Who

- **Fabien Dagnat**, Engineer and Phd in CS
- Associate Professor at ENST de Bretagne, Brest, France
- Researches in:
  - Models, Languages, Semantics, Verifications
  - To support Mobility, Reconfigurability, Coordination / Composition
  - In Object, Concurrent and Component OL
Static Analysis of Communications for Erlang

work made at IRIT in Toulouse
mainly with Marc Pantel
Why

- Concurrent Programming is **hard** but **essential**
- We need Concurrent Oriented Languages
  ⇒ God gave us Erlang
Why

• Concurrent Programming is **hard** but **essential**
• We need Concurrent Oriented Languages
  ⇒ God gave us Erlang
• Programming cannot remains a **craft (an art?)**
• We need to verify (automatically) programs
  ⇒ God gave us Typing
Why

- Concurrent Programming is **hard** but **essential**
- We need Concurrent Oriented Languages
  ⇒ God gave us Erlang
- Programming cannot remains a **craft** (an art?)
- We need to verify (automatically) programs
  ⇒ God gave us Typing

**Let’s build static analyzes for (a Typed) Erlang**
How

• Why typing concurrency?
• TOOL analogy
• Extensions needed (effects, received messages and subtyping)
• The real system
• Futures
fac(N) when N>0 -> N*fac(N-1);
fac(0) -> 1.

> fac(foo).
> fac(18,"hello").

⇒ Works from Wadler et al.
fac(N) when N>0 -> N*fac(N-1);
fac(0) -> 1.
> fac(foo).
> fac(18,"hello").
⇒ Works from Wadler et al.

Processes (and messages) are central
BUT
not typed
fac(N) when N>0 -> N*fac(N-1);
fac(0)       -> 1.
> fac(foo).
> fac(18,"hello").
⇒ Works from Wadler et al.

Processes (and messages) are central
BUT
not typed

ping() -> receive ping -> ping() end.
> (spawn(ping,[[]]))!pong
fac(N) when N>0 -> N*fac(N-1);
fac(0)          -> 1.
> fac(foo).
> fac(18,"hello").
⇒ Works from Wadler et al.

**Processes (and messages) are central**
**BUT not typed**

ping() -> receive ping -> ping() end.
> (spawn(ping,[[]]))!pong

⇒ pong is a **message not understood**
Method Not Understood

- Usual Typed OOL *method not understood*
• Usual Typed OOL method not understood

• class C: typeof(C) set of methods C defines or inherits

```plaintext
class C extends C' {
   .. m1(..){}
   .. m2(..){}
}
```

typeof(C) = \{m1, m2\} ∪ typeof(C')
• Usual Typed OOL **method not understood**

• **class C**: `typeof(C)` set of methods `C` defines or inherits

```java
class C extends C' { .. m1(..){} }
  .. m2(..){} }
```

`typeof(C) = \{m1, m2\} \cup typeof(C')`

• **object o=new C()**: `typeof(o)=typeof(C)`
• Usual Typed OOL **method not understood**

• **class C**: `typeof(C)` set of methods `C` defines or inherits

  ```java
class C extends C' {
    .. m1(..){}
    .. m2(..){}
  }
```

  `typeof(C) = \{m1,m2\} \cup typeof(C')`

• **object o=new C(): typeof(o)=typeof(C)**

• `o.m` is correct **iff** `m \in typeof(o)`
We could make some analogy:

<table>
<thead>
<tr>
<th>Java</th>
<th>Erlang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Process</td>
</tr>
<tr>
<td>new</td>
<td>spawn</td>
</tr>
<tr>
<td><em>·</em></td>
<td><em>!</em></td>
</tr>
<tr>
<td>Class</td>
<td>Function</td>
</tr>
<tr>
<td>Inheritance</td>
<td>Function Call</td>
</tr>
<tr>
<td>Method definition</td>
<td>receive statement</td>
</tr>
</tbody>
</table>
We could make some analogy:

<table>
<thead>
<tr>
<th>Java</th>
<th>Erlang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Process</td>
</tr>
<tr>
<td>new</td>
<td>spawn</td>
</tr>
<tr>
<td><em>.</em></td>
<td><em>!</em></td>
</tr>
<tr>
<td>Class</td>
<td>Function</td>
</tr>
<tr>
<td>Inheritance</td>
<td>Function Call</td>
</tr>
<tr>
<td>Method definition</td>
<td>receive statement</td>
</tr>
</tbody>
</table>

**BUT what is the function type?**

*domain → codomain* is not enough
Let us collect all receive blocks (flow analysis)

\[
\text{state1}(V) \rightarrow \\
\text{receive} \\
\quad \{\text{add}, V1\} \rightarrow \text{state1}(V1+V); \\
\quad \{\text{change}, V1\} \rightarrow \text{state2}(V,V1) \\
\text{end.}
\]

\[
\text{state2}(V1,V2) \rightarrow \\
\text{receive} \\
\quad \{\text{add}, V3,V4\} \rightarrow \text{state2}(V1+V3,V2+V4) \\
\text{end.}
\]

\[
\text{state1}: \text{num} \xrightarrow{\{\text{add: num, change: num, add: num } \times \text{num}\}} \bot
\]

\[
\text{state2}: \text{num } \times \text{num} \xrightarrow{\{\text{add: num } \times \text{num}\}} \bot
\]
state2(V1,V2) ->
  receive
    {add,V3,V4} -> state2(V1+V3,V2+V4);
    {mute,F}   -> F()
  end.

What is the effect of F()?
state2(V1,V2) ->
  receive
  {add,V3,V4} -> state2(V1+V3,V2+V4);
  {mute,F}   -> F()
  end.

What is the effect of \texttt{F()}?

Keep all messages received by a process.
state2(V1,V2) ->
  receive
    {add,V3,V4} -> state2(V1+V3,V2+V4);
    {mute,F}   -> F()
  end.

What is the effect of \( F() \)?

Keep all messages received by a process.

⇒ Process Type:

\[
P : \@\{\ldots, m : (\underbrace{\text{received}, \understood}_{P!\{m,\_\}}), \ldots\}\}
\]
correction iff \( \text{received} \sqsubseteq \text{understood} \)
correction \textbf{iff} received \sqsubseteq \textit{understood}

\textbf{Process Type:}

@\{m_i : (\alpha^i_1, \alpha^i_2)\}_{I_1} \sqsubseteq @\{m_i : (\beta^i_1, \beta^i_2)\}_{I_2}

\textbf{iff}

I_2 \subseteq I_1 \textbf{ and } \forall i \in I_2 \beta^i_1 \sqsubseteq \alpha^i_1 \alpha^i_2 \sqsubseteq \beta^i_2
correction $\textit{iff}$ received $\sqsubseteq$ understood

Process Type:

$$\forall \{m_i : (\alpha^i_1, \alpha^i_2)\} \in I_1 \sqsubseteq \forall \{m_i : (\beta^i_1, \beta^i_2)\} \in I_2$$

$\textit{iff}$

$$I_2 \subseteq I_1 \text{ and } \forall i \in I_2 \ \beta^i_1 \sqsubseteq \alpha^i_1 \ \alpha^i_2 \sqsubseteq \beta^i_2$$

Function Type:

$$\alpha_1 \xrightarrow{I_1} \beta_1 \sqsubseteq \alpha_2 \xrightarrow{I_2} \beta_2$$

$\textit{iff}$

$$\alpha_2 \sqsubseteq \alpha_1 \text{ and } \beta_1 \sqsubseteq \beta_2 \text{ and } I_1 \sqsubseteq I_2$$
state3() -> receive kill -> true end.
> P=(spawn(state1,[1]))![mute,state3].
state3() -> receive kill -> true end.

> P=(spawn(state1,[1]))!{mute,state3}.

\[ P : T_P = @\{...,mute:(state3,type(F)),...\}\]
state3() -> receive kill -> true end.
> P=(spawn(state1,[1]))!{mute,state3}.

$P : T_P = \{...,\text{mute} : (\text{state3}, \text{type}(F)), \ldots\}$

state3 $\sqsubset$ type (F)
state3() -> receive kill -> true end.
> P = (spawn(state1, [1]))!{mute, state3}.

\[ P : T_P = @\{..., mute : (state3, type(F)), ...\} \]

\[ \text{state3} \sqsubseteq \text{type}(F) \]

\[ \text{unit} \xrightarrow{\{\text{kill}:(\perp, \text{unit})\}} \text{true} \sqsubseteq \text{unit} \xrightarrow{I} t \]
state3() -> receive kill -> true end.

> P = (spawn(state1, [1]))!{mute, state3}.

$P : T_P = \{\ldots, \text{mute} : (\text{state3}, \text{type}(F)), \ldots\}$

state3 ⊑ type(F)

\[
\begin{align*}
\text{unit} & \xrightarrow{\{\text{kill} : (\bot, \text{unit})\}} \text{true} \sqsubseteq \text{unit} \quad I \rightarrow t \\
T_p & \sqsubseteq @I \sqsubseteq @\{\text{kill} : (\bot, \text{unit})\} \quad \text{and} \quad \text{true} \sqsubseteq t
\end{align*}
\]
state3() -> receive kill -> true end.
> P=(spawn(state1,[1]))!{mute,state3}.
P : \[ T_P = @\{...,mute:(state3,type(F)),...\} \]

state3 ⊑ type(F)

unit \{kill:(⊥,unit)\} \rightarrow true ⊑ unit I \rightarrow t

\[ T_p ⊑ @I ⊑ @\{kill:(⊥,unit)\} \text{ and true ⊑ t} \]

\[ \Rightarrow P!kill \text{ is correct and } P!\text{sub}(1) \text{ is not} \]
Scaling to Erlang

- Complete type language (Lot of rules!)
- Better precision (*conditional constraints*)
- Dynamic patterns (*name polymorphism*)
- Constraints:

  Program → Analyser → types + constraints → Solver → safety error(s)

<table>
<thead>
<tr>
<th>types + solved constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Printer</td>
</tr>
<tr>
<td>&lt;&lt; readable &gt;&gt; types</td>
</tr>
</tbody>
</table>

- A pragmatic simple prototype written in CaML
Limitations & Results

- “Global” analysis: need (a large part of) the application
- Lots of calculus: “slow” analysis
- Difficult to give the precise source of an error
- (Simplified) Types are sometimes hard to read (tree structure)

```
@{m_1 : (@{m_2 : (@{m_3 : (\ldots) }, \ldots) }, \ldots) \ldots}
```
Limitations & Results

• “Global” analysis: need (a large part of) the application
• Lots of calculus: “slow” analysis
• Difficult to give the precise source of an error
• (Simplified) Types are sometimes hard to read (tree structure)

\[@\{m_1 : (\{m_2 : (\{m_3 : (\ldots)\}, \ldots)\}, \ldots) \ldots\}\]

• A first core for a formal semantics
• A precise type inference system proved
• A toy implementation
• Write a formal semantics of (a larger part of) Erlang
• Produce a less toy prototype (Modules, Records, Exceptions)
• Integration with specweb? Add messages to the dictionary? Have two type checker?
• Take a look at Hot code swapping
Questions?
Insight on Semantics

- Abstract the functional part: configuration[X]

- $\pi$-calculus like term:
  - $\nu a. (\langle a \mid mb \rangle \triangleright exp \parallel a \triangleleft mess)$
  - reception leads to: $\nu a. (\langle a \mid mess \ mb \rangle \triangleright exp)$
  - functional calculus leads to:
    $\nu a. (\nu a'. (\langle a \mid mb' \rangle \triangleright exp' \parallel w))$ if
    $a' \vdash \langle a \mid mess\ mb \rangle, \exp \xrightarrow{w} e \langle a \mid mb' \rangle, \exp'$

- X = Erlang:
  - send: $a' \vdash \alpha, v_1 ! v_2 \xrightarrow{v_1 < v_2} e\ \alpha, v_2$
  - spawn:
    $a \vdash \alpha, spawn (f, [v_1, \ldots, v_n]) \xrightarrow{\langle a|\emptyset\rangle \triangleright f(v_1,\ldots,v_n)} e\ \alpha, a$
Another Example

timer({Pid, Time, Alarm}) ->
  receive {cancel, Pid} -> true
  after Time -> Pid ! Alarm
  end.
timeout({Time, Alarm}) ->
  spawn(timer, {self(), Time, Alarm}).
cancel(Timer) ->
  Timer ! {cancel, self()}.
$f(p_1^1, \ldots, p_n^1) \rightarrow e_1; \ldots \; f(p_1^k, \ldots, p_n^k) \rightarrow e_k.$

\[
\begin{cases}
\text{type}(f) = \bigsqcup_i (\text{type}(p_1^i) \times \cdots \times \text{type}(p_n^i) \rightarrow \text{type}(\mathcal{E}, e_i)) \\
\text{effect}(f) = \bigsqcup_i \text{effect}(\mathcal{E}, e_i)
\end{cases}
\]

\[
\begin{cases}
\text{type}(\mathcal{E}, f(e_1, \ldots, e_n)) = t \\
\text{if type}(f) = \text{type}(\mathcal{E}, e_1) \times \cdots \times \text{type}(\mathcal{E}, e_n) \rightarrow t \\
\text{effect}(\mathcal{E}, f(e_1, \ldots, e_n)) = \text{effect}(f) \cup \bigsqcup_i \text{effect}(\mathcal{E}, e_i)
\end{cases}
\]