ECT: an Object-Oriented Extension to Erlang

Gábor Fehér
Budapest University of Technology and Economics
feherga@gmail.com

András 'Georgy' Békés
Ericsson Hungary
andras.gyorgy.bekes@ericsson.com

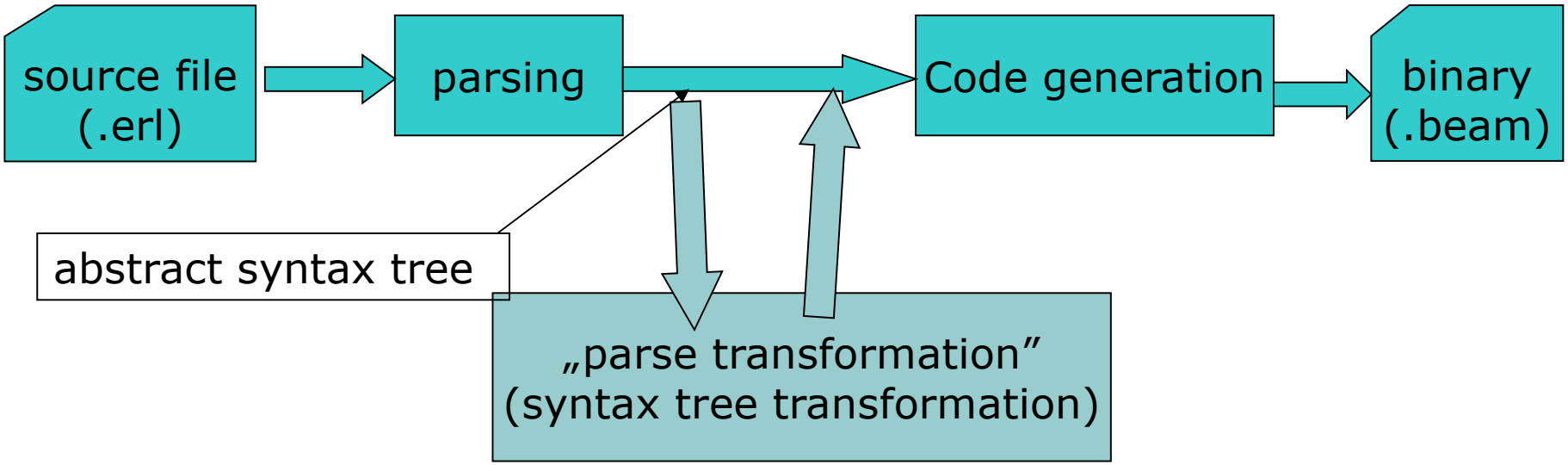
What is ECT?

- ECT = Erlang Class Transformation, an OOP extension to Erlang
- Efficient:
  - O(1) overhead of: method call, field access, dynamic type test (as a function of inheritance chain length)
  - Faster than the alternatives
- Fits better to Erlang than the alternatives:
  - Single-assignment objects
  - Pattern matching on object fields
  - Both object-as-data and object-as-process is supported
- Safe and portable:
  - Compiler or VM not modified
  - Uses parse transformation
- Open source
  - In “beta” state
  - Download from: http://ect.googlecode.com

Erlang and OOP

- (Almost) every advantage of OOP can be used in Erlang
- Using ad-hoc techniques is problematic in the long term
- OOP extensions to Erlang
  - Standardize how a certain OOP feature should be programmed
  - The OOP boilerplate code is written once
  - Just like the `gen_server` behaviour standardize how processes should be programmed
- OOP Features implemented in ECT:
  - Encapsulation of data and operations
  - Inheritance of fields and methods
  - Subtype polymorphism
- Not implemented:
  - Multiple inheritance (debated usefulness)
  - Information hiding (against the spirit of Erlang)

Parse transformation



- Compiles the new language elements into standard Erlang
- Can extend the language only if the new language construct is accepted by the parser
  - Not everything accepted by the parser is valid Erlang (for some reason)

    a parse transform can turn it into valid Erlang
  - Some existing language elements get new meaning

OOP in Erlang

- Object-as-data approach
- Object instance = some data
  - an Erlang term
- Method call = function call
  - The object instance is passed as argument to the method-function
  - If the method modifies the object, a new object (data) is returned
- OOP-extensions using this approach:
  - Parameterized modules
- Pros: faster

- Object-as-process approach
- Object instance = a process
  - The object state is the “loop data”
- Method call = request-response messages
  - If the method modifies the object, the “loop data” is modified
- OOP-extensions using this approach:
  - eXAT, WOOPER
- Pros: multiple processes can use an object

ECT approach

- ECT approach: both
- Write the code once
- An object is object-as-data by default
- You can instantiate object-as-process objects whenever it is more suitable
  - The code for
    - sending method call request-response messages
    - handling the process loop

    is automatically generated
- The “client” code is obviously different
  - Methods of object-as-data objects return the new state of the object
  - Methods of object-as-process objects don't

Defining a class

```
-include_lib("ect/include/ect.hrl").
-class(animal).

?FIELDS({color = red}).
?METHODS([method1/2, method2/2]).

method1(#animal{color=blue}, X) ->
    X;
method1(This, _X) ->
    This#animal.color.

method2(This, A) ->
    This#animal{color=A}.
```

Defining a subclass

```
-include("animal.class.hrl"). %generated when compiling animal.erl
-class(dog).
?SUPERCLASS(animal).

?FIELDS({name}).
?METHODS([method3/2]).

method2(#dog{name=”Bodri”}, A) ->
   whatever();
method2(This, A) ->
   animal:method2(This, A);

method3(#animal{color=Color}, A) ->
   whatever2(Color, A).
```

Using a class (“client code”)

```
-include("dog.class.hrl").
-module(whatever).

f()->
 A = #dog{},

 Color = A#animal.color,
 B = A#animal{color=green}, % B will be an instance of 'dog'

 {A}:method1(42), % animal:method1/2
 {A}:method2(42), % dog:method2/2
 % {Something} is not a valid module name

 {NewA,ReturnValue} = {A}:method_that_modifies_the_object(),
```

Object instances

- An object instance is a record:

  `#dog{name="Bodri"}`
  equals

  `{dog,{animal,dog},red,"Bodri"}`

- The first tuple field defines the module that contains the method-functions
- The second tuple field holds the names of ancestor classes
  - It is used for dynamic type checking:
    - Matching against `#animal{}` will succeed
    - Matching against `#vehicle{}` or `#car{}` will fail
  - Needed for O(1) type check

Method execution

- A method call of the form

  ```
  {Object}:method(Arg1,Arg2,...)
  ```

  is transformed into

  ```
  (element(1,Object)):method(Object,Arg1,Arg2,...)
  ```

- Inherited methods implemented only in a superclass are called via auto-generated proxy-functions:

  ```
  method(This,Arg1,Arg2,...) ->
      SUPERCLASS:method(This,Arg1,Arg2,...).
  ```

  Where `SUPERCLASS` is the latest ancestor class that implements `method/N` (known at compile time).

Pattern matching

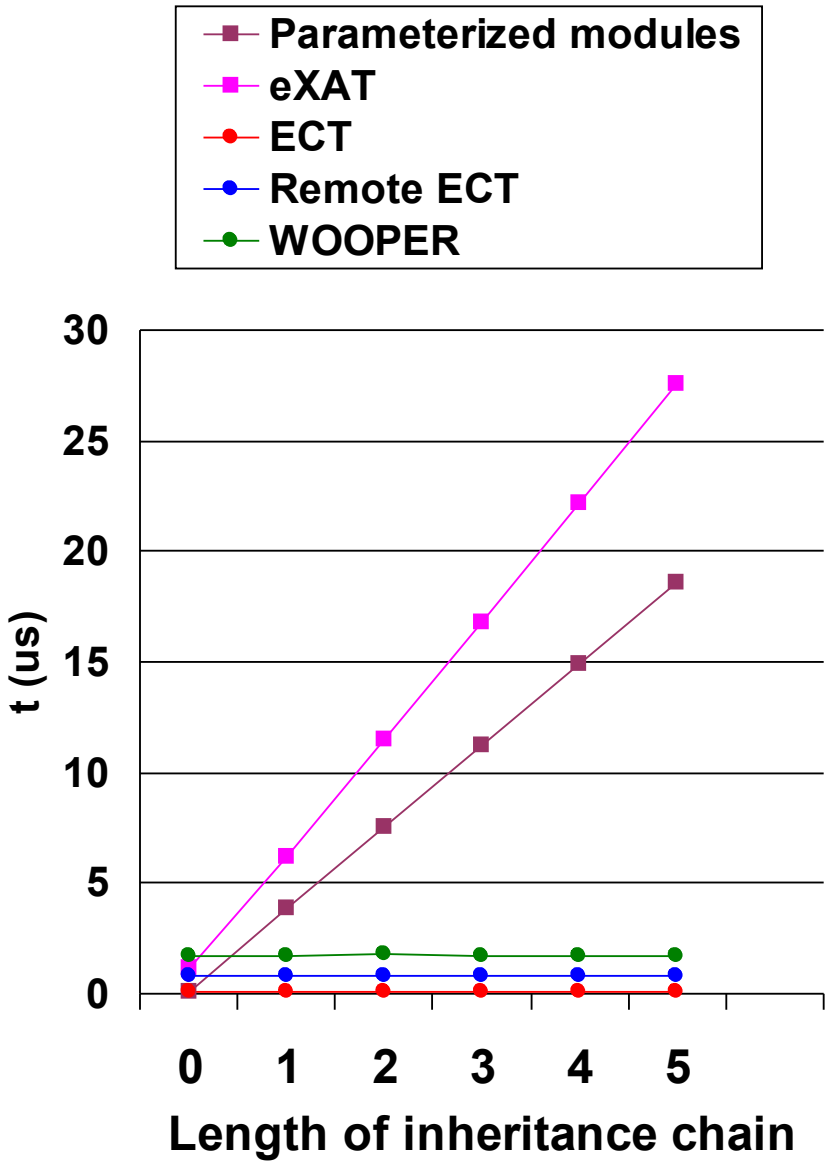
- Class-patterns are translated into a combination of patterns, guards and match expressions. Examples:
- `#dog{color=green, name=Name} = DogExpression`
  is converted into

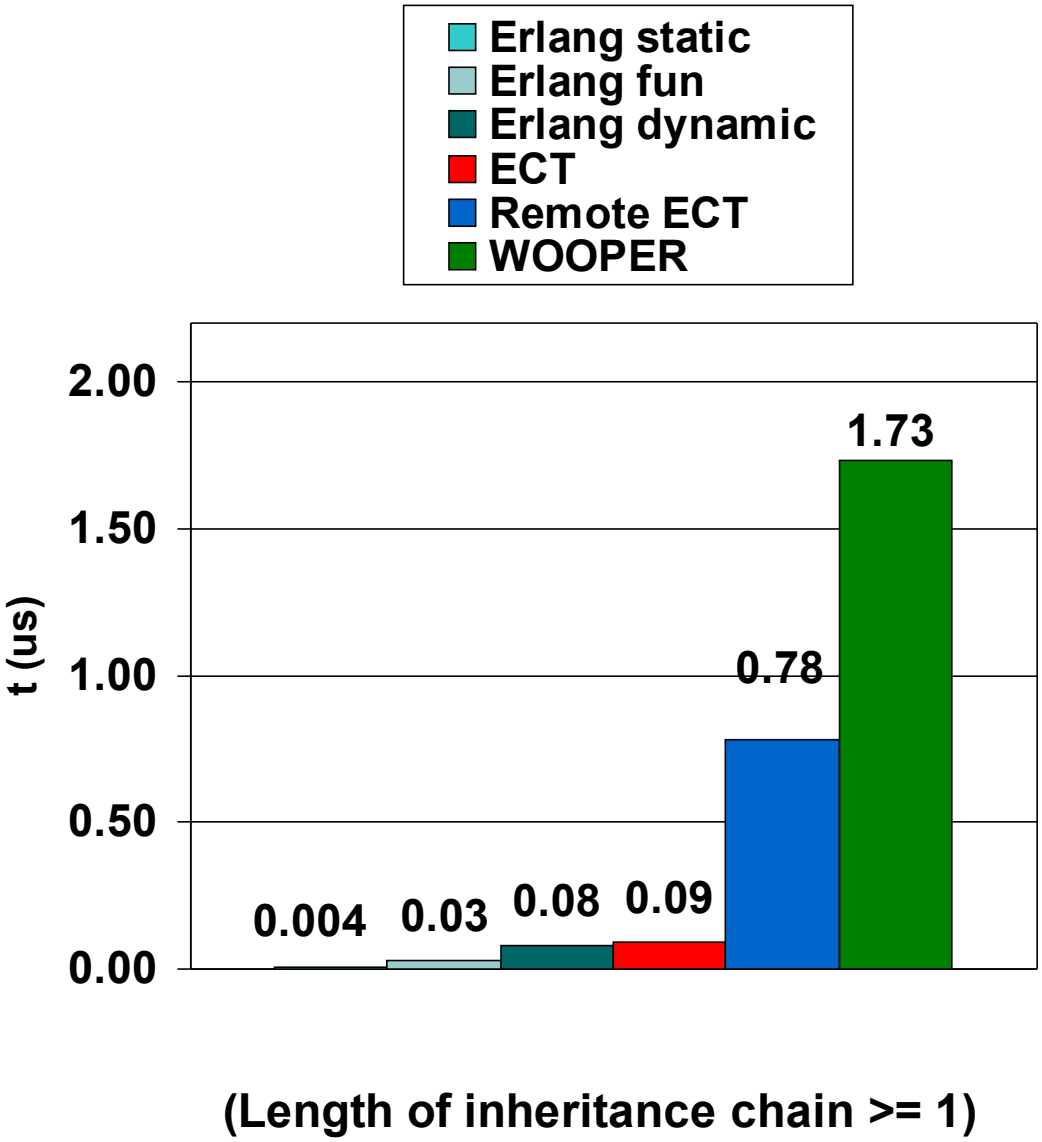
```
begin
    X0 = DogExpression,      % X0 is a generated variable name
    is_object(X0, dog),      % the dynamic-type test
    green = element(3, X0), % fail if not 'green'
    Name = element(4, X0),  % extract field
    X0                       % return value
end
```

- `demo(#cat{color=green, name=Name, x=X}, Name) -> whatever(X).`
  is converted into

```
demo(X0, Name) when is_object(X0,cat),
                    element(3,X0) =:= green,
                    element(4,X0) =:= Name ->
    X=element(5,X0),
    whatever(X).
```

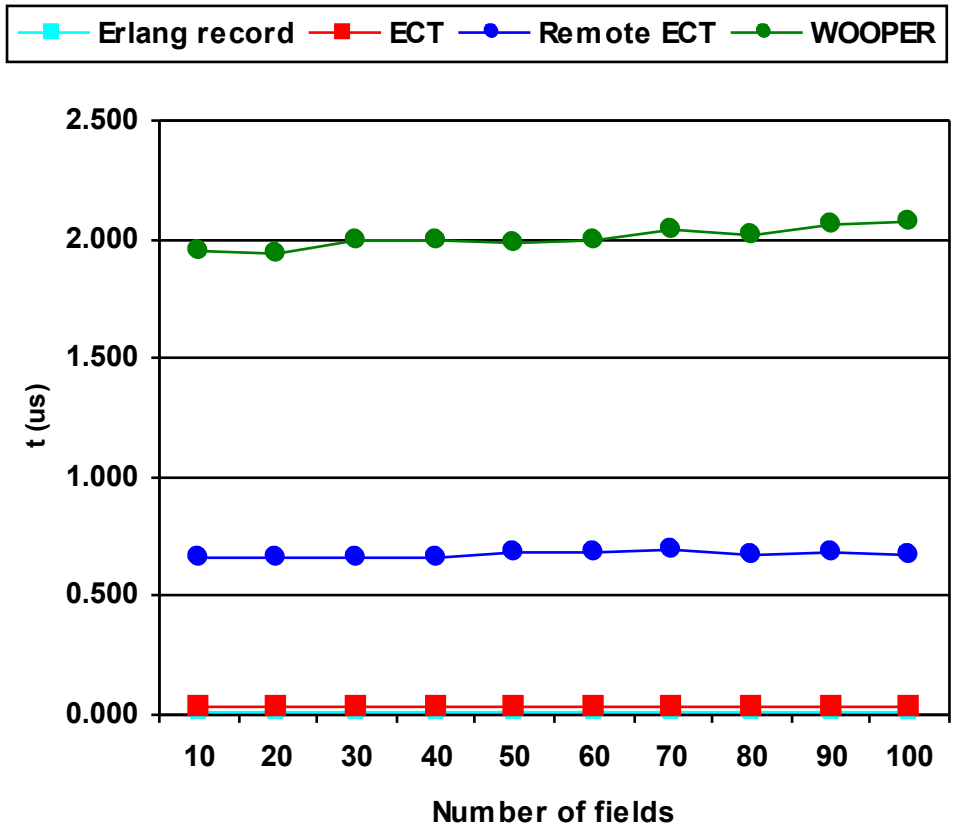
Method and function call times



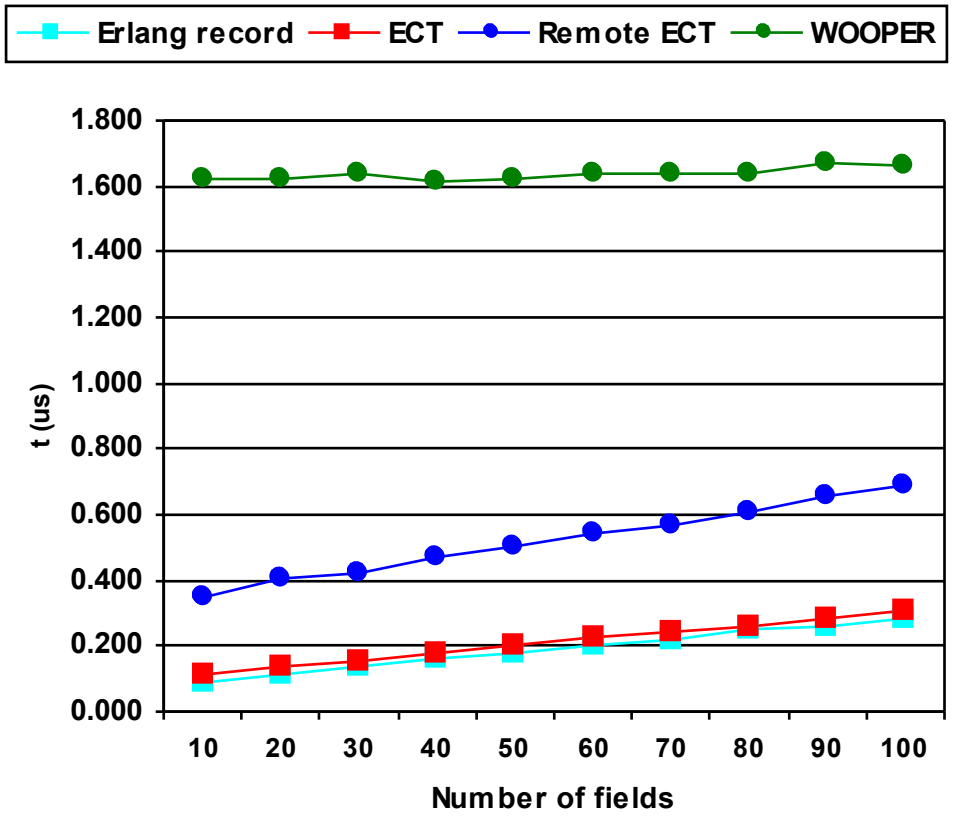


Field access times

Read



Update



Writing a parse transformation

- 1st approach: take `erl_id_trans.erl` (an identity transformation from OTP) and modify it
  - Whenever the Erlang AST changes, you have to modify your code
- 2nd approach: write a higher-order parse transformation that
  - Can be parametrized with a module name and an initial state
  - For each AST element it calls the appropriate function in the callback module
  - The transformation functions in the callback module return the AST and the state either unchanged, or modified
  - the design became better: code for traversing the tree and the code of an actual rewrite became separated in different modules (a behaviour and a callback module)
- 3rd approach: do the same, but you call it OOP (and write less code)
  - an identity-transformation class traverses the AST, while maintaining a state during the traverse
  - a subclass can override any of the methods of this class: the methods that handle those parts of the AST that you want to modify
  - ECT was rewritten using this approach. It can compile itself: proves that ECT works.

Interesting facts

- To extend a class, you only need the result of it's compilation, the `.beam` and the `.hrl` (i.e. the source is not needed).
- Code upgrade: any class module can be replaced any time. When a superclass is upgraded, the subclasses start using the new version of the methods that they've inherited and not overridden.
- With ECT, a method might replace the object with an object of another class. This is not possible in most OOP languages and we don't know if it's useful.
- You can not store members of a descendant class in a MNESIA table, because in MNESIA “the records must be instances of the same record type”.

Conclusion

- What should be the role of ECT?
  - Don't use OOP everywhere, only when it offers an advantage
  - You can use ECT for Object Oriented Programming, or
  - Use only for extensible records
  - Use only for extensible modules
- ECT is in beta state, download it, test it, comments and opinions are welcomed
  - The “remote ECT” part is only a prototype implementation, just to be able to measure its performance and compare with others.

Thanks for your attention.

Questions?