Erlang
Types, Abstract Form & Core

Salvador Tamarit Muñoz

Universitat Politècnica de València
Erlang is **dynamically** and **strongly** typed. It is important to point out that although Erlang is dynamically typed, it is **type safe**. All values are **tagged** with their **type** during **runtime** and an **exception** is thrown if a **type clash** occurs.

```
1  -module(test).
2  -export([main/0]).
3
4  main() -> 6 + 1().
5
6  1() -> "2".
```
Types

Erlang is dynamically and strongly typed. It is important to point out that although Erlang is dynamically typed, it is type safe. All values are tagged with their type during runtime and an exception is thrown if a type clash occurs.

```
-module(test).
-export([main/0]).

main() -> 6 + 1().
1() -> "2".
```

```
1> c(test).
{ok,test}
2> test:main().
** exception error: an error occurred when evaluating an arithmetic expression in function test:main/0 (test.erl, line 4)
```
Developing programs in dynamically typed functional languages, programming is a tranquil and relatively uneventful activity. Since type declarations and annotations need not be typed (in), program development can progress more rapidly.

Unfortunately, this freedom of expression comes with a price. Significantly less typos and other such mundane programming errors are caught by the compiler.

By success typings of functions it is possible to reconstruct a significant portion of the type information which is implicit in a program, automatically annotate function interfaces (command typer), and detect definite type clashes (command dialyzer) without fundamental changes to the philosophy of the language or imposing a type system which unnecessarily rejects perfectly reasonable programs.
Dialyzer

```
-module(test).
-export([main/0]).

main() -> 6 + 1() .

1() -> "2" .
```
Dialyzer

```erlang
-module(test).
-export([[main/0]]).

main() -> 6 + 1().

1() -> "2".
```

$ dialyzer test.erl

test.erl:4: Function main/0 has no local return
test.erl:4: The call erlang:'+'(6,[50,...]) will never return since it differs in the 2nd argument from the success typing arguments: (number(),number())
Type systems based on subtyping try to solve sets of constraints of the form $\alpha \subseteq \beta$ while unification based type systems in the tradition of Hindley and Milner try to solve constraints of the form $\alpha = \beta$.

```
send(Pg, Mess) when is atom(Pg) ->
  global:send(Pg, {send, self(), Mess});
send(Pg, Mess) when is pid(Pg) ->
Pg ! {send, self(), Mess}.
```

A constructor-based type system such as Hindley-Milner can not type this function. If we adopt a subtyping system that allows for disjoint union types we can simply describe the first argument as a union containing atoms and pids.
Dialyzer

Type systems based on subtyping try to solve sets of constraints of the form $\alpha \subseteq \beta$ while unification based type systems in the tradition of Hindley and Milner try to solve constraints of the form $\alpha = \beta$.

```erlang
send(Pg, Mess) when is atom(Pg) ->
    global:send(Pg, {send, self(), Mess});
send(Pg, Mess) when is pid(Pg) ->
    Pg ! {send, self(), Mess}.
```

A constructor-based type system such as Hindley-Milner can not type this function. If we adopt a subtyping system that allows for disjoint union types we can simply describe the first argument as a union containing atoms and pids.
Dialyzer

Another example:

\[
\begin{align*}
\text{and}(true, true) & \rightarrow true; \\
\text{and}(false, _) & \rightarrow false; \\
\text{and}(_, false) & \rightarrow false.
\end{align*}
\]

Hindley-Milner: \((\text{bool}(), \text{bool}()) \rightarrow \text{bool}()\)
Success typing: \((\text{any}(), \text{any}()) \rightarrow \text{bool}()\)
Another example:

\[
\begin{align*}
\text{and}(true, true) & \rightarrow true; \\
\text{and}(false, _) & \rightarrow false; \\
\text{and}(_, \text{false}) & \rightarrow false.
\end{align*}
\]

**Hindley-Milner:** \((\text{bool}(), \text{bool}()) \rightarrow \text{bool}()\)

**Success typing:** \((\text{any}(), \text{any}()) \rightarrow \text{bool}()\)
Dialyzer

- **Types** represents **sets of values**. Since these unions can become large or even infinite, in the analysis Dialyzer imposes a **fixed size limit** after which the union is **widened to a supertype**. For example, if the union limit is three the union type $1 \cup 2 \cup 3 \cup 4$ will be widened to `integer()`.

- A **success typing** is a type signature that **over-approximates** the set of types for which the function can evaluate to a value.

- **The algorithm** for inferring **success typings** has two phases:
  - In the first, the code is traversed and constraints are generated using derivation rules.
  - In the second, Dialyzer tries to find a solution to the constraints and this solution constitutes the **success typing**.
%% (integer()) -> integer()
add1(X) when is_integer(X) -> X + 1.

%% (integer()) -> ok1
bar(X) ->
  case add1(X) of
    42 -> ok1;
    gazonk -> ok2
  end.
% (integer() | list()) -> integer() | atom()
foo(X) when is_integer(X) -> X + 1.
foo(X) -> list_to_atom(X).

% (integer()) -> ok1 | ok2
baz(X) when is_integer(X) ->
    case foo(X) of
        42 -> ok1;
        gazonk -> ok2
    end.

A type signature for `foo` that kept track of input-output type dependencies would make it possible to detect that the second clause is unreachable.
Dialyzer

```erlang
%% (integer() | list()) -> integer() | atom()
foo(X) when is_integer(X) -> X + 1.
foo(X) -> list_to_atom(X).

%% (integer()) -> ok1 | ok2
baz(X) when is_integer(X) ->
    case foo(X) of
        42 -> ok1;
        gazonzk -> ok2
    end.
```

A type signature for `foo` that kept track of **input-output type dependencies** would make it possible to detect that the second clause is unreachable.

```erlang
%% (integer()) -> integer() | (list()) -> atom()
```
main(A) ->
    X = f(A),
    case X of
        0  -> g(X);
        _  -> g(X-1)
    end.

f(X)  -> X+1.
g(42)  -> ok.
Dialyzer

```erlang
main(A) ->
    X = f(A),
    case X of
        0 -> g(X);
        _ -> g(X-1)
    end.

f(X) -> X+1.

g(42) -> ok.
```

$ dialyzer ex1.erl

ex1.erl:8: The call ex1:g(X::0) will never return since it differs in the 1st argument from the success typing arguments: (42)
main(A) ->
    X = f(A),
    Y = g(A,\theta),
    Z = f(Y),
    case Z of
    \theta -> g(X,A);
    _  -> g(X-1,\theta)
end.

f(\theta) -> \theta;
f(1)   -> 1.
g(1,\theta) -> 0.
Dialyzer

```erlang
main(A) ->
    X = f(A),
    Y = g(A,0),
    Z = f(Y),
    case Z of
        0 -> g(X,A);
        _ -> g(X-1,0)
    end.

f(0) -> 0;

f(1) -> 1.

g(1,0) -> 0.
```

```
$ dialyzer ex2.erl
ex2.erl:5: Function main/1 has no local return
ex2.erl:10: The call ex2:g(X::0 | 1,A::1) will never return since it differs in the 2nd argument from the success typing arguments: (1,0)
ex2.erl:11: The call ex2:g(-1 | 0,0) will never return since it differs in the 1st argument from the success typing arguments: (1,0)
```
Dialyzer

```erlang
main(X) ->
    Y = f(X),
    case Y of
        a -> 1;
        b -> 2;
        _ -> 42
    end.

f(one) -> a;

f(two) -> b.
```
Dialyzer

\[
\text{main}(X) \rightarrow \\
Y = f(X), \\
\text{case } Y \text{ of} \\
a \rightarrow 1; \\
b \rightarrow 2; \\
_ \rightarrow 42 \\
\text{end.} \\
f(\text{one}) \rightarrow a; \\
f(\text{two}) \rightarrow b.
\]

$\text{dialyzer ex3.erl}$

ex3.erl:10: The variable _ can never match since previous clauses completely covered the type 'a' | 'b'
Dialyzer

%% (list()) -> integer()
length_1([]) -> 0;
length_1([_|T]) -> 1 + length_1(T).

However, if we decide to do a simple program transformation and make this function tail-recursive:
However, if we decide to do a simple program transformation and make this function tail-recursive:

```erlang
%% (list()) -> any()
length_2(List) -> length_3(List, 0).

%% (list(), any()) -> any()
length_3([], N) -> N;
length_3([_|T], N) -> length_3(T, N+1).
```
A refined success typing is a success typing under some additional constraints.

A refined success typing is a success typing where the domain is restricted to some subtype of the success typing’s domain.

Since the set of possible inputs to the function gets restricted, the set of its possible outputs may also get restricted.

The success typing of a function captures the set of all its possible uses. A refined success typing a restricted set of uses which reflects how the function is actually used in a program.
-module(my_list_utils).
-export([length_2/1]).

%%% (list()) -> integer()
length_2(List) -> length_3(List, 0).

%%% (list(), integer()) -> integer()
length_3([], N) -> N;
length_3([_|T], N) -> length_3(T, N+1).
Dialyzer

\begin{verbatim}
-module(arith).
-export([t/1]).

\texttt{t(N) \rightarrow X = f(3.14) + N, n(42) + n(X).}

\texttt{n(N) \rightarrow N + 1.}

\texttt{f(N) \rightarrow N + 2.}
\end{verbatim}

Success typings

\begin{align*}
\texttt{t (number())} & \rightarrow \texttt{number()} \\
\texttt{n (number())} & \rightarrow \texttt{number()} \\
\texttt{f (number())} & \rightarrow \texttt{number()}
\end{align*}

Refined success typings

\begin{align*}
\texttt{t (number())} & \rightarrow \texttt{float()} \\
\texttt{n (number())} & \rightarrow \texttt{number()} \\
\texttt{f (float())} & \rightarrow \texttt{float()}
\end{align*}
Dialyzer

-module(arith).
-export([t/1]).

t(N) ->
    X = f(3.14) + N,
    n(42) + n(X).

n(N) -> N + 1.

f(N) -> N + 2.

Success typings

\[
t \ (\text{number}()) \rightarrow \text{number}()
\]

\[
n \ (\text{number}()) \rightarrow \text{number}()
\]

\[
f \ (\text{number}()) \rightarrow \text{number}()
\]

Refined success typings

\[
t \ (\text{number}()) \rightarrow \text{float}()
\]

\[
n \ (\text{number}()) \rightarrow \text{number}()
\]

\[
f \ (\text{float}()) \rightarrow \text{float}()
\]
Dialyzer

-module(arith).
-export([t/1]).

t(N) ->
    X = f(3.14) + N,
    n(42) + n(X).

n(N) -> N + 1.

f(N) -> N + 2.

Success typings

\[
\begin{align*}
t &\colon \text{number()} \rightarrow \text{number()} \\
n &\colon \text{number()} \rightarrow \text{number()} \\
f &\colon \text{number()} \rightarrow \text{number()}
\end{align*}
\]

Refined success typings

\[
\begin{align*}
t &\colon \text{number()} \rightarrow \text{float()} \\
n &\colon \text{number()} \rightarrow \text{number()} \\
f &\colon \text{float()} \rightarrow \text{float()}
\end{align*}
\]
-module(ex4).
-export([main/1]).

main(X)->
    case X of
    1 -> a;
    3 -> c
    end,
    f(X).

f(2) -> a.
```erlang
-module(ex4).
-export([main/1]).

main(X) ->
    case X of
        1 -> a;
        3 -> c
    end,
    f(X).

f(2) -> a.
```

$ dialyzer ex4.erl

ex4.erl:4: Function main/1 has no local return
ex4.erl:9: The call main:f(X:1 | 3) will never return since it differs in the 1st argument from the success typing arguments: (2)
ex4.erl:10: Function f/1 has no local return
ex4.erl:10: The pattern 2 can never match the type 1 | 3
-module(ex5).
-export([main/1]).

main(X) ->
  f(X),
  g(X),
  i(X),
  h(X).
  f(1)->a; f(2)->b; f(3)->c.
  g(1)->a; g(4)->b.
  i(1)->a; i(2)->b; i(3)->c.
  h(4)->c.
Dialyzer

-module(ex5).
-export([main/1]).

main(X) ->
    f(X),
    g(X),
    i(X),
    h(X).
    f(1)->a; f(2)->b; f(3)->c.
    g(1)->a; g(4)->b.
    i(1)->a; i(2)->b; i(3)->c.
    h(4)->c.

$ dialyzer ex5.erl
ex5.erl:4: Function main/1 has no local return
ex5.erl:8: The call ex5:h(X::1) will never return since it differs
    in the 1st argument from the success typing arguments: (4)
ex5.erl:10: The pattern 4 can never match the type 2 | 3
ex5.erl:11: The pattern 3 can never match the type 1
ex5.erl:11: The pattern 2 can never match the type 1
ex5.erl:12: Function h/1 has no local return
ex5.erl:12: The pattern 4 can never match the type 1
Abstract Format

The following Erlang expression:

\[
\text{foo:bar(baz,17).}
\]

it is represented in abstract format by the following tuple:

The tuple forms a tree structure:
Abstract Format

The following Erlang expression:

foo:bar(baz,17).

it is represented in abstract format by the following tuple:

{call,1,{remote,1,{atom,1,foo},{atom,1,bar}},{atom,1,baz},{integer,1,17}}.

The tuple forms a tree structure:
Abstract Format

The following Erlang expression:

\[ \text{foo:bar(baz,17)}. \]

it is represented in abstract format by the following tuple:

\[ \{\text{call,1,\{remote,1,\{atom,1,foo\}\,\{atom,1,bar\}\}},[\{\text{atom,1,baz}\,\{\text{integer,1,17}\}\}]}. \]

The tuple forms a tree structure:
Abstract Format

- A **module** is represented with a list \([F_1 \ldots F_n]\), where each \(F\) represents a **form**.
- A **form** is either an **attribute** or a **function declaration**. Concretely:
  - For instance, the abstract format corresponding to the attribute `-module(Mod)` is \(\{\text{attribute,LINE,module,Mod}\}\).
  - The abstract format corresponding to a function declaration is \(\{\text{function,LINE,Name,Arity,}[FC_1 \ldots FC_n]\}\) where each \(FC\) is the abstract format of a function clause, which in turn is represented by \(\{\text{clause,LINE,}[P_1 \ldots P_n], [G_1 \ldots G_n], [E_1 \ldots E_n]\}\) where each \(P\), \(G\) and \(E\) is the abstract representation of one of its **pattern**, one of its **guards** and one of its body’s **expressions** respectively.
The easiest way to get the abstract form corresponding to a source code is using module \texttt{smerl.erl} (Simple Metaprogramming for Erlang) included in the \texttt{erlyweb} library:

\begin{verbatim}
http://code.google.com/p/erlyweb/
\end{verbatim}

With a few lines of code we can get the abstract form of all the function declarations of a module from its source file:
The easiest way to get the abstract form corresponding to a source code is using module `smerl.erl` (Simple Metaprogramming for Erlang) included in the erlyweb library:

http://code.google.com/p/erlyweb/

With a few lines of code we can get the abstract form of all the function declarations of a module from its source file:

```erlang
{ok,AbstractForm} = smerl:for_file("test.erl"),
FunctionForms = [Form||Form={[function,_,_,_,_]}<-smerl:get_forms(AbstractForm)].
```
Metaprogramming

Let’s see an example. Consider the following Erlang module:

```
-module(test).
-export([[fac/1]]).

fac(0) -> 1;
fac(N) when N>0 -> mult(N, fac(N-1)).

mult(N1,N2) -> N1 * N2.
```

The corresponding abstract form for its function declarations is:
Metaprogramming

Let’s see an example. Consider the following Erlang module:

```erlang
-module(test).
-export([fac/1]).

fac(0) -> 1;
fac(N) when N>0 -> mult(N, fac(N-1)).
mult(N1,N2) -> N1 * N2.
```

The corresponding abstract form for its function declarations is:

```
[action 7,mult,2]
  [clause 7,
    [[var 7, 'N1'], [var 7, 'N2']],
    [],
    [[op 7, '*', [var 7, 'N1'], [var 7, 'N2']]]]]
{function 4, fac, 1,
  [clause 4, [{integer 4, 0}]], [{integer 4, 1}]},
[clause 5,
  [[var 5, 'N']],
  [[op 5, '>', [var 5, 'N']], [integer 5, 0]]],
  [call 5,
    {atom 5, mult},
    [var 5, 'N'],
    [call 5,
      {atom 5, fac},
      [[op 5, '-', [var 5, 'N']], [integer 5, 1]]]]]]]]
```
Metaprogramming

Additionally, with smerl it is possible to create and compile easily a new module allowing to call its functions:

```
-module(try_smerl).
-export([test_smerl/0]).

test_smerl() ->
    M1 = smerl:new(foo),
    {ok, M2} = smerl:add_func(M1, "bar() -> 1 + 1."),
    smerl:compile(M2),
    foo:bar().
```
Metaprogramming

Additionally, with smerl it is possible to create and compile easily a new module allowing to call its functions:

```erlang
-module(try_smerl).
-export([test_smerl/0]).

test_smerl() ->
    M1 = smerl:new(foo),
    {ok, M2} = smerl:add_func(M1, "bar() -> 1 + 1."),
    smerl:compile(M2),
    foo:bar().

> c(try_smerl).
{ok,try_smerl}
> try_smerl:test_smerl().
2
```
Metaprogramming

This code replaces all the original variable names by \textit{X}:

```erlang
-module(get_forms).
-export([main/0]).

main() ->
    {ok,Abstract} = smerl:for_file("test.erl"),
    FunForms = [Form||Form={function,_,_,_}<-smerl:get_forms(Abstract)],
    AttributeForms = smerl:get_forms(Abstract) ++ FunForms,
    NForms = [replaceVarFun(Form) || Form <- FunForms],
    NAbstract = smerl:set_forms(Abstract,AttributeForms++NForms),
    PrettyPrinted = smerl:to_src(NAbstract),
    io:format("-s\n",[PrettyPrinted]).

replaceVarFun({function,Line,Name,Ary,Clauses}) ->
    {function,Line,Name,Ary,[replaceVarClause(Clause) || Clause <- Clauses]}.

replaceVarClause({clause,Line,Par,GuardSeq,Exprs}) ->
    {clause,Line,
     [replaceVar(Par) || Par <- Par],
     [replaceVar(GuardSeq) || Guard <- Guard] || Guard <- GuardSeq],
     [replaceVar(Expr) || Expr <- Exprs]}.

replaceVar({var,Line,}) ->
    {var,Line,'X'};
replaceVar({op,Line,Oper,Expr1,Expr2}) ->
    {op,Line,Oper,replaceVar(Expr1),replaceVar(Expr2)};
replaceVar({call,Line,Func,Args}) ->
    {call,Line,Func,[replaceVar(Args) || Arg <- Args]}.
replaceVar(E) ->
    E.
```
Metaprogramming

The resulting code is:

```erlang
-module(test).
-export([[fac/1]]).

fac(0) -> 1;
fac(X) when X > 0 -> mult(X, fac(X - 1)).

mult(X, X) -> X * X.
```
Metaprogramming

Even easier:

```erlang
-module(get_forms2).
-export([[main/0]]).

main() ->
    {ok,Abstract} = smerl:for_file("test.erl"),
    Forms = smerl:get_forms(Abstract),
    NForms = [erl_syntax_lib:map(fun replaceVar/1,Form) || Form <- Forms],
    NAbstract = smerl:set_forms(Abstract,lists:reverse(NForms)),
    PrettyPrinted = smerl:to_src(NAbstract),
    io:format("~s\n",[PrettyPrinted]).

replaceVar([{var,Line,_}]) ->
    {var,Line,'X'};
replaceVar(E) ->
    E.
```
Metaprogramming

The best:

-module(trans_vars).
-export([[parse_transform/2]]).

parse_transform(Forms, _Options) ->
    [erl_syntax:revert(
        erl_syntax_lib:map(
            fun replaceVar/1,Form))
    || Form <- Forms].

replaceVar({var,Line, _}) ->
    {var,Line,'X'};
replaceVar(E) ->
    E.

-module(test).
-export([[fac/1]]).

-compile({parse_transform, trans_vars}).

fac(0) -> 1;
fac(N) when N>0 -> mult(N, fac(N-1)).
mult(N1,N2) -> N1 * N2.
Metaprogramming

It is very easy to get the abstract format corresponding to any expression:

```
main() ->
    {ok, Toks, _} = erl_scan:string("(X/3) + f(g(Y) , [Z || {atom,_,Z} <- L, Z /= []])."),
    {ok, [AExpr|_]} = erl_parse:parse_exprs(Toks),
    io:format("AExpr: ~p\n",[AExpr]).
```
Metaprogramming

It is very easy to get the abstract format corresponding to any expression:

```erlang
main() ->
    {ok, Toks, _} = erl_scan:string("(X/3) + f(g(Y) , [Z || {atom,_,Z} <- L, Z /= []])."),
    {ok, [AExpr|_]} = erl_parse:parse_exprs(Toks),
    io:format("AExpr: ~p
", [AExpr]).
```

\[\text{AExpr: } \{\text{op,1,'+',}
\{\text{op,1,'/',} , \{\text{var,1,'X'}, \{\text{integer,1,3}}\}\},
\{\text{call,1,}
\{\text{atom,1,f},
\{\text{call,1,}\{\text{atom,1,g}, \{\text{var,1,'Y'}\}\}},
\{\text{lc,1,}
\{\text{var,1,'Z'}\},
\{\text{generate,1,}
\{\text{tuple,1,}\{\text{atom,1,atom}, \{\text{var,1,'_'}\}, \{\text{var,1,'Z'}\}\}},
\{\text{var,1,'L'}\}},
\{\text{op,1,'/',} , \{\text{var,1,'Z'}, \{\text{nil,1}}\}\}\}\}}\]
Metaprogramming

We can use the functionality of module `erl_eval` to evaluate abstract format structures:

\[ Y = X + 3. \quad \% Being 4 the value of X \]
Metaprogramming

We can use the functionality of module `erl_eval` to evaluate abstract format structures:

\[ Y = X + 3. \%
\text{Being} 4 \text{ the value of } X \]

\begin{verbatim}
> erl_eval:expr([{match, 1, 
                   {var, 1, 'Y'}, 
                   {op, 1, '+'}, 
                   {var, 1, 'X'}, 
                   {integer, 1, 3}]],
               [{'X', 4}]).

{value, 7, [{'X', 4}, {'Y', 7}]}
\end{verbatim}
Metaprogramming

There are several useful modules to work with the abstract form:

**erl_syntax**

This module defines an abstract data type for representing Erlang source code as syntax trees, in a way that is backwards compatible with the data structures created by the Erlang standard library parser module `erl_parse`.

**erl_syntax_lib**

This module contains utility functions for working with the abstract data type defined in the module `erl_syntax`.

**erl_prettypr**

Pretty printing of abstract Erlang syntax trees.

**erl_id_trans**

An Identity Parse Transform.
Metaprogramming

There are several *useful modules* to work with the *abstract form*:

**erl_tidy**

Tidies and pretty-prints Erlang source code, removing unused functions, updating obsolete constructs and function calls, etc.

**epp_dodger**

Bypasses the Erlang preprocessor - avoids macro expansion, file inclusion, conditional compilation, etc. Allows to *find/modify particular definitions/applications of macros*, etc.

**igor**

It merges the source code of one or more Erlang modules into a single module, which can then replace the original set of modules.
Core Erlang

- A program operating on source code must handle so many cases as to become impractical in general. Core Erlang was designed to overcome this issue.

- The compiler uses it as an intermediate representation between the source code and the byte code. Additionally it helps to perform some optimizations.

- Some interesting features are:
  - A strict, higher-order functional language.
  - Simple and unambiguous grammar.
  - Human-readable textual representation.
  - Language easy to work with.
There are two main ways to get the core representation of a module:

- **Inside the interpreter** with the following instruction:
  
  ```erlang
  >c(test, to_core).
  ```

- **Inside a source code** using the functionality of the module `compile`:
  
  ```erlang
  compile:file("test.erl", [to_core, binary])
  ```
Consider the following **Erlang module**:

```erlang
-module(test).
-export([fac/1]).

fac(0) -> 1;
fac(N) when N>0 -> mult(N, fac(N-1)).

mult(N1,N2) -> N1 * N2.
```

This code produce the following **Core Erlang code**:

```erlang
module 'test' ['fac'/1]
attributes []
'fac'/1 =
  fun (_cor0) ->
    case _cor0 of
      <0> when 'true' -> 1
      <N> when call 'erlang':'+'(N, 0) ->
        let <_cor1> = call 'erlang':'+'(N, 1)
        in let <_cor2> = apply 'fac'/1(_cor1)
        in apply 'mult'/2(N, _cor2)
        end
  end
'mult'/2 =
  fun (_cor1,_cor0) -> call 'erlang':'*'(N, _cor1, _cor0)
end
```
The following function extracts all the name variables of a Core Erlang term:

```
variables(T) ->
    case cerl:type(T) of
        literal -> [];
        var -> [cerl:var_name(T)];
        'let' ->
            Vs = variables(cerl:let_body(T)),
            Vs1 = var_list_names(cerl:let_vars(T)),
            Vs2 = ordsets:union(Vs, Vs1),
            ordsets:union(variables(cerl:let_arg(T)), Vs2);
        apply ->
            ordsets:union(
                variables(cerl:apply_op(T)),
                var_list(cerl:apply_args(T)));
        call ->
            ordsets:union(variables(cerl:call_module(T)),
              ordsets:union(
                  variables(cerl:call_name(T)),
                  var_list(cerl:call_args(T))));
        'case' ->
            ordsets:union(variables(cerl:case_arg(T)),
              var_list(cerl:case_clauses(T)));
        clause ->
            Vs = ordsets:union(variables(cerl:clause_guard(T)),
                                variables(cerl:clause_body(T)));
            Vs1 = var_list(cerl:clause_vars(T)),
            ordsets:union(Vs, Vs1);
        'fun' ->
            Vs = variables(cerl:fun_body(T)),
            Vs1 = var_list_names(cerl:fun_vars(T)),
            ordsets:union(Vs, Vs1);
        module ->
            Vs = var_list_defs(cerl:module_defs(T)),
            Vs1 = ordsets:union(var_list(cerl:module_exports(T)), Vs),
            Vs2 = var_list_names(cerl:module_vars(T)),
            ordsets:union(Vs1, Vs2)
    end.
```
The auxiliar functions used by previous functions are:

```erlang
vars_in_list(Ts) ->
    vars_in_list(Ts, []).

vars_in_list([T | Ts], A) ->
    vars_in_list(Ts, ordsets:union(variables(T), A));
vars_in_list([], A) ->
    A.

vars_in_defs(Ds) ->
    vars_in_defs(Ds, []).

vars_in_defs([_, F | Ds], A) ->
    vars_in_defs(Ds, ordsets:union(variables(F), A));
vars_in_defs([], A) ->
    A.

var_list_names(Vs) ->
    var_list_names(Vs, []).

var_list_names([V | Vs], A) ->
    var_list_names(Vs, ordsets:add_element(cerl:var_name(V), A));
var_list_names([], A) ->
    A.
```
Manipulating Core Erlang

Considering previous Core Erlang module:

```
module 'test' ['fac'/1]
  attributes []
  'fac'/1 =
    fun (_cor0) ->
      case _cor0 of
        <> when 'true' -> 1
        <N> when call 'erlang':>'(_cor0,0) ->
          let _cor1 = call 'erlang':>'(N, 1)
          in let _cor2 = apply 'fac'/1(_cor1)
              in apply 'mult'/2(N, _cor2)
      end
  end
'mult'/2 =
  fun (_cor1,_cor0) -> call 'erlang':>'('*(_cor1, _cor0)
end
```

the result of the function is:

```
['N', cor0, cor1, cor2, {fac, 1}, {mult, 2}]
```
Manipulating Core Erlang

Interesting modules when manipulating Core Erlang are:

- cerl
  This module defines an abstract data type for representing Core Erlang source code as syntax trees.

- cerl_trees
  Basic functions on Core Erlang abstract syntax trees.

- cerl_clauses
  Utility functions for Core Erlang case/receive clauses.

- cerl_inline
  An implementation of the algorithm by Waddell and Dybvig (Fast and Effective Procedure Inlining, 1997), adapted to the Core Erlang.
Manipulating Core Erlang

Interesting modules when manipulating Core Erlang are:

**cerl_prettypr**
Core Erlang prettyprinter.

**cerl_closurean**
Closure analysis of Core Erlang programs.

**cerl_pmatch**
Core Erlang pattern matching compiler.

**cerl_typean**
Type analysis of Core Erlang programs.
Conclusions

- Erlang is language **widely used in the industry with a clear syntax** which allows to create **reliable concurrent and distributed systems**.
- The drawbacks of having a dynamic type system are solved by **Dialyzer**, a static analysis tool that helps to prevent runtime errors through the use of **success typing inference**.
- Erlang **abstract format is easy to manipulate and understand**.
- **Core Erlang** is suitable for program analysis. It is **human-readable** and there exists a **lot of libraries** helping to work with it.
Thanks for listening!