TypEr: A Type Annotator of Erlang Code

Tobias Lindahl and Kostis Sagonas
Dept of Information Technology
Uppsala University

Background to this work

Erlang is dynamically typed and type safe.

It possible to infer types for variables based on their usage.

- Example: The arguments to addition must be numbers or else the call will fail. If the call succeeds the result must also be a number.

Dialyzer exploits this to reconstruct type information and report obvious type clashes to the user.

What is TypEr?

TypEr is a tool that automatically inserts type annotations in Erlang code.

The aims of TypEr:

- Facilitate documentation of Erlang code.
- Provide help to understand legacy code.
- Encourage a type-aware development of Erlang programs.

Design goals of TypEr

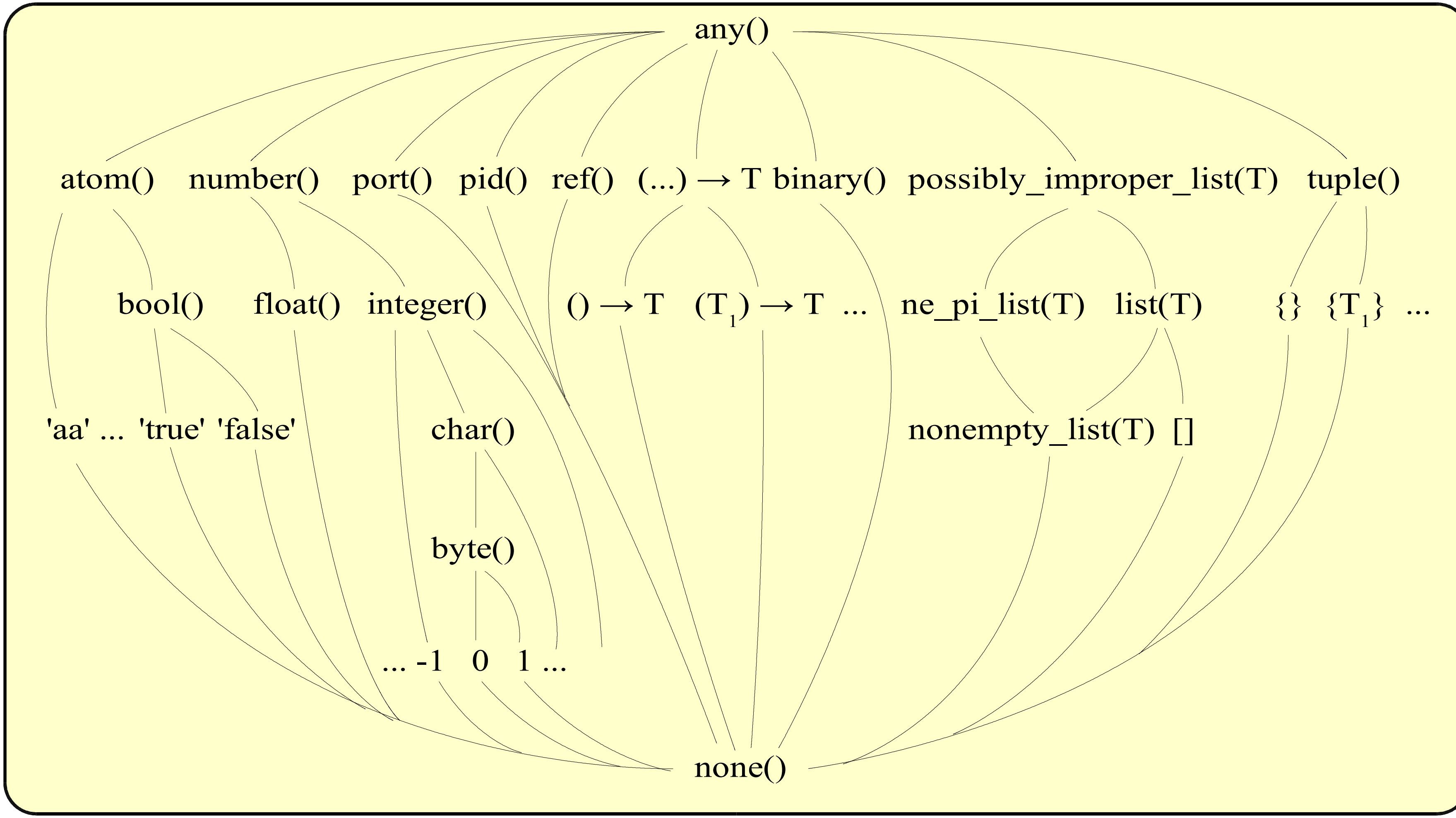
- TypEr should accept all Erlang code.
  - TypEr should not act as a type checker.
- TypEr should be fully automatic.
  - No user annotation of interfaces, etc.
- TypEr should perform reasonably even if all code is not available.
- TypEr should never be wrong.
  - The annotations should be as precise as possible, but still be safe over-approximations.

The type annotation language

The type system is based on subtyping and includes:

- Basic types, including one-point types.
  - *float(), pid(), binary(), atom(), 'ok', 'true', 42, ...*
- Structured types
  - *tuple(), {T_1, ..., T_n}, ...*
- Lists (as the only recursive type)
  - *list(T), [], nonempty_list(T), ...*
- *Disjoint unions*
  - *atom() | float(), -1 | 42, ...*
- *A largest and a smallest type*
  - *any(), none()*

The type lattice



The need for subtyping.

```
tag(N) when is_float(N) -> {float, N};
tag(N) when is_integer(N) -> {int, N}.
```

$$\begin{aligned} tag/1 :: (number()) \rightarrow\ & \{'float', float()\} \\ \mid\ & \{'int', integer()\} \end{aligned}$$

```
tag(N) when is_float(N) -> {float, N};
tag(N) when is_integer(N) -> {int, N};
tag(_) -> not_valid.
```

$$\begin{aligned} tag/1 :: (any()) \rightarrow\ & \{'float', float()\} \\ \mid\ & \{'int', integer()\} \\ \mid\ & 'not\_valid' \end{aligned}$$

Success types

The success type of a function expresses:

- For which domain a function can return, and
- The range for the function if it ever returns.

The type inference allows for type errors inside the function. It simply removes the offending clause.

When determining the success type of a function, only the function itself and the functions it calls are considered.

- No unification with the call sites.
- Allows for a modular type inference.

The analysis at a glance.

The type inference is based on subtype constraints.

The analysis works at the granularity of strongly connected components (SCCs) of the static call graph of the code.

- The static call graph is constructed.
- The SCCs are identified and sorted topologically.
- The SCCs are analyzed bottom-up, all the time using the accumulated information.

Constraint generation

Calls to functions with known type signatures.

Example: The built-in function `length/1` has the signature

$$\text{length/1} :: (\mathit{list}\,()) \rightarrow \mathit{integer}\,()$$

so the call

```
N = length(L),
```

yields the constraints

$$\tau_N \subseteq \mathit{integer}\,() \wedge \tau_L \subseteq \mathit{list}\,()$$

More refined type signatures.

The general signature for addition is

$$+/2 :: (number(), number()) \rightarrow number()$$

But we would expect the function

```
int_add(X, Y) when is_integer(X), is_integer(Y) -> X + Y.
```

to have the signature

$$\mathrm{int\_add}/2 :: (integer(), integer()) \rightarrow integer()$$

This is implemented by having a limited form of dependent types hard-coded in the analysis.

Case expressions

The general form of a case expression is

```
case E of
  P1 when G1 -> B1;
  ...
  Pn when Gn -> Bn
end
```

where

```
E – Expression
P – Pattern
G – Guard
B – Body
```

The generated constraints are

$$C_E \wedge \left(\bigvee_i \tau_E = \tau_{P_i} \wedge C_{G_i} \wedge C_{B_i} \wedge \tau_{out} \subseteq \tau_{B_i} \right)$$

Solving the constraints.

Conjunctive constraints:

- A type is the greatest lower bound (infimum) of all its subtype constraints.

Disjunctive constraints:

- Solve all partial constraints.
- A type is the lowest upper bound (supremum) of all the partial solutions.

Example of constraint solving

```
is_this_the_answer_1(X) ->
  case X of
    42 -> true;
    _  -> false
  end.
```

We have the constraints

$$(\tau_X \subseteq 42 \wedge \tau_{out} \subseteq '\mathit{true}') \vee (\tau_{out} \subseteq '\mathit{false}')$$

Each conjunct is trivial. Taking the supremum of the solutions:

$$\begin{array}{lll} \tau_{out} & \subseteq & sup('\mathit{true}', '\mathit{false}') = bool() \\ \tau_X & \subseteq & sup(42, any()) = any() \end{array}$$

The inferred signature:

$$\text{is\_this\_the\_answer\_1/1} :: (any()) \rightarrow bool()$$

Recursive functions: Fibonacci numbers

```
fib(0) -> 1;
fib(1) -> 1;
fib(X) -> fib(X-1) + fib(X-2).
```

From the first two clauses we have the closed form

$$\begin{pmatrix} \tau_{\text{fib}} = (0) \to 1 \\ \vee\, \tau_{\text{fib}} = (1) \to 1 \end{pmatrix} \Rightarrow \tau_{\text{fib}} = (0|1) \to 1$$

$$\boxed{\text{fib} :: (integer\,()) \to integer\,()}$$

Now solve iteratively:

$$\begin{aligned} &\tau_{X-1} \subseteq (0|1) \\ \wedge\, &\tau_{X-2} \subseteq (0|1) \\ \wedge\, &\tau_{X} \subseteq (1|2|3) \\ \wedge\, &\tau_{\text{fib}} = (integer\,()) \to (1|2) \end{aligned}$$

$$\begin{aligned} &\tau_{X-1} \subseteq integer\,() \\ \wedge\, &\tau_{X-2} \subseteq integer\,() \\ \wedge\, &\tau_{X} \subseteq integer\,() \\ \wedge\, &\tau_{\text{fib}} = (integer\,()) \to (1|2|3|4) \end{aligned}$$

Fibonacci numbers with a twist

```
fib(Zero) when Zero == 0 -> 1;
fib(One)  when One  == 1 -> 1;
fib(X) -> fib(trunc(X-1)) + fib(trunc(X-2)).
```

The constraint generation and solving are left as exercises, but the signature is:

$$\boxed{\text{fib} :: (\mathit{number}\,()) \rightarrow \mathit{integer}\,()}$$

Consequences of inferring success types

```
is_this_the_answer_2(X) when is_atom(X) ->
  case X of
    42 -> true;
    _  -> false
  end.
```

We have the constraints

$$\tau_X \subseteq atom() \wedge \big((\tau_X \subseteq 42 \wedge \tau_{out} \subseteq 'true') \vee (\tau_{out} \subseteq 'false')\big)$$

We have contradictory constraints from the first clause

$$\tau_X \subseteq atom() \wedge \tau_X \subseteq 42$$

The inferred signature: $\text{is\_this\_the\_answer\_2/1} :: (atom()) \rightarrow 'false'$

Success types: Handling exceptions

```
foo(X) when is_atom(X) ->
  io:format("Wrong input: ~w", [X]),
  exit(error);
foo(X) ->
  X + 1.
```

The type signature does not reflect the explicit handling of atoms

$$\text{is\_this\_the\_answer\_2/1} :: (\textit{number}()) \rightarrow \textit{number}()$$

Success typings: Servers

```
loop(Parent) when is_pid(Parent) ->
  receive
    {_Pid, Msg} ->
      Parent ! Msg,
      loop(Parent)
  end.
```

Since this function does not return its type signature becomes

$$\text{loop/1} :: (\mathit{any}()) \rightarrow \mathit{none}()$$

Success typings: Servers (cont'd)

```
loop(Parent) when is_pid(Parent) ->
  receive
    {_Pid, Msg} ->
      Parent ! Msg,
      loop(Parent);
    {Parent, stop} ->
      ok
  end.
```

Now we have a return value from the function and the signature becomes

$$\boxed{\mathrm{loop/1} :: (\,pid\,()) \rightarrow 'ok'}$$

Benefiting from the module system

The module system in Erlang provides means to make the result of the analysis more precise.

The functions in a module are either:

- Escaping - exposed to the outer world.
- Internal - protected against arbitrary calls from the outside.

Type signatures for internal functions are specialized by their uses.

Example 1: An internal function

```
-module(m1).
-export([main/1]).

main(N) when is_integer(N) -> tag(N+42).

tag(N) -> {tag, N}.
```

A first attempt:

$$\begin{aligned} &\text{main/1} :: (integer()) \rightarrow \{'tag', any()\} \\ &\text{tag/1} :: (any()) \rightarrow \{'tag', any()\} \end{aligned}$$

Since we know all call-sites to the function tag/1, we can specialize the signatures.

$$\begin{aligned} &\text{main/1} :: (integer()) \rightarrow \{'tag', integer()\} \\ &\text{tag/1} :: (integer()) \rightarrow \{'tag', integer()\} \end{aligned}$$

Example 2: An “internal” function escapes

```
-module(m2).
-export([main/1]).

main(N) when is_integer(N) -> {tag(N+42), fun tag/1}.

tag(N) -> {tag, N}.
```

Since the function tag/1 now escapes the module, we do not have control over all the call-sites, and we must assume that the function can be called with anything.

$$\begin{array}{ll} \text{main/1} & :: (\mathit{integer}()) \rightarrow \{\{'\mathit{tag}', \mathit{any}()\}, (\mathit{any}()) \rightarrow \{'\mathit{tag}', \mathit{any}()\}\} \\ \text{tag/1} & :: (\mathit{any}()) \rightarrow \{'\mathit{tag}', \mathit{any}()\} \end{array}$$

The analysis revisited

1. The static callgraph is constructed.
2. The SCCs are identified and sorted topologically.
3. The SCCs are analyzed bottom-up, using the constraint based analysis.
4. The SCCs are then traversed top-down to specialize the signatures of internal functions.
5. If no specializations are made we have reached a fix-point, otherwise repeat from 3.

Summary

- The type inference that TypEr employs:
  - Requires no annotations or code alternations.
  - Handles the complete Erlang language.
- The approach is novel and fast:
  - The annotation process is modular and incremental.
  - In the proceedings there are run-times for analyzing the whole Erlang/OTP.

Current and future work

- Currently TypEr is at the prototype stage, but we are working on a release.
- Add the possibility to take user-supplied type signatures into account.
- Investigate how the behavior of non-returning functions can be captured in a better way.
- Integrate the analysis of TypEr in Dialyzer to allow it to find more discrepancies.