Erlang structs

Joe Armstrong
SICS
joe@sics.se

11 - December 2001

Abstract

This note describes a new data type (the *struct*) and how it could be added to Erlang.

Structs are designed to be fully backwards compatible with the current and all previous versions of Erlang.

To test some of these ideas a trial implementation of structs has been made.

1 Structs

We propose a new internal Erlang data structure called *structs*. Structs are similar to Erlang records, but have the important difference that the fields in the structs are dynamic and do not have to be declared in advance, as is the case for records. There are two types of structs, named structs and anonymous structs, here are some examples of both types of structs:

```
~person{name="Claire", age=11}
~{width=80, height=10}
```

The ~ character introduces the struct. The general syntax of a named struct is:

```
~Name{Key1 = E1, Key2 = E2, ..., KeyN = EN}
```

Where `Name` and `Key1 ... KeyN` are Erlang atoms and `E1 ... EN` are Erlang expressions. An anonymous struct has the same syntax, only the name of the structure is missing. The following build-in function are defined on the struct `S`.

- `erlang:arity(S) -> [Key1, Key2, ... KeyN]` returns the keys in `S`.
- `erlang:name(S) -> {name, Name} | anon` can be used to find if the struct `S` is a named or anonymous structure.
- `erlang:fetch_key(S, K) -> Value | EXIT` extracts the value of the key `K`, if the key is not present an exception is generated.
- `erlang:find_key(S, K) -> {ok, Value} | error` searches for the key `K`. It returns the value associated with the key or an error.

- `erlang:delete_key(S, K) -> S1` deletes the key `K` from a struct returning a new struct `S1`.

In all cases if `S` is not a struct an exception is generated. One additional primitive is defined to test if an expression is a struct:

- `erlang:is_struct(X) -> true | false` is `true` if `X` is a struct otherwise `false`

We extend Erlang pattern matching to include structs. We say that the struct pattern `P` matches the struct `S` if the following conditions are true:

- If `S` is a named struct and `P` is a named struct then both structs must have the same name, and all the keys given in `P` must match.
- If `S` is a named struct and `P` is a anonymous struct then all the keys given in `P` must match.
- Forall keys `K` in `P` the key `K` must exist in `S` and the associated value in `S` must match the pattern for this key given in `P`.

Structs can be thought of as "anonymous records". For reasons of backwards compatibility structs are introduced by the ~ character.

Structs are inspired by records in Oz[1] and by previous work on the Erlang 5.0 standard[**?**].

1.1 Examples of structs

Here are some examples[1] of code which uses structs:

```
 1  -module(small).
 2  -include("structs").
 3  -compile(export_all).
 4
 5  test() ->
 6      V  = ~{age=11, name="Claire"},
 7      V1 = give_birthday_present(V, "cat"),
 8      ~{present="cat", name="Claire", age=12} == V1,
 9      erlang:arity(V1).
10
11  give_birthday_present(V=~{age=A}, Thing) ->
12      V~{age=A+1, present=Thing}.
```

The statement `V = ~{age=11, name="Claire"}` creates an anonymous struct, with two fields called `age` and `name`. In line 11 the value of the `age` field in the input struct is bound to the variable `A` and the entire struct itself is bound to the variable `V`. Line 12 creates a new struct from `V` overwriting the `age` field with a new value, and dynamically adding a new field called `present` whose value is `Thing`.

Line 8 is a pattern matching operation to verify that all the resultant structure is correct. Finally the statement `erlang:arity(V1)` returns the field names contained in the struct `V1`.

We can compile and run this in the usual way:

[1]This example assume the prototype implementation of structs described later in this chapter.

```
1> c(small).
{ok,small}
2> small:test().
[present,age,name]
```

The next example shows makes use of a named struct and shows how to access the fields of the struct.

```
-module(demo).

-include("structs").
-compile(export_all).
-import(lists, [foreach/2]).

test() ->
  X = ~person{name=claire, age=12},
  pp(X).

pp(S) ->
  Name = erlang:name(S),
  io:format("<~s>~n",[Name]),
  Arity = erlang:arity(S),
  foreach(fun(Key) ->
            io:format("<~s>",[Key]),
            io:format("~p",
                      [erlang:fetch_field(S,Key)]),
            io:format("</~s>~n",[Key])
          end, Arity),
  io:format("</~s>~n",[Name]).
```

Running this yields:

```
b> c(demo).
{ok,demo}
> demo:test().
<person>
<name>claire</name>
<age>12</age>
</person>
```

This example nicely illustrates the correspondance between Erlang structs and XML terms - indeed we observe in passing that translating XML terms into Erlang structs and manipulating them through function calls involving compiled stuct patterns provides us with an extemely efficient “XML processor”.

1.2 Advantages and Disadvantages of structs

Structs are fully dynamic declarative data structures - that fit in very nicely with eh Erlang programming model. This type of data structure is “self-describing” to the extent that the field names of the different elements in the struct are often sufficient to provide adequate documentation of the meaning of the element in question.

For small fixed data structures, Erlang tuples are adequate - but as the number of elements in a tuple increases it becomes increasingly more difficult to remember which element means what, and modifications to the program which require the addition of subtraction pf elements in the tuple is a tedious and error-prone task.

Structures are very similar to Erlang records - the difference being that the file names are dynamic - new fields can be added or removed at run-time. Use of structs is slightly less efficient than that of records, but more flexible - the efficiency of pattern matching on structs should be the same as the efficiency of matching regular function head clauses.

The advantages of struct (compared to the current implementation) are:

- No include (.hrl) files.
- No record definitions.
- Can write generic methods on structs.
- Structs are ”self-describing”

Disadvantages

These are disadvantages compared to the current record implementation.

- Mis-spelling of field names can cause problems.
- Size overhead.
- CPU overhead.
- You cannot have default fields
- No way of cross checking consistency between modules (using the typed property of records)

2 Prototyping structs

A simple[2] prototype implementation was made using a single parse transform. The transformation necessary are described below:

Firstly, observe that struct patterns occur in three different contexts:

- In function heads.
- In `case` alternatives.
- In equalities

We give examples of how each of these can be transformed:

[2]not that simple :-)

2.0.1 Structs in function heads

The schema used for translating structs pattern matching in function heads is illustrated in the following example:

```
1 foo(~person{name=N, arg=12}, {g,a,12}) -> Rhs1
2 foo(xxx, yyy) -> Rhs2
```

Is translated into:

```
1 foo(F1, F2={g,a,12}) ->
2     case match_struct(F1, person, [name, arg]) of
3        {ok, [N, 12]} -> Rhs1;
4        _ -> foo_1(F1, F2)
5     end;
6 foo(F1, F2) -> foo_1(F1, F2).
7
8 foo_1(xxx, yyy) -> Rhs2.
```

In this, and all subsequent examples, variables of the form `Fn` are assumed to be new variables, which are not mentioned elsewhere in the body of the function.

In line 1 the new struct in argument one is replaced by a free variable `F1` and a copy of the the tuple in argument two is kept in the variable `F2`, this is to avoid re-building the term `{g,a,12}` in the call to the added function `foo_1/2` if the struct match fails. Here is a second example:

```
bar({cat, B} , ~{age=N, arg={a,B}},  {dog,N}) ->
bar(...)
```

Is transformed into:

```


bar(F1={cat, B}, F2, F3={dog, N}) ->
    case match_struct(F2, [age, arg]) of
        {ok, [N, {a,B}]} ->
            Rhs;
        _ ->
            bar_1(F1, F2, F3)
    end;
bar(F1, F2, F3) ->
        bar_1(F1, F2, F3).

bar_1(F1, F2, F3) ->
  %% the second clause (possibly transformed)
  %% of bar/3
```

Nested structs, and guard tests are also illustrated:

```
boo({a,C}, ~{a=1,b=B,c=~{d=B,e=1}}) when integer(B)->
```

Is transformed into

```
boo(F1={a,C}, F2) ->
    case match_struct(F2, [a,b,c]) of
        {ok, [1,B,F3]} ->
            case match_struct(F3, [d,e]) of
                {ok, } ->
                    %% Rhs guard
                    if integer(B) ->
                            Rhs;
                       true ->
                            fail
                    end;
                 ->
        ... etc ...
```

Embedded structs must also be handled, thus:

```
foo(abc, {a,b,~joe{a=K}}) -> Rhs1
foo(def, ho) -> Rhs2
```

Is transformed into

```
foo(F1 = abc, F2={a,b,F3}) ->
   case match_struct(F3, joe, [a]) of
        {ok, [K]} ->
            Rhs1;
        _ ->
            foo_1(F1, F2)
   end;
foo(F1, F2) ->
   foo_1(F1, F2)
```

2.0.2 Structs in case alternatives

Structs that occur in case alternative are trnasfored as in the following example:

```
case Var of
    ~{name=joe, age=A} ->
        Rhs1;
    Lhs2 ->
        Rhs2
end
```

becomes

```
case match_struct(Var, anon, [name,age]) of
    {ok, [joe,A]} ->
        Rhs1;
    _ ->
        case Var of
            Lhs2 ->
                Rhs2
        end
end.
```

When the subject of the `case` statement involves a function call the translation is as in the following example:

```
case f(X) of
    ~{name=joe, age=A} ->
        Rhs1;
    Lhs2 ->
        Rhs2
end
```

becomes

```
case match_struct(Var = f(X), anon, [name,age]) of
    {ok, [joe,A]} ->
        Rhs1;
    _ ->
        case Var of
            Lhs2 ->
                Rhs2
        end
end.
```

Where the `Var` introduced in is a new variable.

2.0.3 Structs in equality

Structs in equality are transformed as in the following example:

```
~{name=N, age=A} = Y
```

becomes

```
{ok, [N,A]} = match:struct(Y, [name, age])
```

2.1 Constructing a struct

```
    X = person~{name=joe, age=12}.
```

becomes

```
    mk_struct(person, [..]).
```

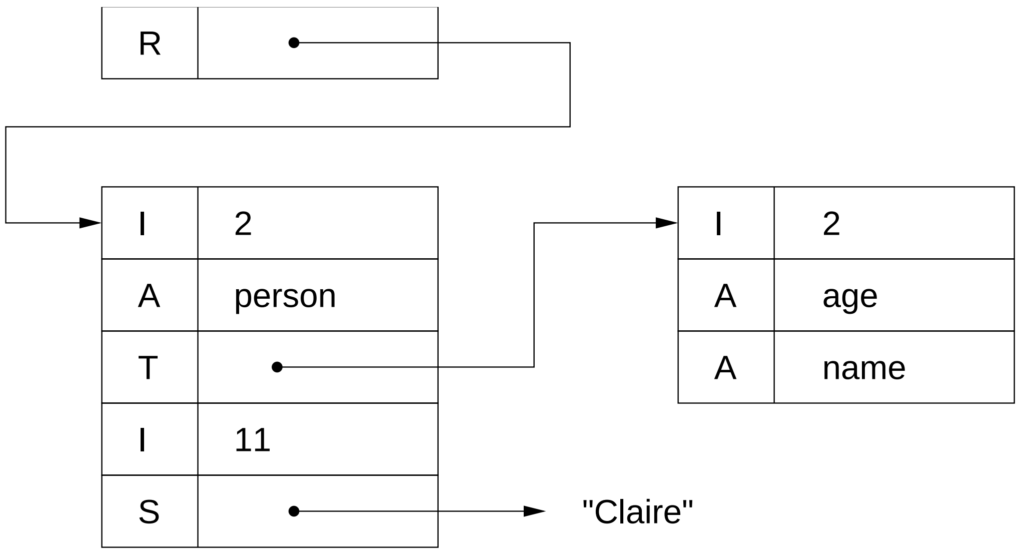

Figure 1: An example struct.

3 Compiling Structs

We assume the internal representation of struct as shown in Figure 1.

The `R` tag is a new tag field which points to a struct data type on the heap.

The heap representation of a struct starts with an atom containing the name of the struct, and is followed by an alternating sequence of atom-value pairs of words.

We also define the constructor `~{Tag=...}` which creates an ”anonymous” struct whose struct name is `[]`.

Assuming a simple JAM like instruction set the operations on struct can be compiled as follows:

3.0.1 Pattern matching

```
foo(~person{age=A, name=N}) ->
   ...

tryMeElse, Label
  getStruct Arg1, person
  getKey age,A
  getKey name,N

foo(~{age=A, name=N}) ->
  ...

tryMeElse, Label
   getAnonStruct, Arg1
   getKey age,A
   getKey name, N
```

3.0.2 Struct creation

We create a struct as follows:

```
X = ˜person{name="Claire", age=11}

pushAtom, name
pushString, "Claire"
pushAtom, age
pushInt, 11
pushAtom, person
mkStruct,2
```

Here we just push all the arguments of the struct onto the stack and call `mkStruct,2` the second argument is the number of fields in the struct.

3.0.3 Struct update

```
X1 = X˜{age=12}

pushAtom, age
pushInt, 12
updateStruct X, 1

X1 = X˜person{age=12}

pushAtom, age
pushInt, 12
pushAtom, person
updateNamedStruct X, 1
```

3.1 Primitives

- `arity(X) -> [Key1, Key2, ..]` - return the key set of the struct `X`.
- `name(X) -> Atom` - return the name of the current struct.

3.2 Implementation

An implementation of the struct type described above needs the following changes to the system.

- Parser ...
- Compiler. Change compiler for structs. While the current document describes structs in terms of an old (Jam Like :-) instruction set I'm not sure how this would look in the current compiler. I assuming some kind of PJ pattern matching optimization is in order.

- Emulator. Seems like adding a new type is easy if there are any unused tag bits, otherwise (ouch ...) - Are their any unused tag bits??
- GC. Easy (I think) they are just like tuples.

3.2.1 PJ like optimizations

The following optimization applies:

```

foo(~person{name=N, age=A, married=B}) ->
  ...
foo(~person{name=N, age=A, job=J}) ->
  ...

tryMeElse, Label1
   getStruct Arg1, person
   getKey name, N
   getkey age, A
   tryMeElse, L2
      getKey married, B
      ...

L2: getKey job, J
```

This is presumably the same optimization as for compiling lists with common prefixes, for example:

```
foo([a,b,c|T]) ->
  ...
foo([a,b,d|T]) ->
  ...
```

References

[1] Mozart Consortium. The Mozart Programming System, January 1999. Available at `http://www.mozart-oz.org/`.