NAME

ms_transform − A parse transformation that translates fun syntax into match specifications.

DESCRIPTION

This module provides the parse transformation that makes calls to *ets* and *dbg:fun2ms/1* translate into literal match specifications. It also provides the back end for the same functions when called from the Erlang shell.

The translation from funs to match specifications is accessed through the two "pseudo functions" *ets:fun2ms/1* and *dbg:fun2ms/1*.

As everyone trying to use *ets:select/2* or *dbg* seems to end up reading this manual page, this description is an introduction to the concept of match specifications.

Read the whole manual page if it is the first time you are using the transformations.

Match specifications are used more or less as filters. They resemble usual Erlang matching in a list comprehension or in a fun used with *lists:foldl/3*, and so on. However, the syntax of pure match specifications is awkward, as they are made up purely by Erlang terms, and the language has no syntax to make the match specifications more readable.

As the execution and structure of the match specifications are like that of a fun, it is more straightforward to write it using the familiar fun syntax and to have that translated into a match specification automatically. A real fun is clearly more powerful than the match specifications allow, but bearing the match specifications in mind, and what they can do, it is still more convenient to write it all as a fun. This module contains the code that translates the fun syntax into match specification terms.

EXAMPLE 1

Using *ets:select/2* and a match specification, one can filter out rows of a table and construct a list of tuples containing relevant parts of the data in these rows. One can use *ets:foldl/3* instead, but the *ets:select/2* call is far more efficient. Without the translation provided by *ms_transform*, one must struggle with writing match specifications terms to accommodate this.

Consider a simple table of employees:

```
-record(emp, {empno,     %Employee number as a string, the key
              surname,   %Surname of the employee
              givenname, %Given name of employee
              dept,      %Department, one of {dev,sales,prod,adm}
              empyear}). %Year the employee was employed
```

We create the table using:

```
ets:new(emp_tab, [{keypos,#emp.empno},named_table,ordered_set]).
```

We fill the table with randomly chosen data:

```
[{emp,"011103","Black","Alfred",sales,2000},
 {emp,"041231","Doe","John",prod,2001},
 {emp,"052341","Smith","John",dev,1997},
 {emp,"076324","Smith","Ella",sales,1995},
 {emp,"122334","Weston","Anna",prod,2002},
 {emp,"535216","Chalker","Samuel",adm,1998},
 {emp,"789789","Harrysson","Joe",adm,1996},
 {emp,"963721","Scott","Juliana",dev,2003},
 {emp,"989891","Brown","Gabriel",prod,1999}]
```

Assuming that we want the employee numbers of everyone in the sales department, there are several ways.

ets:match/2 can be used:

```
1> ets:match(emp_tab, {'_', '$1', '_', '_', sales, '_'}).
[["011103"],["076324"]]
```

ets:match/2 uses a simpler type of match specification, but it is still unreadable, and one has little control over the returned result. It is always a list of lists.

ets:foldl/3 or *ets:foldr/3* can be used to avoid the nested lists:

```
ets:foldr(fun(#emp{empno = E, dept = sales},Acc) -> [E | Acc];
             (_,Acc) -> Acc
          end,
          [],
          emp_tab).
```

The result is *["011103","076324"]*. The fun is straightforward, so the only problem is that all the data from the table must be transferred from the table to the calling process for filtering. That is inefficient compared to the *ets:match/2* call where the filtering can be done "inside" the emulator and only the result is transferred to the process.

Consider a "pure" *ets:select/2* call that does what *ets:foldr* does:

```
ets:select(emp_tab, [{#emp{empno = '$1', dept = sales, _='_'},[],['$1']}]).
```

Although the record syntax is used, it is still hard to read and even harder to write. The first element of the tuple, *#emp{empno = '$1', dept = sales, _='_'}*, tells what to match. Elements not matching this are not returned, as in the *ets:match/2* example. The second element, the empty list, is a list of guard expressions, which we do not need. The third element is the list of expressions constructing the return value (in ETS this is almost always a list containing one single term). In our case *'$1'* is bound to the employee number in the head (first element of the tuple), and hence the employee number is returned. The result is *["011103","076324"]*, as in the *ets:foldr/3* example, but the result is retrieved much more efficiently in terms of execution speed and memory consumption.

Using *ets:fun2ms/1*, we can combine the ease of use of the *ets:foldr/3* and the efficiency of the pure *ets:select/2* example:

```
-include_lib("stdlib/include/ms_transform.hrl").

ets:select(emp_tab, ets:fun2ms(
                      fun(#emp{empno = E, dept = sales}) ->
                            E
                      end)).
```

This example requires no special knowledge of match specifications to understand. The head of the fun matches what you want to filter out and the body returns what you want returned. As long as the fun can be kept within the limits of the match specifications, there is no need to transfer all table data to the process for filtering as in the *ets:foldr/3* example. It is easier to read than the *ets:foldr/3* example, as the select call in itself discards anything that does not match, while the fun of the *ets:foldr/3* call needs to handle both the elements matching and the ones not matching.

In the *ets:fun2ms/1* example above, it is needed to include *ms_transform.hrl* in the source code, as this is what triggers the parse transformation of the *ets:fun2ms/1* call to a valid match specification. This also implies that the transformation is done at compile time (except when called from the shell) and therefore takes no resources in runtime. That is, although you use the more intuitive fun syntax, it gets as efficient in runtime as writing match specifications by hand.

EXAMPLE 2

Assume that we want to get all the employee numbers of employees hired before year 2000. Using *ets:match/2* is not an alternative here, as relational operators cannot be expressed there. Once again, *ets:foldr/3* can do it (slowly, but correct):

```
ets:foldr(fun(#emp{empno = E, empyear = Y},Acc) when Y < 2000 -> [E | Acc];
             (_,Acc) -> Acc
          end,
          [],
          emp_tab).
```

The result is *["052341","076324","535216","789789","989891"]*, as expected. The equivalent expression using a handwritten match specification would look like this:

```
ets:select(emp_tab, [{#emp{empno = '$1', empyear = '$2', _='_'},
                      [{'<', '$2', 2000}],
                      ['$1']}]).
```

This gives the same result. *[{'<', '$2', 2000}]* is in the guard part and therefore discards anything that does not have an *empyear* (bound to *'$2'* in the head) less than 2000, as the guard in the *foldr/3* example.

We write it using *ets:fun2ms/1*:

```
-include_lib("stdlib/include/ms_transform.hrl").

ets:select(emp_tab, ets:fun2ms(
                fun(#emp{empno = E, empyear = Y}) when Y < 2000 ->
                    E
                end)).
```

EXAMPLE 3

Assume that we want the whole object matching instead of only one element. One alternative is to assign a variable to every part of the record and build it up once again in the body of the fun, but the following is easier:

```
ets:select(emp_tab, ets:fun2ms(
                fun(Obj = #emp{empno = E, empyear = Y})
                  when Y < 2000 ->
                      Obj
                end)).
```

As in ordinary Erlang matching, you can bind a variable to the whole matched object using a "match inside the match", that is, a =. Unfortunately in funs translated to match specifications, it is allowed only at the "top-level", that is, matching the *whole* object arriving to be matched into a separate variable. If you are used to writing match specifications by hand, we mention that variable A is simply translated into '$_'. Alternatively, pseudo function *object/0* also returns the whole matched object, see section **Warnings and Restrictions**.

EXAMPLE 4

This example concerns the body of the fun. Assume that all employee numbers beginning with zero (*0*) must be changed to begin with one (*1*) instead, and that we want to create the list *[{<Old empno>,<New empno>}]*:

```
ets:select(emp_tab, ets:fun2ms(
                fun(#emp{empno = [$0 | Rest] }) ->
                    {[$0|Rest],[$1|Rest]}
                end)).
```

This query hits the feature of partially bound keys in table type *ordered_set*, so that not the whole table needs to be searched, only the part containing keys beginning with *0* is looked into.

EXAMPLE 5

The fun can have many clauses. Assume that we want to do the following:

* If an employee started before 1997, return the tuple *{inventory, <employee number>}*.
* If an employee started 1997 or later, but before 2001, return *{rookie, <employee number>}*.
* For all other employees, return *{newbie, <employee number>}*, except for those named *Smith* as they would be affronted by anything other than the tag *guru* and that is also what is returned for their numbers: *{guru, <employee number>}*.

This is accomplished as follows:

```
ets:select(emp_tab, ets:fun2ms(
                fun(#emp{empno = E, surname = "Smith" }) ->
                    {guru,E};
                  (#emp{empno = E, empyear = Y}) when Y < 1997  ->
                    {inventory, E};
                  (#emp{empno = E, empyear = Y}) when Y > 2001  ->
                    {newbie, E};
                  (#emp{empno = E, empyear = Y}) -> % 1997 -- 2001
                    {rookie, E}
                end)).
```

The result is as follows:

```
[{rookie,"011103"},
 {rookie,"041231"},
 {guru,"052341"},
 {guru,"076324"},
 {newbie,"122334"},
 {rookie,"535216"},
 {inventory,"789789"},
 {newbie,"963721"},
 {rookie,"989891"}]
```

USEFUL BIFS

What more can you do? A simple answer is: see the documentation of **match specifications** in ERTS User's Guide. However, the following is a brief overview of the most useful "built-in functions" that you can use when the fun is to be translated into a match specification by *ets:fun2ms/1*. It is not possible to call other functions than those allowed in match specifications. No "usual" Erlang code can be executed by the fun that is translated by *ets:fun2ms/1*. The fun is limited exactly to the power of the match specifications, which is unfortunate, but the price one must pay for the execution speed of *ets:select/2* compared to *ets:foldl/foldr*.

The head of the fun is a head matching (or mismatching) *one* parameter, one object of the table we select from. The object is always a single variable (can be _) or a tuple, as ETS, Dets, and Mnesia tables include that. The match specification returned by *ets:fun2ms/1* can be used with *dets:select/2* and *mnesia:select/2*, and with *ets:select/2*. The use of = in the head is allowed (and encouraged) at the top-level.

The guard section can contain any guard expression of Erlang. The following is a list of BIFs and expressions:

* Type tests: *is_atom*, *is_float*, *is_integer*, *is_list*, *is_number*, *is_pid*, *is_port*, *is_reference*, *is_tuple*, *is_binary*, *is_function*, *is_record*
* Boolean operators: *not*, *and*, *or*, *andalso*, *orelse*
* Relational operators: >, >=, <, =<, =:=, ==, =/=, /=
* Arithmetics: +, -, *, *div*, *rem*
* Bitwise operators: *band*, *bor*, *bxor*, *bnot*, *bsl*, *bsr*
* The guard BIFs: *abs*, *element*, *hd*, *length*, *node*, *round*, *size*, *tl*, *trunc*, *self*

Contrary to the fact with "handwritten" match specifications, the *is_record* guard works as in ordinary Erlang code.

Semicolons (;) in guards are allowed, the result is (as expected) one "match specification clause" for each semicolon-separated part of the guard. The semantics is identical to the Erlang semantics.

The body of the fun is used to construct the resulting value. When selecting from tables, one usually construct a suiting term here, using ordinary Erlang term construction, like tuple parentheses, list brackets, and variables matched out in the head, possibly with the occasional constant. Whatever expressions are allowed in guards are also allowed here, but no special functions exist except *object* and *bindings* (see further down), which returns the whole matched object and all known variable bindings, respectively.

The *dbg* variants of match specifications have an imperative approach to the match specification body, the ETS dialect has not. The fun body for *ets:fun2ms/1* returns the result without side effects. As matching (=) in the body of the match specifications is not allowed (for performance reasons) the only thing left, more or less, is term construction.

EXAMPLE WITH DBG

This section describes the slightly different match specifications translated by *dbg:fun2ms/1*.

The same reasons for using the parse transformation apply to *dbg*, maybe even more, as filtering using Erlang code is not a good idea when tracing (except afterwards, if you trace to file). The concept is similar to that of *ets:fun2ms/1* except that you usually use it directly from the shell (which can also be done with *ets:fun2ms/1*).

The following is an example module to trace on:

```
-module(toy).

-export([start/1, store/2, retrieve/1]).

start(Args) ->
   toy_table = ets:new(toy_table, Args).

store(Key, Value) ->
   ets:insert(toy_table, {Key,Value}).

retrieve(Key) ->
   [{Key, Value}] = ets:lookup(toy_table, Key),
   Value.
```

During model testing, the first test results in *{badmatch,16}* in *{toy,start,1}*, why?

We suspect the *ets:new/2* call, as we match hard on the return value, but want only the particular *new/2* call with *toy_table* as first parameter. So we start a default tracer on the node:

```
1> dbg:tracer().
{ok,<0.88.0>}
```

We turn on call tracing for all processes, we want to make a pretty restrictive trace pattern, so there is no need to call trace only a few processes (usually it is not):

```
2> dbg:p(all,call).
{ok,[{matched,nonode@nohost,25}]}
```

We specify the filter, we want to view calls that resemble *ets:new(toy_table, <something>)*:

```
3> dbg:tp(ets,new,dbg:fun2ms(fun([toy_table,_]) -> true end)).
{ok,[{matched,nonode@nohost,1},{saved,1}]}
```

As can be seen, the fun used with *dbg:fun2ms/1* takes a single list as parameter instead of a single tuple. The list matches a list of the parameters to the traced function. A single variable can also be used. The body of the fun expresses, in a more imperative way, actions to be taken if the fun head (and the guards) matches. *true* is returned here, only because the body of a fun cannot be empty. The return value is discarded.

The following trace output is received during test:

```
(<0.86.0>) call ets:new(toy_table, [ordered_set])
```

Assume that we have not found the problem yet, and want to see what *ets:new/2* returns. We use a slightly different trace pattern:

```
4> dbg:tp(ets,new,dbg:fun2ms(fun([toy_table,_]) -> return_trace() end)).
```

The following trace output is received during test:

```
(<0.86.0>) call ets:new(toy_table,[ordered_set])
(<0.86.0>) returned from ets:new/2 -> 24
```

The call to *return_trace* results in a trace message when the function returns. It applies only to the specific function call triggering the match specification (and matching the head/guards of the match specification). This is by far the most common call in the body of a *dbg* match specification.

The test now fails with *{badmatch,24}* because the atom *toy_table* does not match the number returned for an unnamed table. So, the problem is found, the table is to be named, and the arguments supplied by the test program do not include *named_table*. We rewrite the start function:

```
start(Args) ->
   toy_table = ets:new(toy_table, [named_table|Args]).
```

With the same tracing turned on, the following trace output is received:

```
(<0.86.0>) call ets:new(toy_table,[named_table,ordered_set])
(<0.86.0>) returned from ets:new/2 -> toy_table
```

Assume that the module now passes all testing and goes into the system. After a while, it is found that table *toy_table* grows while the system is running and that there are many elements with atoms as keys. We expected only integer keys and so does the rest of the system, but clearly not the entire system. We turn on call tracing and try to see calls to the module with an atom as the key:

```
1> dbg:tracer().
{ok,<0.88.0>}
2> dbg:p(all,call).
{ok,[{matched,nonode@nohost,25}]}
3> dbg:tpl(toy,store,dbg:fun2ms(fun([A,_]) when is_atom(A) -> true end)).
{ok,[{matched,nonode@nohost,1},{saved,1}]}
```

We use *dbg:tpl/3* to ensure to catch local calls (assume that the module has grown since the smaller version and we are unsure if this inserting of atoms is not done locally). When in doubt, always use local call tracing.

Assume that nothing happens when tracing in this way. The function is never called with these parameters. We conclude that someone else (some other module) is doing it and realize that we must trace on *ets:insert/2* and want to see the calling function. The calling function can be retrieved using the match specification function *caller*. To get it into the trace message, the match specification function *message* must be used. The filter call looks like this (looking for calls to *ets:insert/2*):

```
4> dbg:tpl(ets,insert,dbg:fun2ms(fun([toy_table,{A,_}]) when is_atom(A) ->
 message(caller())
 end)).
{ok,[{matched,nonode@nohost,1},{saved,2}]}
```

The caller is now displayed in the "additional message" part of the trace output, and the following is displayed after a while:

```
(<0.86.0>) call ets:insert(toy_table,{garbage,can}) ({evil_mod,evil_fun,2})
```

You have realized that function *evil_fun* of the *evil_mod* module, with arity *2*, is causing all this trouble.

This example illustrates the most used calls in match specifications for *dbg*. The other, more esoteric, calls are listed and explained in **Match specifications in Erlang** in ERTS User's Guide, as they are beyond the scope of this description.

WARNINGS AND RESTRICTIONS

The following warnings and restrictions apply to the funs used in with *ets:fun2ms/1* and *dbg:fun2ms/1*.

Warning:

To use the pseudo functions triggering the translation, ensure to include the header file *ms_transform.hrl* in the source code. Failure to do so possibly results in runtime errors rather than compile time, as the expression can be valid as a plain Erlang program without translation.

Warning:

The fun must be literally constructed inside the parameter list to the pseudo functions. The fun cannot be bound to a variable first and then passed to *ets:fun2ms/1* or *dbg:fun2ms/1*. For example, *ets:fun2ms(fun(A) -> A end)* works, but not *F = fun(A) -> A end, ets:fun2ms(F)*. The latter results in a compile-time error if the header is included, otherwise a runtime error.

Many restrictions apply to the fun that is translated into a match specification. To put it simple: you cannot use anything in the fun that you cannot use in a match specification. This means that, among others, the following restrictions apply to the fun itself:

* Functions written in Erlang cannot be called, neither can local functions, global functions, or real funs.
* Everything that is written as a function call is translated into a match specification call to a built-in function, so that the call *is_list(X)* is translated to *{'is_list', '$1'}* (*'$1'* is only an example, the numbering can vary). If one tries to call a function that is not a match specification built-in, it causes an error.
* Variables occurring in the head of the fun are replaced by match specification variables in the order of occurrence, so that fragment *fun({A,B,C})* is replaced by *{'$1', '$2', '$3'}*, and so on.

Every occurrence of such a variable in the match specification is replaced by a match specification variable in the same way, so that the fun *fun({A,B}) when is_atom(A) -> B end* is translated into *[{{'$1','$2'},[{is_atom,'$1'}],['$2']}]*.

* Variables that are not included in the head are imported from the environment and made into match specification *const* expressions. Example from the shell:

```
1> X = 25.
25
2> ets:fun2ms(fun({A,B}) when A > X -> B end).
[{{'$1','$2'},[{'>','$1',{const,25}}],['$2']}]
```

* Matching with = cannot be used in the body. It can only be used on the top-level in the head of the fun. Example from the shell again:

```
1> ets:fun2ms(fun({A,[B|C]} = D) when A > B -> D end).
[{{'$1',['$2'|'$3']},[{'>','$1','$2'}],['$_']}]
2> ets:fun2ms(fun({A,[B|C]=D}) when A > B -> D end).
Error: fun with head matching ('=' in head) cannot be translated into
match_spec
{error,transform_error}
3> ets:fun2ms(fun({A,[B|C]}) when A > B -> D = [B|C], D end).
Error: fun with body matching ('=' in body) is illegal as match_spec
{error,transform_error}
```

All variables are bound in the head of a match specification, so the translator cannot allow multiple bindings. The special case when matching is done on the top-level makes the variable bind to *'$_'* in the resulting match specification. It is to allow a more natural access to the whole matched object. Pseudo function *object()* can be used instead, see below.

The following expressions are translated equally:

```
ets:fun2ms(fun({a,_} = A) -> A end).
ets:fun2ms(fun({a,_}) -> object() end).
```

* The special match specification variables *'$_'* and *'$*'* can be accessed through the pseudo functions *object()* (for *'$_'*) and *bindings()* (for *'$*'*). As an example, one can translate the following *ets:match_object/2* call to a *ets:select/2* call:

```
ets:match_object(Table, {'$1',test,'$2'}).
```

This is the same as:

```
ets:select(Table, ets:fun2ms(fun({A,test,B}) -> object() end)).
```

In this simple case, the former expression is probably preferable in terms of readability.

The *ets:select/2* call conceptually looks like this in the resulting code:

```
ets:select(Table, [{{'$1',test,'$2'},[],['$_']}]).
```

Matching on the top-level of the fun head can be a more natural way to access *'$_'*, see above.

* Term constructions/literals are translated as much as is needed to get them into valid match specification. This way tuples are made into match specification tuple constructions (a one element tuple containing the tuple) and constant expressions are used when importing variables from the environment. Records are also translated into plain tuple constructions, calls to element, and so on. The guard test *is_record/2* is translated into match specification code using the three parameter version that is built into match specification, so that *is_record(A,t)* is translated into *{is_record,'$1',t,5}* if the record size of record type *t* is 5.
* Language constructions such as *case*, *if*, and *catch* that are not present in match specifications are not allowed.

* If header file *ms_transform.hrl* is not included, the fun is not translated, which can result in a *run-time error* (depending on whether the fun is valid in a pure Erlang context).

  Ensure that the header is included when using *ets* and *dbg:fun2ms/1* in compiled code.

* If pseudo function triggering the translation is *ets:fun2ms/1*, the head of the fun must contain a single variable or a single tuple. If the pseudo function is *dbg:fun2ms/1*, the head of the fun must contain a single variable or a single list.

The translation from funs to match specifications is done at compile time, so runtime performance is not affected by using these pseudo functions.

For more information about match specifications, see the **Match specifications in Erlang** in ERTS User's Guide.

EXPORTS

format_error(Error) -> Chars

Types:

Error = {error, module(), term()}
Chars = **io_lib:chars()**

Takes an error code returned by one of the other functions in the module and creates a textual description of the error.

parse_transform(Forms, Options) -> Forms2

Types:

Forms = Forms2 = [**erl_parse:abstract_form()** | **erl_parse:form_info()**]
Options = term()
Option list, required but not used.

Implements the transformation at compile time. This function is called by the compiler to do the source code transformation if and when header file *ms_transform.hrl* is included in the source code.

For information about how to use this parse transformation, see *ets* and *dbg:fun2ms/1*.

For a description of match specifications, see section **Match Specification in Erlang** in ERTS User's Guide.

transform_from_shell(Dialect, Clauses, BoundEnvironment) -> term()

Types:

Dialect = ets | dbg
Clauses = [**erl_parse:abstract_clause()**]
BoundEnvironment = **erl_eval:binding_struct()**
List of variable bindings in the shell environment.

Implements the transformation when the *fun2ms/1* functions are called from the shell. In this case, the abstract form is for one single fun (parsed by the Erlang shell). All imported variables are to be in the key-value list passed as *BoundEnvironment*. The result is a term, normalized, that is, not in abstract format.