CAA

03 November 2020 14:39

Erlang abstract format

https://erlang.org/doc/apps/erts/absform.html

Things in my mind:

- anything outside, en let's just focus on "receives blocks"
  works sequentially.
- so let's first focus at receive block and send term, abstract format

send term !!

- Example + abstract format of send term:

```
send(X) ->
    X!s.
```

⇒

```
{function,4,send,1,
        [{clause,4,
                 [{var,4,'X'}],
                 [],
                 [{op,5,'!',{var,5,'X'},{atom,5,s}}]}]}
ok
```

Here we can see that a send "op" is represented through an operator expression, where inside the tuple:

1st ⇒ op ⇒ just an atom, saying here its an operation

2nd ⇒ ANNO

3rd ⇒ "OP" variable represent the binary operator (taking two argument) "!"

except "=" (coz this is what as unary operator and the representation is a bit different)

4th ⇒ the left hand side "var" of "Op"

5th ⇒ the right hand side "var" of "OP"

- If E is an operator expression `E_1 Op E_2`, where `Op` is a binary operator other than match operator =, then Rep(E) = `{Op,ANNO,Op,Rep(E_1),Rep(E_2)}`.
- If E is an operator expression `Op E_0`, where `Op` is a unary operator, then Rep(E) = `{Op,ANNO,Op,Rep(E_0)}`.

Receive block

+ Expressions:

- If E is a receive expression `receive Cc_1 ; ... ; Cc_k end`, where each `Cc_i` is a case clause, then Rep(E) = `{'receive',ANNO,[Rep(Cc_1), ..., Rep(Cc_k)]}`.
- If E is a receive expression `receive Cc_1 ; ... ; Cc_k after E_0 -> B_t end`, where each `Cc_i` is a case clause, `E_0` is an expression, and `B_t` is a body, then Rep(E) = `{'receive',ANNO,[Rep(Cc_1), ..., Rep(Cc_k)],Rep(E_0),Rep(B_t)}`.

example

```
rec() ->
    receive
        hello -> hi;
        hyd -> good
end.
```

⇒

```
{function,4,rec,0,
    [{clause,4,[],[],
        [{'receive',5,
            [{clause,6,[{atom,6,hello}],[],[{atom,6,hi}]},
             {clause,7,[{atom,7,hyd}],[],[{atom,7,good}]}]}]}]}
ok
```

```
rec2() ->
    receive
        hello -> hi;
        hyd -> good
    after
        500 -> bye
end.
```

⇒

```
{function,12,rec2,0,
    [{clause,12,[],[],
        [{'receive',13,
            [{clause,14,[{atom,14,hello}],[],[{atom,14,hi}]},
             {clause,15,[{atom,15,hyd}],[],[{atom,15,good}]}],
            {integer,17,500},
            [{atom,17,bye}]}]}]}
ok
```

} 3/receive1.erl

- So by analysing these examples, we can see that inside a receive block there can be only two types of expressions (as a defined unit by themselves) 'clauses' and a 'after' block (only one after block is allowed).

} otherwise it will throw an error. can be seen in 3/receive1.erl.

And this bit of information is important because:

① if the length of the 'receive' form is more than 3 then we know there is a 'after' block which is defined here: {'receive', —, —, expression, body}

and if it is more than 3, then in the CAM/delta function we can ....

} How we going to represent that in the CAM/delta function?

- And we're not going to represent a recursive call which is made inside the receive block in the CAM/delta function. Because in dominic's paper, first we are not representing them and it's because we can easily identify it through to which state we are transitioning to.

} what about call to other function? should we show that as a transition to other state (that's all?), what if there is a call to this function from that

Example 2.2. Consider the following Erlang code:

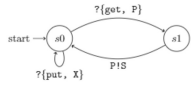
```
mem(S) ->
  receive
    {get, P} -> P!S, mem(S);
    {put, X} -> mem(X)
  end.
```

This can be modelled as an eCFSM by the states $S = \{s_0, s_1\}$, initial state $o = s_0$, final states $F = \emptyset$, send labels $\mathcal{L}_t = Terms$ and receive labels $\mathcal{L}_r =$

{{get, P}, {put, X}} and transitions:

$$\begin{aligned} \delta(s_0, ?\{\texttt{get, P}\}) &= s_1 \\ \delta(s_0, ?\{\texttt{put, P}\}) &= s_0 \\ \delta(s_1, P!S) &= s_0 \end{aligned}$$

which can represented by the automata diagram:



* And the Abstract format for this example would be something like:

```
rec(S) ->
    receive
        {get, X} -> X!S, rec(S);
        {put, X} -> rec(X)
end.
```

⇒

```
{function,5,rec,1,
    [{clause,5,
        [{var,5,'S'}],
        [],
        [{'receive',6,
            [{clause,7,
                [{tuple,7,[{atom,7,get},{var,7,'X'}]}],
                [],
                [{op,7,'!',{var,7,'X'},{var,7,'S'}},
                 {call,7,{atom,7,rec},[{var,7,'S'}]}]},
            {clause,8,
                [{tuple,8,[{atom,8,put},{var,8,'X'}]}],
                [],
                [{call,8,{atom,8,rec},[{var,8,'X'}]}]}]}]}]}
```

as we can see the call to the function (here its recursive) is represented as a tuple with four things in it:

- 1st → 'call' atom to represent its a call
- 2nd → annotation ANNO
- 3rd → it contains the information about the method name
- 4th → a list representing the number of parameter this method call is taking.

*expression:

- If E is a function call `E_0(E_1, ..., E_k)`, then Rep(E) = `{call,ANNO,Rep(E_0),[Rep(E_1), ..., Rep(E_k)]}`.
- If E is a function call `E_m:E_0(E_1, ..., E_k)`, then Rep(E) = `{call,ANNO,{remote,ANNO,Rep(E_m),Rep(E_0)},[Rep(E_1), ..., Rep(E_k)]}`.

* So to check if its a recursive call we can, if the third element of the java representation tuple is a tuple like,

{atom, _, Method_Name}
↑ this. method

And, if does the length of the 4th element matches the number of parameters (arguments) this method_name takes?

} i don't think we can do pattern matching here since the argument name may change here than the name it has above.