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
  $\Rightarrow$God gave us Erlang

Why

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

Why

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

Typing ?

```
fac(N) when N>0 -> N*fac(N-1);
fac(0)          -> 1.
> fac(foo).
> fac(18,"hello").
```

⇒Works from Wadler et al.

Typing ?

```
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

Typing ?

```
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
```

Typing ?

```
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**

Method Not Understood

- Usual Typed OOL **method not understood**
- class `C`: `typeof(C)` set of methods `C` defines or inherits

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

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

Method Not Understood

- Usual Typed OOL **method not understood**
- class `C`: `typeof(C)` set of methods `C` defines or inherits

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

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

- object `o=new C()`: `typeof(o)=typeof(C)`

Method Not Understood

- Usual Typed OOL **method not understood**
- class `C`: `typeof(C)` set of methods `C` defines or inherits

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

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

- object `o=new C()`: `typeof(o)=typeof(C)`
- `o.m` is correct **iff** $\texttt{m} \in \texttt{typeof(o)}$

Typed OOL Analogy

We could make some analogy:

Java	Erlang
Object	Process
`new`	`spawn`
`_._`	`_!_`
Class	Function
Inheritance	Function Call
Method definition	`receive` statement

Typed OOL Analogy

We could make some analogy:

Java	Erlang
Object	Process
`new`	`spawn`
`_._`	`_!_`
Class	Function
Inheritance	Function Call
Method definition	`receive` statement

BUT what is the function type?

$domain \rightarrow codomain$ **is not enough**

Adding Effects

Let us collect all receive blocks (flow analysis)

```
state1(V) ->
  receive
    {add,V1}     -> state1(V1+V);
    {change,V1} -> state2(V,V1)
  end.
state2(V1,V2) ->
  receive
    {add,V3,V4} -> state2(V1+V3,V2+V4)
  end.
```

$$\texttt{state1} : \mathbf{num} \xrightarrow{\{\text{add}:\mathbf{num},\ \text{change}:\mathbf{num},\ \text{add}:\mathbf{num}\times\mathbf{num}\}} \bot$$

$$\texttt{state2} : \mathbf{num} \times \mathbf{num} \xrightarrow{\{\text{add}:\mathbf{num}\times\mathbf{num}\}} \bot$$

Process Types

```
state2(V1,V2) ->
  receive
    {add,V3,V4} -> state2(V1+V3,V2+V4);
    {mute,F}    -> F()
  end.
```

What is the effect of `F()`?

Process Types

```
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 Types

```
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.

$\Rightarrow$ Process Type:

$$P : @\{\ldots, \mathtt{m} : (\underbrace{\textit{received}}_{P\,\mathtt{!}\,\{\mathtt{m},_\}}, \overbrace{\textit{understood}}^{P \text{ execute a } \mathtt{receive}\ \{\mathtt{m},_\}}), \ldots\}$$

Subtyping

correction **iff** *received* $\sqsubseteq$ *understood*

Subtyping

correction **iff** *received* $\sqsubseteq$ *understood*

Process Type:

$$@\{m_i : (\alpha_1^i, \alpha_2^i)\}_{I_1} \sqsubseteq @\{m_i : (\beta_1^i, \beta_2^i)\}_{I_2}$$

iff

$I_2 \subseteq I_1$ and $\forall i \in I_2 \;\; \beta_1^i \sqsubseteq \alpha_1^i \; \alpha_2^i \sqsubseteq \beta_2^i$

Subtyping

correction **iff** *received* $\sqsubseteq$ *understood*

Process Type:

$$@\{m_i : (\alpha_1^i, \alpha_2^i)\}_{I_1} \sqsubseteq @\{m_i : (\beta_1^i, \beta_2^i)\}_{I_2}$$

iff

$I_2 \subseteq I_1$ and $\forall i \in I_2 \;\; \beta_1^i \sqsubseteq \alpha_1^i \; \alpha_2^i \sqsubseteq \beta_2^i$

Function Type:

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

iff

$\alpha_2 \sqsubseteq \alpha_1$ and $\beta_1 \sqsubseteq \beta_2$ and $I_1 \sqsubseteq I_2$

An Example

```
state3() -> receive kill -> true end.
> P=(spawn(state1,[1]))!{mute,state3}.
```

An Example

```
state3() -> receive kill -> true end.
> P=(spawn(state1,[1]))!{mute,state3}.
```

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

An Example

```
state3() -> receive kill -> true end.
> P=(spawn(state1,[1]))!{mute,state3}.
```

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

$\texttt{state3} \sqsubseteq \texttt{type(F)}$

An Example

```
state3() -> receive kill -> true end.
> P=(spawn(state1,[1]))!{mute,state3}.
```

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

$\texttt{state3} \sqsubseteq \texttt{type(F)}$

$unit \xrightarrow{\{\texttt{kill}:(\bot, unit)\}} \texttt{true} \sqsubseteq unit \xrightarrow{I} t$

An Example

```
state3() -> receive kill -> true end.
> P=(spawn(state1,[1]))!{mute,state3}.
```

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

$\texttt{state3} \sqsubseteq \texttt{type(F)}$

$unit \xrightarrow{\{\texttt{kill}:(\bot, unit)\}} \texttt{true} \sqsubseteq unit \xrightarrow{I} t$

$T_p \sqsubseteq @I \sqsubseteq @\{\texttt{kill} : (\bot, unit)\}$ and $\texttt{true} \sqsubseteq t$

An Example

```
state3() -> receive kill -> true end.
> P=(spawn(state1,[1]))!{mute,state3}.
```

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

$\texttt{state3} \sqsubseteq \texttt{type(F)}$

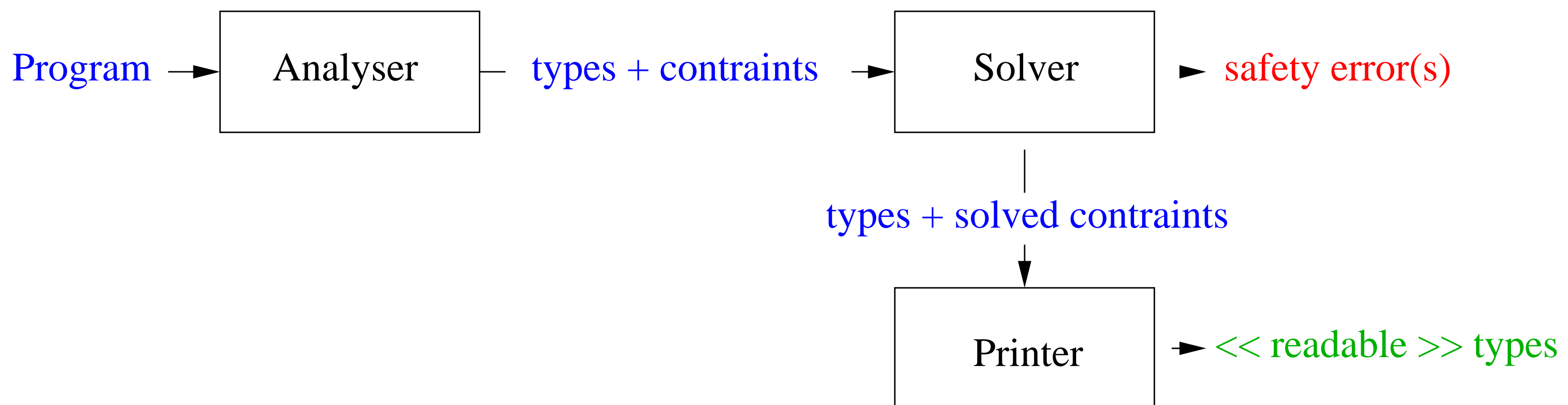
$unit \xrightarrow{\{\texttt{kill}:(\bot, unit)\}} \texttt{true} \sqsubseteq unit \xrightarrow{I} t$

$T_p \sqsubseteq @I \sqsubseteq @\{\texttt{kill} : (\bot, unit)\}$ and $\texttt{true} \sqsubseteq t$

$\Rightarrow$ `P!kill` is correct and `P!sub(1)` is not

Scaling to Erlang

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



- 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

Futures

- 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]
- π-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**: $a' \vdash \alpha, v_1 \,!\, v_2 \xrightarrow{v_1 \triangleleft v_2}_e \alpha, v_2$
  - **spawn**:

$$a \vdash \alpha, \texttt{spawn(}f\texttt{, [}v_1, \ldots, v_n\texttt{])} \xrightarrow{\langle a|\varnothing\rangle \triangleright f(v_1,\ldots,v_n)}_e \alpha, a$$

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()}.
```

$$\texttt{timer:} \quad @\widehat{\alpha} \times \mathbf{num} \times \alpha \xrightarrow{\{\texttt{cancel}:(\bot,\, @\widehat{\alpha})\}} \texttt{true} \sqcup \alpha$$

$$\texttt{timeout:} \quad \mathbf{num} \times \alpha \xrightarrow{\widehat{\alpha}} @\{\texttt{cancel} : (\bot,\, @\widehat{\alpha})\}$$

$$\texttt{cancel:} \quad @\{\texttt{cancel} : (@\phi,\, \top)\} \xrightarrow{\phi} \texttt{cancel} \times @\phi$$

Types and Effects

$$f(p_1^1, \ldots, p_n^1) \rightarrow e_1; \quad \ldots \quad f(p_1^k, \ldots, p_n^k) \rightarrow e_k.$$

$$\begin{cases} \texttt{type}(f) = \sqcup_i(\texttt{type}(p_1^i) \times \cdots \times \texttt{type}(p_n^i) \rightarrow \texttt{type}(\mathcal{E}, e_i)) \\ \texttt{effect}(f) = \cup_i \texttt{effect}(\mathcal{E}, e_i) \end{cases}$$

$$\begin{cases} \texttt{type}(\mathcal{E}, f\,\texttt{(}\,e_1, \ldots, e_n\,\texttt{)}) = t \\ \quad \text{if } \texttt{type}(f) = \texttt{type}(\mathcal{E}, e_1) \times \cdots \times \texttt{type}(\mathcal{E}, e_n) \rightarrow t \\ \texttt{effect}(\mathcal{E}, f\,\texttt{(}\,e_1, \ldots, e_n\,\texttt{)}) = \texttt{effect}(f) \cup \cup_i \texttt{effect}(\mathcal{E}, e_i) \end{cases}$$