Metaprogramming for Erlang

Abstract Format & Core

Salvador Tamarit Muñoz

Universitat Politècnica de València

Contents

Abstract Format

Description

The following Erlang expression:

```
foo:bar(baz,17).
```

it is represented in abstract format by the following tuple:

The tuple forms a tree structure:

Description

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:

Description

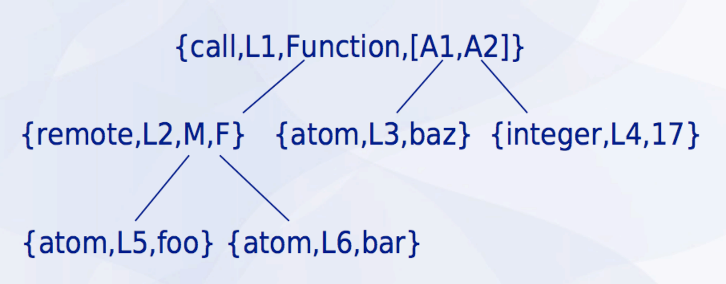
The following Erlang expression:

```
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:



Description

- A module is represented with a list `[F1...Fn]`, 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 `{attribute,LINE,module,Mod}`.
  - The abstract format corresponding to a function declaration is `{function,LINE,Name,Arity,[FC1...FCn]}` where each `FC` is the abstract format of a function clause, wich in turn is represented by `{clause,LINE,[P1...Pn],[G1...Gn],[E1...En]}` 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 whole abstract format description is publicly available at:

  http://www.erlang.org/doc/apps/erts/absform.html

How to get it

One line of code:

```
{ok, Forms} = epp:parse_file(File, [], [])
```

Let's see an example:

factorial.erl

How to get it

One line of code:

```
{ok, Forms} = epp:parse_file(File, [], [])
```

Let's see an example:

factorial.erl

Manipulating Abstract Format

We want to write a program transformation to replace each occurrence of an append operator (++) where its left operand is single element list (e.g. [1]) with a list construction. For instance:

```
[1] ++ [2,3] → [1 | [2,3]]
```

Additionally, we will replace append operations where one of their operand is an empty list with its other operand. For instance:

```
[] ++ [2,3] → [2,3]
[2,3] ++[] → [2,3]
```

The code is in:

`refactorer.erl`

And a test with some cases in:

`refactorer_test.erl`

Manipulating Abstract Format

We want to write a program transformation to replace each occurrence of an append operator (++) where its left operand is single element list (e.g. [1]) with a list construction. For instance:

```
[1] ++ [2,3] → [1 | [2,3]]
```

Additionally, we will replace append operations where one of their operand is an empty list with its other operand. For instance:

```
[] ++ [2,3] → [2,3]
[2,3] ++[] → [2,3]
```

The code is in:

`refactorer.erl`

And a test with some cases in:

`refactorer_test.erl`

Manipulating Abstract Format

We can write a simpler version using function `erl_syntax_lib:map/2`:

`refactorer_simpler.erl`

We have been accessing directly to the raw abstract format. We can use library `erl_syntax` to avoid having to do so:

`refactorer_simpler_alt.erl`

Manipulating Abstract Format

We can write a simpler version using function `erl_syntax_lib:map/2`:

`refactorer_simpler.erl`

We have been accessing directly to the raw abstract format. We can use library `erl_syntax` to avoid having to do so:

`refactorer_simpler_alt.erl`

Manipulating Abstract Format

If we want easily reuse our transformations, we can use parse transformations. In order to implement a parse transformation, we need to write in our module a function named`parse_transform/2` and export it.

`refactorer_pt.erl`

Then, we can reuse this transformation via a complier option:

```
c(refactorer_test,[{parse_transform,refactorer_pt}]).
```

The loaded code will have its content changed according to the transformation defined in `refactorer_pt`.

When a program transformation is faced for first time, it is a good idea to take a look in the following module to consult the cases to be treated:

`erl_id_trans.erl`

Manipulating Abstract Format

If we want easily reuse our transformations, we can use parse transformations. In order to implement a parse transformation, we need to write in our module a function named`parse_transform/2` and export it.

`refactorer_pt.erl`

Then, we can reuse this transformation via a complier option:

```
c(refactorer_test,[{parse_transform,refactorer_pt}]).
```

The loaded code will have its content changed according to the transformation defined in `refactorer_pt`.

When a program transformation is faced for first time, it is a good idea to take a look in the following module to consult the cases to be treated:

`erl_id_trans.erl`

Manipulating Abstract Format

If we want easily reuse our transformations, we can use parse transformations. In order to implement a parse transformation, we need to write in our module a function named`parse_transform/2` and export it.

`refactorer_pt.erl`

Then, we can reuse this transformation via a complier option:

```
c(refactorer_test,[{parse_transform,refactorer_pt}]).
```

The loaded code will have its content changed according to the transformation defined in `refactorer_pt`.

When a program transformation is faced for first time, it is a good idea to take a look in the following module to consult the cases to be treated:

`erl_id_trans.erl`

Get the Abstract Format from an Erlang expression

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]).
```

Get the Abstract Format from an Erlang expression

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]).
```

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

Smerl

Module `smerl.erl` provides functionality to simplify the metaprogramming task, along with some interesting features.

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

`smerl_test.erl`

Evaluating Abstract Format

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

```
Y = X + 3. % Being 4 the value of X
```

Evaluating Abstract Format

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

```
Y = X + 3. % Being 4 the value of X
```

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

```
{value,7,[{'X',4},{'Y',7}]}
```

Libraries

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.

Libraries

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

Description

- 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.

Syntax

$$
\begin{array}{lcl}
\text{fname} & ::= & \text{Atom} \,/\, \text{Integer} \\
\text{lit} & ::= & \text{Atom} \mid \text{Integer} \mid \text{Float} \mid \text{Char} \mid \text{String} \mid [\;] \\
\text{fun} & ::= & \texttt{fun}(\text{var}_1 \,, \ldots, \text{var}_n) \texttt{ -> } \text{exprs} \\
\text{clause} & ::= & \text{pats } \texttt{when } \text{exprs}_1 \texttt{ -> } \text{exprs}_2 \\
\text{pat} & ::= & \text{var} \mid \text{lit} \mid [\,\text{pats} \mid \text{pats}\,] \mid \{\,\text{pats}_1, \ldots, \text{pats}_n\,\} \mid \text{var} = \text{pats} \\
\text{pats} & ::= & \text{pat} \mid < \text{pat}, \ldots, \text{pat} > \\
\text{exprs} & ::= & \text{expr} \mid < \text{expr}, \ldots, \text{expr} > \\
\text{expr} & ::= & \text{var} \mid \text{fname} \mid \text{fun} \mid [\,\text{exprs} \mid \text{exprs}\,] \mid \{\,\text{exprs}_1, \ldots, \text{exprs}_n\,\} \\
 & & \mid \texttt{let } \text{vars} = \text{exprs}_1 \texttt{ in } \text{exprs}_2 \\
 & & \mid \texttt{letrec } \text{fname}_1 = \text{fun}_1 \ldots \text{fname}_n = \text{fun}_n \texttt{ in } \text{exprs} \\
 & & \mid \texttt{apply } \text{exprs} \;(\, \text{exprs}_1 \,, \ldots, \text{exprs}_n \,) \\
 & & \mid \texttt{call } \text{exprs}_{n+1}\texttt{:}\text{exprs}_{n+2} \;(\, \text{exprs}_1 \,, \ldots, \text{exprs}_n \,) \\
 & & \mid \texttt{primop } \text{Atom} \;(\, \text{exprs}_1 \,, \ldots, \text{exprs}_n \,) \\
 & & \mid \texttt{try } \text{exprs}_1 \texttt{ of } <\text{var}_1 \,, \ldots, \text{var}_n> \texttt{ -> } \text{exprs}_2 \texttt{ catch } <\text{var}'_1 \,, \ldots, \text{var}'_m> \texttt{ -> } \text{exprs}_3 \\
 & & \mid \texttt{case } \text{exprs} \texttt{ of } \text{clause}_1 \ldots \text{clause}_n \texttt{ end} \mid \texttt{do } \text{exprs}_1 \ \text{exprs}_2 \mid \texttt{catch } \text{exprs} \\
\xi & ::= & \text{Exception}(\overline{val_m}) \\
\text{val} & ::= & \text{lit} \mid \text{fname} \mid \text{fun} \mid [\,\text{vals} \mid \text{vals}\,] \mid \{\text{vals}_1, \ldots, \text{vals}_n\} \mid \xi \\
\text{vals} & ::= & \text{val} \mid < \text{val}, \ldots, \text{val} > \\
\text{vars} & ::= & \text{var} \mid < \text{var}, \ldots, \text{var} >
\end{array}
$$

How to get it

There are two main ways to get the core representation of a module:

- In the interpreter, with the following instruction:

```
> c(factorial,to_core).
```

- In a source code, using the functionality of the module `compile`:

```
{ok,_,Core} =
compile:file("factorial.erl",[to_core, binary])
```

Try it:

factorial.erl

How to get it

There are two main ways to get the core representation of a module:

- In the interpreter, with the following instruction:

```
> c(factorial,to_core).
```

- In a source code, using the functionality of the module `compile`:

```
{ok,_,Core} =
compile:file("factorial.erl",[to_core, binary])
```

Try it:

factorial.erl

From Erlang to Core

It is important to know some basis of how the translation from Erlang to Core Erlang is done. It is going to be definitely very useful when treating the Core Erlang code. Let see an example:

`core_transformations.erl`

There is a compiler option that let the resulting Core Erlang closer to the original Erlang source. We should use it when we are facing a problem where the correspondence between Core and Erlang is important.

```
> c(core_transformations, [to_core, no_copt])
```

From Erlang to Core

It is important to know some basis of how the translation from Erlang to Core Erlang is done. It is going to be definitely very useful when treating the Core Erlang code. Let see an example:

core_transformations.erl

There is a compiler option that let the resulting Core Erlang closer to the original Erlang source. We should use it when we are facing a problem where the correspondence between Core and Erlang is important.

```
> c(core_transformations, [to_core, no_copt])
```

Manipulating Core Erlang

Suppose we need to know all the variables names used in a module. We should only return the names of user defined variables, in other words, we should ignore variables introduces by Core (i.e. cor variables). Function names are also considered as variables names and they are represented by tuples of two elements. We want these function names to be ignored also. The code is in:

`core_vars.erl`

As in the case of Abstract Format, there are some libraries that can simplify this code. In this case, library `cerl_trees` contain some useful methods. We can get a simpler version of our previous code:

`core_vars_simpler.erl`

Manipulating Core Erlang

Suppose we need to know all the variables names used in a module. We should only return the names of user defined variables, in other words, we should ignore variables introduces by Core (i.e. cor variables). Function names are also considered as variables names and they are represented by tuples of two elements. We want these function names to be ignored also. The code is in:

`core_vars.erl`

As in the case of Abstract Format, there are some libraries that can simplify this code. In this case, library `cerl_trees` contain some useful methods. We can get a simpler version of our previous code:

`core_vars_simpler.erl`

Get the Core Erlang from an Erlang expression

We could need to get the Core Erlang corresponding to a single expression. We can build an auxiliary module, and then compile it to core. The following module shows how to get this:

`core_from_expression.erl`

Get the Core Erlang from an external module

Sometimes we need to load Core Erlang from a module included in the OTP libraries. The following code allow us to get this:

core_from_module.erl

Note that this code can also load Abstract Format changing the call to function `compile:file/2` in the the last expression to a call to function `epp:parse_file/3`.

Get the Core Erlang from an external module

Sometimes we need to load Core Erlang from a module included in the OTP libraries. The following code allow us to get this:

core_from_module.erl

Note that this code can also load Abstract Format changing the call to function `compile:file/2` in the the last expression to a call to function `epp:parse_file/3`.

Retrieving Erlang Code from Core Erlang

It is not easy to return from Core Erlang to Erlang. There are different problems:

- The language is different, and some Core constructions are not in Erlang and vice versa.
- Providing that the expression that we want to retrieve could be retrieved, there is still a problem. Its line is the only way to identify an expression as well as its type. Therefore, if two expression of the same type share the same line is impossible to know which is the one that we want to retrieve. For example, suppose we have this expression in one line of a Erlang module `{{1,2},{3,4}}`. During the Core processing, we want to retrieve a tuple from this line. We will have three possibilities.

Retrieving Erlang Code from Core Erlang

The problem can be seen using this module:

core_to_abstract_bad.erl

with this module as test:

core_to_abstract_test.erl

Retrieving Erlang Code from Core Erlang

We can build a parse transformation to give to each individual expression a unique line, and then compile to Core from the resulting forms. The module implementing this transformation is:

`line_changer_pt.erl`

We can now modify the previous module to retrieve correctly the Erlang expression for a given Core Erlang:

`core_to_abstract.erl`

Libraries

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.

Libraries

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

Abstract format

- It is easy to manipulate and understand.
- We have a lot of libraries to ease our work.
- We can evaluate its forms in two different ways.

Core Erlang

- It is suitable for program analysis.
- It is simpler that Abstract Format but it stills human-readable.
- There exists a lot of libraries helping to work with it.
- It is possible to retrieve the corresponding Erlang expressions using some tricky methods.

Thanks for listening

ERLANG