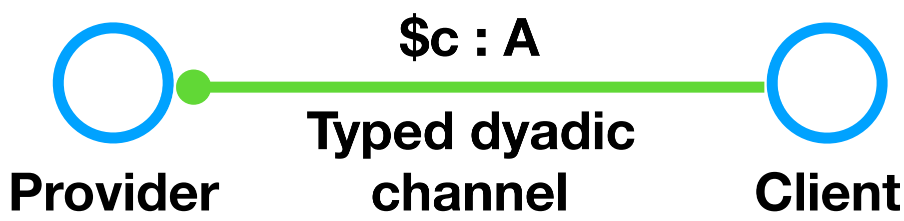
- Protocol
    - Client: ins; x; *recurse…*

Example: A Store (Stack or Queue)

- Protocol
  - Client: ins; x; *recurse…*
  - Client: del

Example: A Store (Stack or Queue)

- Protocol
  - Client: ins; x; *recurse…*
  - Client: del
    - Provider: none; close.

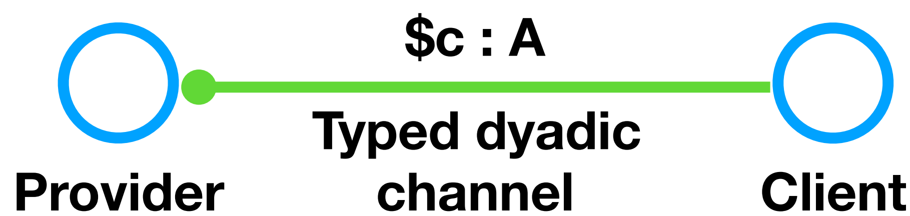


Example: A Store (Stack or Queue)

- Protocol
  - Client: ins; x; *recurse…*
  - Client: del
    - Provider: none; close.
    - Provider: some; x; *recurse…*

Example: A Store (Stack or Queue)

- Protocol
  - Client: ins; x; *recurse…*
  - Client: del
    - Provider: none; close.
    - Provider: some; x; *recurse…*
- Protocol should be expressed by a type!



Example: A Store (Stack or Queue)

- Protocol
  - Client: ins; x; *recurse…*
  - Client: del

Client's choice (external)

    - Provider: none; close.
    - Provider: some; x; *recurse…*
- Protocol should be expressed by a type!

Example: A Store (Stack or Queue)

- Protocol
  - Client: ins; x; *recurse…*

    Client's choice (external)
  - Client: del
    - Provider: none; close.

      Provider's choice (internal)
    - Provider: some; x; *recurse…*
- Protocol should be expressed by a type!

A Simple Client, in CC0

A Simple Client, in CC0

```
int main() {
```

A Simple Client, in CC0

```
int main() {
  int n = 10;
```

A Simple Client, in CC0

```
int main() {
  int n = 10;
  stack $s = empty();
```

A Simple Client, in CC0

```
int main() {
  int n = 10;
  stack $s = empty();
  for (int i = 0; i < n; i++) {
```

A Simple Client, in CC0

```
int main() {
  int n = 10;
  stack $s = empty();
  for (int i = 0; i < n; i++) {
    $s.ins; send($s, i);
```

A Simple Client, in CC0

```
int main() {
  int n = 10;
  stack $s = empty();
  for (int i = 0; i < n; i++) {
    $s.ins; send($s, i);
  }
```

A Simple Client, in CC0

```
int main() {
  int n = 10;
  stack $s = empty();
  for (int i = 0; i < n; i++) {
    $s.ins; send($s, i);
  }
  print_stack($s);
```

A Simple Client, in CC0

```
int main() {
  int n = 10;
  stack $s = empty();
  for (int i = 0; i < n; i++) {
    $s.ins; send($s, i);
  }
  print_stack($s);
  return 0;
```

A Simple Client, in CC0

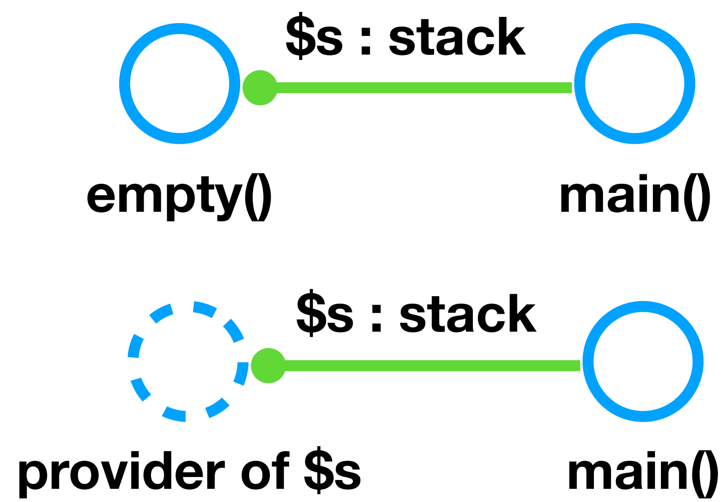
```
int main() {
  int n = 10;
  stack $s = empty();
  for (int i = 0; i < n; i++) {
    $s.ins; send($s, i);
  }
  print_stack($s);
  return 0;
}
```

A Simple Client, in CC0


```
int main() {
  int n = 10;
  stack $s = empty();
  for (int i = 0; i < n; i++) {
    $s.ins; send($s, i);
  }
  print_stack($s);
  return 0;
}
```

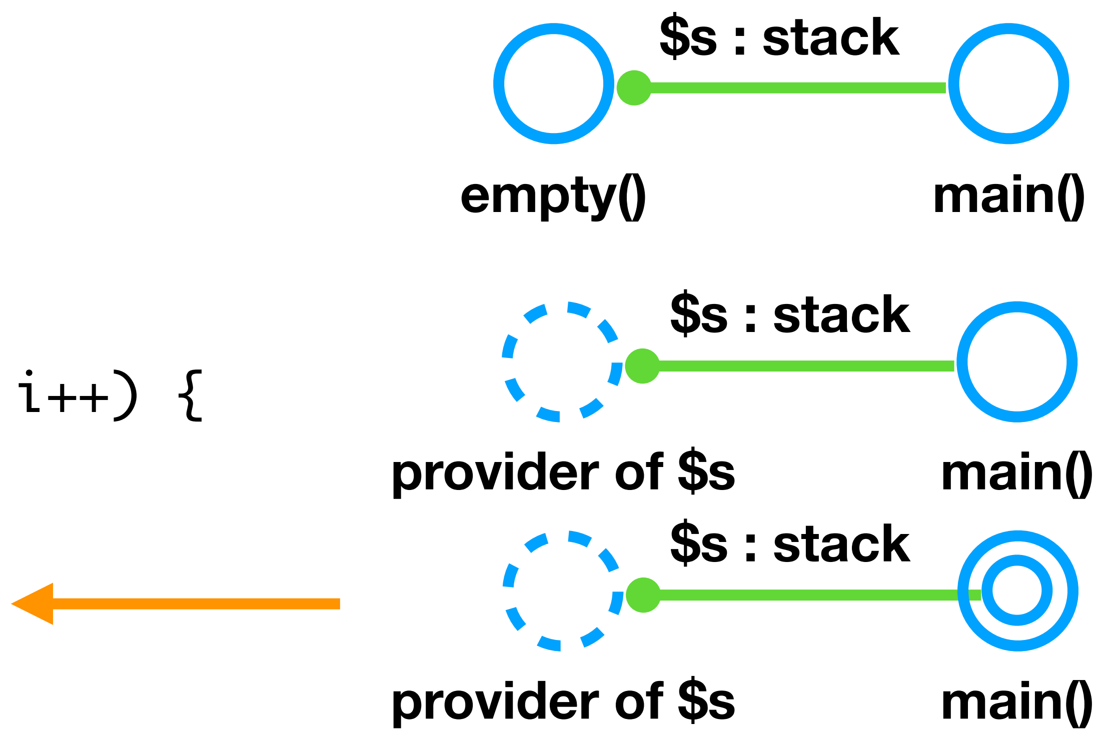
A Simple Client, in CC0

```
int main() {
  int n = 10;
  stack $s = empty();
  for (int i = 0; i < n; i++) {
    $s.ins; send($s, i);
  }
  print_stack($s);
  return 0;
}
```



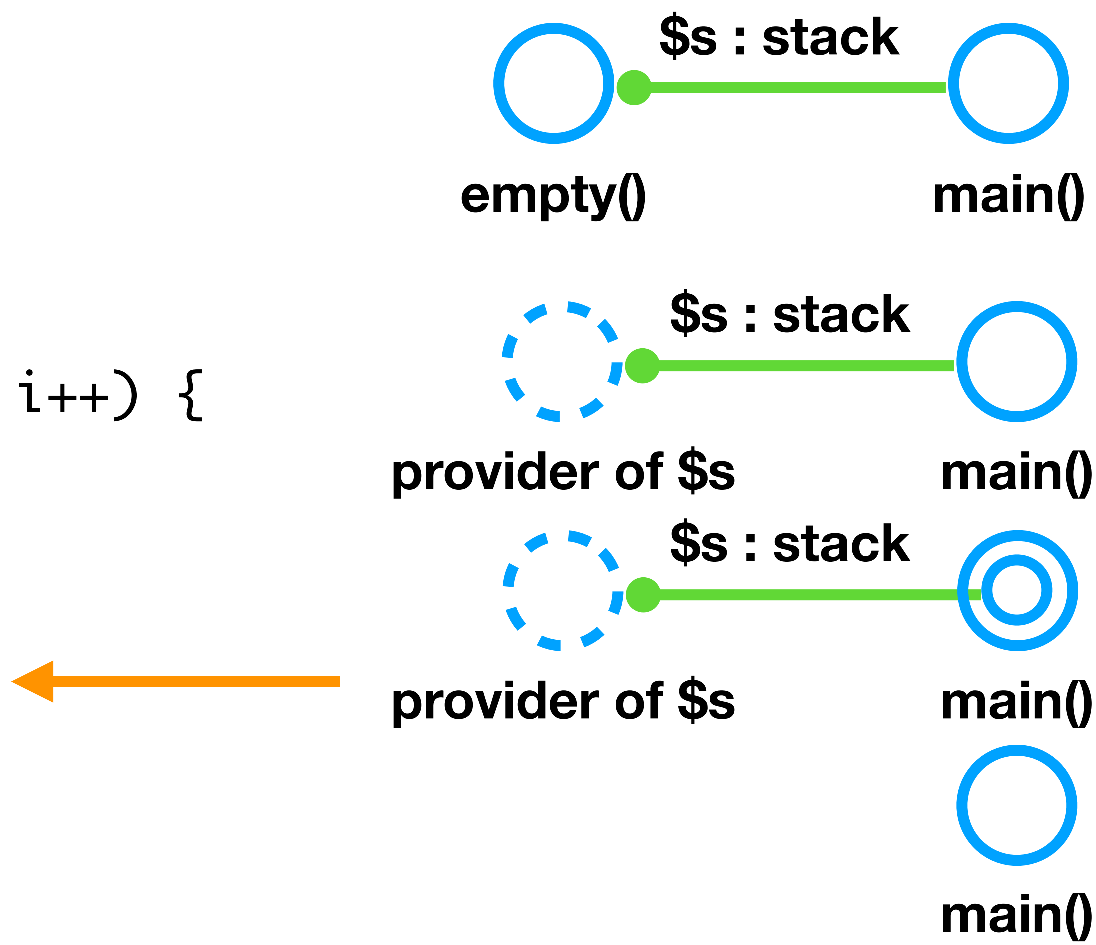
A Simple Client, in CC0

```
int main() {
  int n = 10;
  stack $s = empty();
  for (int i = 0; i < n; i++) {
    $s.ins; send($s, i);
  }
  print_stack($s);
  return 0;
}
```



A Simple Client, in CC0

```
int main() {
  int n = 10;
  stack $s = empty();
  for (int i = 0; i < n; i++) {
    $s.ins; send($s, i);
  }
  print_stack($s);
  return 0;
}
```



A Simple Provider, in CC0

A Simple Provider, in CC0

```
stack $s elem(int x, stack $t) {
```

A Simple Provider, in CC0

```
stack $s elem(int x, stack $t) {
  switch ($s) {
```

A Simple Provider, in CC0

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
```

A Simple Provider, in CC0

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
```

A Simple Provider, in CC0

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
```

A Simple Provider, in CC0

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
```

A Simple Provider, in CC0

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
```

A Simple Provider, in CC0

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
    case del: {
```

A Simple Provider, in CC0

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
    case del: {
      $s.some;
```

A Simple Provider, in CC0

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
    case del: {
      $s.some;
      send($s, x);
```

A Simple Provider, in CC0

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
    case del: {
      $s.some;
      send($s, x);
      $s = $t;
```

A Simple Provider, in CC0

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
    case del: {
      $s.some;
      send($s, x);
      $s = $t;
    }
```

A Simple Provider, in CC0

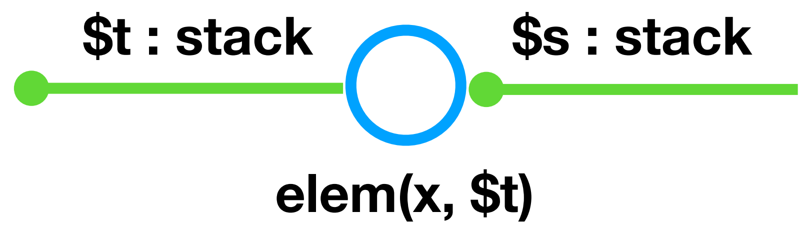
```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
    case del: {
      $s.some;
      send($s, x);
      $s = $t;
    }
  }
```

A Simple Provider, in CC0

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
    case del: {
      $s.some;
      send($s, x);
      $s = $t;
    }
  }
}
```

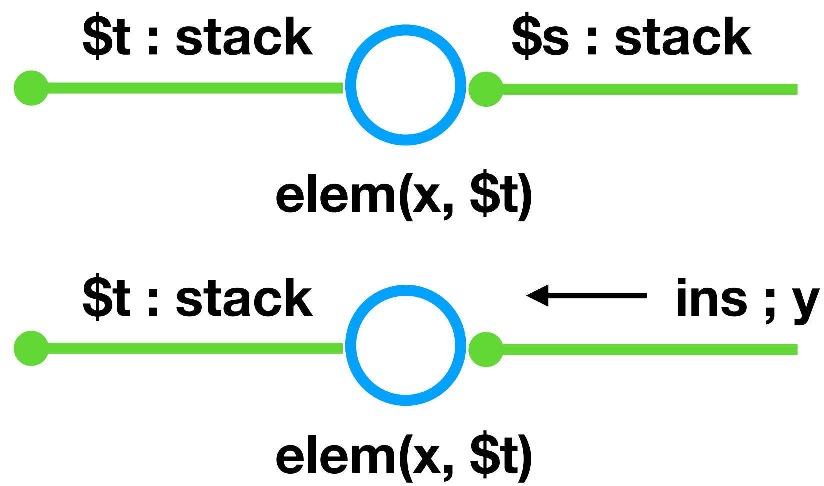
A Simple Provider, in CC0

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
    case del: {
      $s.some;
      send($s, x);
      $s = $t;
    }
  }
}
```



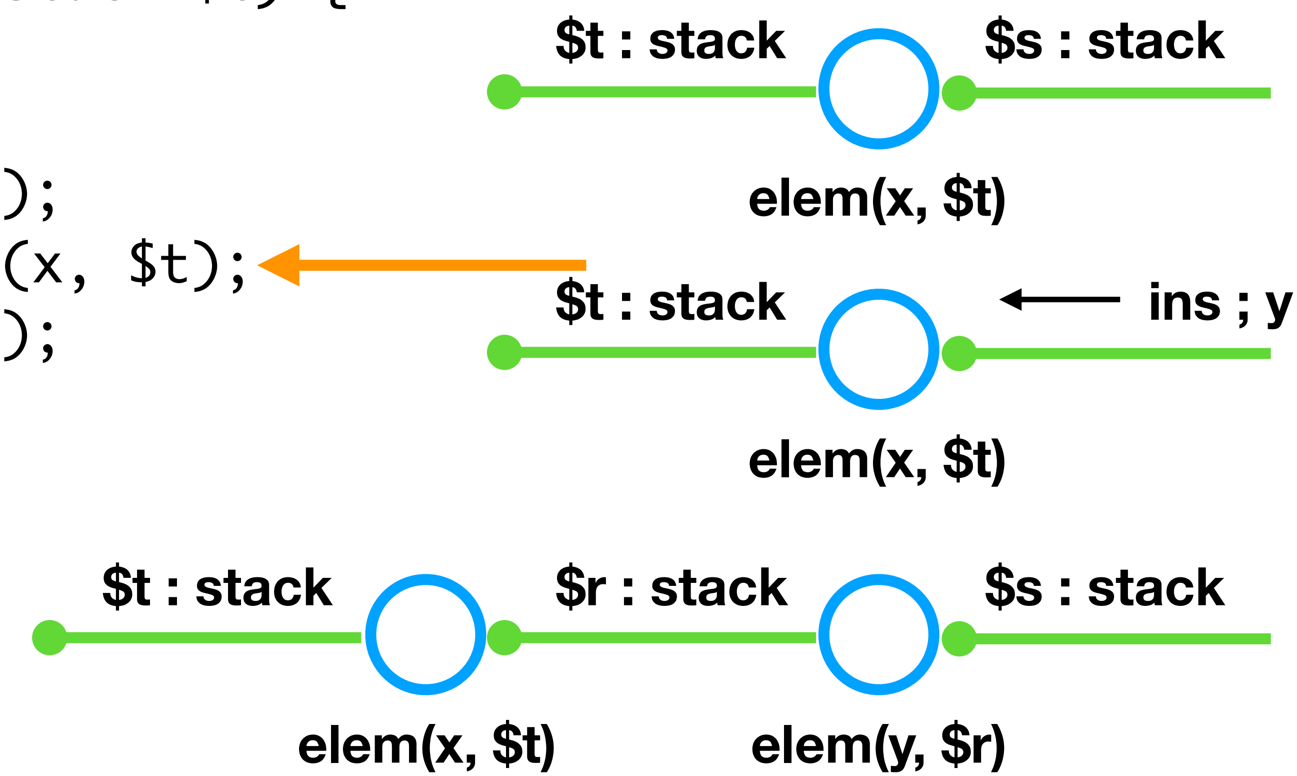
A Simple Provider, in CC0

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
    case del: {
      $s.some;
      send($s, x);
      $s = $t;
    }
  }
}
```



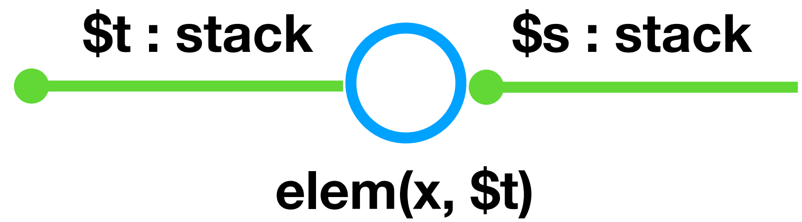
A Simple Provider, in CC0

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
    case del: {
      $s.some;
      send($s, x);
      $s = $t;
    }
  }
}
```

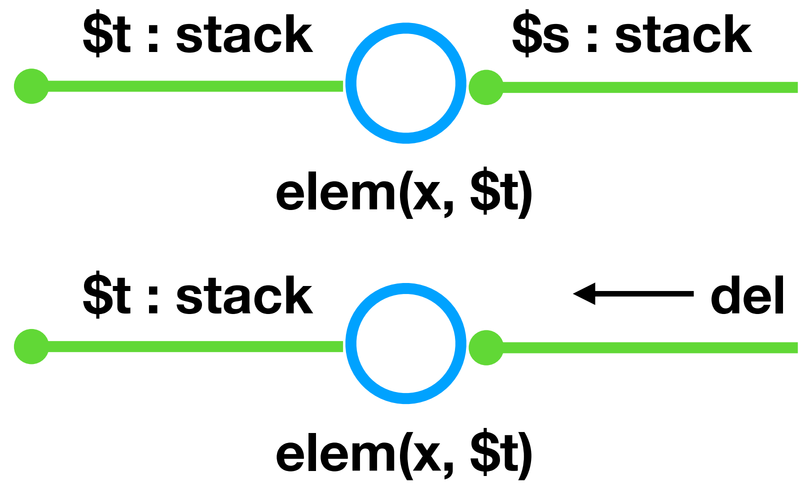


```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
    case del: {
      $s.some;
      send($s, x);
      $s = $t;
    }
  }
}
```

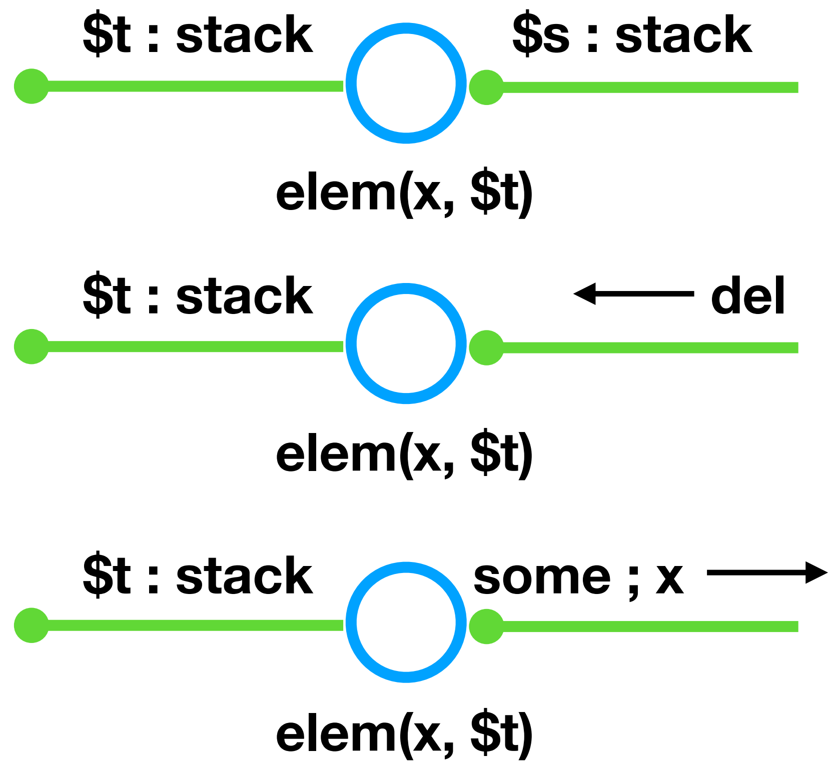
```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
    case del: {
      $s.some;
      send($s, x);
      $s = $t;
    }
  }
}
```



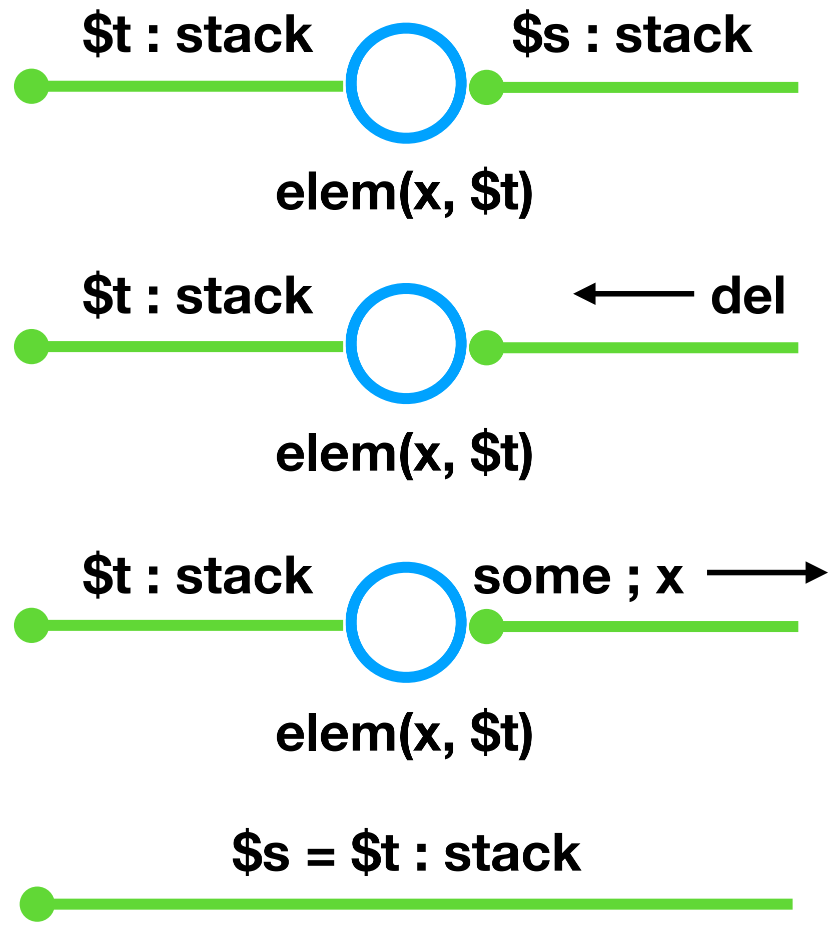
```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
    case del: {
      $s.some;
      send($s, x);
      $s = $t;
    }
  }
}
```



```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
    case del: {
      $s.some;
      send($s, x);
      $s = $t;
    }
  }
}
```

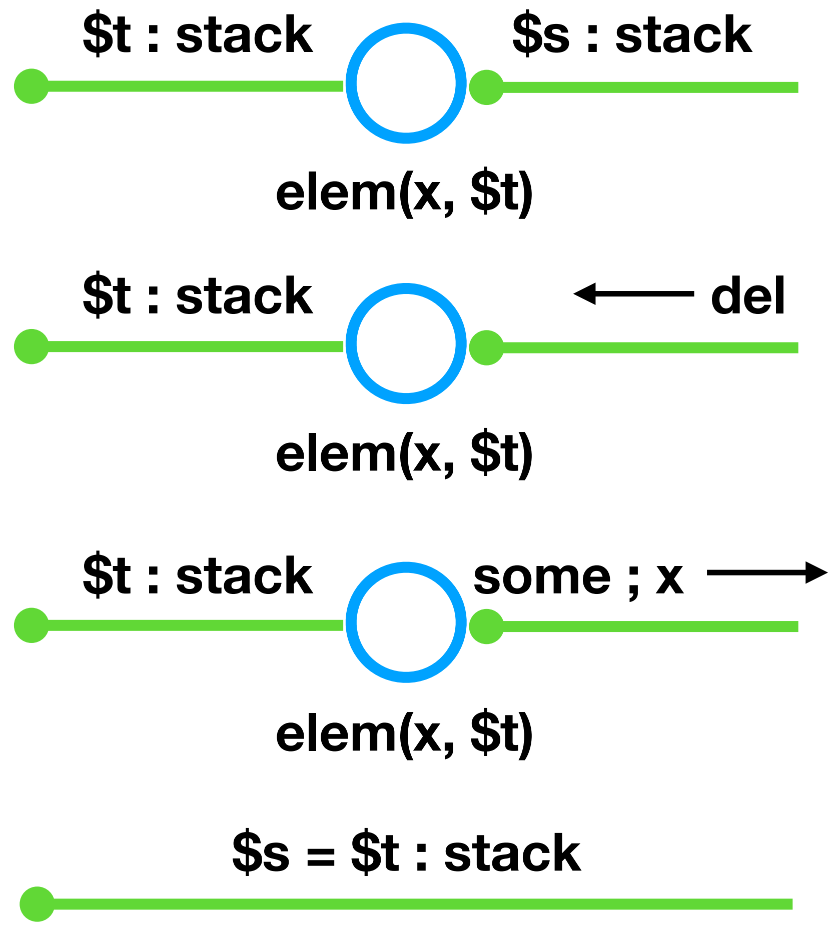


```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
    case del: {
      $s.some;
      send($s, x);
      $s = $t;
    }
  }
}
```



```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {
      int y = recv($s);
      stack $r = elem(x, $t);
      $s = elem(y, $r);
    }
    case del: {
      $s.some;
      send($s, x);
      $s = $t;
    }
  }
}
```

Forwarding (or *channel identification*) is not part of the π-calculus



```
stack $s empty() {
```

```
stack $s empty() {
  switch ($s) {
```

```
stack $s empty() {
  switch ($s) {
    case ins: {
```

```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
```

```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
      stack $e = empty();
```

```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
      stack $e = empty();
      $s = elem(x, $e);
```

```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
      stack $e = empty();
      $s = elem(x, $e);
    }
```

```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
      stack $e = empty();
      $s = elem(x, $e);
    }
    case del: {
```

```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
      stack $e = empty();
      $s = elem(x, $e);
    }
    case del: {
      $s.none;
```

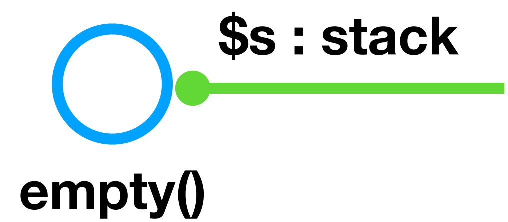
```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
      stack $e = empty();
      $s = elem(x, $e);
    }
    case del: {
      $s.none;
      close($s);
```

```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
      stack $e = empty();
      $s = elem(x, $e);
    }
    case del: {
      $s.none;
      close($s);
    }
```

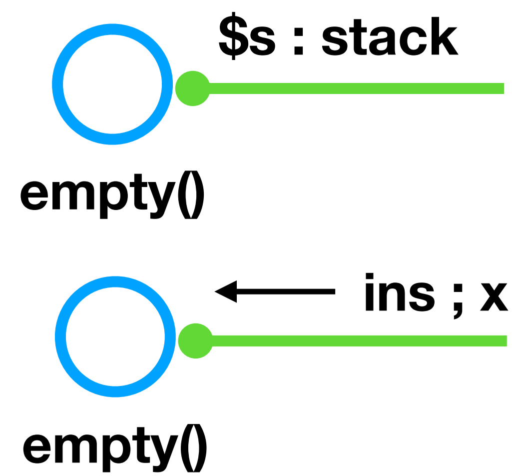
```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
      stack $e = empty();
      $s = elem(x, $e);
    }
    case del: {
      $s.none;
      close($s);
    }
  }
```

```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
      stack $e = empty();
      $s = elem(x, $e);
    }
    case del: {
      $s.none;
      close($s);
    }
  }
}
```

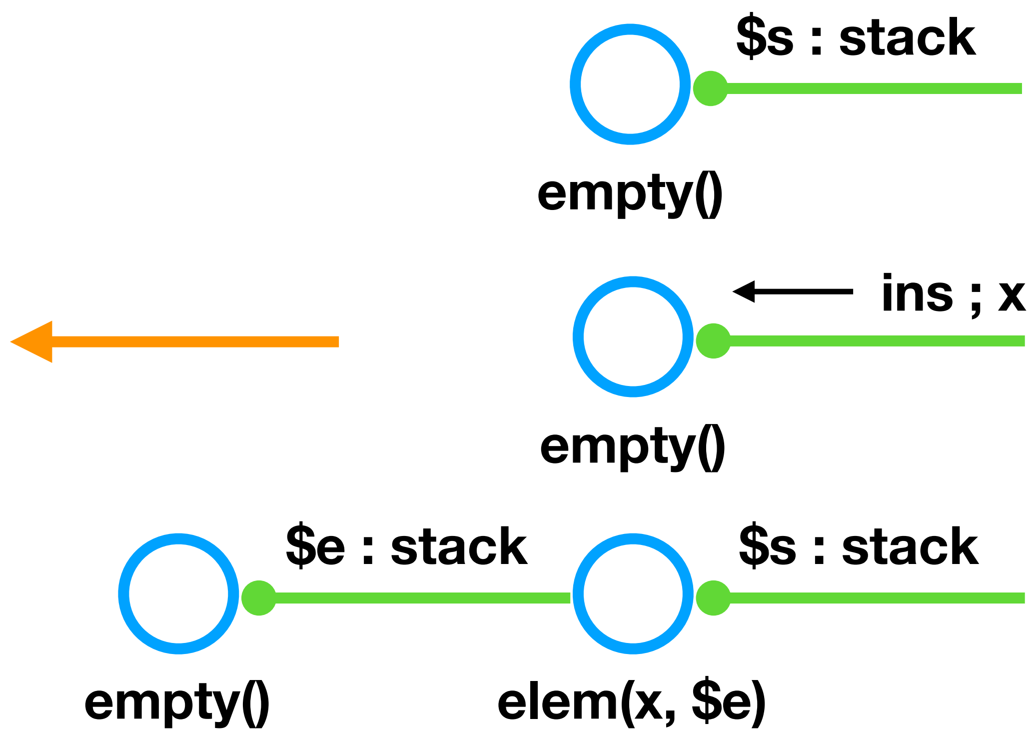
```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
      stack $e = empty();
      $s = elem(x, $e);
    }
    case del: {
      $s.none;
      close($s);
    }
  }
}
```



```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
      stack $e = empty();
      $s = elem(x, $e);
    }
    case del: {
      $s.none;
      close($s);
    }
  }
}
```

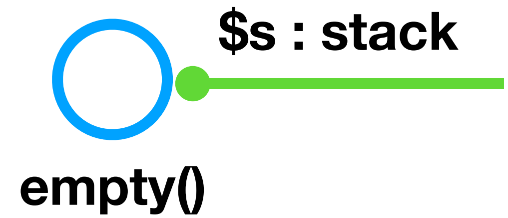


```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
      stack $e = empty();
      $s = elem(x, $e);
    }
    case del: {
      $s.none;
      close($s);
    }
  }
}
```

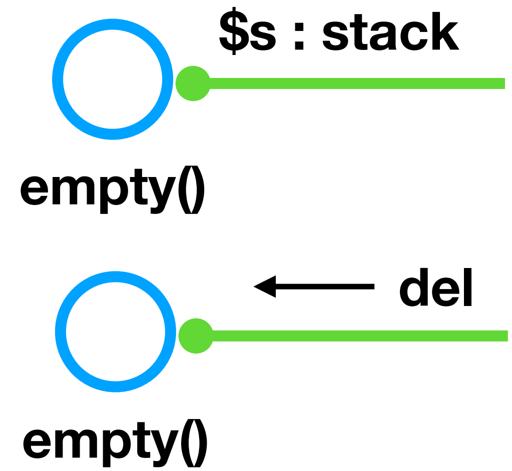


```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
      stack $e = empty();
      $s = elem(x, $e);
    }
    case del: {
      $s.none;
      close($s);
    }
  }
}
```

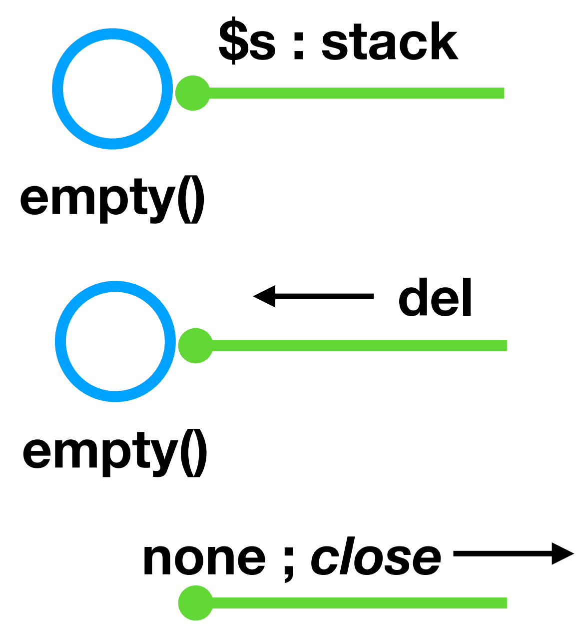
```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
      stack $e = empty();
      $s = elem(x, $e);
    }
    case del: {
      $s.none;
      close($s);
    }
  }
}
```



```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
      stack $e = empty();
      $s = elem(x, $e);
    }
    case del: {
      $s.none;
      close($s);
    }
  }
}
```



```
stack $s empty() {
  switch ($s) {
    case ins: {
      int x = recv($s);
      stack $e = empty();
      $s = elem(x, $e);
    }
    case del: {
      $s.none;
      close($s);
    }
  }
}
```



Summary So Far

- Processes provide one channel and are clients to other channels
- Spawning a process “returns” a fresh channel $c, with two endpoints
  - New process provides $c
  - Spawning process is client of $c
- Processes can terminate by forwarding
- Communication is bidirectional
  - Processes send and receive labels or integers

Typing Channels

- Channel types should encode protocol of communication
  - Provider and client must execute complementary actions
- External choice: Provider branches on label / Client sends label
- Internal choice: Provider sends label / Client branches on label
- Termination: Provider terminates / Client waits for termination
- Basic data: sending or receiving atomic values

Session Types, Abstractly

$$
\begin{array}{lllll}
 & & \text{Type} & \text{Provider action} & \text{Continuation} \\[1em]
A & ::= & \&\{\ell : A_\ell\}_{\ell \in L} & \text{receive some } k \in L & A_k \\
 & \mid & \oplus\{\ell : A_\ell\}_{\ell \in L} & \text{send some } k \in L & A_k \\
 & \mid & A \multimap B & \text{receive channel } c : A & B \\
 & \mid & A \otimes B & \text{send channel } c : A & B \\
 & \mid & \mathbf{1} & \text{terminate} & \textit{none} \\
 & \mid & \forall x{:}\tau.\, A & \text{receive } v : \tau & [v/x]A \\
 & \mid & \exists x{:}\tau.\, A & \text{send } v : \tau & [v/x]A
\end{array}
$$

Session Types, Abstractly

		Type	Provider action	Continuation
A	$::=$	$\&\{\ell : A_\ell\}_{\ell \in L}$	receive some $k \in L$	A_k
	$\mid$	$\oplus\{\ell : A_\ell\}_{\ell \in L}$	send some $k \in L$	A_k
	$\mid$	$A \multimap B$	receive channel $c : A$	B
	$\mid$	$A \otimes B$	send channel $c : A$	B
	$\mid$	$\mathbf{1}$	terminate	*none*
	$\mid$	$\forall x{:}\tau.\, A$	receive $v : \tau$	$[v/x]A$
	$\mid$	$\exists x{:}\tau.\, A$	send $v : \tau$	$[v/x]A$

$$\boxed{\begin{aligned} \mathit{stack}_A = \&\{\ & \mathsf{ins} : A \multimap \mathit{stack}_A, \\ & \mathsf{del} : \oplus\{\ \mathsf{none} : \mathbf{1}, \\ & \qquad\qquad \mathsf{some} : A \otimes \mathit{stack}_A\ \}\ \} \end{aligned}}$$

Types as Propositions

Types as Propositions

- These types are exactly the propositions of linear logic, except ‘!’

Types as Propositions

- These types are exactly the propositions of linear logic, except ‘!’
- An instance of the Curry-Howard correspondence

Types as Propositions

- These types are exactly the propositions of linear logic, except ‘!’
- An instance of the Curry-Howard correspondence
    - Typing rules correspond to sequent calculus inference rules

Types as Propositions

- These types are exactly the propositions of linear logic, except ‘!’
- An instance of the Curry-Howard correspondence
  - Typing rules correspond to sequent calculus inference rules
  - Programs correspond to process expressions

Types as Propositions

- These types are exactly the propositions of linear logic, except ‘!’
- An instance of the Curry-Howard correspondence
  - Typing rules correspond to sequent calculus inference rules
  - Programs correspond to process expressions
  - Communication corresponds to cut reduction

Session Typing Judgments

Session Typing Judgments

channels used by P

$$\underbrace{x_1 : A_1, \ldots, x_n : A_n}_{\Gamma} \vdash P :: (x : A)$$

channel provided by P

Session Typing Judgments

channels used by P

$$\underbrace{x_1 : A_1, \ldots, x_n : A_n}_{\Gamma} \vdash P :: (x : A)$$

channel provided by P

Configuration $\quad \Omega ::= (P_1 \mid \cdots \mid P_n)$

Session Typing Judgments

channels used by P

$$\underbrace{x_1 : A_1, \ldots, x_n : A_n}_{\Gamma} \vdash P :: (x : A)$$

channel provided by P

Configuration $\Omega ::= (P_1 \mid \cdots \mid P_n)$

Configuration Typing $\Gamma \models \Omega :: \Gamma'$

channels used by Ω **channels provided by Ω**

Theoretical Properties

Theoretical Properties

- Without recursive types and processes
  - Session fidelity (Preservation)
  - Deadlock freedom
  - Termination
- With recursion
  - Session fidelity (Preservation)
  - Deadlock freedom

Theoretical Properties

- Without recursive types and processes
  - Session fidelity (Preservation) $\text{If } \Gamma \models \Omega :: \Gamma' \text{ and } \Omega \mapsto \Omega' \text{ then } \Gamma \models \Omega :: \Omega'$
  - Deadlock freedom
  - Termination
- With recursion
  - Session fidelity (Preservation)
  - Deadlock freedom

Theoretical Properties

- Without recursive types and processes
  - Session fidelity (Preservation) $\text{If } \Gamma \models \Omega :: \Gamma' \text{ and } \Omega \mapsto \Omega' \text{ then } \Gamma \models \Omega :: \Omega'$
  - Deadlock freedom $\text{If } \Gamma \models \Omega :: \Gamma' \text{ then } \Omega \textit{ poised} \text{ or } \Omega \mapsto \Omega' \text{ for some } \Omega'$
  - Termination
- With recursion
  - Session fidelity (Preservation)
  - Deadlock freedom

Theoretical Properties

- Without recursive types and processes
  - Session fidelity (Preservation) $\text{If } \Gamma \models \Omega :: \Gamma' \text{ and } \Omega \mapsto \Omega' \text{ then } \Gamma \models \Omega :: \Omega'$
  - Deadlock freedom $\text{If } \Gamma \models \Omega :: \Gamma' \text{ then } \Omega \textit{ poised} \text{ or } \Omega \mapsto \Omega' \text{ for some } \Omega'$
  - Termination $\text{If } \Gamma \vdash \Omega :: \Gamma' \text{ then } \Omega \mapsto^* \Omega' \text{ for } \Omega' \textit{ poised}$
- With recursion
  - Session fidelity (Preservation)
  - Deadlock freedom

Theoretical Properties

- Without recursive types and processes
  - Session fidelity (Preservation) $\quad \text{If } \Gamma \models \Omega :: \Gamma' \text{ and } \Omega \mapsto \Omega' \text{ then } \Gamma \models \Omega :: \Omega'$
  - Deadlock freedom $\quad \text{If } \Gamma \models \Omega :: \Gamma' \text{ then } \Omega \textit{ poised} \text{ or } \Omega \mapsto \Omega' \text{ for some } \Omega'$
  - Termination $\quad \text{If } \Gamma \vdash \Omega :: \Gamma' \text{ then } \Omega \mapsto^* \Omega' \text{ for } \Omega' \textit{ poised}$
- With recursion
  - Session fidelity (Preservation)
  - Deadlock freedom

Ω is *poised* if all processes in Ω attempt to communicate along a channel in the external interface

Mode of Communication

- Both synchronous and asynchronous communication can be supported
- Asynchronous: messages still must appear in order (for session fidelity)
  - Synchronization via polarization of the types
- Synchronous: messages can be coded via one-action processes
- Asynchronous seems to be the right default
  - Closer to reasonable implementation
  - Generalizes to channels with multiple endpoints

Session Types, in CC0

? = receive
! = send
; = sequence of interaction
<…> session type

Session Types, in CC0

```
choice stack_req {
  <?int ; ?choice stack_req> ins;
  <!choice stack_response>   del;
};

choice stack_response {
  < >                        none;
  <!int ; ?choice stack_req> some;
};

typedef <?choice stack_req> stack;
```

? = receive
! = send
; = sequence of interaction
<…> session type

Tracing the Type-Checker

Tracing the Type-Checker

```
stack $s elem(int x, stack $t) {
```

Tracing the Type-Checker

```
stack $s elem(int x, stack $t) {
  switch ($s) {
```

Tracing the Type-Checker

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {                 % $s : <?int ; stack> -| $t : stack
```

Tracing the Type-Checker

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {                % $s : <?int ; stack> -| $t : stack
      int y = recv($s);        % $s : stack          -| $t : stack
```

Tracing the Type-Checker

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {                % $s : <?int ; stack> -| $t : stack
      int y = recv($s);        % $s : stack          -| $t : stack
      stack $r = elem(x, $t);  % $s : stack          -| $r : stack
```

Tracing the Type-Checker

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {                % $s : <?int ; stack> -| $t : stack
      int y = recv($s);        % $s : stack          -| $t : stack
      stack $r = elem(x, $t);  % $s : stack          -| $r : stack
      $s = elem(y, $r);
```

Tracing the Type-Checker

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {                % $s : <?int ; stack> -| $t : stack
      int y = recv($s);        % $s : stack          -| $t : stack
      stack $r = elem(x, $t);  % $s : stack          -| $r : stack
      $s = elem(y, $r);
    }
```

Tracing the Type-Checker

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {                % $s : <?int ; stack> -| $t : stack
      int y = recv($s);        % $s : stack          -| $t : stack
      stack $r = elem(x, $t);  % $s : stack          -| $r : stack
      $s = elem(y, $r);
    }
    case del: {                % $s : <!choice stack_response> -| …
```

Tracing the Type-Checker

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {                % $s : <?int ; stack> -| $t : stack
      int y = recv($s);        % $s : stack          -| $t : stack
      stack $r = elem(x, $t);  % $s : stack          -| $r : stack
      $s = elem(y, $r);
    }
    case del: {                % $s : <!choice stack_response> -| …
      $s.some;                 % $s : <!int ; stack> -| $t : stack
```

Tracing the Type-Checker

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {                  % $s : <?int ; stack> -| $t : stack
      int y = recv($s);          % $s : stack          -| $t : stack
      stack $r = elem(x, $t);    % $s : stack          -| $r : stack
      $s = elem(y, $r);
    }
    case del: {                  % $s : <!choice stack_response> -| …
      $s.some;                   % $s : <!int ; stack> -| $t : stack
      send($s, x);               % $s : stack          -| $t : stack
```

Tracing the Type-Checker

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {                % $s : <?int ; stack> -| $t : stack
      int y = recv($s);        % $s : stack          -| $t : stack
      stack $r = elem(x, $t);  % $s : stack          -| $r : stack
      $s = elem(y, $r);
    }
    case del: {                % $s : <!choice stack_response> -| …
      $s.some;                 % $s : <!int ; stack> -| $t : stack
      send($s, x);             % $s : stack          -| $t : stack
      $s = $t;
```

Tracing the Type-Checker

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {                 % $s : <?int ; stack> -| $t : stack
      int y = recv($s);         % $s : stack          -| $t : stack
      stack $r = elem(x, $t);   % $s : stack          -| $r : stack
      $s = elem(y, $r);
    }
    case del: {                 % $s : <!choice stack_response> -| …
      $s.some;                  % $s : <!int ; stack> -| $t : stack
      send($s, x);              % $s : stack          -| $t : stack
      $s = $t;
    }
```

Tracing the Type-Checker

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {                 % $s : <?int ; stack> -| $t : stack
      int y = recv($s);         % $s : stack          -| $t : stack
      stack $r = elem(x, $t);   % $s : stack          -| $r : stack
      $s = elem(y, $r);
    }
    case del: {                 % $s : <!choice stack_response> -| …
      $s.some;                  % $s : <!int ; stack> -| $t : stack
      send($s, x);              % $s : stack          -| $t : stack
      $s = $t;
    }
  }
```

Tracing the Type-Checker

```
stack $s elem(int x, stack $t) {
  switch ($s) {
    case ins: {                 % $s : <?int ; stack> -| $t : stack
      int y = recv($s);         % $s : stack          -| $t : stack
      stack $r = elem(x, $t);   % $s : stack          -| $r : stack
      $s = elem(y, $r);
    }
    case del: {                 % $s : <!choice stack_response> -| …
      $s.some;                  % $s : <!int ; stack> -| $t : stack
      send($s, x);              % $s : stack          -| $t : stack
      $s = $t;
    }
  }
}
```

Tracing the Type-Checker

Tracing the Type-Checker

```
stack $s empty() {
```

Tracing the Type-Checker

```
stack $s empty() {
  switch ($s) {
```

Tracing the Type-Checker

```
stack $s empty() {
  switch ($s) {
    case ins: {              % $s : <?int ; stack> -| .
```

Tracing the Type-Checker

```
stack $s empty() {
  switch ($s) {
    case ins: {              % $s : <?int ; stack> -| .
      int x = recv($s);      % $s : stack          -| .
```

Tracing the Type-Checker

```
stack $s empty() {
  switch ($s) {
    case ins: {              % $s : <?int ; stack> -| .
      int x = recv($s);      % $s : stack          -| .
      stack $e = empty();    % $s : stack          -| $e : stack
```

Tracing the Type-Checker

```
stack $s empty() {
  switch ($s) {
    case ins: {                 % $s : <?int ; stack> -| .
      int x = recv($s);         % $s : stack          -| .
      stack $e = empty();       % $s : stack          -| $e : stack
      $s = elem(x, $e);
```

Tracing the Type-Checker

```
stack $s empty() {
  switch ($s) {
    case ins: {                % $s : <?int ; stack> -| .
      int x = recv($s);        % $s : stack          -| .
      stack $e = empty();      % $s : stack          -| $e : stack
      $s = elem(x, $e);
    }
```

Tracing the Type-Checker

```
stack $s empty() {
  switch ($s) {
    case ins: {               % $s : <?int ; stack> -| .
      int x = recv($s);       % $s : stack          -| .
      stack $e = empty();     % $s : stack          -| $e : stack
      $s = elem(x, $e);
    }
    case del: {               % $s : <!stack_response> -| .
```

Tracing the Type-Checker

```
stack $s empty() {
  switch ($s) {
    case ins: {                % $s : <?int ; stack> -| .
      int x = recv($s);        % $s : stack          -| .
      stack $e = empty();      % $s : stack          -| $e : stack
      $s = elem(x, $e);
    }
    case del: {                % $s : <!stack_response> -| .
      $s.none;                 % $s : < >               -| .
```

Tracing the Type-Checker

```
stack $s empty() {
  switch ($s) {
    case ins: {                 % $s : <?int ; stack> -| .
      int x = recv($s);         % $s : stack          -| .
      stack $e = empty();       % $s : stack          -| $e : stack
      $s = elem(x, $e);
    }
    case del: {                 % $s : <!stack_response> -| .
      $s.none;                  % $s : < >               -| .
      close($s);
```

Tracing the Type-Checker

```
stack $s empty() {
  switch ($s) {
    case ins: {                 % $s : <?int ; stack> -| .
      int x = recv($s);         % $s : stack          -| .
      stack $e = empty();       % $s : stack          -| $e : stack
      $s = elem(x, $e);
    }
    case del: {                 % $s : <!stack_response> -| .
      $s.none;                  % $s : < >               -| .
      close($s);
    }
```

Tracing the Type-Checker

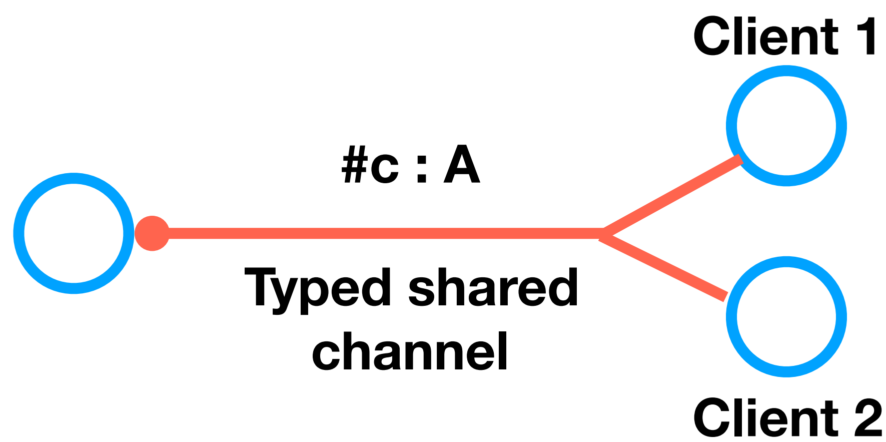
```
stack $s empty() {
  switch ($s) {
    case ins: {               % $s : <?int ; stack> -| .
      int x = recv($s);       % $s : stack          -| .
      stack $e = empty();     % $s : stack          -| $e : stack
      $s = elem(x, $e);
    }
    case del: {               % $s : <!stack_response> -| .
      $s.none;                % $s : < >               -| .
      close($s);
    }
  }
```

Tracing the Type-Checker

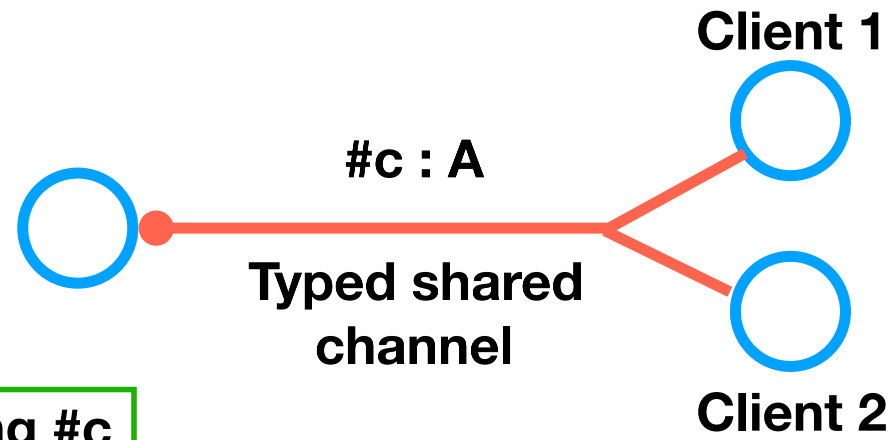
```
stack $s empty() {
  switch ($s) {
    case ins: {              % $s : <?int ; stack> -| .
      int x = recv($s);      % $s : stack          -| .
      stack $e = empty();    % $s : stack          -| $e : stack
      $s = elem(x, $e);
    }
    case del: {              % $s : <!stack_response> -| .
      $s.none;               % $s : < >               -| .
      close($s);
    }
  }
}
```

The Problem with Sharing

The Problem with Sharing

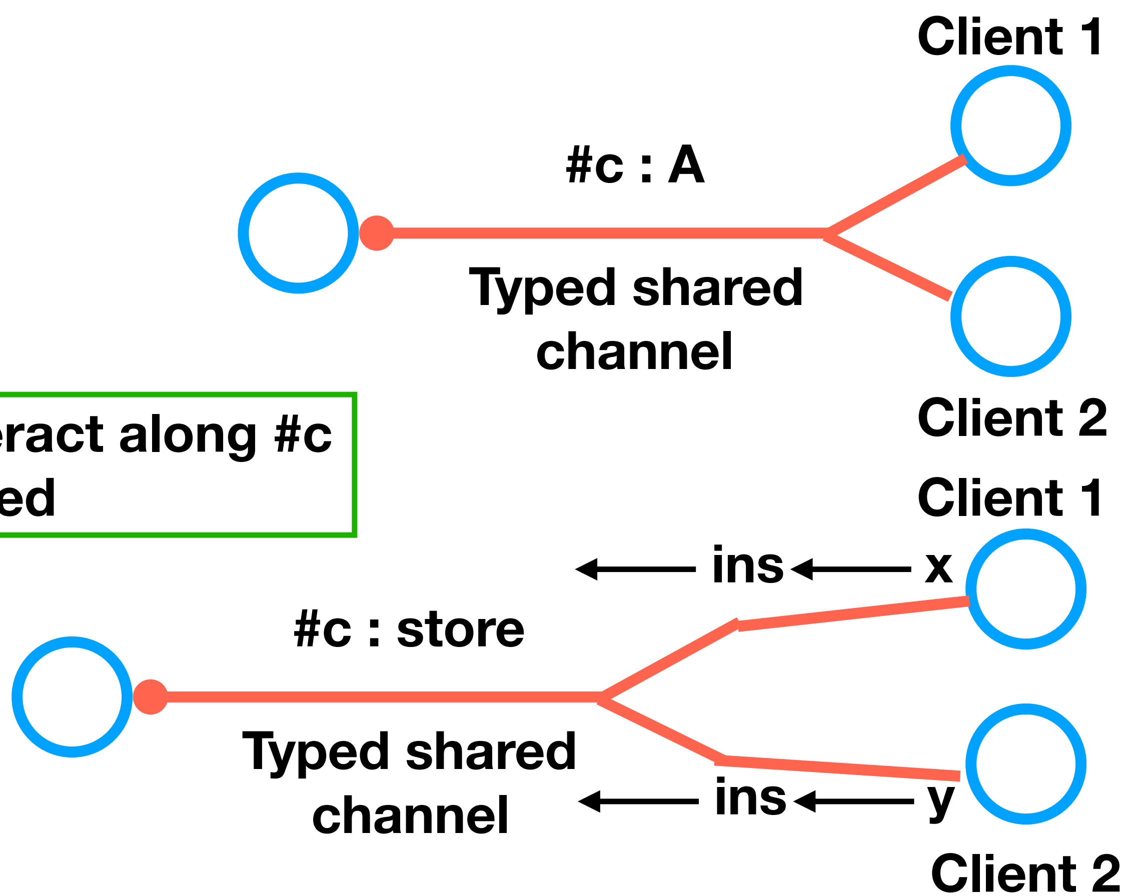


The Problem with Sharing



If both clients can freely interact along #c session fidelity will be violated

The Problem with Sharing



If both clients can freely interact along #c session fidelity will be violated

Linear and Shared Channels

		Type	Provider action	Continuation
A	$::=$	$\&\{\ell : A_\ell\}_{\ell \in L}$	receive some $k \in L$	A_k
	$\mid$	$\oplus\{\ell : A_\ell\}_{\ell \in L}$	send some $k \in L$	A_k
	$\mid$	$A \multimap B$	receive channel $c : A$	B
	$\mid$	$A \otimes B$	send channel $c : A$	B
	$\mid$	$\mathbf{1}$	terminate	*none*
	$\mid$	$\forall x{:}\tau.\, A$	receive $v : \tau$	$[v/x]A$
	$\mid$	$\exists x{:}\tau.\, A$	send $v : \tau$	$[v/x]A$
	$\mid$	$\downarrow S$	detach from client	S
S	$::=$	$\uparrow A$	accept client	A

Linear and Shared Channels

		Type	Provider action	Continuation
A	$::=$	$\&\{\ell : A_\ell\}_{\ell \in L}$	receive some $k \in L$	A_k
	$\mid$	$\oplus\{\ell : A_\ell\}_{\ell \in L}$	send some $k \in L$	A_k
	$\mid$	$A \multimap B$	receive channel $c : A$	B
	$\mid$	$A \otimes B$	send channel $c : A$	B
	$\mid$	$\mathbf{1}$	terminate	*none*
	$\mid$	$\forall x{:}\tau.\, A$	receive $v : \tau$	$[v/x]A$
	$\mid$	$\exists x{:}\tau.\, A$	send $v : \tau$	$[v/x]A$
	$\mid$	$\downarrow S$	detach from client	S
S	$::=$	$\uparrow A$	accept client	A

$$!A \triangleq \downarrow\uparrow A$$

A Shared Queue

$$\begin{aligned} queue_A = {\uparrow}\&\{\ & \mathsf{ins} : A \multimap {\downarrow}queue_A, \\ & \mathsf{del} : \oplus\{\ \mathsf{none} : {\downarrow}queue_A, \\ & \qquad\qquad \mathsf{some} : A \otimes {\downarrow}queue_A\ \}\ \} \end{aligned}$$

A Shared Queue

$$
\begin{aligned}
\textcolor{red}{queue}_A = {\uparrow}\&\{\ & \mathsf{ins} : A \multimap {\downarrow}\textcolor{red}{queue}_A, \\
& \mathsf{del} : \oplus\{\ \mathsf{none} : {\downarrow}\textcolor{red}{queue}_A, \\
& \qquad\qquad\ \mathsf{some} : A \otimes {\downarrow}\textcolor{red}{queue}_A\ \}\ \}
\end{aligned}
$$

The section ↑…↓ describes a *critical region*

A Shared Queue

$$\begin{aligned} \mathit{queue}_A = {\uparrow}\&\{\ & \mathsf{ins} : A \multimap {\downarrow}\mathit{queue}_A, \\ & \mathsf{del} : \oplus\{\ \mathsf{none} : {\downarrow}\mathit{queue}_A, \\ & \qquad\quad \mathsf{some} : A \otimes {\downarrow}\mathit{queue}_A\ \}\ \} \end{aligned}$$

The section ↑…↓ describes a *critical region*

Types must be *equisynchronizing*
(released at the same type they are acquired to guarantee session fidelity)

A Shared Queue

$$
\begin{aligned}
\color{red}{queue_A} = {\uparrow}\&\{\ &\mathsf{ins} : A \multimap {\downarrow}\color{red}{queue_A}, \\
&\mathsf{del} : \oplus\{\ \mathsf{none} : {\downarrow}\color{red}{queue_A}, \\
&\qquad\qquad \mathsf{some} : A \otimes {\downarrow}\color{red}{queue_A}\ \}\ \}
\end{aligned}
$$

The section ↑…↓ describes a *critical region*

Types must be *equisynchronizing* (released at the same type they are acquired to guarantee session fidelity)

Certain deadlocks can now arise

A Shared Queue

$$
\begin{aligned}
\color{red}{queue_A} = {\uparrow}\&\{\ & \mathsf{ins} : A \multimap {\downarrow}\color{red}{queue_A}, \\
& \mathsf{del} : \oplus\{\ \mathsf{none} : {\downarrow}\color{red}{queue_A}, \\
& \qquad\qquad \mathsf{some} : A \otimes {\downarrow}\color{red}{queue_A}\ \}\ \}
\end{aligned}
$$

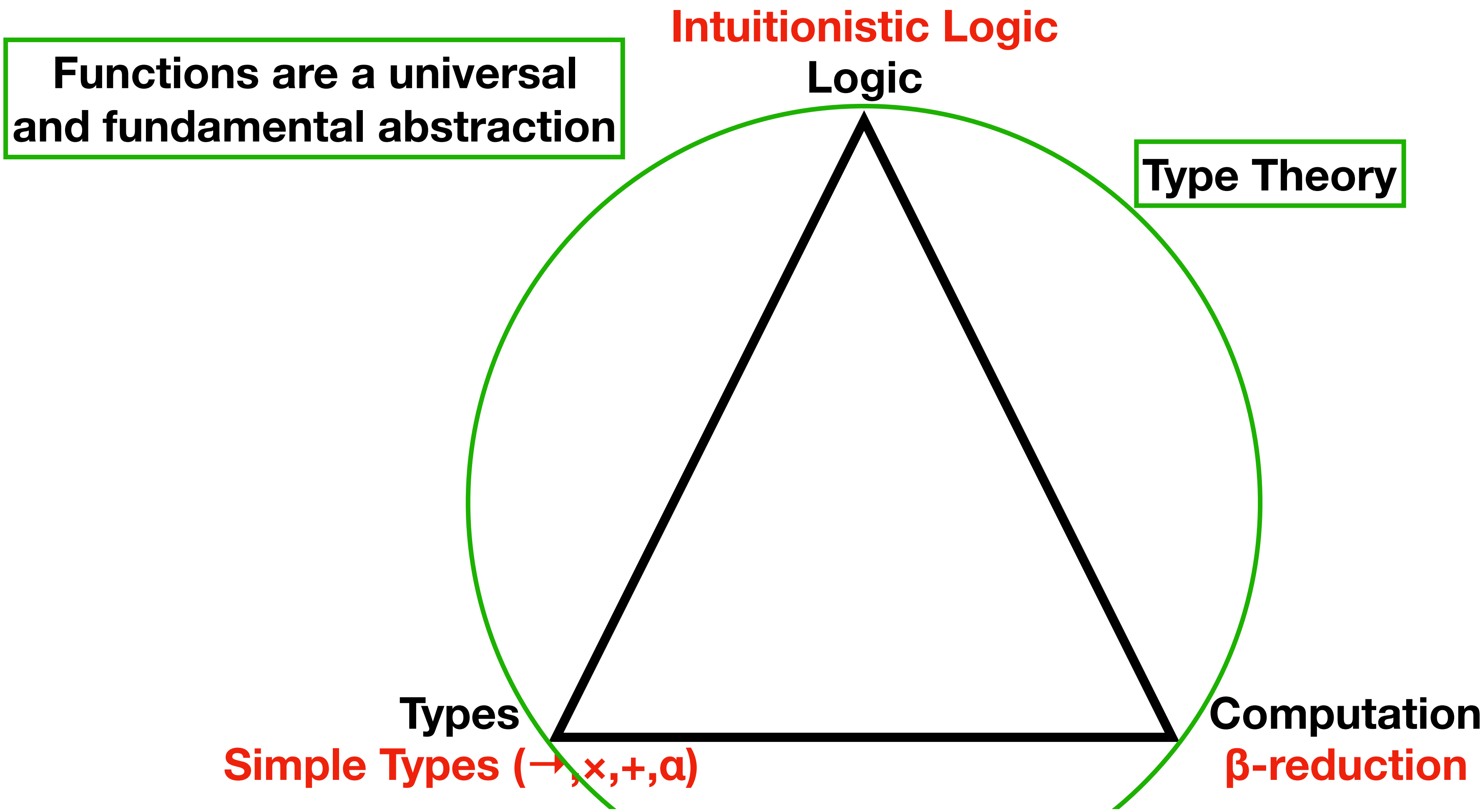
The section ↑…↓ describes a *critical region*

Types must be *equisynchronizing* (released at the same type they are acquired to guarantee session fidelity)

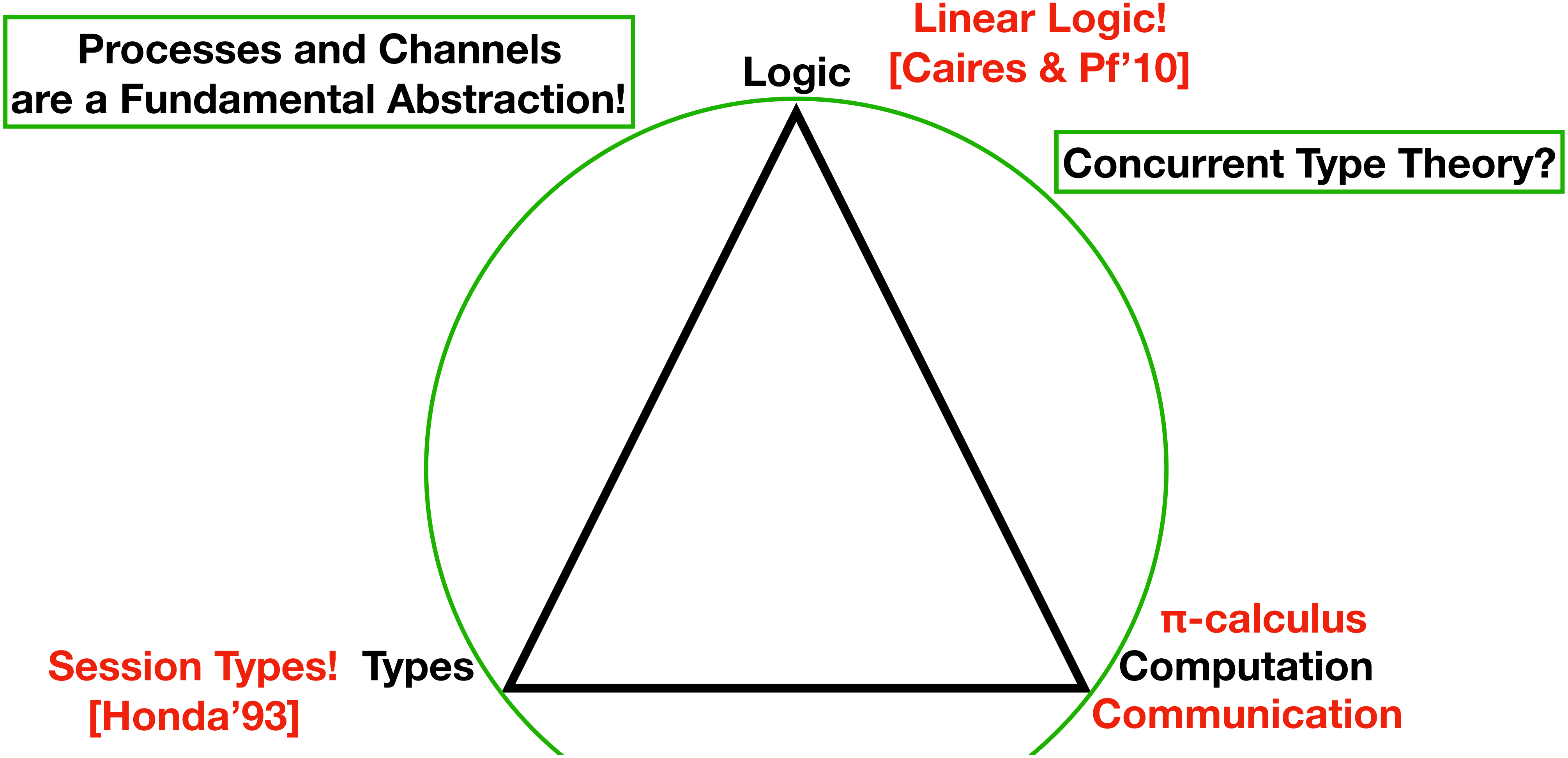
Certain deadlocks can now arise

Sharing and critical regions are manifest in the type!

Why is Functional Programming So Effective?

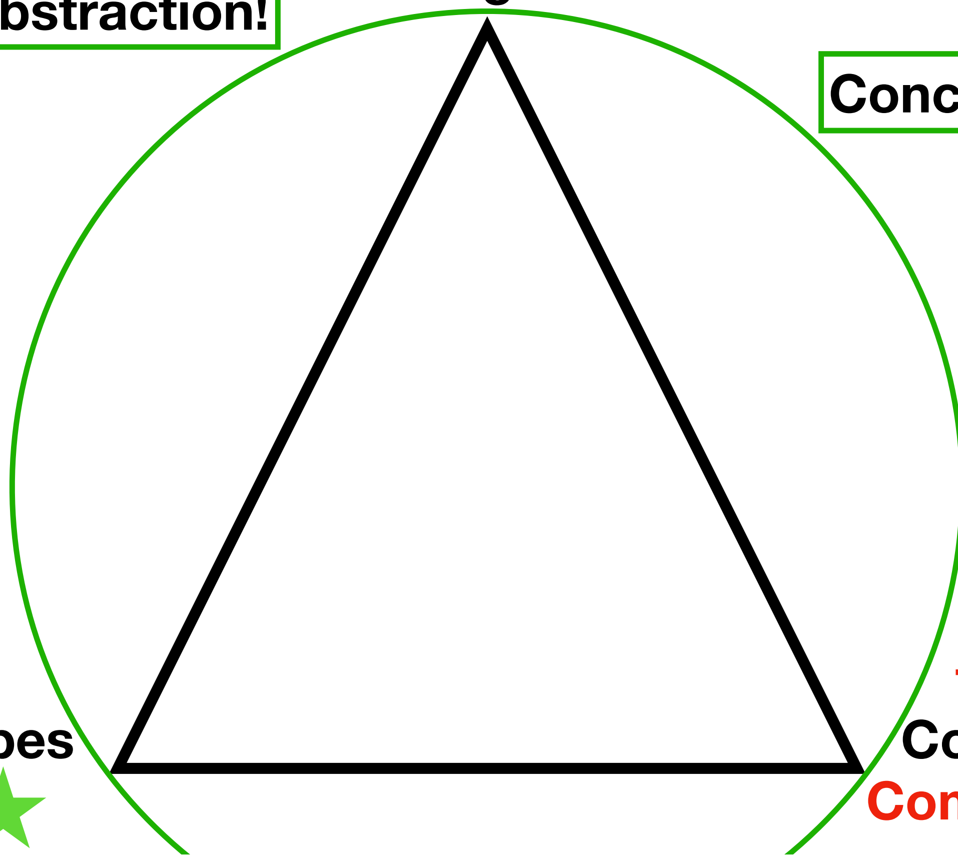


What about Concurrency?

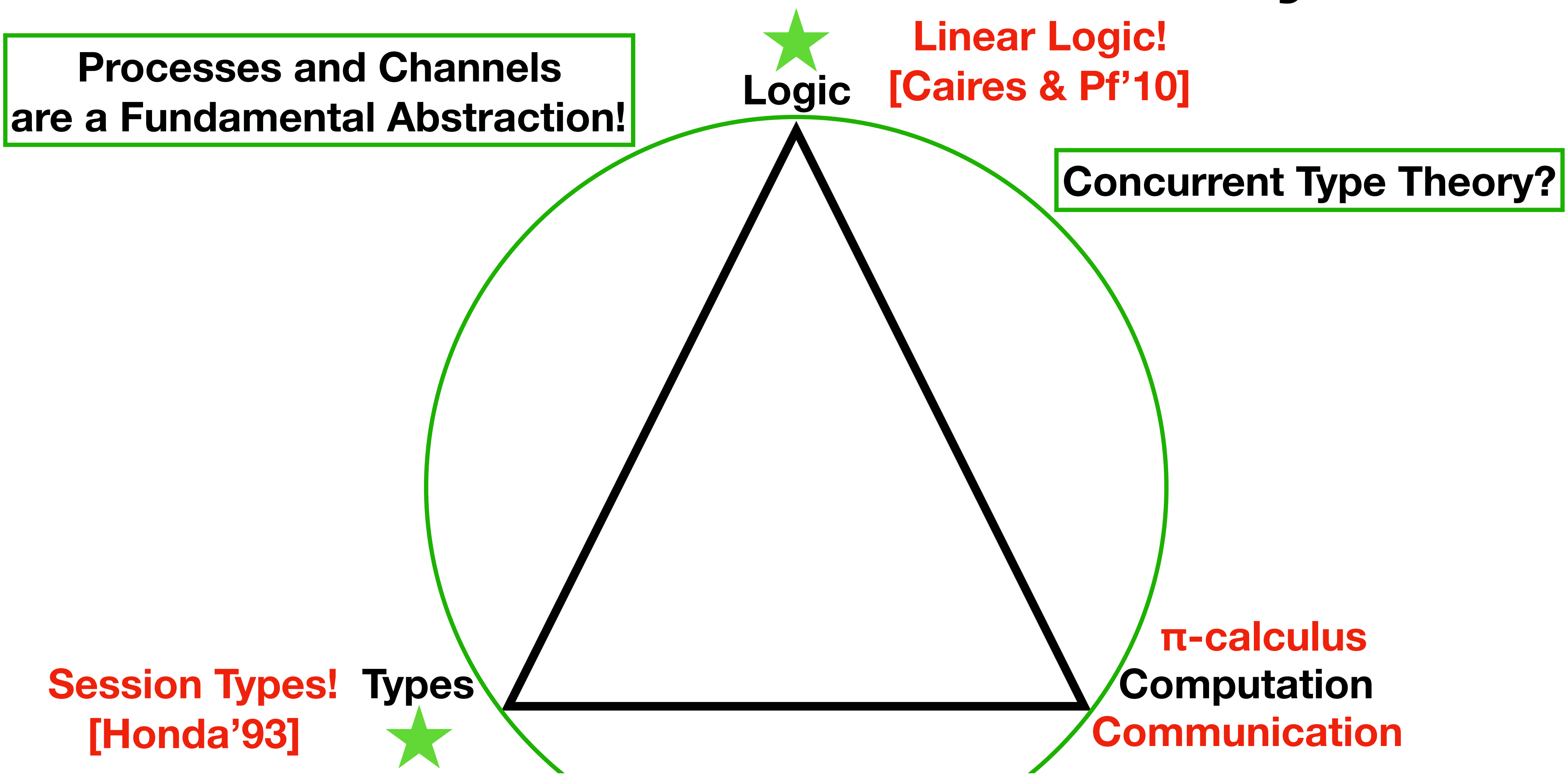


What about Concurrency?

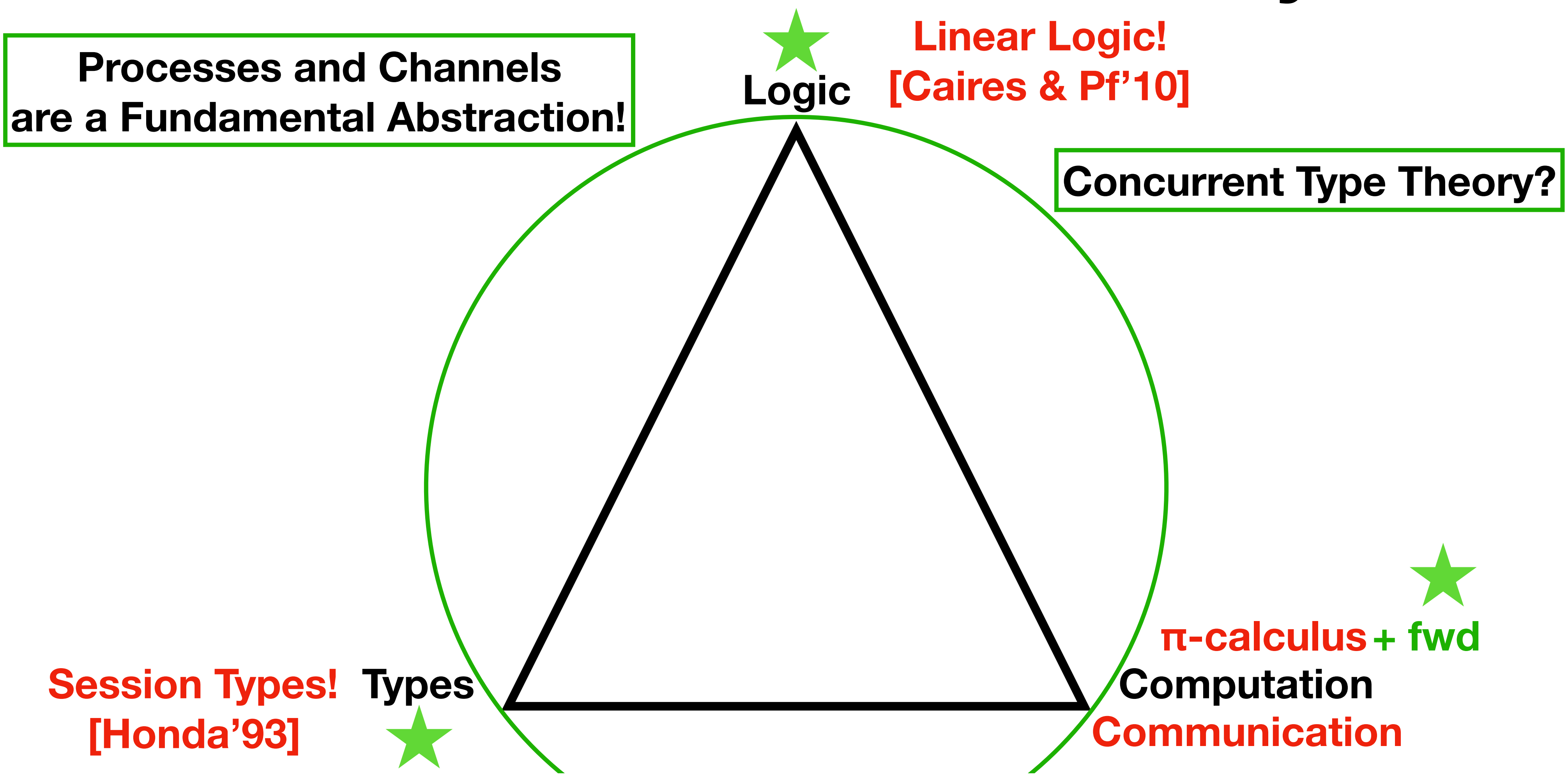
Processes and Channels
are a Fundamental Abstraction!



What about Concurrency?



What about Concurrency?



Session Types at Present

- Scribble — www.scribble.org — multiparty session types
- ABCD project — groups.inf.ed.ac.uk/abcd/ — Simon Gay, Nobuko Yoshida, Philip Wadler
- At CMU — SILL (functional), CC0 (imperative), RSILL (time and work)
- Thanks to my collaborators: Coşku Acay, Stephanie Balzer, Luís Caires, William Chargin, Ankush Das, Henry DeYoung, Anna Gommerstadt, Dennis Griffith, Jan Hoffmann, Limin Jia, Jorge Pérez, Rokhini Prabhu, Klaas Pruiksma, Miguel Silva, Mário Florido, Bernardo Toninho, Max Willsey

A Paper I Love

- *Types for Dyadic Interaction*, Kohei Honda, CONCUR 1993

Kohei Honda, 1959—2012

Types for Dyadic Interaction*

Kohei Honda
kohei@mt.cs.keio.ac.jp

Department of Computer Science, Keio University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223, Japan

Abstract

We formulate a typed formalism for concurrency where types denote freely composable structure of dyadic interaction in the symmetric scheme. The resulting calculus is a typed reconstruction of name passing process calculi. Systems with both the explicit and implicit typing disciplines, where types form a simple hierarchy of types, are presented, which are proved to be in accordance with each other. A typed variant of bisimilarity is formulated and it is shown that typed β-equality has a clean embedding in the bisimilarity. Name reference structure induced by the simple hierarchy of types is studied, which fully characterises the typable terms in the set of untyped terms. It turns out that the name reference structure results in the deadlock-free property for a subset of terms with a certain regular structure, showing behavioural significance of the simple type discipline.