A Rehabilitation of Message-Passing Concurrency

Frank Pfenning
Carnegie Mellon University

PWLCNF 2018, St. Louis
A Paper I Love

Types for Dyadic Interaction*

Kohei Honda

kohei@nt.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 \(\beta\)-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

**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 $\beta$-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
The Activity of Programming
The Activity of Programming

\[
\text{sort}(A); \\
x = A[0];
\]
The Activity of Programming

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

hd(sort(A))
```
The Activity of Programming

- Every programmer, all the time, reasons

```plaintext
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

\[
\text{sort}(A); \\
x = A[0]; \\
\text{hd}(\text{sort}(A))
\]
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

```plaintext
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

- Logic
- Types
- Computation
Why is Functional Programming So Effective?

Functions are a universal and fundamental abstraction.

Intuitionistic Logic

Types
Simple Types ($\to, \times, +, \alpha$)

Computation
$\beta$-reduction
Why is Functional Programming So Effective?

Functions are a universal and fundamental abstraction

Intuitionistic Logic

Logic

Type Theory

Types

Simple Types (¬, ×, +, α)

Computation

β-reduction
What about Concurrency?
What about Concurrency?

Is Shared Memory a Fundamental Abstraction?
What about Concurrency?

Is Shared Memory a Fundamental Abstraction?
What about Concurrency?

Concurrent Separation Logic

Logic

Is Shared Memory a Fundamental Abstraction?

Types

Computation

Read/Write Shared Memory
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!

Logic

π-calculus

Computation

Communication
What about Concurrency?

Processes and Channels are a Fundamental Abstraction!

Session Types! [Honda’93]  Types  ?  \(\pi\)-calculus
Computation
Communication
What about Concurrency?

Processes and Channels are a Fundamental Abstraction!

Session Types! [Honda’93]  Types  ?

Logic

Linear Logic! [Caires & Pf’10]

π-calculus
Computation
Communication
What about Concurrency?

Processes and Channels are a Fundamental Abstraction!

Logic

Linear Logic!
[Caires & Pf’10]

Concurrent Type Theory?

Session Types!
[Honda’93]

Types

π-calculus

Computation

Communication
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 τ) 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 $\tau$) 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

- Previous talk!

- From the language point of view: Go
  
  - Goroutines (threads/processes) as a fundamental abstraction
  
  - Channels (\texttt{chan} \ \tau) as a fundamental abstraction

- Connection to logic is missing

- Types are not expressive enough

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

—Effective Go
Example: A Store (Stack or Queue)
Example: A Store (Stack or Queue)

$\mathcal{C} : A$

Provider

Typed dyadic channel

Client
Example: A Store (Stack or Queue)

- Protocol

![Diagram showing typed dyadic channel between Provider and Client with $c : A$]
Example: A Store (Stack or Queue)

- Protocol
  - Client: ins; x; recurse…

$\text{Client} \xrightarrow{\text{Typed dyadic channel}} \text{Provider}$
Example: A Store (Stack or Queue)

- Protocol
  - Client: ins; x; `recurse`…
  - Client: del
Example: A Store (Stack or Queue)

- Protocol
  - Client: ins; x; \textit{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
    - Provider: none; close.
    - Provider: some; x; *recurse*…
  - Protocol should be expressed by a type!
Example: A Store (Stack or Queue)

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

Client's choice (external)

Provider's choice (internal)
A Simple Client, in CC0
int main() {

}
int main() {
    int n = 10;
A Simple Client, in CC0

int main() {
    int n = 10;
    stack $s = empty();
}
A Simple Client, in CC0

```c
int main() {
    int n = 10;
    stack $s = empty();
    for (int i = 0; i < n; i++) {
```
A Simple Client, in CC0

```c
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

```c
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

```c
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;
}
```
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

```c
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;
}
```
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;
}
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

```c
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

```c
stack $s$ elem(int $x$, stack $t$) {
  switch ($s$) {
    case ins: {
```
A Simple Provider, in CC0

```c
stack $s elem(int x, stack $t) {
    switch ($s) {
        case ins: {
            int y = recv($s);
        }
        case ins: {
            int y = recv($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, $r$);
    }
    }
}
stack $s$ elem(int $x$, stack $t$) {
    switch ($s$) {
        case ins: {
            int $y = \mathrm{recv}(s);$
            stack $r = \mathrm{elem}(x, t);$
            $s = \mathrm{elem}(y, r);$
        }
        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 $y = \text{recv}(s)$;
            stack $r = \text{elem}(x, t)$;
            $s = \text{elem}(y, r)$;
        }
        case del: {
            $s$.some;
            send($s$, $x$);
            $s = t$;
        }
    }
}
A Simple Provider, in CC0

```c
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: {
            $s$.somme;
            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

```c
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 $y = \text{recv}($s$);
            stack $r = \text{elem}(x, t);
            $s = \text{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 $y = \text{recv}(s)$;
            stack $r = \text{elem}(x, t)$;
            $s = \text{elem}(y, r)$;
        }
        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 \( \pi \)-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

<table>
<thead>
<tr>
<th>Type</th>
<th>Provider action</th>
<th>Continuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A ::= ) &amp;{(\ell : A_\ell)}_{\ell \in L}</td>
<td>receive some (k \in L)</td>
<td>(A_k)</td>
</tr>
<tr>
<td></td>
<td>(\oplus{(\ell : A_\ell)}_{\ell \in L})</td>
<td>send some (k \in L)</td>
</tr>
<tr>
<td></td>
<td>(A \to B)</td>
<td>receive channel (c : A)</td>
</tr>
<tr>
<td></td>
<td>(A \otimes B)</td>
<td>send channel (c : A)</td>
</tr>
<tr>
<td></td>
<td>(1)</td>
<td>terminate</td>
</tr>
<tr>
<td></td>
<td>(\forall x:\tau. A)</td>
<td>receive (v : \tau)</td>
</tr>
<tr>
<td></td>
<td>(\exists x:\tau. A)</td>
<td>send (v : \tau)</td>
</tr>
</tbody>
</table>
### Session Types, Abstractly

<table>
<thead>
<tr>
<th>Type</th>
<th>Provider action</th>
<th>Continuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A ::= &amp; { \ell : A_{\ell} }_{\ell \in L}$</td>
<td>receive some $k \in L$</td>
<td>$A_k$</td>
</tr>
<tr>
<td></td>
<td>$\oplus { \ell : A_{\ell} }_{\ell \in L}$</td>
<td>send some $k \in L$</td>
</tr>
<tr>
<td>$A \rightarrow B$</td>
<td>receive channel $c : A$</td>
<td>$B$</td>
</tr>
<tr>
<td>$A \otimes B$</td>
<td>send channel $c : A$</td>
<td>$B$</td>
</tr>
<tr>
<td>$1$</td>
<td>terminate</td>
<td>none</td>
</tr>
<tr>
<td>$\forall x: \tau. A$</td>
<td>receive $v : \tau$</td>
<td>$[v/x]A$</td>
</tr>
<tr>
<td>$\exists x: \tau. A$</td>
<td>send $v : \tau$</td>
<td>$[v/x]A$</td>
</tr>
</tbody>
</table>

$$stack_A = \& \{ \text{ins : } A \rightarrow stack_A,\
\text{del : } \oplus \{ \text{none : } 1,\
\text{some : } A \otimes stack_A \} \}$$
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

\[
\Gamma \vdash x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A)
\]

channel provided by P
Session Typing Judgments

channels used by P

\[
\begin{align*}
\underbrace{x_1 : A_1, \ldots, x_n : A_n}_{\Gamma} & \vdash P :: (x : A) \\
\text{channel provided by P}
\end{align*}
\]

Configuration

\[
\Omega ::= (P_1 | \cdots | P_n)
\]
Session Typing Judgments

channels used by $P$

\[
x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A)
\]

\[\Gamma\]

channel provided by $P$

Configuration

\[\Omega ::= (P_1 \mid \cdots \mid P_n)\]

Configuration Typing

\[\Gamma \models \Omega :: \Gamma'\]

channels used by $\Omega$ channels provided by $\Omega$
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) \[\Gamma \models \Omega :: \Gamma' \text{ and } \Omega \rightarrow \Omega' \text{ then } \Gamma \models \Omega :: \Omega'\]
  - Deadlock freedom
  - Termination
- With recursion
  - Session fidelity (Preservation)
  - Deadlock freedom
Theoretical Properties

• Without recursive types and processes

  • Session fidelity (Preservation) \[ \text{If } \Gamma \models \Omega :: \Gamma' \text{ and } \Omega \rightarrow \Omega' \text{ then } \Gamma \models \Omega :: \Omega' \]

  • Deadlock freedom \[ \text{If } \Gamma \models \Omega :: \Gamma' \text{ then } \Omega \text{ poised or } \Omega \rightarrow \Omega' \text{ for some } \Omega' \]

  • Termination

• With recursion

  • Session fidelity (Preservation)

  • Deadlock freedom
Theoretical Properties

- Without recursive types and processes
  - Session fidelity (Preservation) \[\Gamma \vdash \Omega :: \Gamma' \text{ and } \Omega \rightarrow \Omega' \text{ then } \Gamma \vdash \Omega :: \Omega'\]
  - Deadlock freedom \[\Gamma \vdash \Omega :: \Gamma' \text{ then } \Omega \text{ poised or } \Omega \rightarrow \Omega' \text{ for some } \Omega'\]
  - Termination \[\text{If } \Gamma \vdash \Omega :: \Gamma' \text{ then } \Omega \rightarrow^* \Omega' \text{ for } \Omega' \text{ poised}\]
- With recursion
  - Session fidelity (Preservation)
  - Deadlock freedom
Theoretical Properties

• Without recursive types and processes
  • Session fidelity (Preservation) \[ \Gamma \vdash \Omega :: \Gamma' \] and \( \Omega \rightarrow \Omega' \) then \( \Gamma \vdash \Omega :: \Omega' \)
  • Deadlock freedom \[ \text{If } \Gamma \vdash \Omega :: \Gamma' \text{ then } \Omega \text{ poised or } \Omega \rightarrow \Omega' \text{ for some } \Omega' \]
  • Termination \[ \text{If } \Gamma \vdash \Omega :: \Gamma' \text{ then } \Omega \rightarrow^* \Omega' \text{ for } \Omega' \text{ poised} \]

• With recursion
  • Session fidelity (Preservation)
  • Deadlock freedom

\( \Omega \text{ is poised if all processes in } \Omega \text{ attempt to communicate along a channel in the external interface} \)
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;
Tracing the Type-Checker
Tracing the Type-Checker

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

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

Tracing the Type-Checker
Tracing the Type-Checker

stack $s$ elem(int $x$, stack $t$) {
    switch ($s$) {
        case ins: {
            % $s : <?int ; stack> -l $t : stack
        }
    }
}
stack $s$ elem(int $x$, stack $t$) {
    switch ($s$) {
    case ins: {
        int $y$ = recv($s$); 
        % $s$ : stack  -l $t$ : stack
        % $s$ : stack -l $t$ : stack
    }
    }
Tracing the Type-Checker

stack $s$ elem(int $x$, stack $t$) {
    switch ($s$) {
        case ins: {
            int $y = \text{recv}($s$);$
            stack $r = \text{elem}(x, t);$} % $s : <?int ; stack> -| $t : stack
    } % $s : stack -| $r : stack
}
Tracing the Type-Checker

stack $s$ elem(int $x$, stack $t$) {
    switch ($s$) {
        case ins: {
            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: {
      int y = recv($s$);
      stack $r = elem(x, $t)$;
      $s = elem(y, $r)$;
    }
  }
}
Tracing the Type-Checker

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: { ...}
    }
}
Tracing the Type-Checker

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;
        }
    }
}

% $s : <?int ; stack> -| $t : stack
% $s : stack -| $t : stack
% $s : stack -| $r : stack
% $s : <!int ; stack> -| $t : stack
Tracing the Type-Checker

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

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

Tracing the Type-Checker
```
stack $s$ elem(int x, stack $t$) {
    switch ($s$) {
    case ins: {
        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> -| ...
        $s.some;
        send($s, x);
        $s = $t;
    }
    }
}
Tracing the Type-Checker

stack $s$ elem(int x, stack $t$) {
  switch ($s$) {
  case ins: {
    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;
    send($s$, x);
    $s = $t;
  }
  }
}
Tracing the Type-Checker

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

stack $s$ empty() {
    switch ($s$) {
        case ins: {
            int $x$ = recv($s$); % $s$ : stack -1.
        }
    }
}

```
Tracing the Type-Checker

stack $s$ empty() {
    switch ($s$) {
        case ins: {
            int $x$ = recv($s$);
            stack $e$ = empty();
        }
    }
}
Tracing the Type-Checker

stack $s$ empty() {
    switch ($s$) {
        case ins: {
            int x = recv($s$); % $s : stack -l .
            stack $e = empty(); % $s : stack $e : stack -l
            $s = elem(x, $e);
        }
    }
}
Tracing the Type-Checker

stack $s$ empty() {
    switch ($s$) {
        case ins: {
            int $x$ = recv($s$);
            stack $e$ = empty();
            $s$ = elem($x$, $e$);
        }
    }
}
Tracing the Type-Checker

```c
stack $s$ empty() {
    switch ($s$) {
        case ins: {
            int x = recv($s$);
            stack $e$ = empty();
            $s$ = elem(x, $e$);
        }
        case del: {
        }
    }
}
```
Tracing the Type-Checker

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($s$);
    }
  }
}
Tracing the Type-Checker

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($s$); % $s : <?int ; stack> -l .
            stack $e = empty(); % $s : stack -l $e : stack
            $s = elem(x, $e);
        }
        case del: {
            $s$.none; % $s : <!stack_response> -l .
            close($s);
        }
    }
}
Tracing the Type-Checker

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

Client 1

Typed shared channel

#c : A

Client 2
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

<table>
<thead>
<tr>
<th>Type</th>
<th>Provider action</th>
<th>Continuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A := &amp;{\ell : A_\ell}_{\ell \in L})</td>
<td>receive some (k \in L)</td>
<td>(A_k)</td>
</tr>
<tr>
<td>(\oplus{\ell : A_\ell}_{\ell \in L})</td>
<td>send some (k \in L)</td>
<td>(A_k)</td>
</tr>
<tr>
<td>(A \rightarrow B)</td>
<td>receive channel (c : A)</td>
<td>(B)</td>
</tr>
<tr>
<td>(A \otimes B)</td>
<td>send channel (c : A)</td>
<td>(B)</td>
</tr>
<tr>
<td>(\downarrow S)</td>
<td>detach from client</td>
<td>(S)</td>
</tr>
<tr>
<td>(\uparrow S)</td>
<td>accept client</td>
<td>(A)</td>
</tr>
</tbody>
</table>
## Linear and Shared Channels

<table>
<thead>
<tr>
<th>Type</th>
<th>Provider action</th>
<th>Continuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A ::= &amp; { \ell : A_\ell }_{\ell \in L}$</td>
<td>receive some $k \in L$</td>
<td>$A_k$</td>
</tr>
<tr>
<td></td>
<td>$\oplus { \ell : A_\ell }_{\ell \in L}$</td>
<td>send some $k \in L$</td>
</tr>
<tr>
<td></td>
<td>$A \to B$</td>
<td>receive channel $c : A$</td>
</tr>
<tr>
<td></td>
<td>$A \otimes B$</td>
<td>send channel $c : A$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
<td>terminate</td>
</tr>
<tr>
<td></td>
<td>$\forall x : \tau. A$</td>
<td>receive $v : \tau$</td>
</tr>
<tr>
<td></td>
<td>$\exists x : \tau. A$</td>
<td>send $v : \tau$</td>
</tr>
<tr>
<td></td>
<td>$\downarrow S$</td>
<td>detach from client</td>
</tr>
<tr>
<td>$S ::= \uparrow A$</td>
<td>accept client</td>
<td>$A$</td>
</tr>
</tbody>
</table>

$!A \triangleq \downarrow \uparrow A$
A Shared Queue

\[ queue_A = \uparrow \& \{ \text{ins} : A \rightarrow \downarrow queue_A, \]
\[ \text{del} : \oplus \{ \text{none} : \downarrow queue_A, \]
\[ \text{some} : A \otimes \downarrow queue_A \} \} \]
A Shared Queue

\[ \text{queue}_A = \uparrow \& \{ \text{ins} : A \rightarrow \downarrow \text{queue}_A, \]
\[ \quad \text{del} : \oplus \{ \text{none} : \downarrow \text{queue}_A, \]
\[ \quad \quad \text{some} : A \otimes \downarrow \text{queue}_A \} \} \]

The section \( \uparrow \ldots \downarrow \) describes a critical region
A Shared Queue

\[ \text{queue}_A = \uparrow \& \{ \text{ins} : A \rightarrow \downarrow \text{queue}_A, \]
\[ \text{del} : \oplus \{ \text{none} : \downarrow \text{queue}_A, \]
\[ \text{some} : A \otimes \downarrow \text{queue}_A \} \} \]

The section \(\uparrow \ldots \downarrow\) describes a critical region

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

\[ queue_A = \uparrow\&\{ \text{ins} : A \rightarrow \downarrow queue_A, \]
\[ \text{del} : \oplus\{ \text{none} : \downarrow queue_A, \]
\[ \text{some} : A \otimes \downarrow queue_A \} \} \]

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

\[ \text{queue}_A = \uparrow \& \{ \text{ins} : A \rightarrow \downarrow \text{queue}_A, \]
\[ \text{del} : \oplus \{ \text{none} : \downarrow \text{queue}_A, \]
\[ \text{some} : A \otimes \downarrow \text{queue}_A \} \}\]

The section \(\uparrow \ldots \downarrow\) 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?

Functions are a universal and fundamental abstraction.

Intuitionistic Logic

Types
Simple Types ($\neg, \times, +, \alpha$)

Type Theory

Computation
$\beta$-reduction
What about Concurrency?

Processes and Channels are a Fundamental Abstraction!

Logic

Linear Logic!
[Caires & Pf’10]

Concurrent Type Theory?

Session Types!
[Honda’93]

Types

π-calculus

Computation

Communication
What about Concurrency?

Processes and Channels are a Fundamental Abstraction!

Logic

Linear Logic!
[Caires & Pf’10]

Concurrent Type Theory?

Session Types!
[Honda’93]

π-calculus

Computation

Communication

Types
What about Concurrency?

Processes and Channels are a Fundamental Abstraction!

Logic

Linear Logic!
[Caires & Pf’10]

Concurrent Type Theory?

Session Types!
[Honda’93]

π-calculus
Computation
Communication

Types
What about Concurrency?

Processes and Channels are a Fundamental Abstraction!

Logic

Linear Logic!
[Caires & Pf’10]

Concurrent Type Theory?

Session Types!
[Honda’93]

Types

π-calculus + fwd

Computation

Communication
Session Types at Present

• Scribble — [www.scribble.org](http://www.scribble.org) — multiparty session types

• ABCD project — [groups.inf.ed.ac.uk/abcd/](http://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

---

Types for Dyadic Interaction*

Kohei Honda
kohei@cs.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 $\beta$-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.