Typing the Wild in Erlang

Nachiappan V

John Hughes

CHALMERS
UNIVERSITY OF TECHNOLOGY

OCTOPI

Erlang has no (static) types!

Icon made by Pixel-Perfect from www.flaticon.com

No types in a distributed language ⇒ distributed debugging!

```
...
spawn(DistantNode, mod, badfun, [42]).
...
```

A Practical Subtyping System For Erlang

Simon Marlow
simonm@dcs.gla.ac.uk
University of Glasgow

Philip Wadler
wadler@research.bell-labs.com
Bell Labs, Lucent Technologies

Typing Erlang

John Hughes, David Sands, Karol Ostrovský

December 12, 2002

TypEr: A Type Annotator of Erlang Code

Tobias Lindahl Konstantinos Sagonas
Department of Information Technology
Uppsala University, Sweden
{tobiasl,kostis}@it.uu.se

**Experience from Developing the Dialyzer:
A Static Analysis Tool Detecting Defects in Erlang Applicatior**

Konstantinos Sagonas
Department of Information Technology
Uppsala University, Sweden
kostis@it.uu.se

Practical Type Inference Based on Success Typings

Tobias Lindahl [1] Konstantinos Sagonas [1,2]
[1] Department of Information Technology, Uppsala University, Sweden
[2] School of Electrical and Computer Engineering, National Technical University of Athens, Greece
{tobiasl,kostis}@it.uu.se

Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story

Tobias Lindahl and Konstantinos Sagonas
Computing Science, Dept. of Information Technology, Uppsala University, Sweden
{Tobias.Lindahl,Konstantinos.Sagonas}@it.uu.se

Point Of No Local Return:

The Continuing Story Of Erlang Type Systems

Zeeshan Lakhani
Papers We Love, Basho Technologies
@zeeshanlakhani

Our good friend Dialyzer

```
-spec zip(List1, List2) -> List3
    when List1 :: [A],
         List2 :: [B],
         List3 :: [{A, B}],
         A :: term(), B :: term().
```

Dialyzer in action

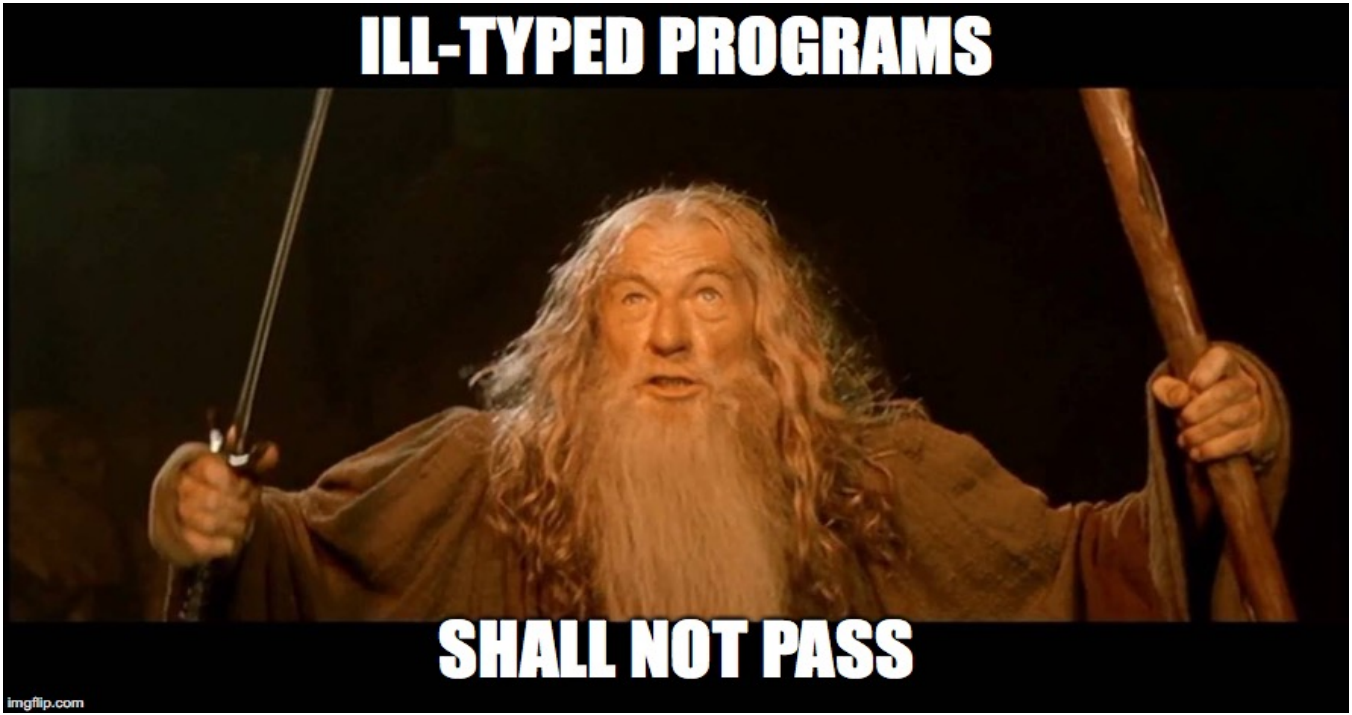
```
$ dialyzer test.erl

  Checking ..
  Proceeding with analysis...
  done in 0m0.13s

done (passed successfully)
```

UH OH!

```
find() ->
   {ok,"two"} = lookup(0,[{0,2.0}]).
```

{ok, 2.0}

Goal of our type system



When is something ill-typed?

- An expression is ill-typed if the *type system* cannot derive a type for it

- A type system which identifies more type errors rejects more programs!

The Question is

What do Erlang programs written with a *Hindley-Milner* type system look like?

The Question is *not*

Can we accept (succesfully type) as many existing Erlang programs as possible?

Why Hindley-Milner?

- Has been very successful in typing
- Strong type inference properties

"The difficulty is that with Hindley-Milner each type must involve a set of *constructors* distinct from those used in any other types" — Marlow and Wadler, 96

Why Hindley-Milner?

- Has been very successful in typing

"The difficulty is that with Hindley-Milner each type must involve a set of *constructors* distinct from those used in any other types" — Marlow and Wadler, 96

- Strong type inference properties

Algebraic Data Types (ADTs)

```haskell
data Tree a = Nil
    | Node a (Tree a) (Tree a)
```


```erlang
-type tree(A) :: nil
    | {node,A,tree(A),tree(A)}.
```

Type inference, an example

```
findNode(_,nil) ->
    false;
findNode(N,{_,Lt,_}) when N<Lt -> findNode(N,Lt);
findNode(N,{_,_,Rt}) -> findNode(N,Rt).
```

```
findNode/2 ::
(A,tree(A)) → boolean()
```

Assigning types to constructors

```
-type cl() :: {response,integer()}
-type sr() :: {request,integer()}
```

```
response/1 :: integer() → cl()
```

```
request/1 :: integer() → sr()
```

Overloading constructors

- Contemporary implementations of Hindley–Milner restrict constructors to have a *unique* type

- In Erlang, restricting constructors to a unique type is practically impossible
  Example: `{ok,Value}`

Constructor overloading problem

```
-type cl(R) :: {'EXIT',pid(),R}
     | {response,integer()}
-type sr(R) :: {'EXIT',pid(),R}
     | {request,integer()}
```

```
EXIT/2 :: ?
```

Constructor overloading problem

```
-type cl(R) :: {'EXIT',pid(),R}
     | {response,integer()}
-type sr(R) :: {'EXIT',pid(),R}
     | {request,integer()}
```

```
EXIT/2 :: (pid(), R) → cl(R) ?
```

```
EXIT/2 :: (pid(), R) → sr(R) ?
```

Constructor overloading solution

```
-type cl(R) :: {'                    d(),R}
     | {response,
-type sr(R) :: {                     ),R}
     | {request,in
```

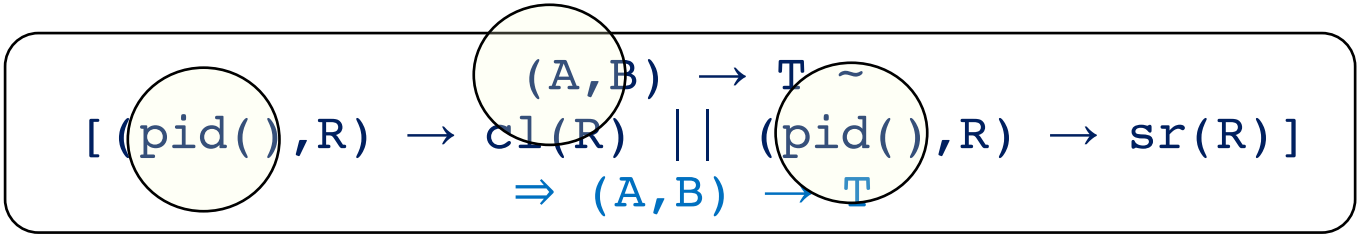
deferred unification constraint (duc)

```
EXIT/2 :: T ~ [cl(R) || sr(R)]
        ⇒ (pid(),R) → T
```

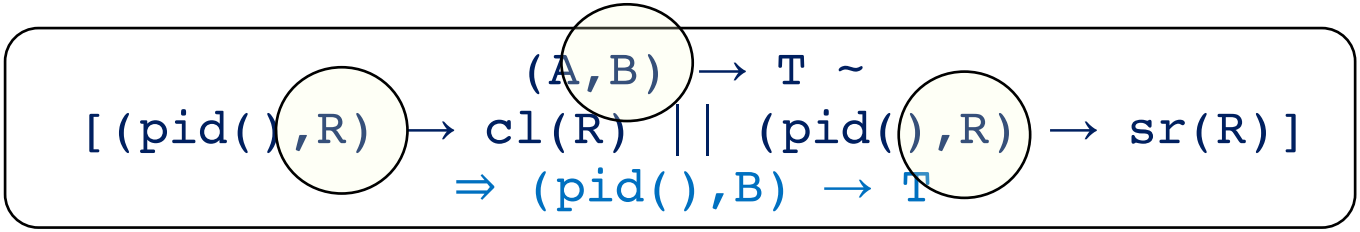
Deriving the type of `EXIT/2`

```
(A,B) → T
```

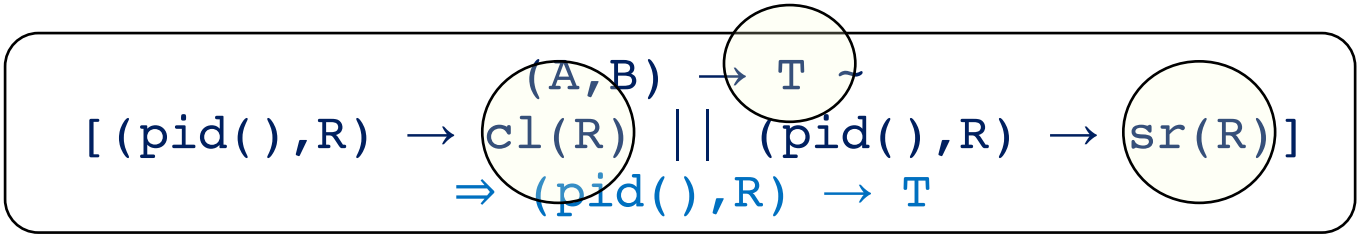
Deriving the type of `EXIT/2`



Deriving the type of `EXIT/2`



Deriving the type of `EXIT/2`



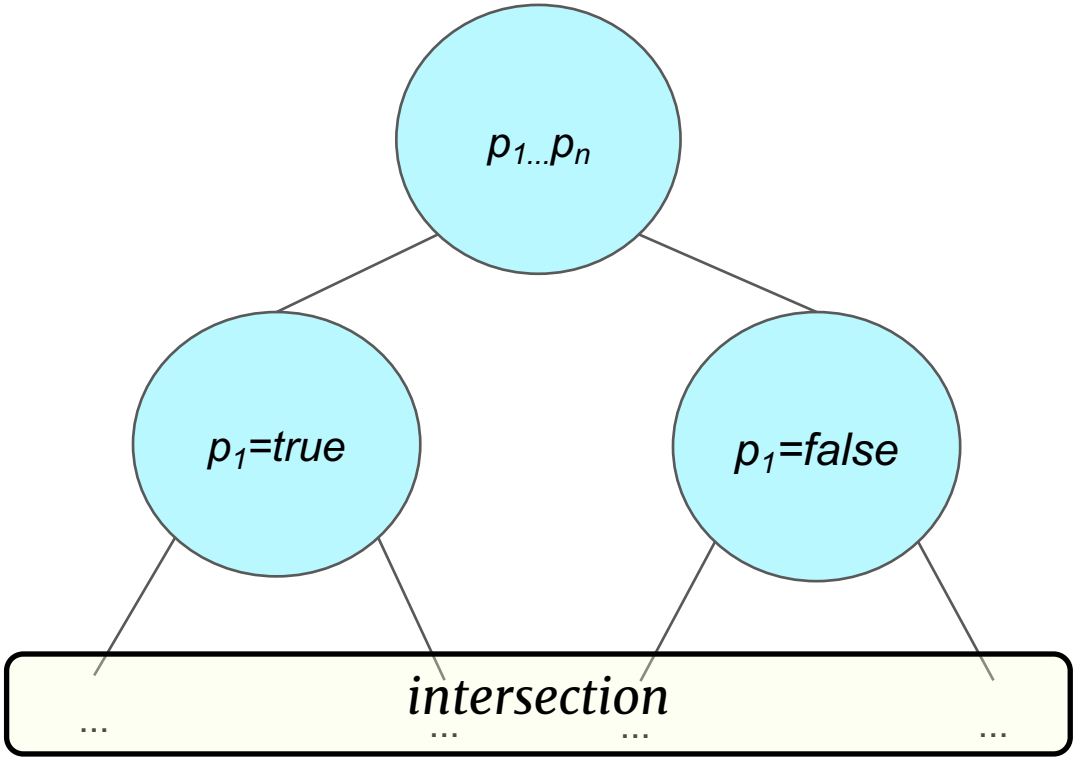
Deriving the type of `EXIT/2`

```
                    (A,B) → T ~
[(pid(),R) → cl(R) || (pid(),R) → sr(R)]
          ⇒ (pid(),R) → T
```



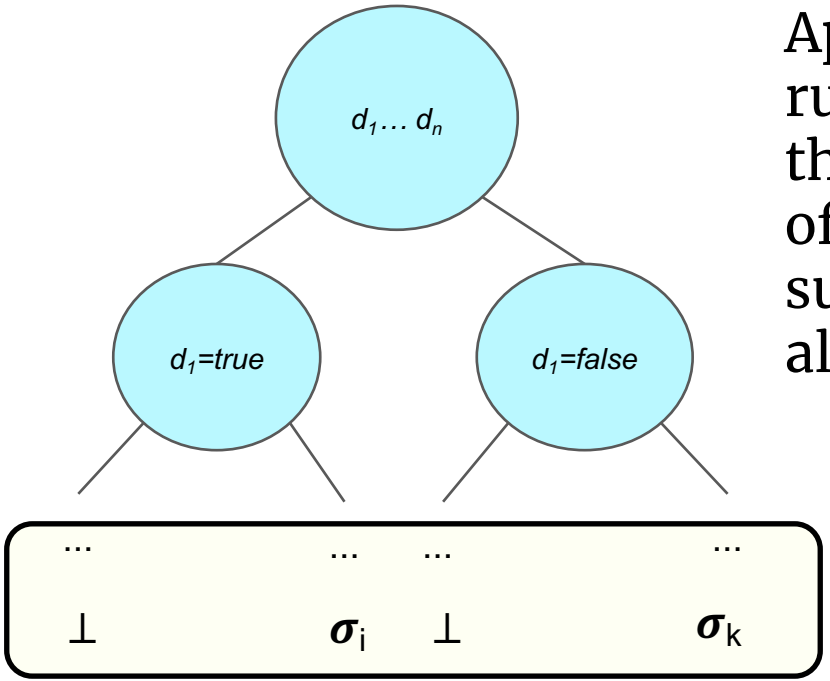
```
T ~ [cl(R) || sr(R)] ⇒ (pid(),R) → T
```

Stålmarck's Dilemma Rule [1]



[1] Sheeran & Stålmarck, *A Tutorial on Stålmarck's Proof Procedure for Propositional Logic*, 2000

Propositions as constraints!



Applying the Dilemma rule on ducs gives us that the intersection of all valid type substitutions must always hold

Allowing flexible programming

Branches of different type? No problem!
...*provided* return value is not used

```
case X of
    {request,N} ->
        ClientId ! N + 42;
    {'EXIT',_,R} ->
        log(R)
end
```

Difficult to type in Hindley-Milner

- `element(Position, Tuple)`

  `element(2,{a,b,c}) = b`

- `is_tuple(Tuple)`

  `is_tuple({}) = true`

- `spawn(Module, Function, Args)`

- …

Simplifying by Partial Evaluation

Before

```
T = {F(X),G(X)},
  element(1,T).
```

After

```
T1 = F(X),
T2 = G(X),
     T1.
```

Results

- Type inference applied successfully to a few small libraries (< 3 LOC modifed)
  - OTP libraries: *orddict* and *orddsets* (~ 200 LOC)
  - An implementation of a distributed fault tolerant resource pool (~100 LOC)
- Fast!
- Looks like Haskell programs

Vs Dialyzer

```
-type maybe(A) :: none | {ok,A}.

lookup(K,[])          -> none;
```

```
Type error:
Cannot unify [char()] with float()
```

```
find() ->
  {ok,"two"} = lookup(0,[{0,2.0}]).
```

Limitations!

- Can't do generic programming over constructors!

```
-type rbt(K,V) :: empty
   | {r,rbt(K,V),K,V,rbt(K,V)}
   | {b,rbt(K,V),K,V,rbt(K,V)}
to_list(empty, List) -> List;
to_list({_,L,Ks,Vs,R}, List) ->
  to_list(L, [{Ks,Vs}|to_list(R, List)]).
```

- PE helps only when at least *some* static information is available

Future Work

- Type inference for modules & error handling
- Typing concurrency by adding *effects*
- Better ways to integrate type inference and partial evaluation
- *Gradual typing?*

Good stuff in the paper

- Rules for typing records
- Type classes (for overloding)
- Discussion on type errors

That's all folks!

Type checker source at:
https://github.com/nachivpn/mt

CHALMERS
UNIVERSITY OF TECHNOLOGY

OCTOPI

Specializing type of a constructor

```
case X of
  {'EXIT',_,R} ->
      log(R)
  {request,N} ->
```

```
X ::
T ~ [cl(R) || sr(R)]
⇒ T
```

Specializing type of a constructor

```
case X of
  {'EXIT',_,R} ->
      log(R)
  {request,N} ->
```

```
                    X ::
T ~ [cl(R) || sr(R)] && T ~ [sr(R)]
                    ⇒ T
```

Specializing type of a constructor

```
case X of
  {'EXIT',_,R} ->
      log(R)
  {request,N} ->
```

```
X :: sr(R)
```

Typing Records

```
-record(person, {
    name :: string(),
    age  :: integer(),
    id
}).
```


generates

```
-type person(A) ::
    {person,string(),integer(),A}
```

Typing Records, an example

```
-type person(A) ::
      {person,string(),integer(),A}

me() -> #person{
      name = "Joe",
      age  = 20,
      id   = "order66"
}
```

```
me/0 :: person(string())
```