Typing the Untypeable

in Erlang using Partial Evaluation

Nachiappan Valliappan

CHALMERS
UNIVERSITY OF TECHNOLOGY

Er..what?

- Functional programming language

- Language based solution to distributed software

- Riak, RabbitMQ, ejabberd, WhatsApp...

But, no (static) types!

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

No types in a distributed language
$\Rightarrow$ distributed debugging!

```
...

spawn(FarAwayNode, my_module, non_existent_function,[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

TypEr: A Type Annotator of Erlang Code

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

Typing Erlang

John Hughes, David Sands, Karol Ostrovský

December 12, 2002

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

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

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

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

Our good friend Dialyzer

```
-spec zip(List1, List2) -> List3

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

Dialyzer in action

```
-module(test).



lookup(K,[])         -> none;

lookup(K,[{K,V}|_]) -> {ok,V};

lookup(K,[_|KVs])    -> lookup(K,KVs).



find() ->  {ok,"s"} = lookup(0,[{0,1}]).
```

```
… $ dialyzer test.erl

  Checking whether the PLT
/Users/nachi/.dialyzer_plt is
up-to-date... yes

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

done (passed successfully)
```

UH OH!

Goals of this thesis

- Retrofit a static type system to Erlang

- Respect Erlang’s philosophy: allow flexible programming

Hindley-Milner Type system

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

- Rejected by previous attempts to type Erlang
- “The **difficulty** is that with Hindley-Milner each type must involve a set of *constructors* distinct from those used in any other types, a convention not adhered to by Erlang programmers.” — Marlow and Wadler, 96

  But we show that this need not necessarily be the case!

Haskell ADT

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

Erlang ADT

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

Type inference for ADTs, an example

```
findNode(_,nil) ->
     false;
findNode(N,{node,N,Lt,Rt}) ->
     true;
findNode(N,{node,_,Lt,Rt}) ->
     findNode(N, Lt) or findNode(N,Rt).
```

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

The constructor overloading problem

- Restricting constructors to a unique type is practically impossible

  Example: `{ok,Value}, {'EXIT',pid(),Reason}`

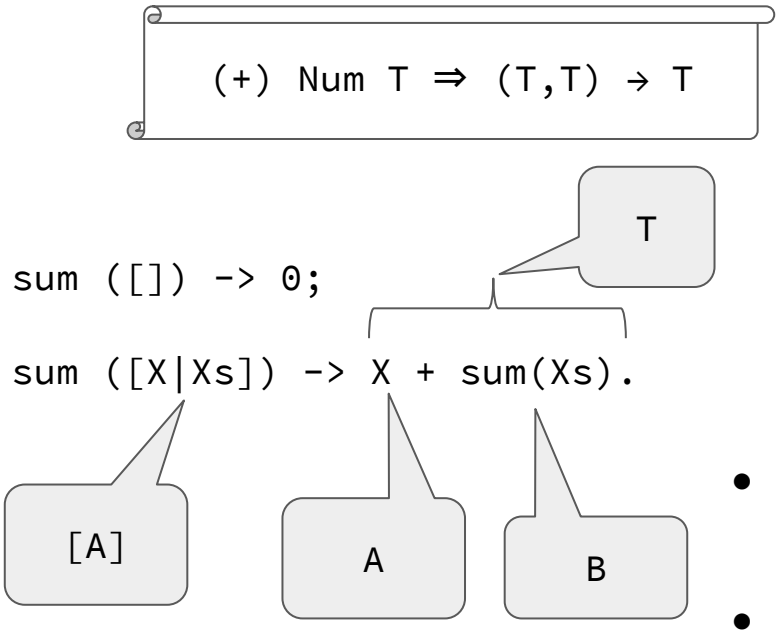
- But, consider these ADTS:

```
-type server(R) :: {'EXIT',pid(),R} | …
-type client(R) :: {'EXIT',pid(),R} | …
```

  What should be the *inferred* type of this constructor be?

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

Type inference walkthrough



Substitution

```
A = T
B = T
```

Predicates

```
{Num,T}
```

```
Intermediate: ([A]) → T

Final: Num T ⇒ ([T]) → T
```

- Collect type information and class constraints
  - Type information as a *substitution*
  - Class constraints as *predicates*
- Specialize the inferred type using collected information

Overloading constructors – the main idea

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

```
‘EXIT’/2 :: A ~ [cl(R),sr(R)] ⇒ (pid(),R) → A
```

deferred unification constraint (duc)

Overloading constructors – specialization

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

    end,
    serverHandler(ClientId,X).
```

```
serverHandler :: Padd D ⇒
    (D,sr(B)) → C
```

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

```
'EXIT'/2 :: (pid(),R) → sr(R)
```

Overloading constructors – general cases

```
getReason({‘EXIT’,_,R}) -> R.
```

```
foo() ->
    getReason({‘EXIT’,self(),true})
```

```
getReason/1 ::
A ~ [cl(R),sr(R)]
    ⇒ (A) → R
```

```
foo/0 ::(A,B) → C ~
[(pid(),B) → cl(B), (pid(),B) → sr(B)];
C ~ [cl(boolean()), sr(boolean())]
    ⇒ () → B
```

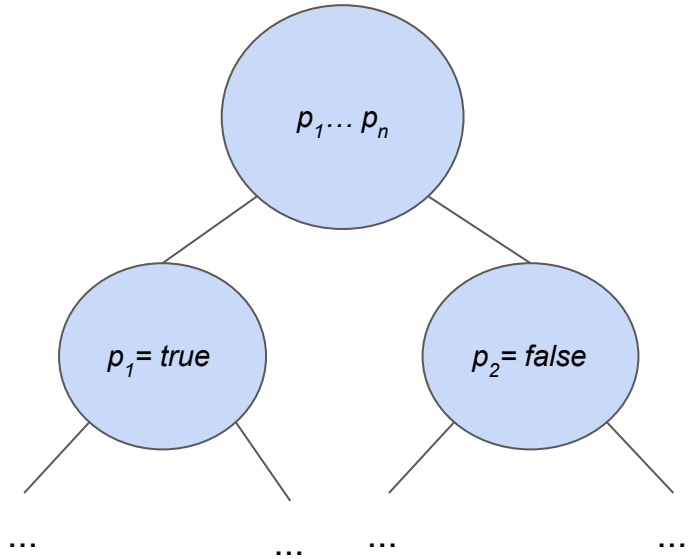
Extracting type information from ducs

```
foo/0 ::(A,B) → C ~ [(pid(),B) → cl(B), (pid(),B) → sr(B)];
          C ~ [cl(boolean()), sr(boolean())]
                    ⇒ () → B
```

```
   foo/0 :: C ~ [cl(B), sr(B)];
          C ~ [cl(boolean()), sr(boolean())]
                    ⇒ () → B
```

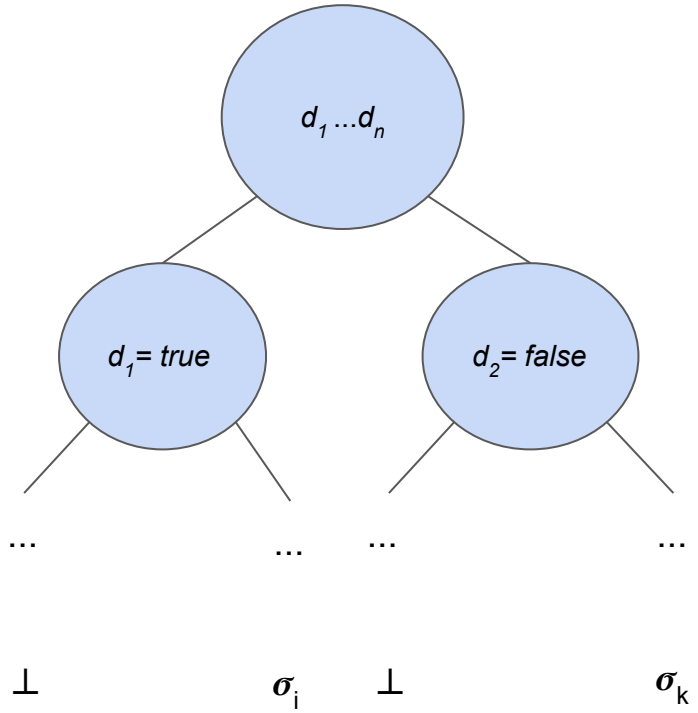
```
foo/0 :: () → boolean()
```

Stålmarck's Dilemma Rule



"Any information gained both from assuming that a proposition is true and from assuming that it is false must hold independent of the value of the proposition"

Applying Stålmarck's Dilemma Rule



- Valid proofs lead to a substitution
- Invalid proofs to a contradiction

Applying the Dilemma rule gives us that the intersection of all valid substitutions must be always true

Allowing flexible programming

- Branches of different type? No problem!
  (provided return value is not used)

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

- Want to compare values of different types? No problem!

```
(==) :: (A,B) → boolean()
```

Allowing flexible programming: Untypeable?

- element(Position,Tuple)

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

- is_function(Function,Arity)

```
is_function(fun (X) -> X end,1) = true
```

- is_tuple(Tuple)

```
is_tuple({}) = true
```

- spawn(Module,Function,Args) ...

Partial Evaluation: Overview

```
power(0,_) -> 1;
power(N,X) -> X * power(N-1,X).
```

```
power(3,X)
```

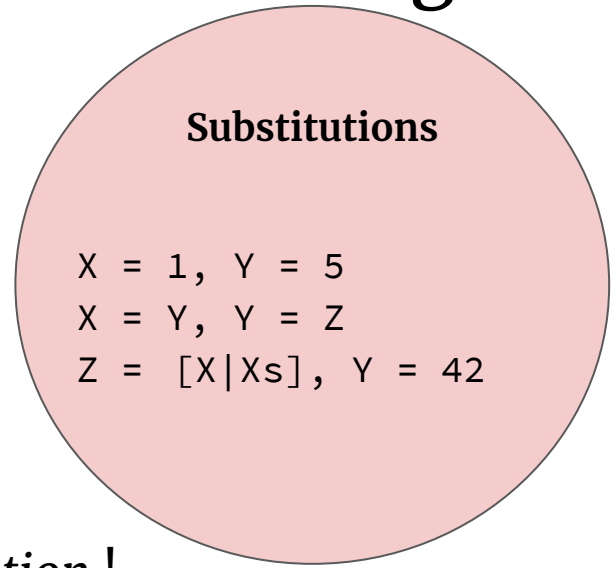
```
X * X * X * 1
```

Partial Evaluation for Erlang

Pattern matching

```
{X,Y} = {1,5}          ⇒
{X,X} = {Y,Z}          ⇒
{[X|Xs],42} = {Z,Y}    ⇒
```

…



looks like a job for *unification* !

Typing the Untypeable Using Partial Evaluation

Demo

Results

- Type inference applied successfully to a few small libraries
  - OTP libraries: *orddict* and *orddsets* (~ 200 LOC)
  - An implementation of a distributed fault tolerant resource pool (~100 LOC)
  - < 3 LOC added/modified in each case (mainly ADT definitions)
- Partial evaluation driven type inference applied to several handcrafted examples

Limitations

- Erlang programmers 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
- Current PE implementation unfolds only fully static function calls

Future Work

- Type inference for records, modules and error handling
- Partial evaluation for function calls with dynamic arguments
- Better ways to integrate type inference and partial evaluation

```
foo(Z) →
    [{A,B}|[X|Xs]] = Z, element(2,X)
```

X is a tuple of 2 elements according to TI, put PE doesn't know that!

- Typing concurrency by adding *effects* to the type system

THE END

Type classes for overloading other operators

```
(!) :: Padd A ⇒ (A,B) → B
```

```
unlink :: Port A ⇒ (A) → boolean()
```

...