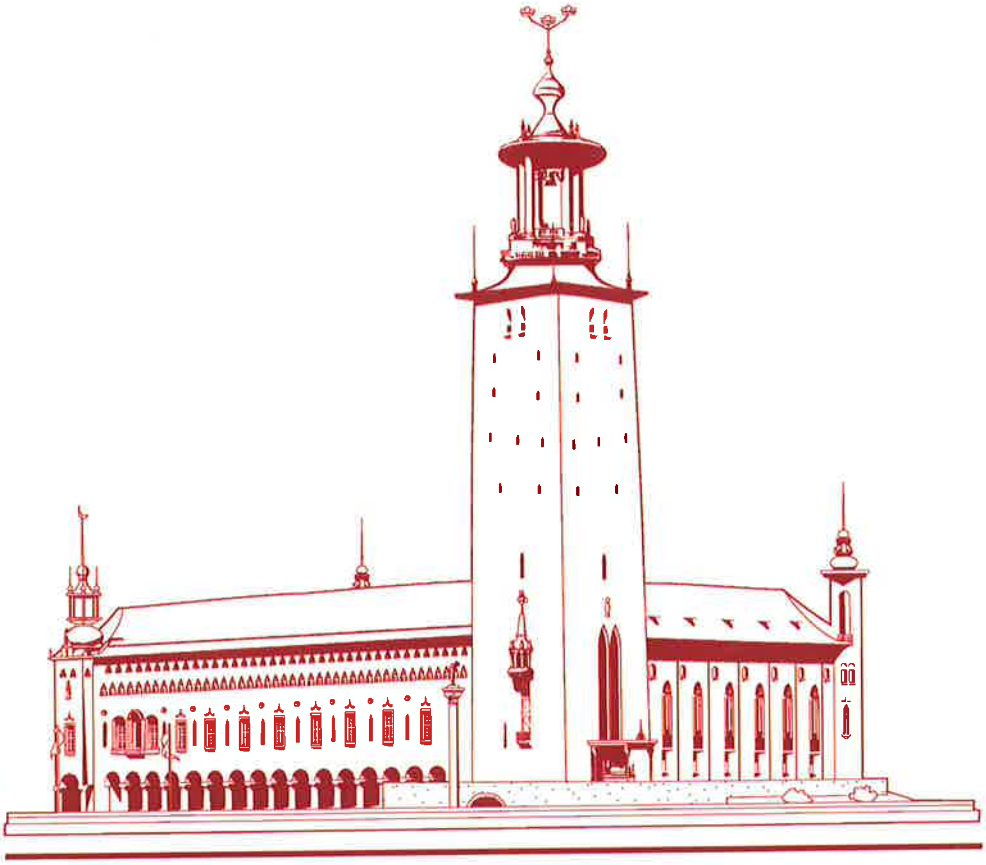


Conference Proceedings

15th International Erlang User Conference

STOCKHOLM, SWEDEN

12th November 2009

ERLANG

SPONSORS

ERICSSON

Erlang

Erlang Training and Consulting Ltd

synapse

mobile networks

klarna

Mobile Arts

tail-f

process one

INSTANT MESSAGES CREATIVE BUSINESSES

O'REILLY

corelatus

Sjöland&Thyselius

Creative Software Solutions

QuviQ

15th International Erlang User Conference

Conference Programme

Bjarne Däcker
Manager of the CSLab at Ericsson
- the birthplace of Erlang

Welcome and introduction

Abstract

Bjarne will open the conference and give a short introduction.

Biography

Bjarne Däcker joined Ericsson in 1966 as programmer and systems analyst. In 1984 he set up the Computer Science Lab together with Mike Williams to explore, develop and introduce new software technology in Ericsson often in collaboration with university research. The CSLab pioneered things like Unix, A.I., Lisp, Prolog and workstations in Ericsson. Erlang was created at the CSLab by an initial team of Joe Armstrong, Mike Williams and Robert Virding. Bjarne organised the first Erlang User Conference in 1994. He has had various external committments such as chairman of the steering committee of the Swedish national research programme in Computer Science 1987-1992 and member of the Evaluation Committee of European Union's ICT-Prize. The CSLab was closed in 2002 in the IT crash. Bjarne holds a Technology Licentiate degree of the Royal Institute of Technology in Stockholm and was promoted to an Honorary Doctorate of Technology at Linköping University in 1993. He is also a member of the Royal Swedish Academy of the Engineering Sciences.

Welcome and introduction

It is our great pleasure to welcome you to the Fifteenth International Erlang User Conference (EUC 2009). For ten years the Erlang User Conference has been held in the Ericsson Conference facilities in Älvsjö Southwest of Stockholm. However last year, the Erlang User Conference outgrew the lecture hall. With a capacity of 140 delegates, several interested people had to be turned away. Thus this year's Erlang User Conference is a turning point with the classical Astoria cinema as the venue and with nearly 250 people attending. Otherwise we keep to the established format of a combination of papers about fun, exciting new applications and presentations of technological developments. The applications show how Erlang is rapidly moving from its initial base in telecommunications to new areas like cloud computing.

Bjarne Däcker

Erlang User Conference 2009 Chairman
Manager 1984-2002 of the Computer Science Laboratory
at Ericsson where Erlang initiated

Rusty Klophaus
Author of Nitrogen Web Framework,
Riak committer

Nitrogen and Riak By Example

Abstract

Nitrogen has gained a quick and active community by providing extensive example-based documentation. In this talk, Rusty will continue this example-based approach by walking through a simple application built on Nitrogen and Riak, highlighting common patterns and best practices.

Biography

Rusty Klophaus is the author of the Nitrogen Web Framework and a Riak code committer. Rusty typed his first line of Erlang code in early 2008 after trying (and quickly discarding) a number of other functional languages. He now ferociously evangelizes the merits of Erlang to anyone who will listen. Rusty grew up on a farm, earned a degree in Computer Science from Princeton University, spent a summer in Philadelphia as a professional musician, co-founded a .NET software consulting company, and has managed multi-million dollar technology projects. He recently joined Basho Technologies, a company focused on providing large data storage, access, and analysis solutions powered by Erlang.

Nitrogen Web Framework For Erlang

riak

Nitrogen and Riak by Example

Rusty Klophaus
@rklophaus
http://www.basho.com

basho

Erlang User Conference
Stockholm, Sweden
November 12, 2009

"50 line code snippets are useful, but how do you build a real application?"

Problem / Use Case

Slide-focused conference call with a prospective client or investor:

“Should I try to use Webex, or just email the slides?”

“The file is 5MB, what if their email server blocks it?”

“Should I send this as a .pdf, .ppt, .pptx, or Keynote?”

“Slide 3 will have no impact without an explanation.”

“I hope they don’t read slide 10 out of context.

“Is the audience even paying attention, or are they reading ahead?”

Solution: Web Slideshow Tool

Step 1 - Upload Your Slides

- Accepts .ZIP containing images, text, code.
- Accepts .PDF (requires Ghostscript)

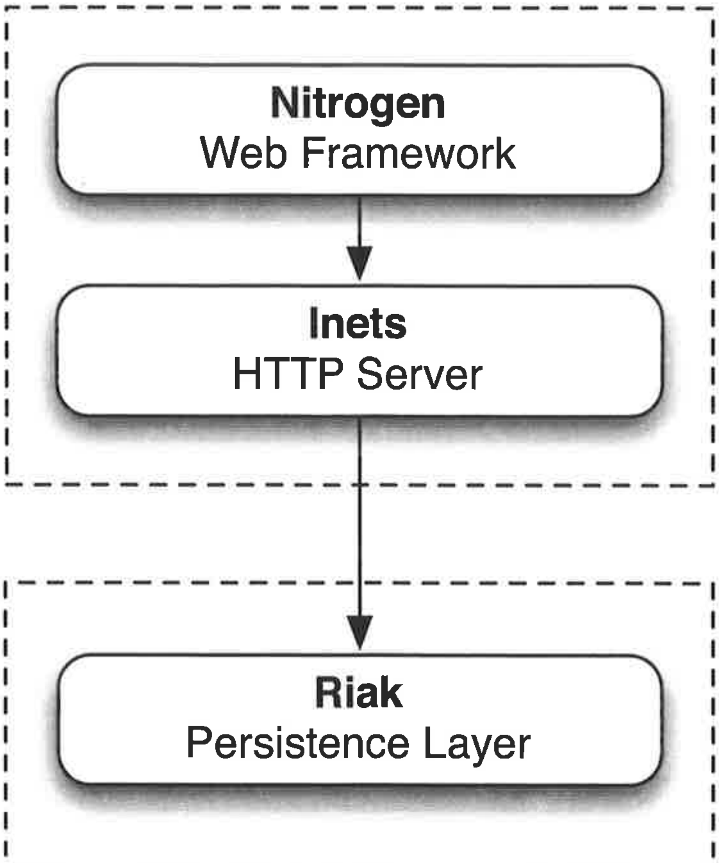
Step 2 - Share a URL

- Via email, chat, etc.

Step 3 - Page Through Slides

- When you advance a slide, everybody in the audience sees the same thing.
- Near-Instant feedback if somebody disconnects.

Components



Nitrogen
Web Framework

Inets
HTTP Server

Riak
Persistence Layer

Nitrogen Web Framework For Erlang

Nitrogen: The Basics

Put elements on a page

```
#link { id=myLink, text="Login" }
```

Listen for Events

```
wf:wire(myLink,
 #event { type=click, postback=click})
```

When an Event Happens, Update the Page

```
wf:update(myPanel,
 #span { text="You clicked!" })
```

Nitrogen: Web 2.0 in Erlang

Rapid Development

- ~40 built-in elements, ~15 actions, 8 validators
- One-line Ajax and Comet
- Abstraction layer for JQuery features:
  - Effects, Sorting, Drag and Drop

Extensible

- Create custom elements and actions

Powerful

- Streaming File Uploads
- APIs for session state, page state, cookies, security

Riak: The Basics

Store Data

```
Obj = riak_object:new(Bucket, Key, Value),
ok = Client:put(Obj, 3)
```

Retrieve Data

```
{ok, Obj} = Client:get(Bucket, Key, 2)
```

Schema Agnostic

Bucket and Key are both binaries

Value can be any term

Riak: Inspired by Dynamo

Scalable

- Add a machine: Gain capacity, speed, and reliability.
- Remove a machine: blocks of data (partitions) are moved to rebalance the cluster.

Reliable & Resilient

- When a machine dies, the other nodes cover for it. (Hinted handoff.)
- Conflicting edits are either last write wins, or can bubble up to your application, if desired.

Riak: Inspired by Dynamo

Flexible

- Tune N per bucket (Number of data replicas.)
- Tune R and W per operation. (How many replicas must respond?)
- Swappable storage engines. Choose one that fits your data.

Riak: Innovation

Map/Reduce for Deep Queries

- Streaming, multi-stage maps and reduces
- The code runs where the data is stored

Linked Data

- A link is a pointer from one object to another
- HTTP interface to traverse links to get related objects

Riak: Innovation

Eventing System

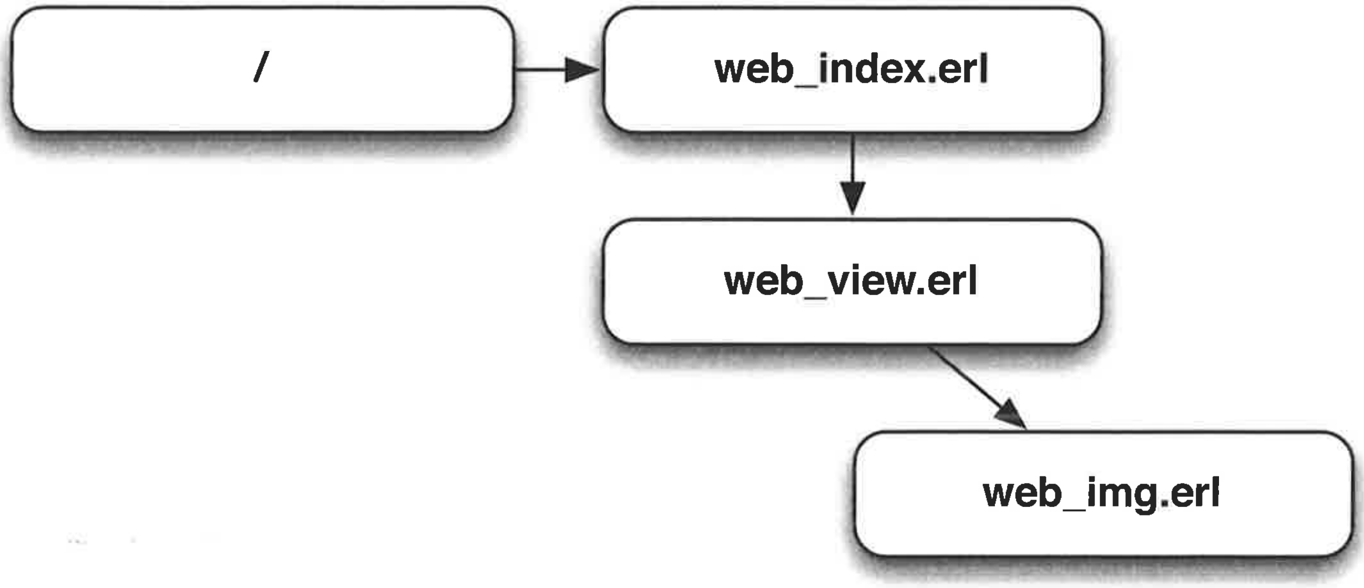
- Subscribe to events using a matchspec

Multi-Lingual

- Erlang, Javascript, Java, Ruby, PHP, Python, & HTTP

Back to our application...

Structure



web_index.erl

Nitrogen Concepts

- Template
- Upload and Upload Event
- Flash
- Redirect

Riak Concepts

- Connect a Client
- Objects, Buckets, and Keys
- Put

web_view.erl

Nitrogen Concepts

- Custom Elements
- Comet
- Session
- Series ID

Riak Concepts

- Get

slide_list_element.erl

Nitrogen Concepts

- Sorting
- Click Events / Actions

slide_controls_element.erl

Nitrogen Concepts

- KeyPress Events / Actions

web_img.erl

Nitrogen Concepts

- Content Types
- Path Info

Riak Concepts

- Get an Object

Sample Code http://github.com/rklophaus/caster

Nitrogen http://nitrogenproject.com
@nitrogenproject

Riak http://nitrogenproject.com
@justinsheehy, @argv0,
@hobbyist, @jrecursive, @rklophaus

Rusty Klophaus http://rklophaus.com
@rklophaus

Jacob Vorreuter
Erlang hacker at Electronic Arts

Hacking Erlang through preprocessing

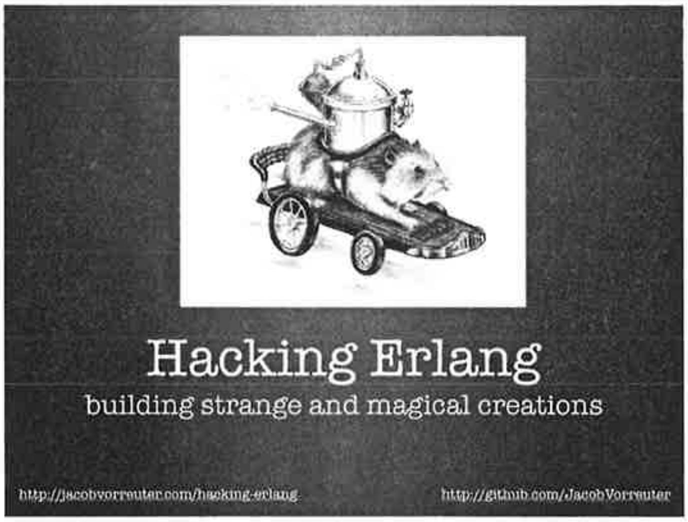
Abstract

The preprocessing step in Erlang code compilation is largely undocumented, but very powerful. The language can be extended to include custom guards, syntax and constructs. Included in the talk are the following:

- Dynamic compilation with the erl_scan, erl_parse, epp and compile modules
- Reverse engineering compiled BEAM code into forms
- Preprocessing vs macros
- The parse_transform compile directive and example usages like:
- adding helper functions into modules that take advantage of record definitions that aren't available at runtime
- performing data integrity checks by expanding custom guards into additional function clauses
- Example usages of the custom_guards, dynamic_compile and excavator projects in production environments at EA.

Biography

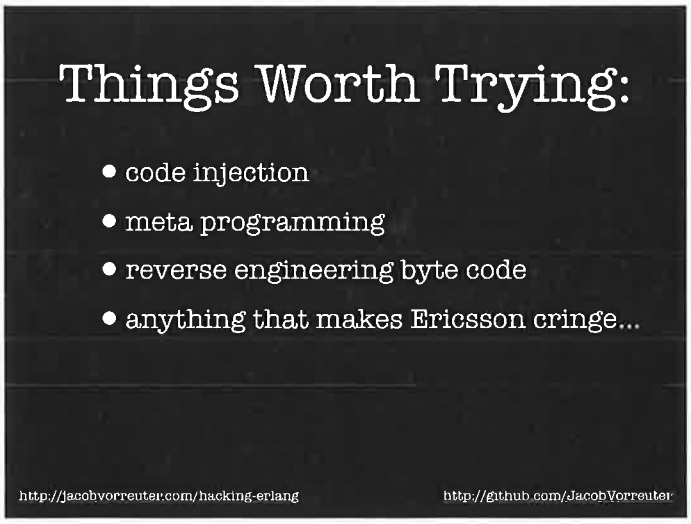
Jacob writes Erlang code for Electronic Arts. His work there has included deploying scalable ejabberd chat clusters and developing a high-traffic Erlang api used by game teams to read and write real-time game data. He's written or contributed to many open source Erlang projects, including dynamic_compile, emongo, erlang_protobuffs, etap, emysql and the binary protocol memcached client.



Hacking Erlang

building strange and magical creations

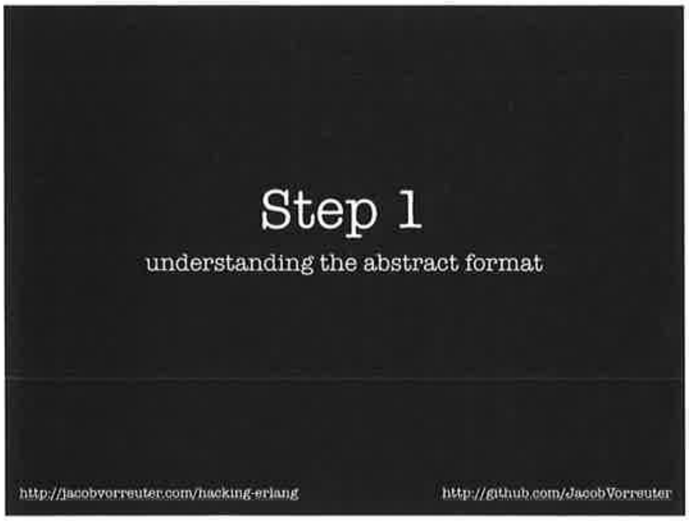
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



Things Worth Trying:

- code injection
- meta programming
- reverse engineering byte code
- anything that makes Ericsson cringe...

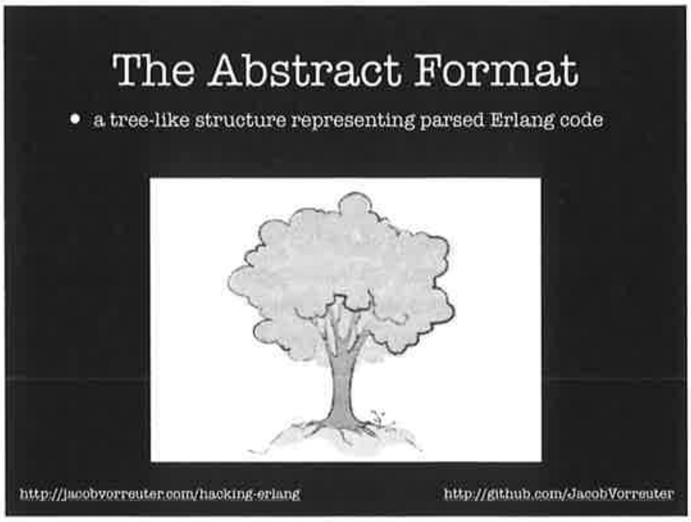
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



Step 1

understanding the abstract format

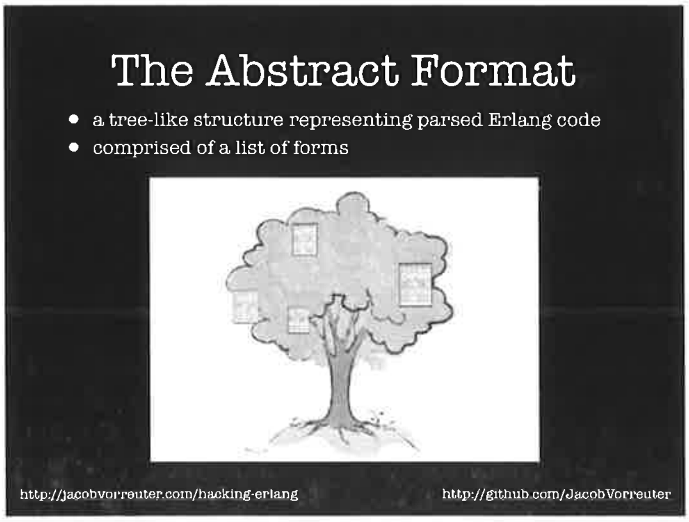
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



The Abstract Format

- a tree-like structure representing parsed Erlang code

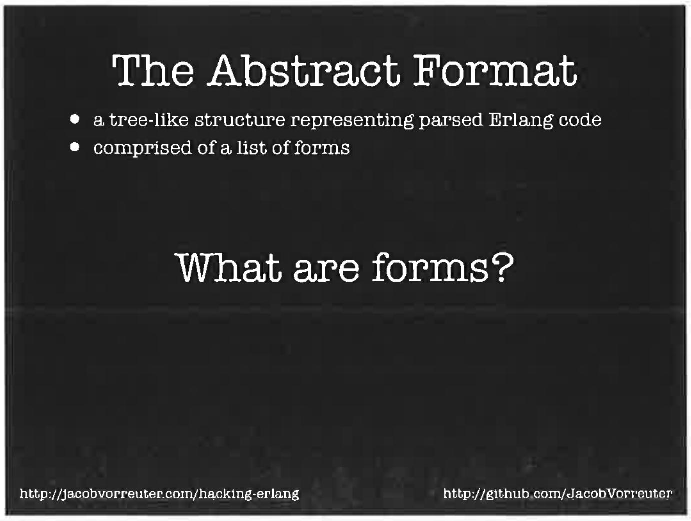
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

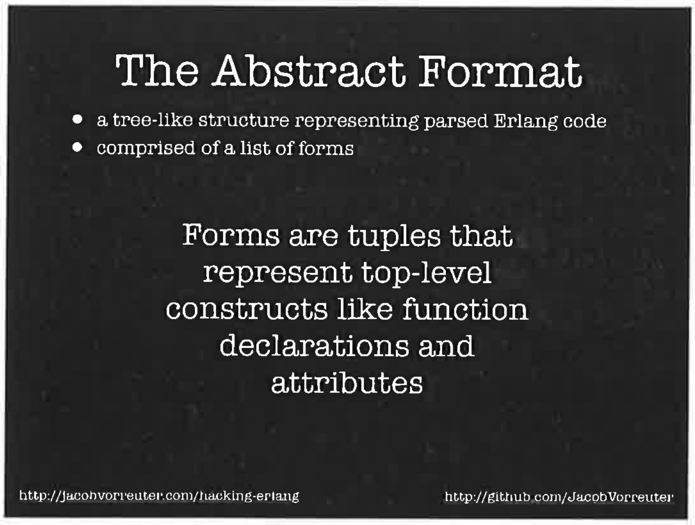


The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

What are forms?

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

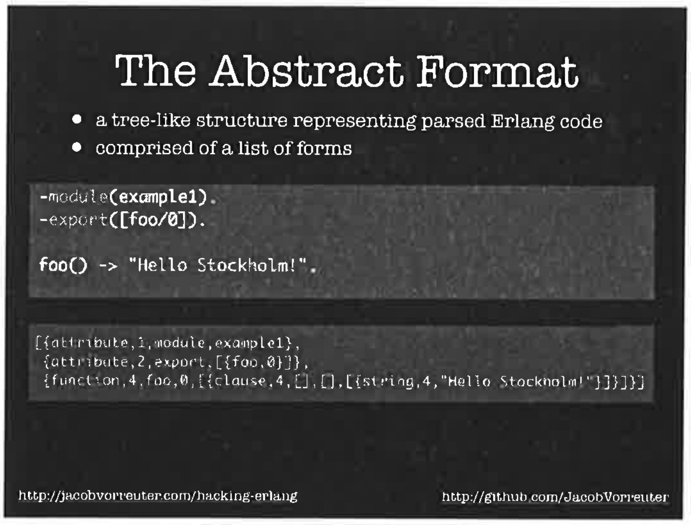


The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

Forms are tuples that represent top-level constructs like function declarations and attributes

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



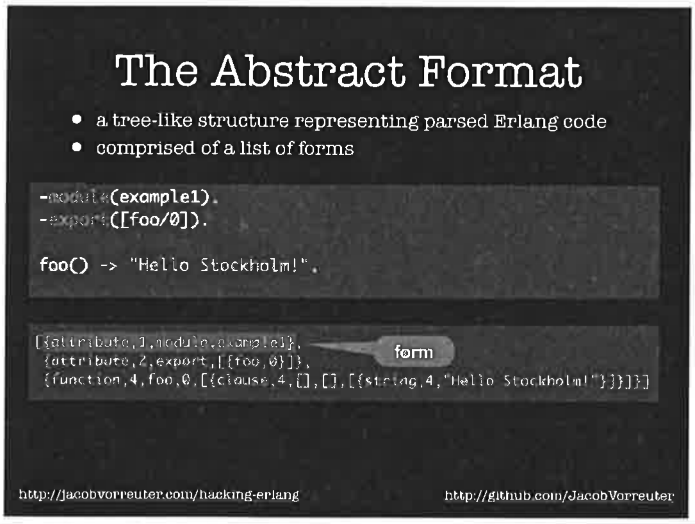
The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".
```

```
[{attribute,1,module,example1},
 {attribute,2,export,[{foo,0}]},
 {function,4,foo,0,[{clause,4,[],[],[{string,4,"Hello Stockholm!"}]}]}]
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



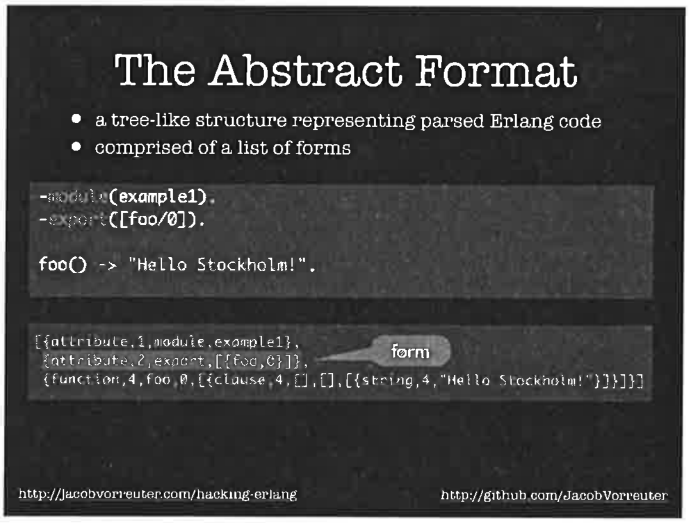
The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".
```

```
[{attribute,1,module,example1},
 {attribute,2,export,[{foo,0}]},
 {function,4,foo,0,[{clause,4,[],[],[{string,4,"Hello Stockholm!"}]}]}]
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



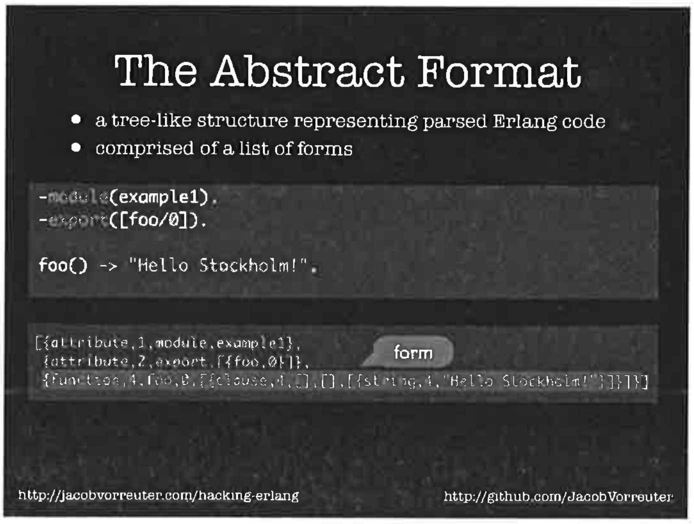
The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".
```

```
[{attribute,1,module,example1},
 {attribute,2,export,[{foo,0}]},
 {function,4,foo,0,[{clause,4,[],[],[{string,4,"Hello Stockholm!"}]}]}]
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



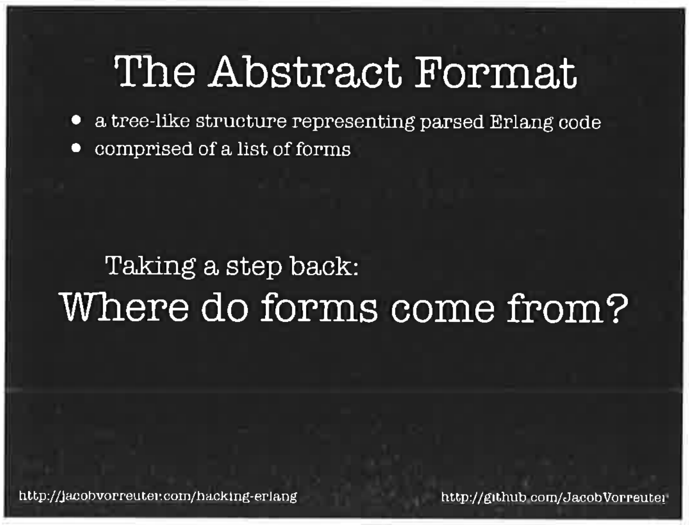
The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".
```

```
[{attribute,1,module,example1},
 {attribute,2,export,[{foo,0}]},
 {function,4,foo,0,[{clause,4,[],[],[{string,4,"Hello Stockholm!"}]}]}]
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



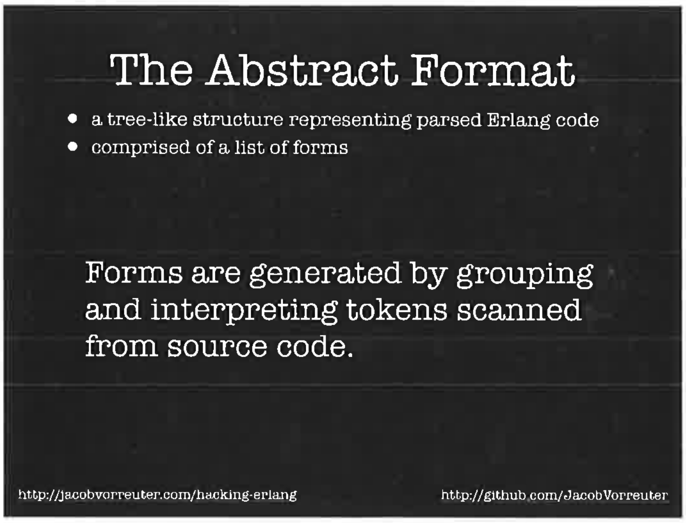
The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

Taking a step back:

Where do forms come from?

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

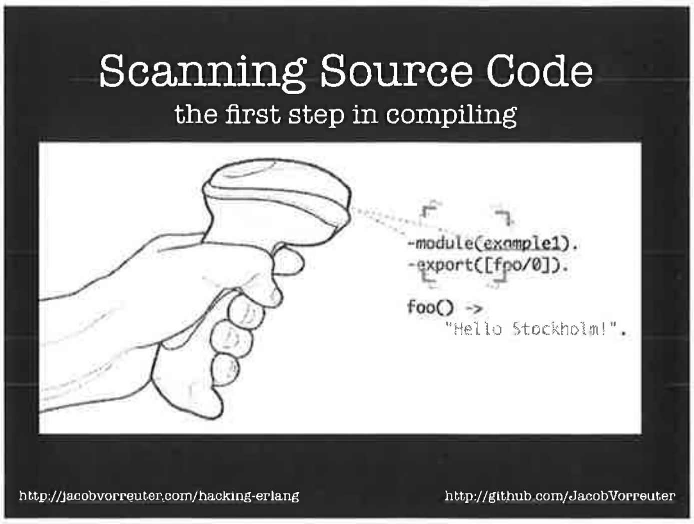


The Abstract Format

- a tree-like structure representing parsed Erlang code
- comprised of a list of forms

Forms are generated by grouping and interpreting tokens scanned from source code.

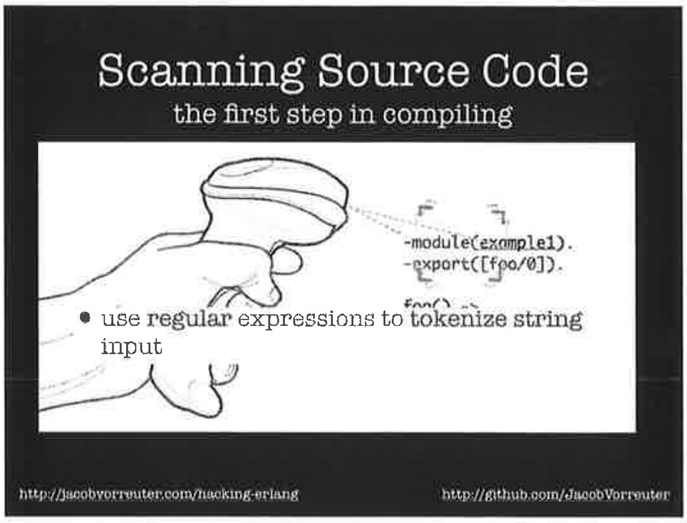
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



Scanning Source Code

the first step in compiling

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

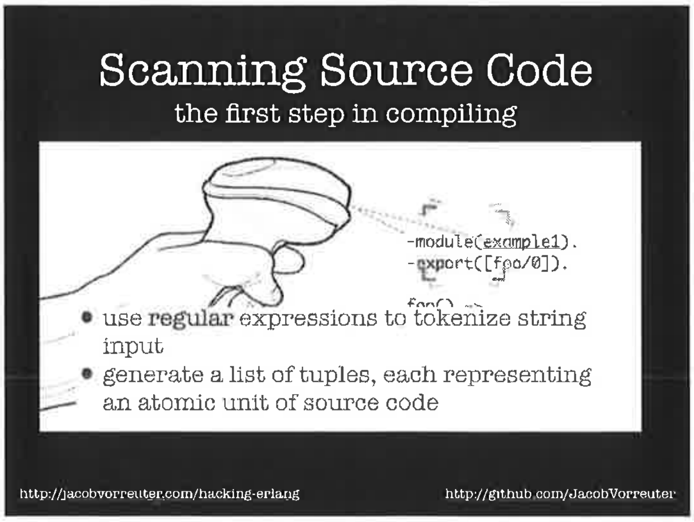


Scanning Source Code

the first step in compiling

- use regular expressions to tokenize string input

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

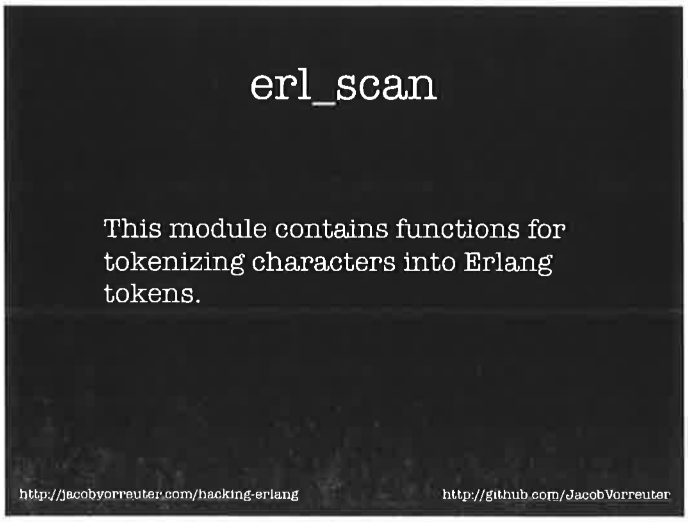


Scanning Source Code

the first step in compiling

- use regular expressions to tokenize string input
- generate a list of tuples, each representing an atomic unit of source code

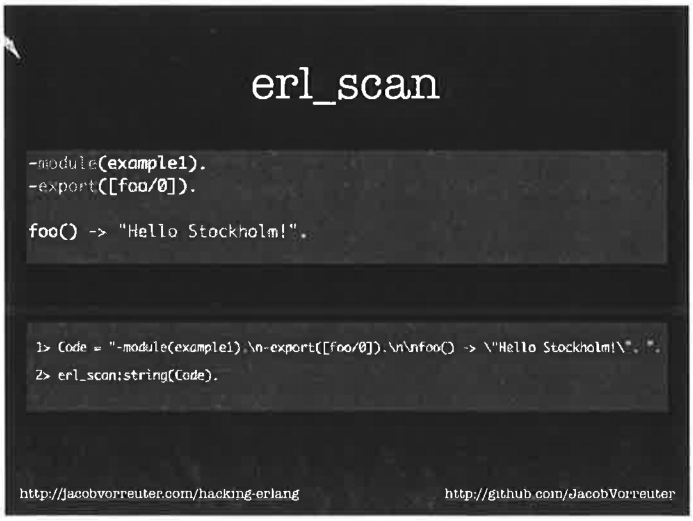
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



erl_scan

This module contains functions for tokenizing characters into Erlang tokens.

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

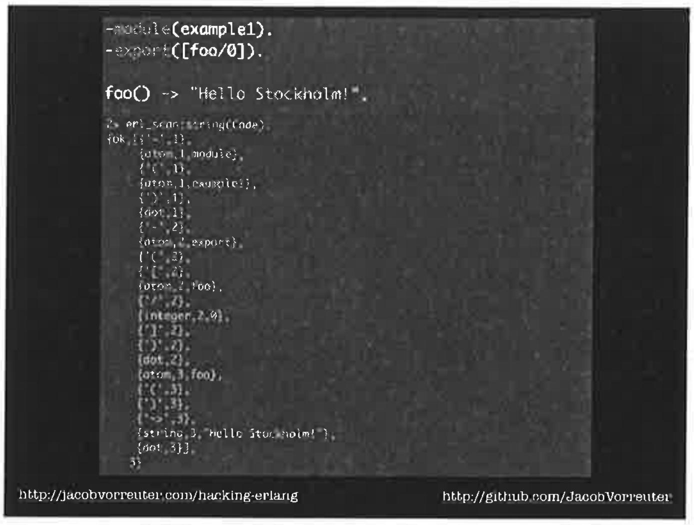


erl_scan

```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".
```

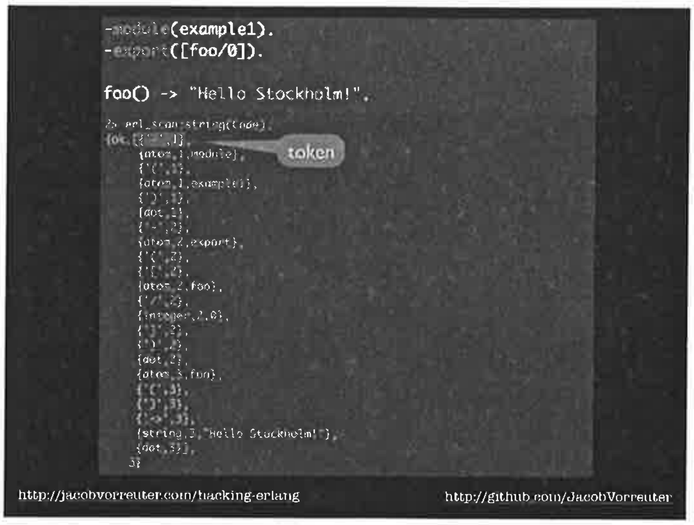
```
1> Code = "-module(example1).\n-export([foo/0]).\n\nfoo() -> \"Hello Stockholm!\". ".
2> erl_scan:string(Code).
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



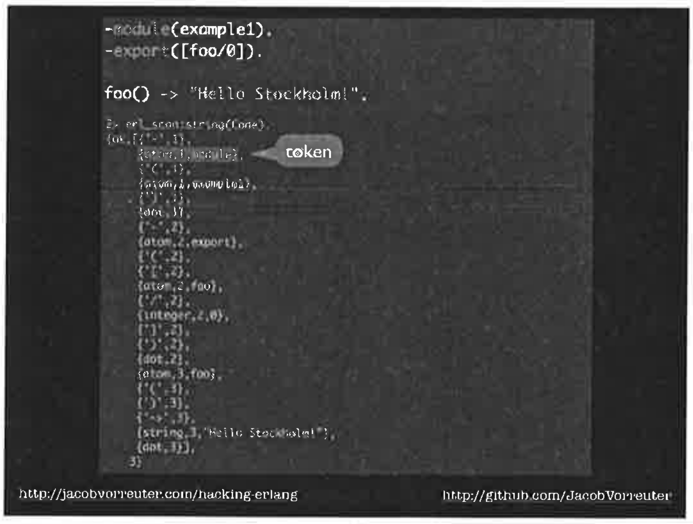
```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".

2> erl_scan:string(Code).
{ok,[{'-',1},
     {atom,1,module},
     {'(',1},
     {atom,1,example1},
     {')',1},
     {dot,1},
     {'-',2},
     {atom,2,export},
     {'(',2},
     {'[',2},
     {atom,2,foo},
     {'/',2},
     {integer,2,0},
     {']',2},
     {')',2},
     {dot,2},
     {atom,3,foo},
     {'(',3},
     {')',3},
     {'->',3},
     {string,3,"Hello Stockholm!"},
     {dot,3}],
 3}
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



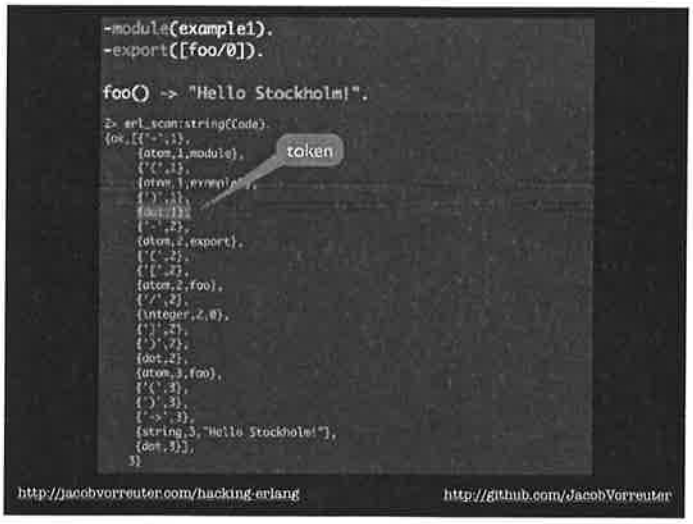
```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".

2> erl_scan:string(Code).
{ok,[{'-',1},
     {atom,1,module},
     {'(',1},
     {atom,1,example1},
     {')',1},
     {dot,1},
     {'-',2},
     {atom,2,export},
     {'(',2},
     {'[',2},
     {atom,2,foo},
     {'/',2},
     {integer,2,0},
     {']',2},
     {')',2},
     {dot,2},
     {atom,3,foo},
     {'(',3},
     {')',3},
     {'->',3},
     {string,3,"Hello Stockholm!"},
     {dot,3}],
 3}
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



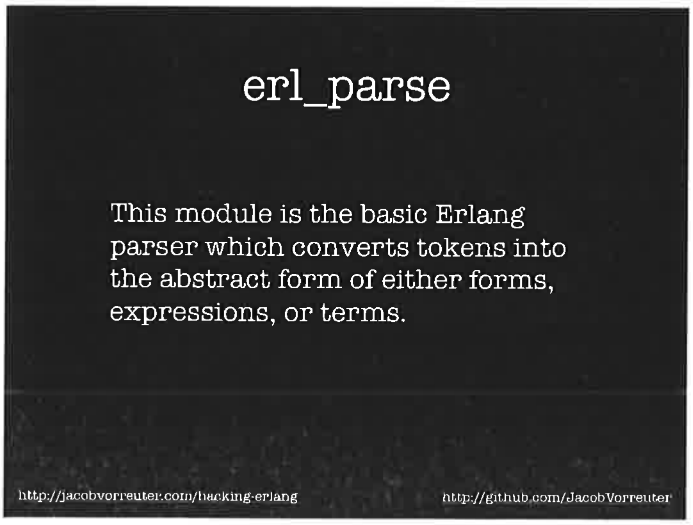
```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".

2> erl_scan:string(Code).
{ok,[{'-',1},
     {atom,1,module},
     {'(',1},
     {atom,1,example1},
     {')',1},
     {dot,1},
     {'-',2},
     {atom,2,export},
     {'(',2},
     {'[',2},
     {atom,2,foo},
     {'/',2},
     {integer,2,0},
     {']',2},
     {')',2},
     {dot,2},
     {atom,3,foo},
     {'(',3},
     {')',3},
     {'->',3},
     {string,3,"Hello Stockholm!"},
     {dot,3}],
 3}
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



```
-module(example1).
-export([foo/0]).

foo() -> "Hello Stockholm!".

2> erl_scan:string(Code).
{ok,[{'-',1},
     {atom,1,module},
     {'(',1},
     {atom,1,example1},
     {')',1},
     {dot,1},
     {'-',2},
     {atom,2,export},
     {'(',2},
     {'[',2},
     {atom,2,foo},
     {'/',2},
     {integer,2,0},
     {']',2},
     {')',2},
     {dot,2},
     {atom,3,foo},
     {'(',3},
     {')',3},
     {'->',3},
     {string,3,"Hello Stockholm!"},
     {dot,3}],
 3}
```

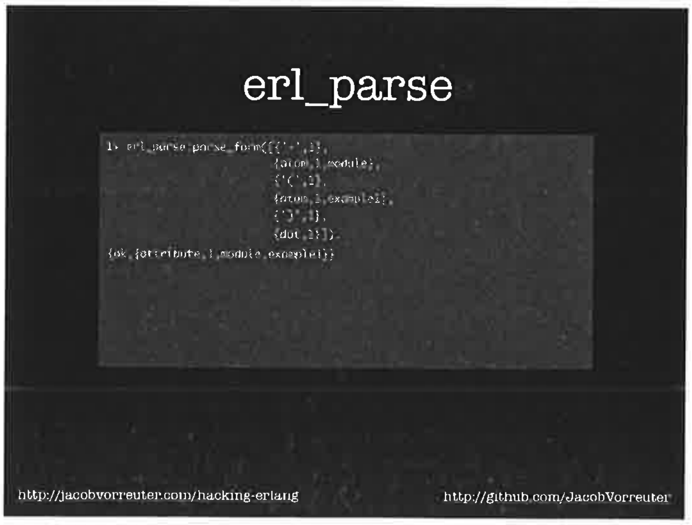
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



erl_parse

This module is the basic Erlang parser which converts tokens into the abstract form of either forms, expressions, or terms.

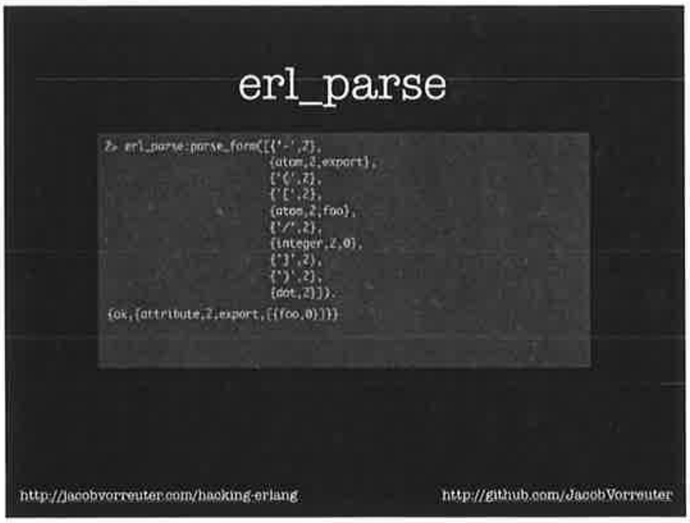
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



erl_parse

```
1> erl_parse:parse_form([{'-',1},
                         {atom,1,module},
                         {'(',1},
                         {atom,1,example1},
                         {')',1},
                         {dot,1}]).
{ok,{attribute,1,module,example1}}
```

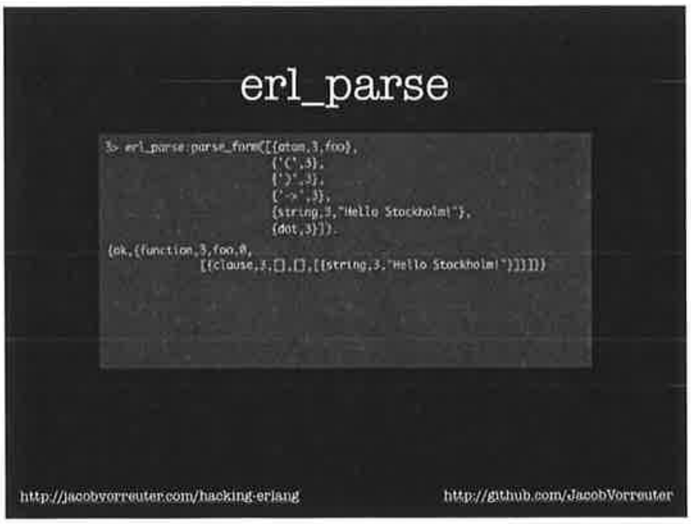
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



erl_parse

```
2> erl_parse:parse_form([{'-',2},
                        {atom,2,export},
                        {'(',2},
                        {'[',2},
                        {atom,2,foo},
                        {'/',2},
                        {integer,2,0},
                        {']',2},
                        {')',2},
                        {dot,2}]).
{ok,{attribute,2,export,[{foo,0}]}}
```

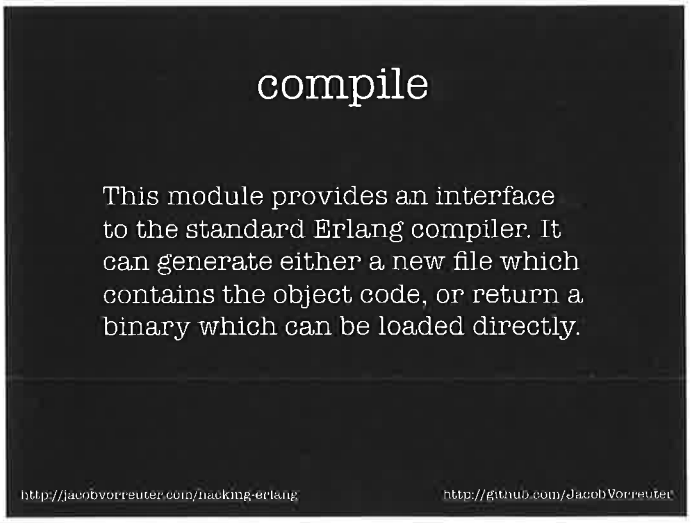
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



erl_parse

```
3> erl_parse:parse_form([{atom,3,foo},
                        {'(',3},
                        {')',3},
                        {'->',3},
                        {string,3,"Hello Stockholm!"},
                        {dot,3}]).
{ok,{function,3,foo,0,
          [{clause,3,[],[],[{string,3,"Hello Stockholm!"}]}]}}
```

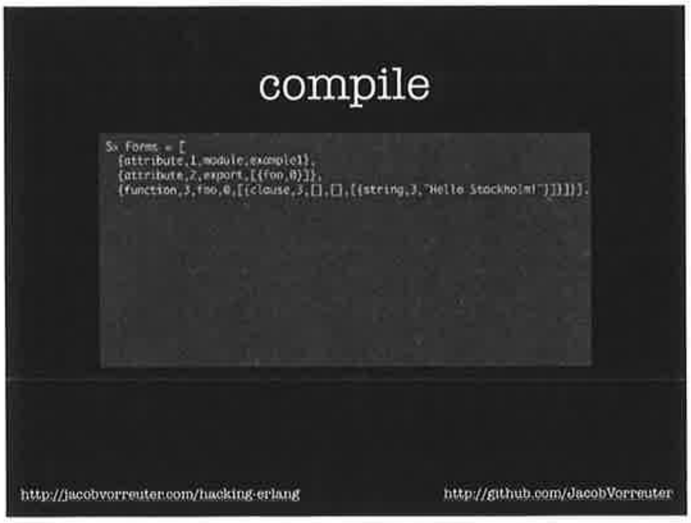
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



compile

This module provides an interface to the standard Erlang compiler. It can generate either a new file which contains the object code, or return a binary which can be loaded directly.

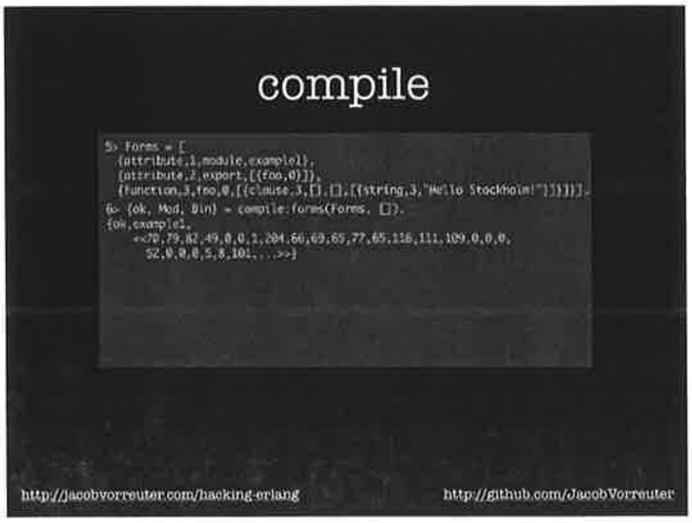
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



compile

```
5> Forms = [
  {attribute,1,module,example1},
  {attribute,2,export,[{foo,0}]},
  {function,3,foo,0,[{clause,3,[],[],[{string,3,"Hello Stockholm!"}]}]}].
```

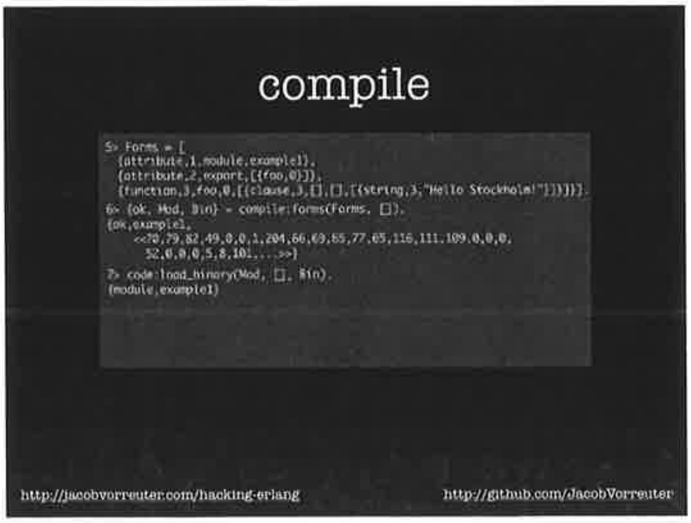
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



compile

```
5> Forms = [
  {attribute,1,module,example1},
  {attribute,2,export,[{foo,0}]},
  {function,3,foo,0,[{clause,3,[],[],[{string,3,"Hello Stockholm!"}]}]}].
6> {ok, Mod, Bin} = compile:forms(Forms, []).
{ok,example1,
    <<70,79,82,49,0,0,1,204,66,69,65,77,65,116,111,109,0,0,0,
      52,0,0,0,5,8,101,...>>}
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

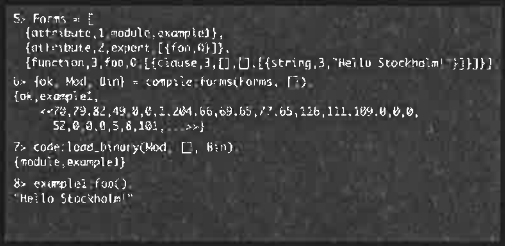


compile

```
5> Forms = [
  {attribute,1,module,example1},
  {attribute,2,export,[{foo,0}]},
  {function,3,foo,0,[{clause,3,[],[],[{string,3,"Hello Stockholm!"}]}]}].
6> {ok, Mod, Bin} = compile:forms(Forms, []).
{ok,example1,
    <<70,79,82,49,0,0,1,204,66,69,65,77,65,116,111,109,0,0,0,
      52,0,0,0,5,8,101,...>>}
7> code:load_binary(Mod, [], Bin).
{module,example1}
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

compile



```
5> Forms = [
  {attribute,1,module,example1},
  {attribute,2,export,[{foo,0}]},
  {function,3,foo,0,[{clause,3,[],[],[{string,3,"Hello Stockholm!"}]}]}].
6> {ok, Mod, Bin} = compile:forms(Forms, []).
{ok,example1,
    <<70,79,82,49,0,0,1,204,66,69,65,77,65,116,111,109,0,0,0,
      52,0,0,0,5,8,101,...>>}
7> code:load_binary(Mod, [], Bin).
{module,example1}
8> example1:foo().
"Hello Stockholm!"
```

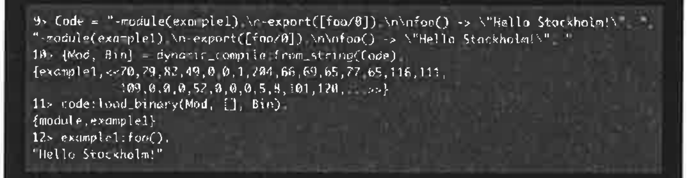
IS THERE A MODULE THAT CAN PERFORM ALL OF THOSE STEPS FOR ME?!?!?

dynamic_compile

The dynamic_compile module performs the actions we've just seen, plus takes care of macro expansion and inclusion of external header files.

http://github.com/JacobVorreuter/dynamic_compile

dynamic_compile



```
9> Code = "-module(example1).\n-export([foo/0]).\n\nfoo() -> \"Hello Stockholm!\".".
"-module(example1).\n-export([foo/0]).\n\nfoo() -> \"Hello Stockholm!\"."
10> {Mod, Bin} = dynamic_compile:from_string(Code).
{example1,<<70,79,82,49,0,0,1,204,66,69,65,77,65,116,111,
            109,0,0,0,52,0,0,0,5,8,101,120,...>>}
11> code:load_binary(Mod, [], Bin).
{module,example1}
12> example1:foo().
"Hello Stockholm!"
```

moving on...

the parse_transform debate...

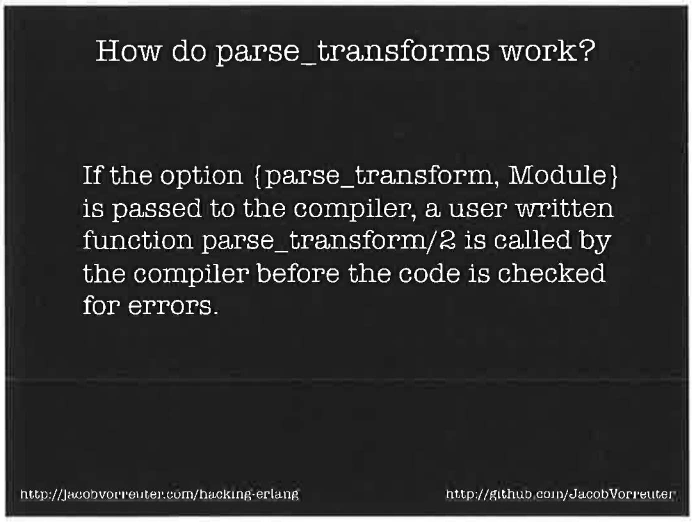
Programmers are strongly advised NOT to engage in parse transformations

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

wait! parse_transforms are cool and have their place in the language...in moderation.

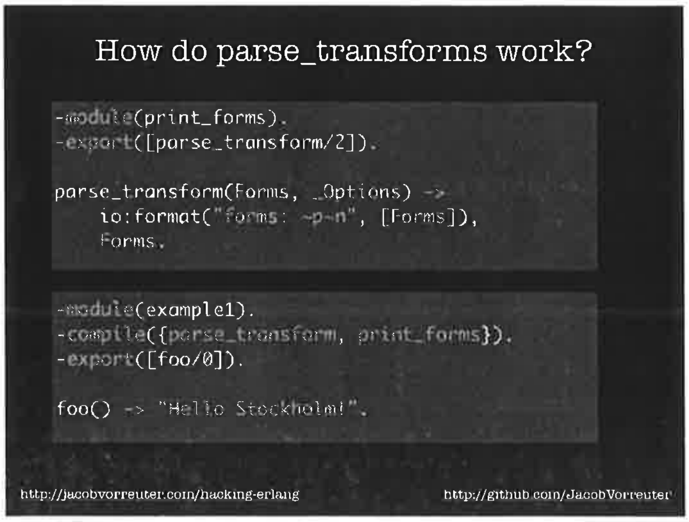
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



How do parse_transforms work?

If the option {parse_transform, Module} is passed to the compiler, a user written function parse_transform/2 is called by the compiler before the code is checked for errors.

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

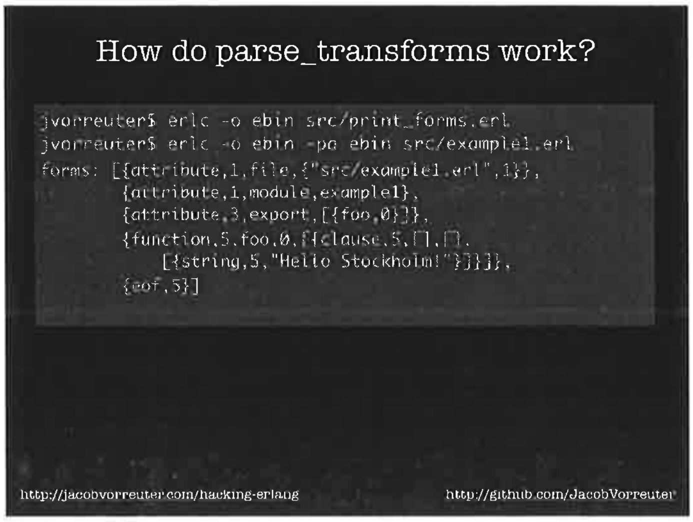


How do parse_transforms work?

```
-module(print_forms).
-export([parse_transform/2]).

parse_transform(Forms, _Options) ->
    io:format("forms: ~p~n", [Forms]),
    Forms.
```

```
-module(example1).
-compile({parse_transform, print_forms}).
-export([foo/0]).

foo() -> "Hello Stockholm!".
```

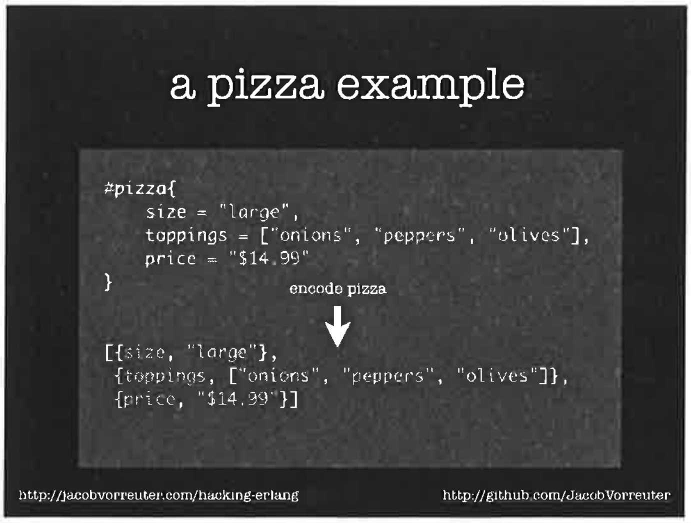
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



How do parse_transforms work?

```
jvorreuter$ erlc -o ebin src/print_forms.erl
jvorreuter$ erlc -o ebin -pa ebin src/example1.erl
forms: [{attribute,1,file,{"src/example1.erl",1}},
        {attribute,1,module,example1},
        {attribute,3,export,[{foo,0}]},
        {function,5,foo,0,[{clause,5,[],[],
            [{string,5,"Hello Stockholm!"}]}]},
        {eof,5}]
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



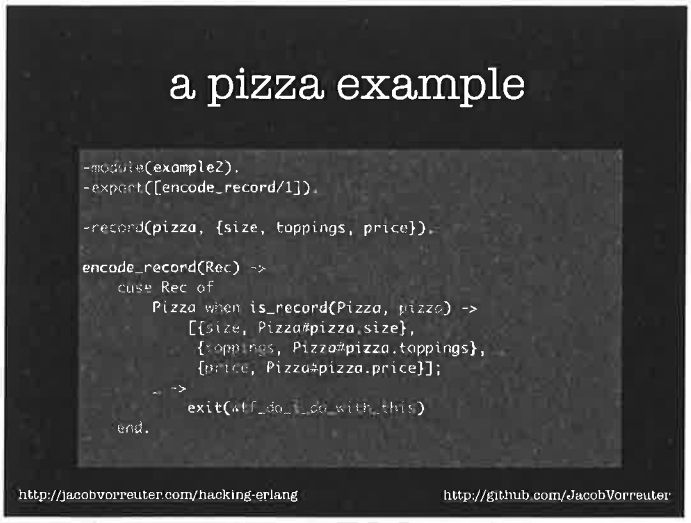
a pizza example

```
#pizza{
    size = "large",
    toppings = ["onions", "peppers", "olives"],
    price = "$14.99"
}
```

encode pizza

```
[{size, "large"},
 {toppings, ["onions", "peppers", "olives"]},
 {price, "$14.99"}]
```

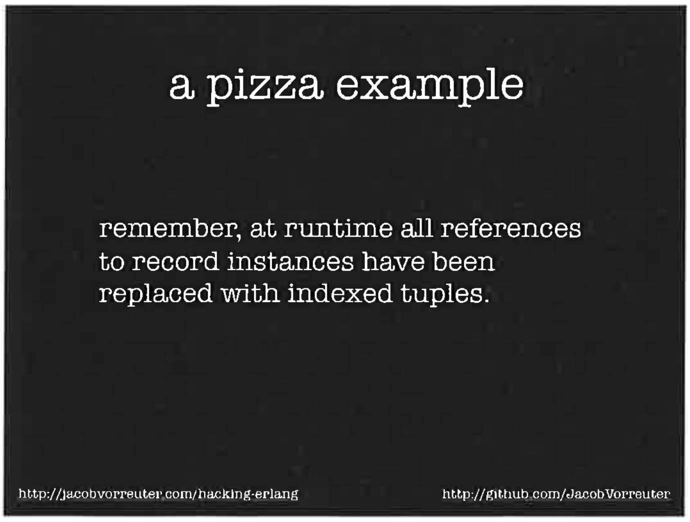
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



a pizza example

```
-module(example2).
-export([encode_record/1]).

-record(pizza, {size, toppings, price}).

encode_record(Rec) ->
    case Rec of
        Pizza when is_record(Pizza, pizza) ->
            [{size, Pizza#pizza.size},
             {toppings, Pizza#pizza.toppings},
             {price, Pizza#pizza.price}];
        _ ->
            exit(wtf_do_i_do_with_this)
    end.
```

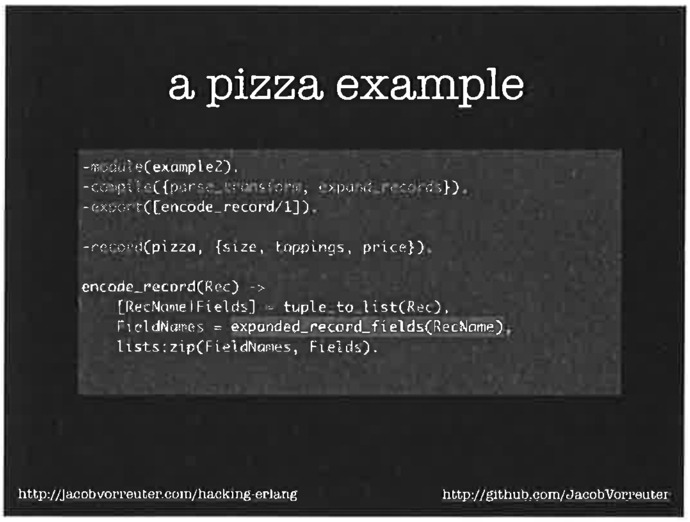
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



a pizza example

remember, at runtime all references to record instances have been replaced with indexed tuples.

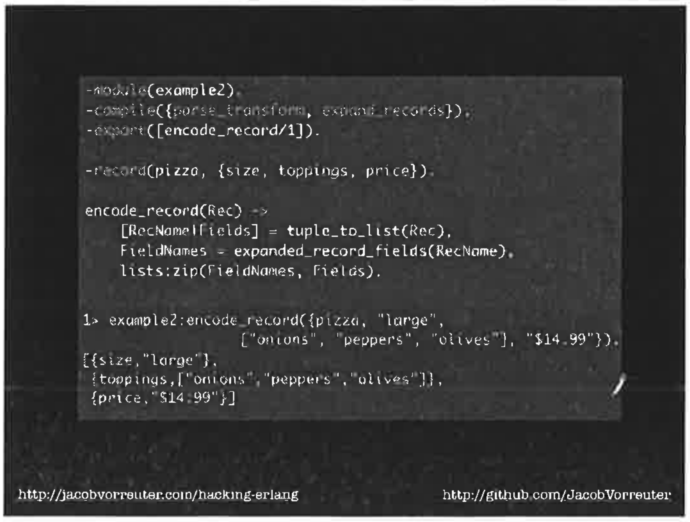
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



a pizza example

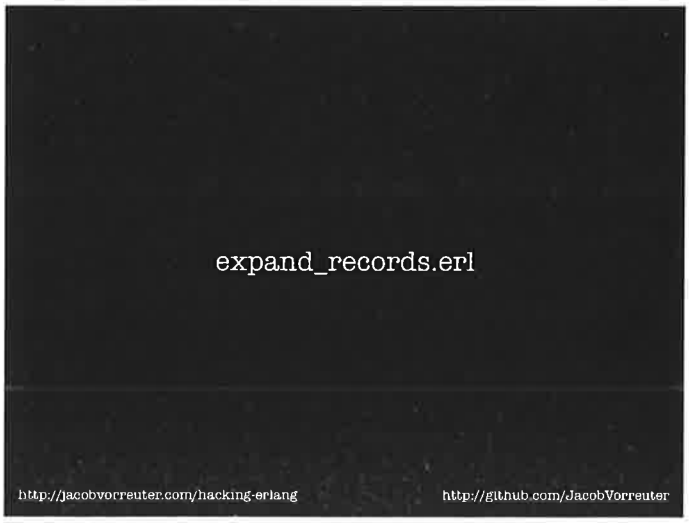
```
-module(example2).
-compile({parse_transform, expand_records}).
-export([encode_record/1]).

-record(pizza, {size, toppings, price}).

encode_record(Rec) ->
    [RecName|Fields] = tuple_to_list(Rec),
    FieldNames = expanded_record_fields(RecName),
    lists:zip(FieldNames, Fields).
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



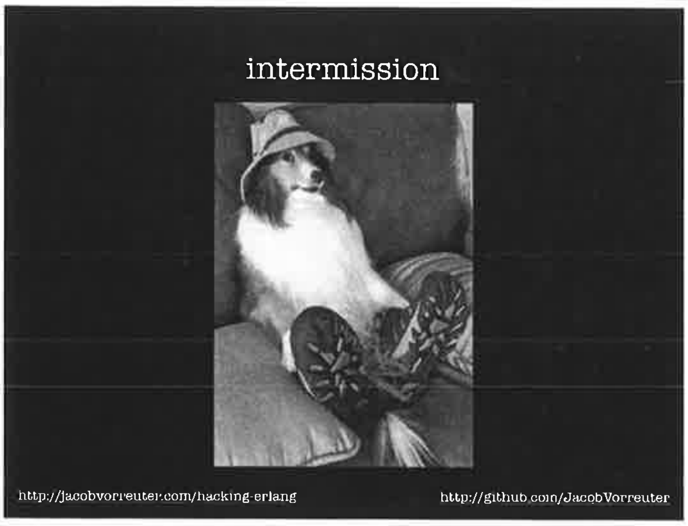
```
-module(example2).
-compile({parse_transform, expand_records}).
-export([encode_record/1]).

-record(pizza, {size, toppings, price}).

encode_record(Rec) ->
    [RecName|Fields] = tuple_to_list(Rec),
    FieldNames = expanded_record_fields(RecName),
    lists:zip(FieldNames, Fields).


1> example2:encode_record({pizza, "large",
                          ["onions", "peppers", "olives"], "$14.99"}).
[{size,"large"},
 {toppings,["onions","peppers","olives"]},
 {price,"$14.99"}]
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

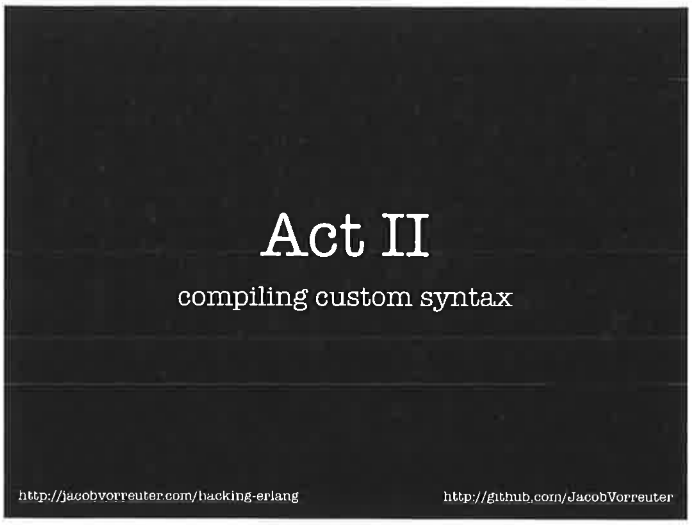


expand_records.erl

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

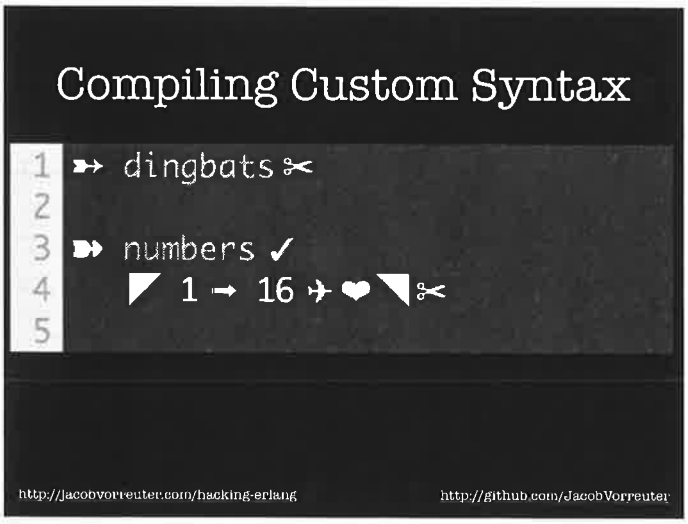


intermission



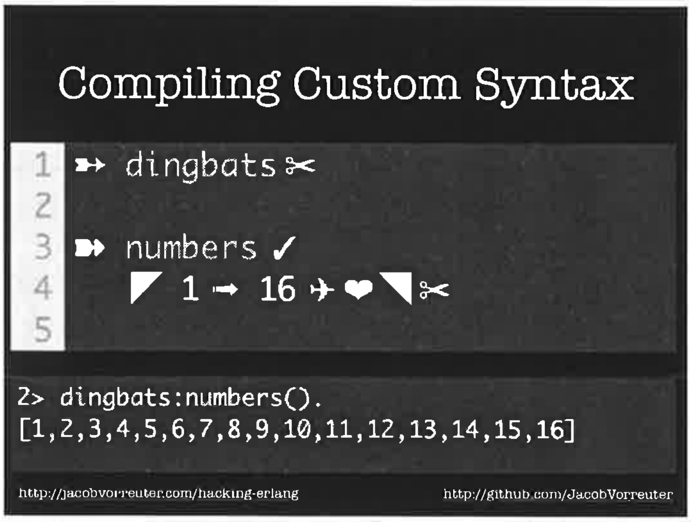
Act II

compiling custom syntax



Compiling Custom Syntax

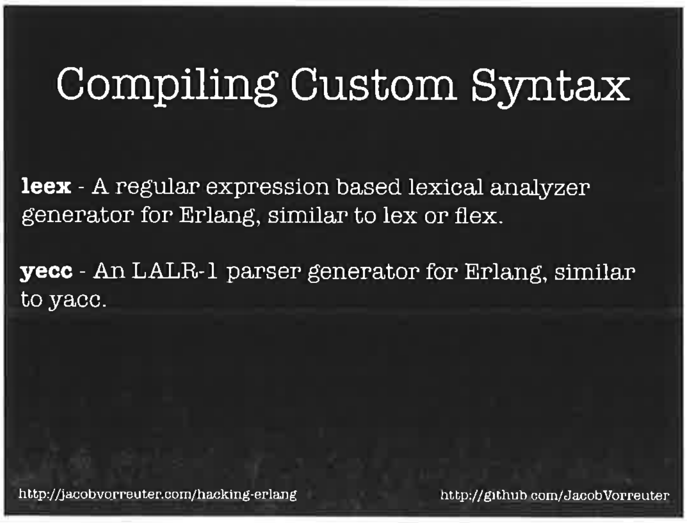
```
➻ dingbats ✂

➽ numbers ✓
   ◤ 1 ➞ 16 ✈ ❤ ◥ ✂

```



Compiling Custom Syntax

```
➻ dingbats ✂

➽ numbers ✓
   ◤ 1 ➞ 16 ✈ ❤ ◥ ✂

```

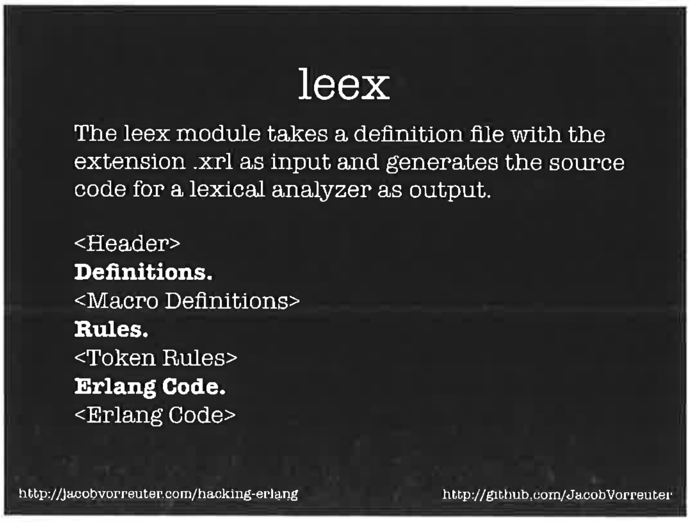
```
2> dingbats:numbers().
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
```



Compiling Custom Syntax

leex - A regular expression based lexical analyzer generator for Erlang, similar to lex or flex.

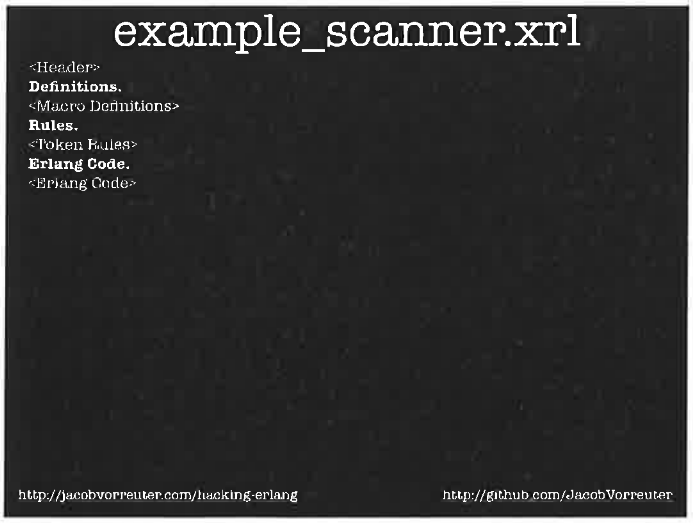
yecc - An LALR-1 parser generator for Erlang, similar to yacc.



leex

The leex module takes a definition file with the extension .xrl as input and generates the source code for a lexical analyzer as output.

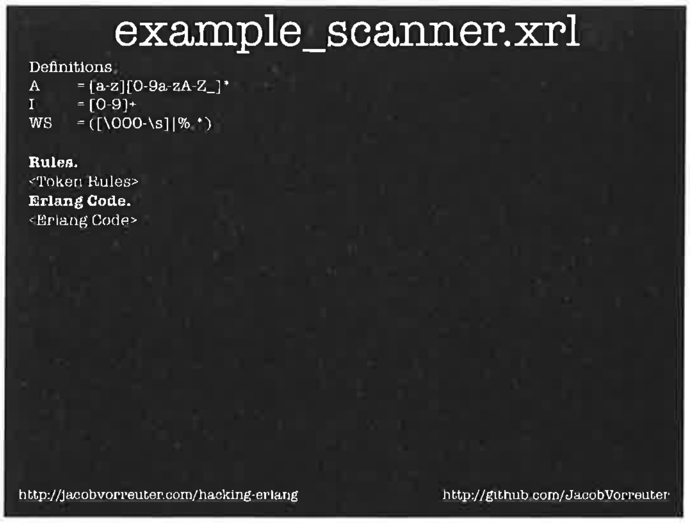
<Header>
Definitions.
<Macro Definitions>
Rules.
<Token Rules>
Erlang Code.
<Erlang Code>



example_scanner.xrl

```
<Header>
Definitions.
<Macro Definitions>
Rules.
<Token Rules>
Erlang Code.
<Erlang Code>
```

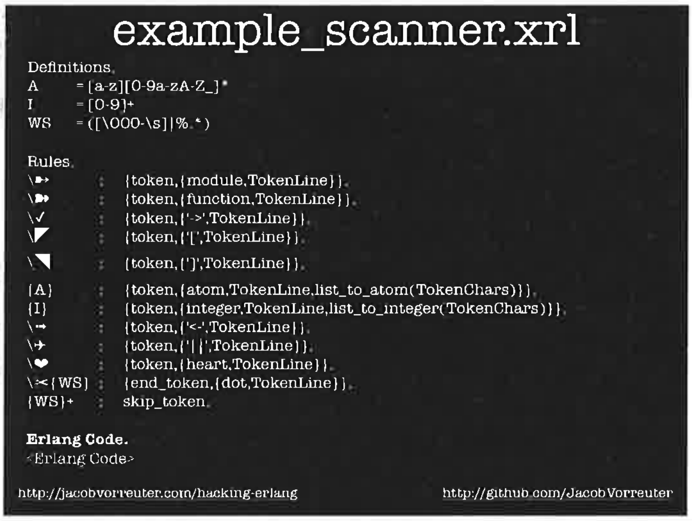
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_scanner.xrl

```
Definitions.
A      = [a-z][0-9a-zA-Z_]*
I      = [0-9]+
WS     = ([\000-\s]|%.*)

Rules.
<Token Rules>
Erlang Code.
<Erlang Code>
```

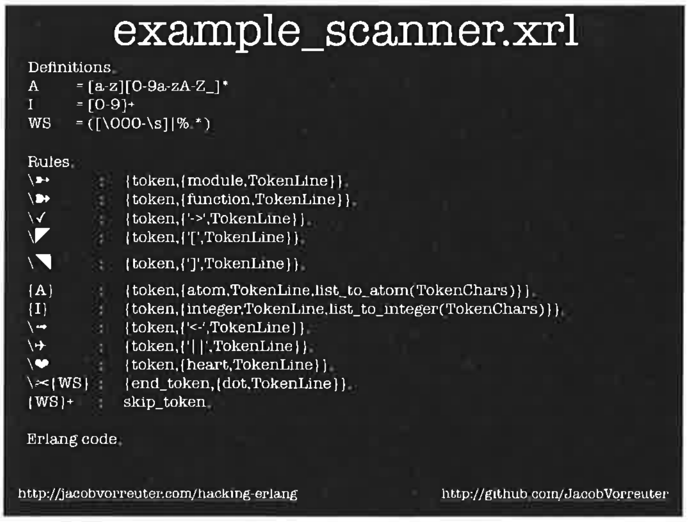
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_scanner.xrl

```
Definitions.
A      = [a-z][0-9a-zA-Z_]*
I      = [0-9]+
WS     = ([\000-\s]|%.*)

Rules.
\➼        :  {token,{module,TokenLine}}.
\➽        :  {token,{function,TokenLine}}.
\✓        :  {token,{'->',TokenLine}}.
\◤        :  {token,{'[',TokenLine}}.
\◥        :  {token,{']',TokenLine}}.
{A}       :  {token,{atom,TokenLine,list_to_atom(TokenChars)}}.
{I}       :  {token,{integer,TokenLine,list_to_integer(TokenChars)}}.
\➙        :  {token,{'<-',TokenLine}}.
\✈        :  {token,{'||',TokenLine}}.
\❤        :  {token,{heart,TokenLine}}.
\✂{WS}    :  {end_token,{dot,TokenLine}}.
{WS}+     :  skip_token.

Erlang Code.
<Erlang Code>
```

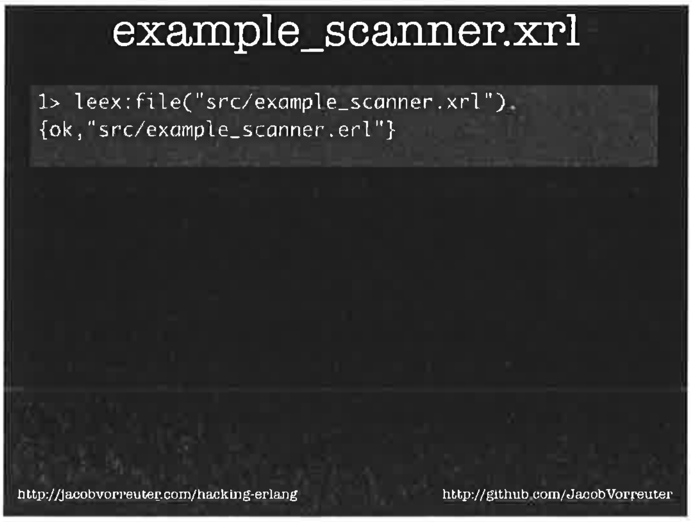
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_scanner.xrl

```
Definitions.
A      = [a-z][0-9a-zA-Z_]*
I      = [0-9]+
WS     = ([\000-\s]|%.*)

Rules.
\➼        :  {token,{module,TokenLine}}.
\➽        :  {token,{function,TokenLine}}.
\✓        :  {token,{'->',TokenLine}}.
\◤        :  {token,{'[',TokenLine}}.
\◥        :  {token,{']',TokenLine}}.
{A}       :  {token,{atom,TokenLine,list_to_atom(TokenChars)}}.
{I}       :  {token,{integer,TokenLine,list_to_integer(TokenChars)}}.
\➙        :  {token,{'<-',TokenLine}}.
\✈        :  {token,{'||',TokenLine}}.
\❤        :  {token,{heart,TokenLine}}.
\✂{WS}    :  {end_token,{dot,TokenLine}}.
{WS}+     :  skip_token.

Erlang code.
```

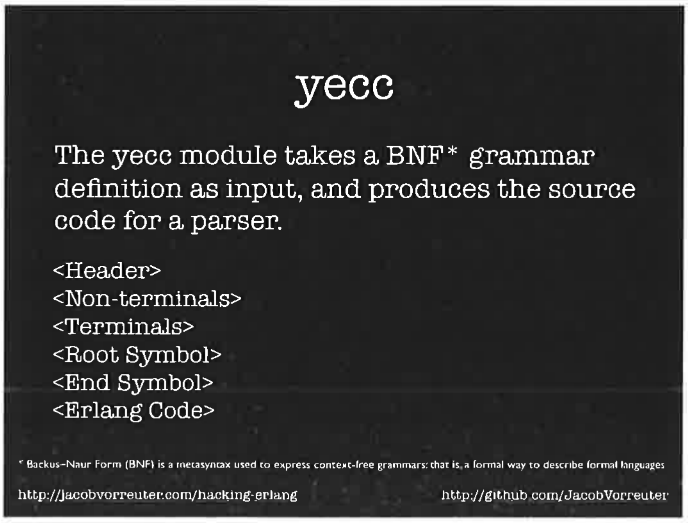
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_scanner.xrl

```
1> leex:file("src/example_scanner.xrl").
{ok,"src/example_scanner.erl"}
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



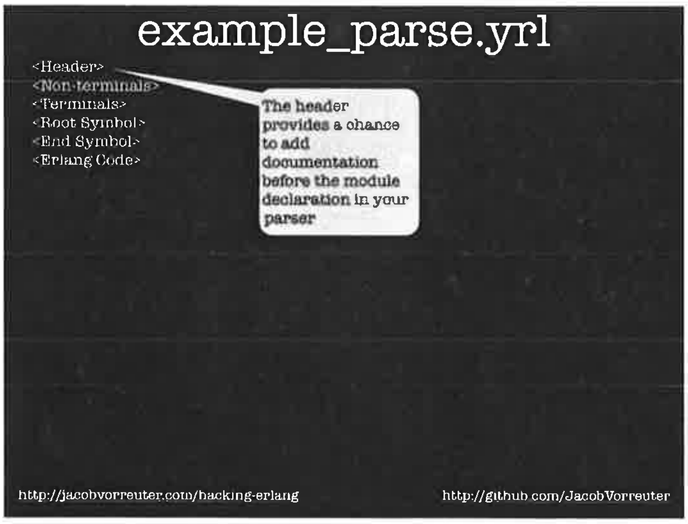
yecc

The yecc module takes a BNF* grammar definition as input, and produces the source code for a parser.

<Header>
<Non-terminals>
<Terminals>
<Root Symbol>
<End Symbol>
<Erlang Code>

* Backus-Naur Form (BNF) is a metasyntax used to express context-free grammars: that is, a formal way to describe formal languages

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

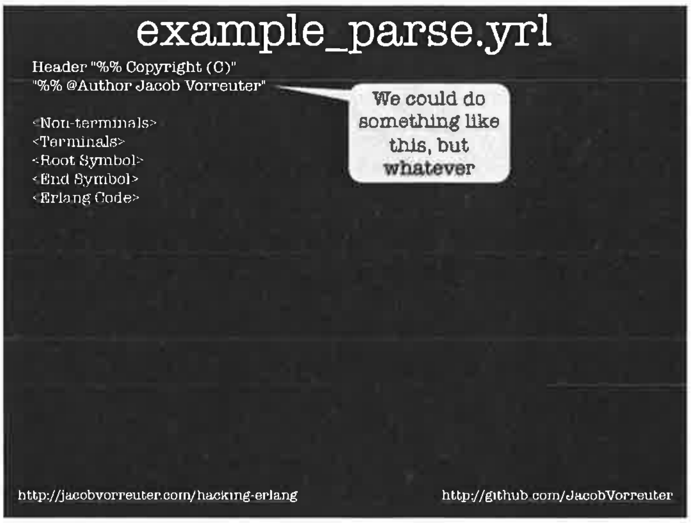


example_parse.yrl

<Header>
<Non-terminals>
<Terminals>
<Root Symbol>
<End Symbol>
<Erlang Code>

The header provides a chance to add documentation before the module declaration in your parser

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



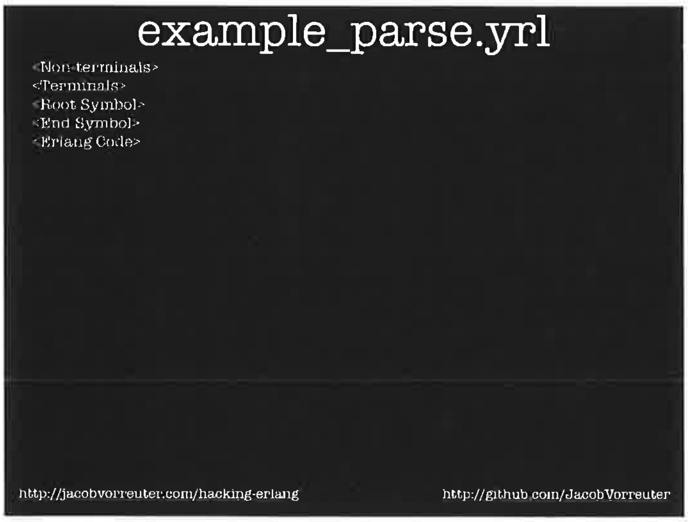
example_parse.yrl

Header "%% Copyright (C)"
"%% @Author Jacob Vorreuter"

<Non-terminals>
<Terminals>
<Root Symbol>
<End Symbol>
<Erlang Code>

We could do something like this, but whatever

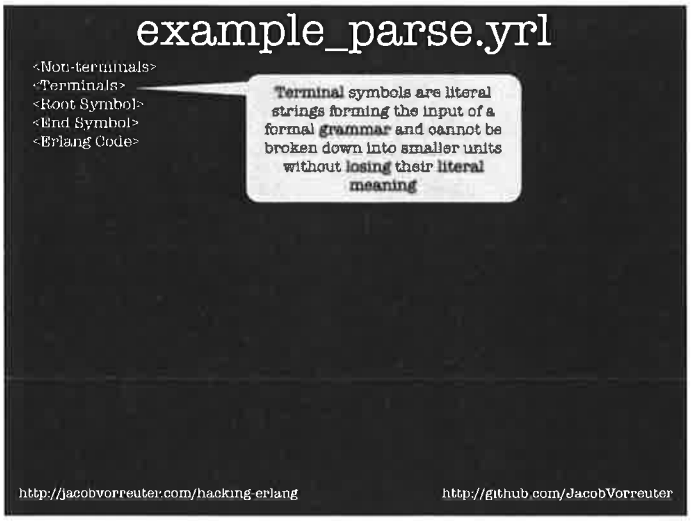
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

<Non-terminals>
<Terminals>
<Root Symbol>
<End Symbol>
<Erlang Code>

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

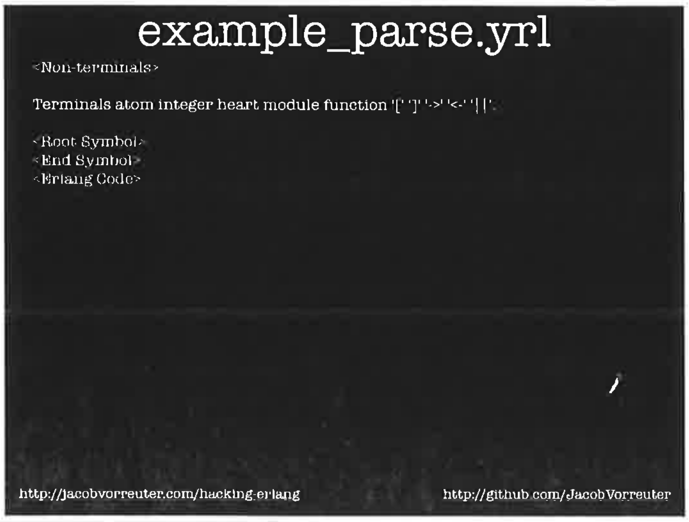


example_parse.yrl

<Non-terminals>
<Terminals>
<Root Symbol>
<End Symbol>
<Erlang Code>

Terminal symbols are literal strings forming the input of a formal grammar and cannot be broken down into smaller units without losing their literal meaning

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



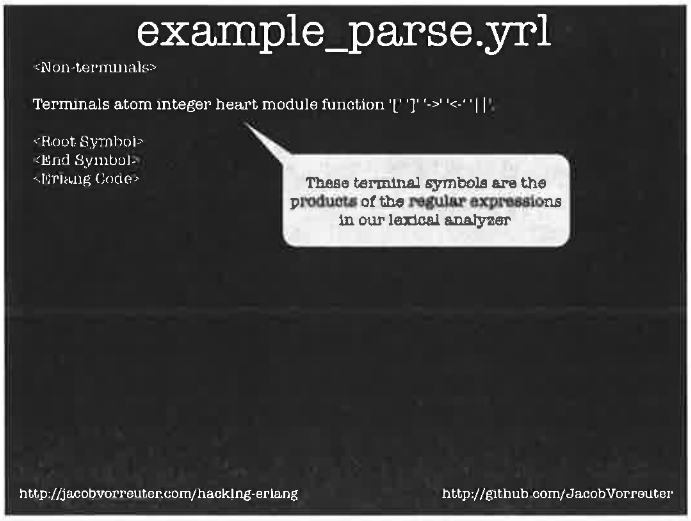
example_parse.yrl

<Non-terminals>

Terminals atom integer heart module function '[' ']' '->' '<-' '||'.

<Root Symbol>
<End Symbol>
<Erlang Code>

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

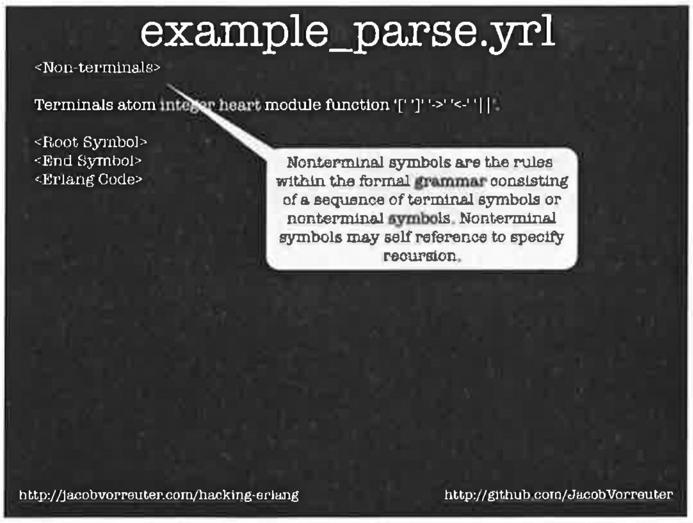
<Non-terminals>

Terminals atom integer heart module function '[' ']' '->' '<-' '||'.

<Root Symbol>
<End Symbol>
<Erlang Code>

These terminal symbols are the products of the regular expressions in our lexical analyzer

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

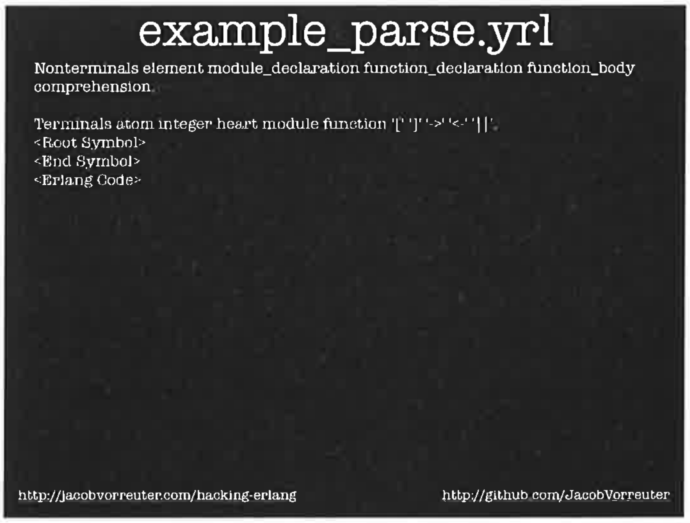


example_parse.yrl

```
<Non-terminals>

Terminals atom integer heart module function '[' ']' '->' '<-' '||'.

<Root Symbol>
<End Symbol>
<Erlang Code>
```

Nonterminal symbols are the rules within the formal grammar consisting of a sequence of terminal symbols or nonterminal symbols. Nonterminal symbols may self reference to specify recursion.

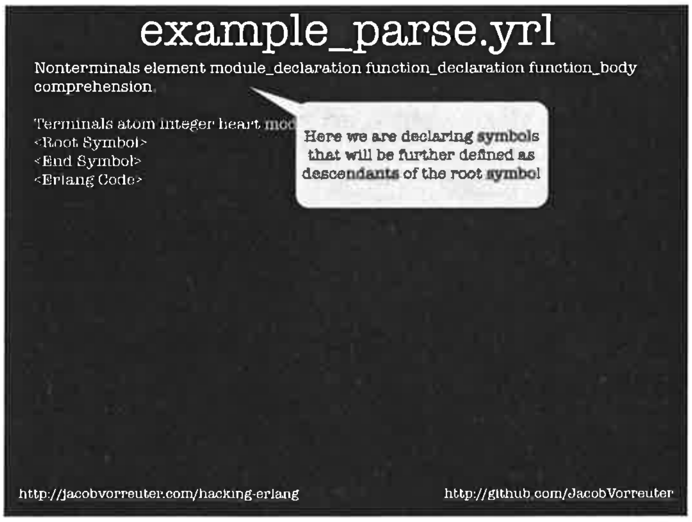
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

```
Nonterminals element module_declaration function_declaration function_body
comprehension.

Terminals atom integer heart module function '[' ']' '->' '<-' '||'.
<Root Symbol>
<End Symbol>
<Erlang Code>
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

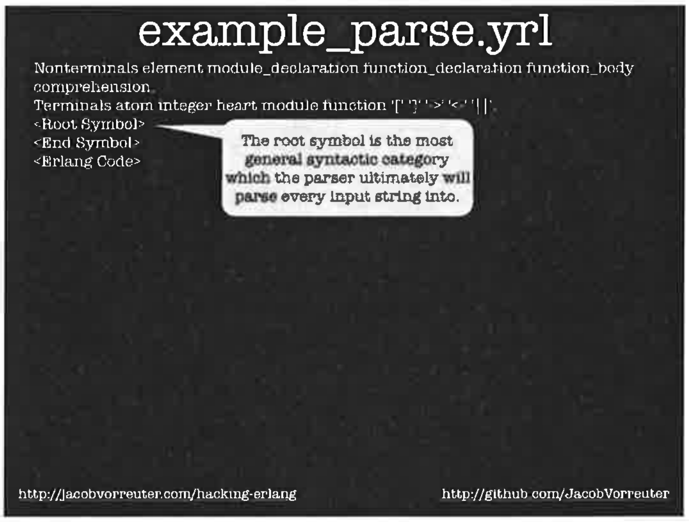


example_parse.yrl

```
Nonterminals element module_declaration function_declaration function_body
comprehension.

Terminals atom integer heart mod
<Root Symbol>
<End Symbol>
<Erlang Code>
```

Here we are declaring symbols that will be further defined as descendants of the root symbol

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

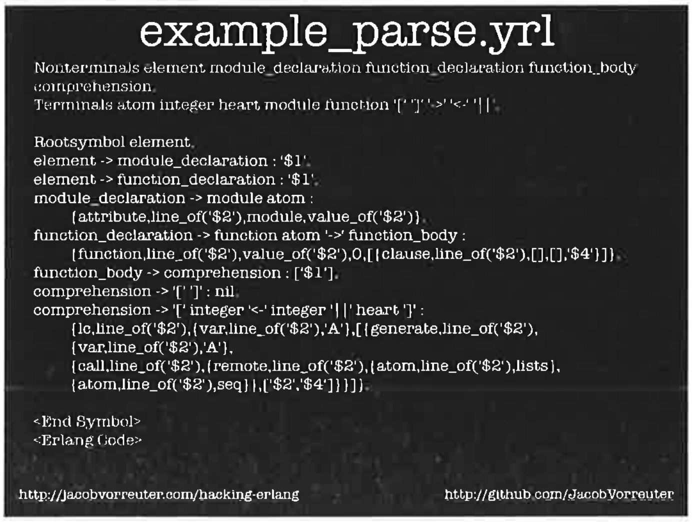


example_parse.yrl

```
Nonterminals element module_declaration function_declaration function_body
comprehension.
Terminals atom integer heart module function '[' ']' '->' '<-' '||'.
<Root Symbol>
<End Symbol>
<Erlang Code>
```

The root symbol is the most general syntactic category which the parser ultimately will parse every input string into.

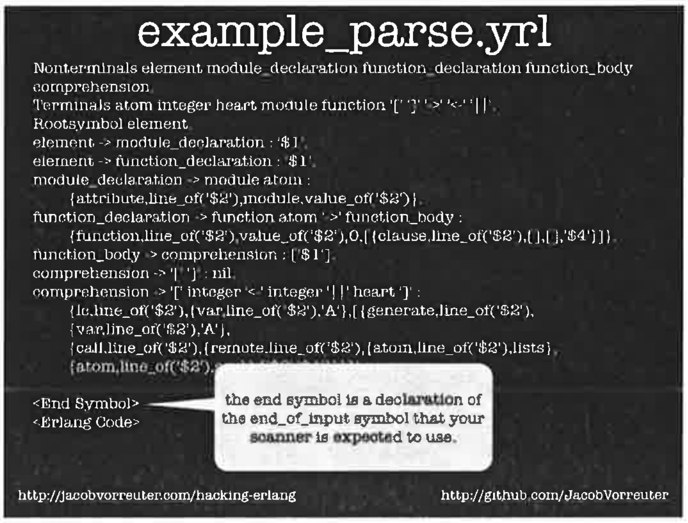
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

```
Nonterminals element module_declaration function_declaration function_body
comprehension.
Terminals atom integer heart module function '[' ']' '->' '<-' '||'.

Rootsymbol element.
element -> module_declaration : '$1'.
element -> function_declaration : '$1'.
module_declaration -> module atom :
        {attribute,line_of('$2'),module,value_of('$2')}.
function_declaration -> function atom '->' function_body :
        {function,line_of('$2'),value_of('$2'),0,[{clause,line_of('$2'),[],[],'$4'}]}.
function_body -> comprehension : ['$1'].
comprehension -> '[' ']' : nil.
comprehension -> '[' integer '<-' integer '||' heart ']' :
        {lc,line_of('$2'),{var,line_of('$2'),'A'},[{generate,line_of('$2'),
        {var,line_of('$2'),'A'},
        {call,line_of('$2'),{remote,line_of('$2'),{atom,line_of('$2'),lists},
        {atom,line_of('$2'),seq}},['$2','$4']}}]}.

<End Symbol>
<Erlang Code>
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

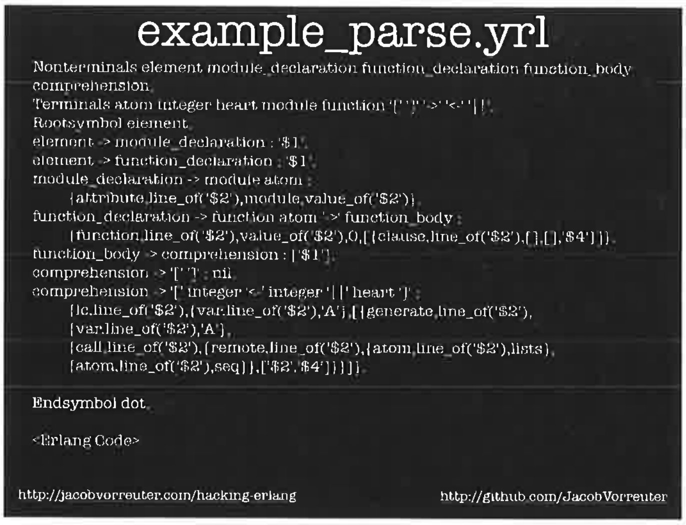


example_parse.yrl

```
Nonterminals element module_declaration function_declaration function_body
comprehension.
Terminals atom integer heart module function '[' ']' '->' '<-' '||'.
Rootsymbol element.
element -> module_declaration : '$1'.
element -> function_declaration : '$1'.
module_declaration -> module atom :
        {attribute,line_of('$2'),module,value_of('$2')}.
function_declaration -> function atom '->' function_body :
        {function,line_of('$2'),value_of('$2'),0,[{clause,line_of('$2'),[],[],'$4'}]}.
function_body -> comprehension : ['$1'].
comprehension -> '[' ']' : nil.
comprehension -> '[' integer '<-' integer '||' heart ']' :
        {lc,line_of('$2'),{var,line_of('$2'),'A'},[{generate,line_of('$2'),
        {var,line_of('$2'),'A'},
        {call,line_of('$2'),{remote,line_of('$2'),{atom,line_of('$2'),lists},
        {atom,line_of('$2'),

<End Symbol>
<Erlang Code>
```

the end symbol is a declaration of the end_of_input symbol that your scanner is expected to use.

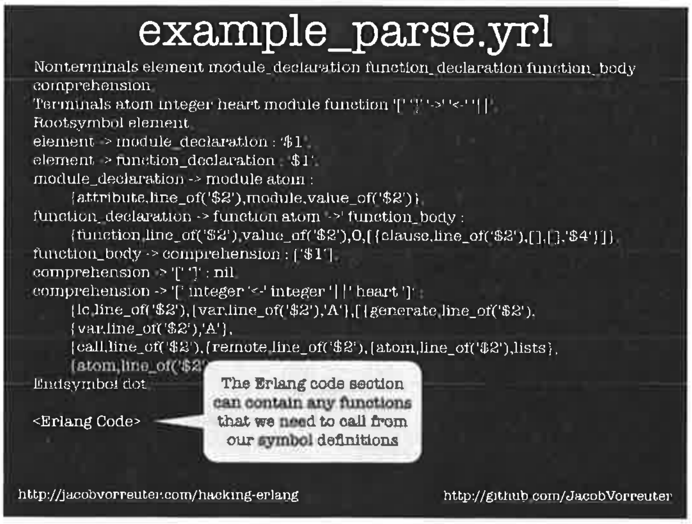
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

```
Nonterminals element module_declaration function_declaration function_body
comprehension.
Terminals atom integer heart module function '[' ']' '->' '<-' '||'.
Rootsymbol element.
element -> module_declaration : '$1'.
element -> function_declaration : '$1'.
module_declaration -> module atom :
        {attribute,line_of('$2'),module,value_of('$2')}.
function_declaration -> function atom '->' function_body :
        {function,line_of('$2'),value_of('$2'),0,[{clause,line_of('$2'),[],[],'$4'}]}.
function_body -> comprehension : ['$1'].
comprehension -> '[' ']' : nil.
comprehension -> '[' integer '<-' integer '||' heart ']' :
        {lc,line_of('$2'),{var,line_of('$2'),'A'},[{generate,line_of('$2'),
        {var,line_of('$2'),'A'},
        {call,line_of('$2'),{remote,line_of('$2'),{atom,line_of('$2'),lists},
        {atom,line_of('$2'),seq}},['$2','$4']}}]}.

Endsymbol dot.

<Erlang Code>
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

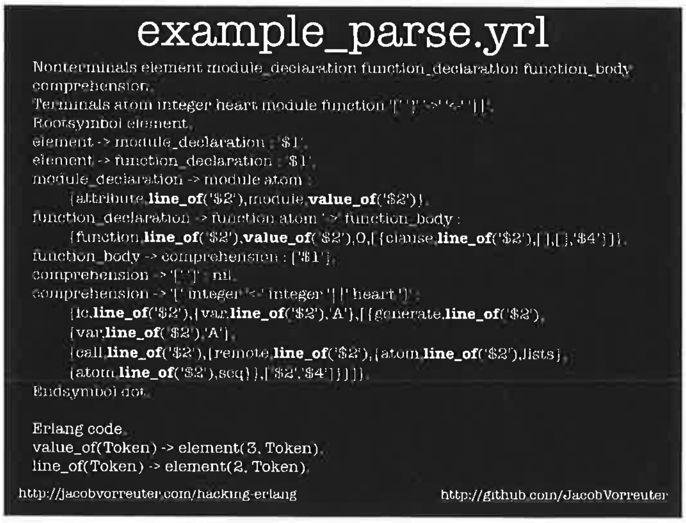


example_parse.yrl

```
Nonterminals element module_declaration function_declaration function_body
comprehension.
Terminals atom integer heart module function '[' ']' '->' '<-' '||'.
Rootsymbol element.
element -> module_declaration : '$1'.
element -> function_declaration : '$1'.
module_declaration -> module atom :
        {attribute,line_of('$2'),module,value_of('$2')}.
function_declaration -> function atom '->' function_body :
        {function,line_of('$2'),value_of('$2'),0,[{clause,line_of('$2'),[],[],'$4'}]}.
function_body -> comprehension : ['$1'].
comprehension -> '[' ']' : nil.
comprehension -> '[' integer '<-' integer '||' heart ']' :
        {lc,line_of('$2'),{var,line_of('$2'),'A'},[{generate,line_of('$2'),
        {var,line_of('$2'),'A'},
        {call,line_of('$2'),{remote,line_of('$2'),{atom,line_of('$2'),lists},
        {atom,line_of('$2'
Endsymbol dot.

<Erlang Code>
```

The Erlang code section can contain any functions that we need to call from our symbol definitions

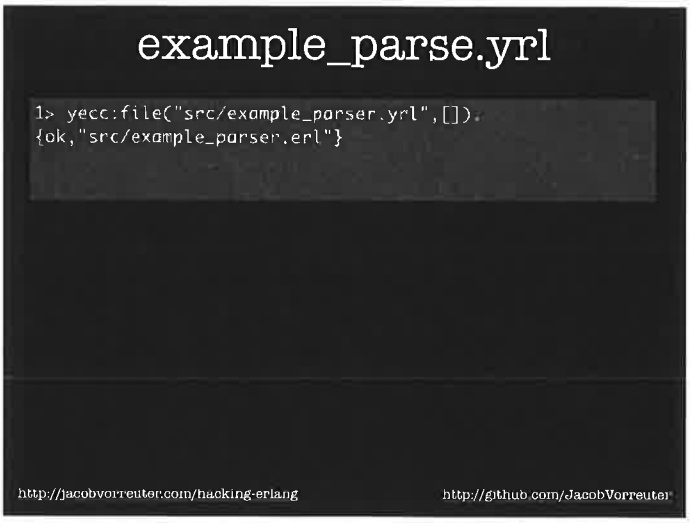
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

```
Nonterminals element module_declaration function_declaration function_body
comprehension.
Terminals atom integer heart module function '[' ']' '->' '<-' '||'.
Rootsymbol element.
element -> module_declaration : '$1'.
element -> function_declaration : '$1'.
module_declaration -> module atom :
        {attribute,line_of('$2'),module,value_of('$2')}.
function_declaration -> function atom '->' function_body :
        {function,line_of('$2'),value_of('$2'),0,[{clause,line_of('$2'),[],[],'$4'}]}.
function_body -> comprehension : ['$1'].
comprehension -> '[' ']' : nil.
comprehension -> '[' integer '<-' integer '||' heart ']' :
        {lc,line_of('$2'),{var,line_of('$2'),'A'},[{generate,line_of('$2'),
        {var,line_of('$2'),'A'},
        {call,line_of('$2'),{remote,line_of('$2'),{atom,line_of('$2'),lists},
        {atom,line_of('$2'),seq}},['$2','$4']}}]}.
Endsymbol dot.

Erlang code.
value_of(Token) -> element(3, Token).
line_of(Token) -> element(2, Token).
```

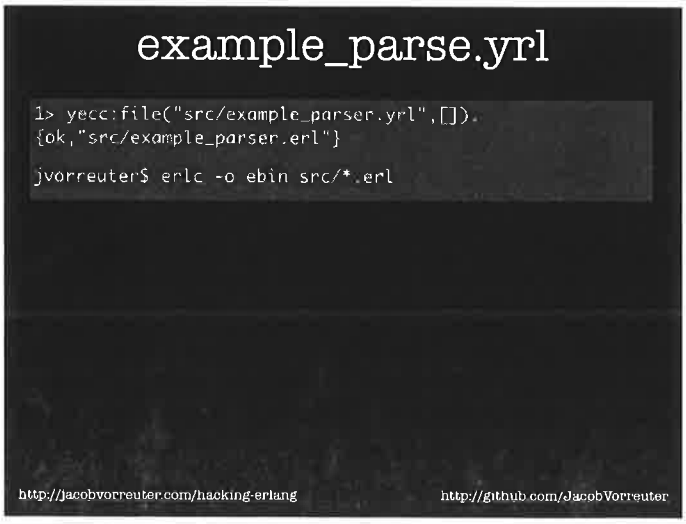
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

```
1> yecc:file("src/example_parser.yrl",[]).
{ok,"src/example_parser.erl"}
```

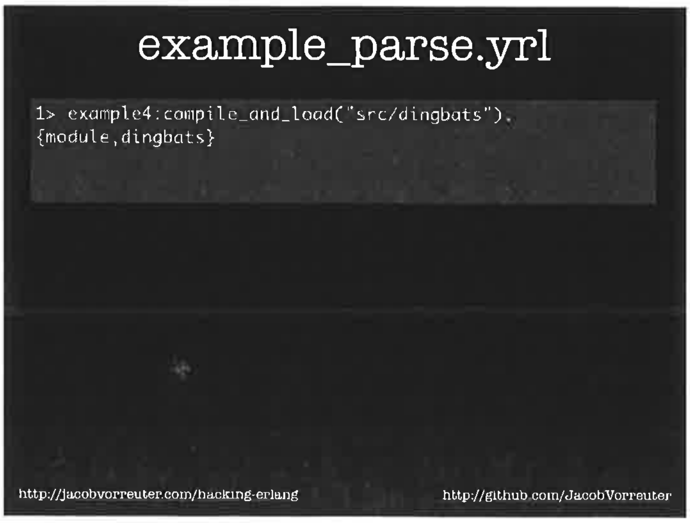
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

```
1> yecc:file("src/example_parser.yrl",[]).
{ok,"src/example_parser.erl"}

jvorreuter$ erlc -o ebin src/*.erl
```

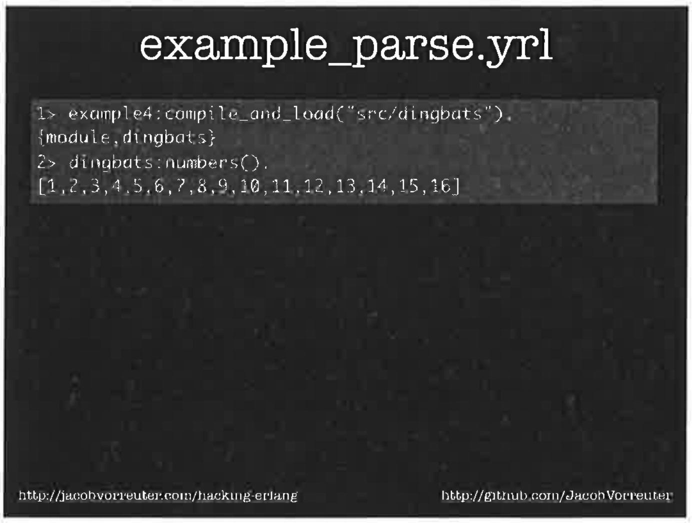
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

```
1> example4:compile_and_load("src/dingbats").
{module,dingbats}
```

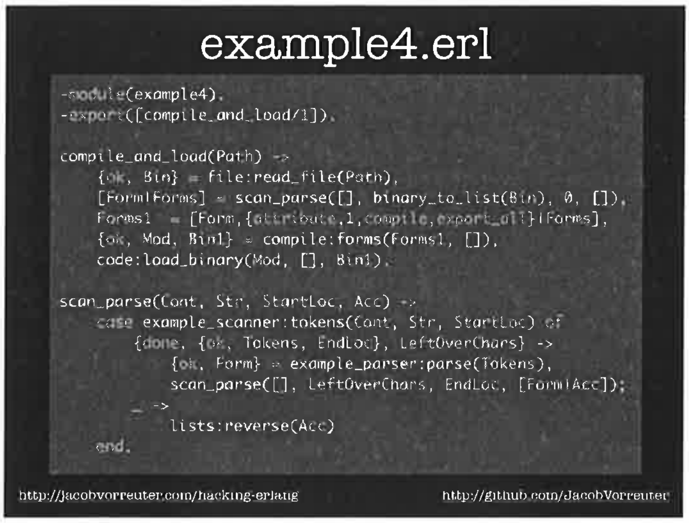
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example_parse.yrl

```
1> example4:compile_and_load("src/dingbats").
{module,dingbats}
2> dingbats:numbers().
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
```

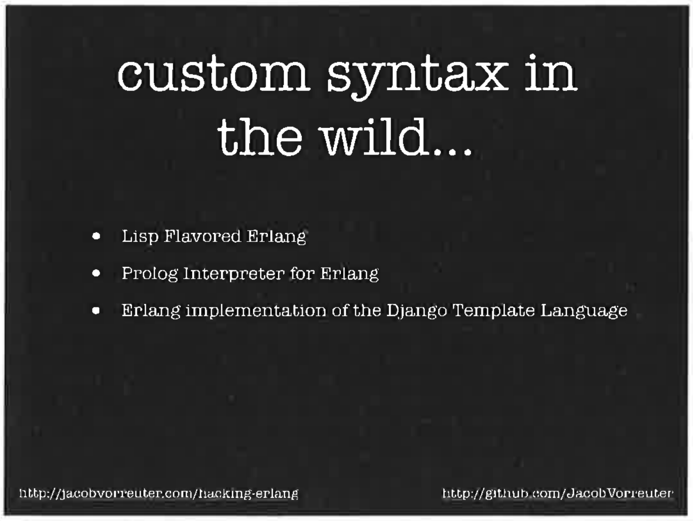
http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



example4.erl

```
-module(example4).
-export([compile_and_load/1]).

compile_and_load(Path) ->
    {ok, Bin} = file:read_file(Path),
    [Form|Forms] = scan_parse([], binary_to_list(Bin), 0, []),
    Forms1  = [Form,{attribute,1,compile,export_all}|Forms],
    {ok, Mod, Bin1} = compile:forms(Forms1, []),
    code:load_binary(Mod, [], Bin1).

scan_parse(Cont, Str, StartLoc, Acc) ->
    case example_scanner:tokens(Cont, Str, StartLoc) of
        {done, {ok, Tokens, EndLoc}, LeftOverChars} ->
            {ok, Form} = example_parser:parse(Tokens),
            scan_parse([], LeftOverChars, EndLoc, [Form|Acc]);
        _ ->
            lists:reverse(Acc)
    end.
```

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter



custom syntax in the wild...

- Lisp Flavored Erlang
- Prolog Interpreter for Erlang
- Erlang implementation of the Django Template Language

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

END

http://jacobvorreuter.com/hacking-erlang http://github.com/JacobVorreuter

Michael Truog
Creator of Cloudi

A Cloud as an Interface

Abstract

Cloudi is a free Erlang based private cloud for efficient processing in C++ to maximize hardware utilization with dynamic load balancing. Cloudi relies on external databases for keeping the work fault-tolerant by preserving the work data. Implementing work for the cloud is as simple as declaring the cloud interface. The presentation provides an introduction to the Cloudi framework.

Biography

Michael received diverse distributed systems experience from Mobile, Online Gaming, Special Effects/Animation, and Government industries that led to an appreciation of Erlang for creating maintainable systems. A focus on real live systems with a pragmatic view on efficiency directed the creation of the Cloudi framework for free private cloud computing.

CloudI

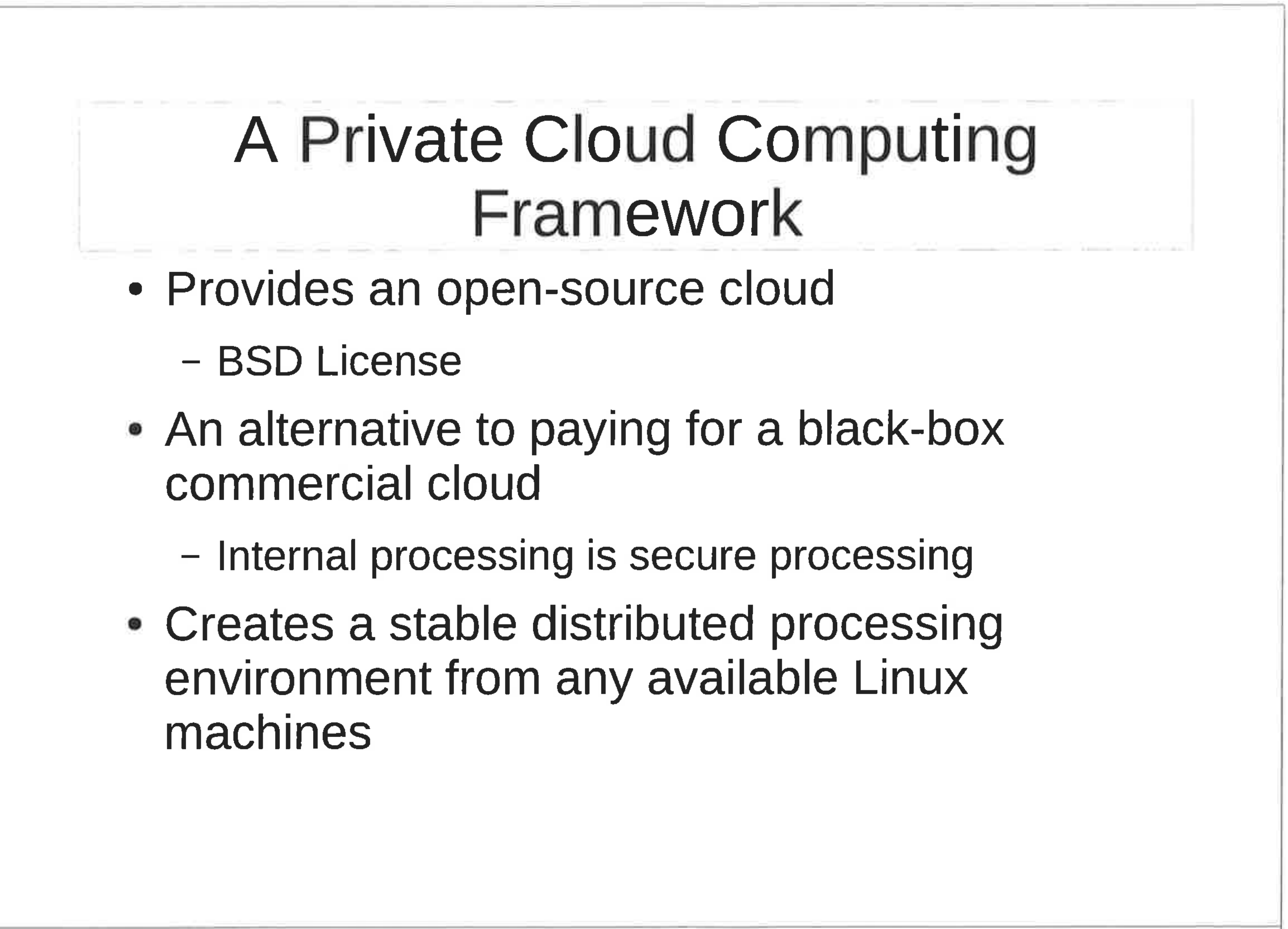
A Cloud as an Interface

Erlang User Conference, Stockholm, Sweden
November 12, 2009

Michael Truog
mjtruog@gmail.com

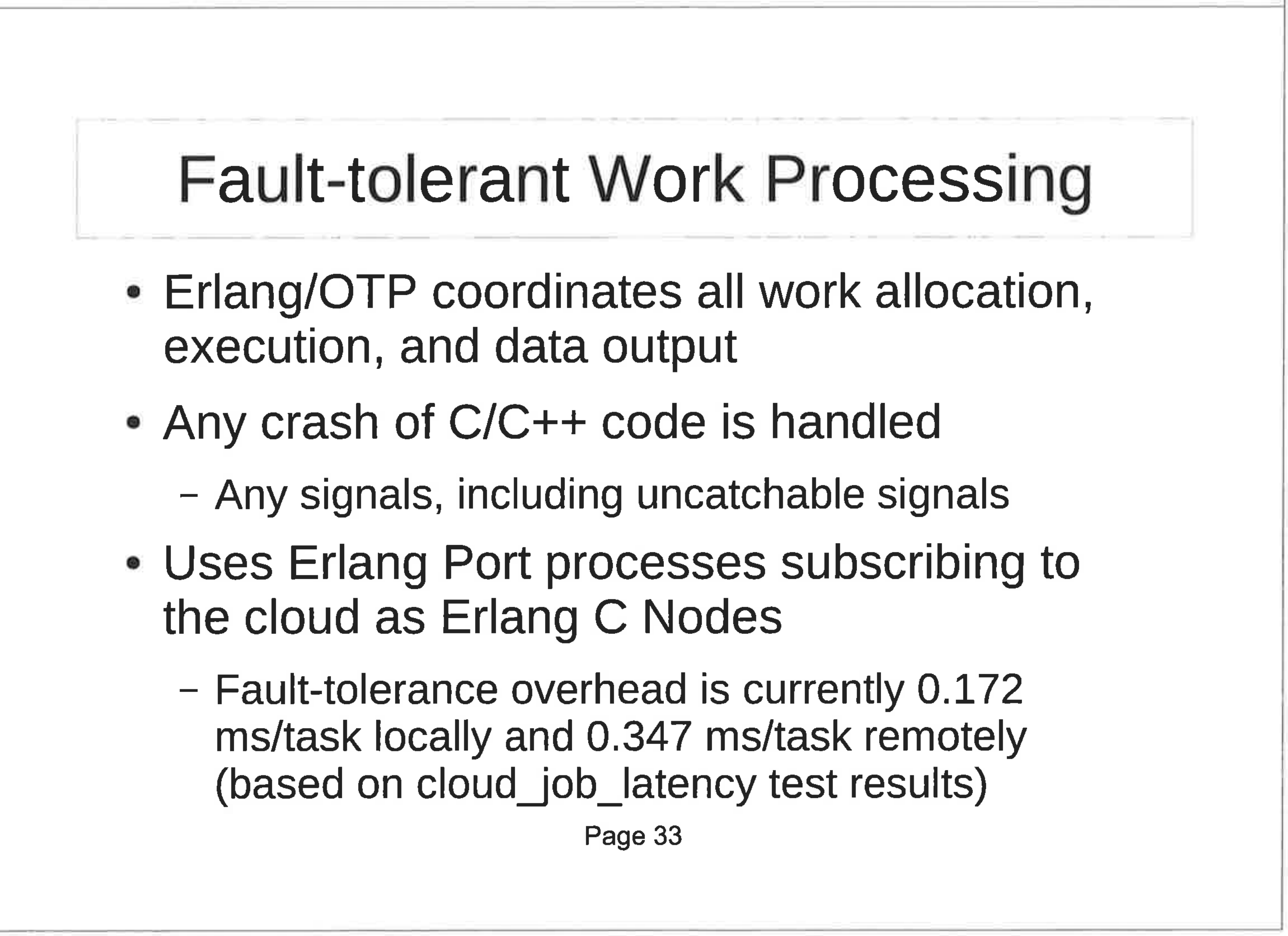
What is Cloudi?

1. Private Cloud Computing Framework
2. Fault-tolerant Work Processing
3. Dynamic Load Balancing and Scheduling
4. Ordered Work Input/Output
5. Distributed Execution of C/C++ Work
6. The Future



A Private Cloud Computing Framework

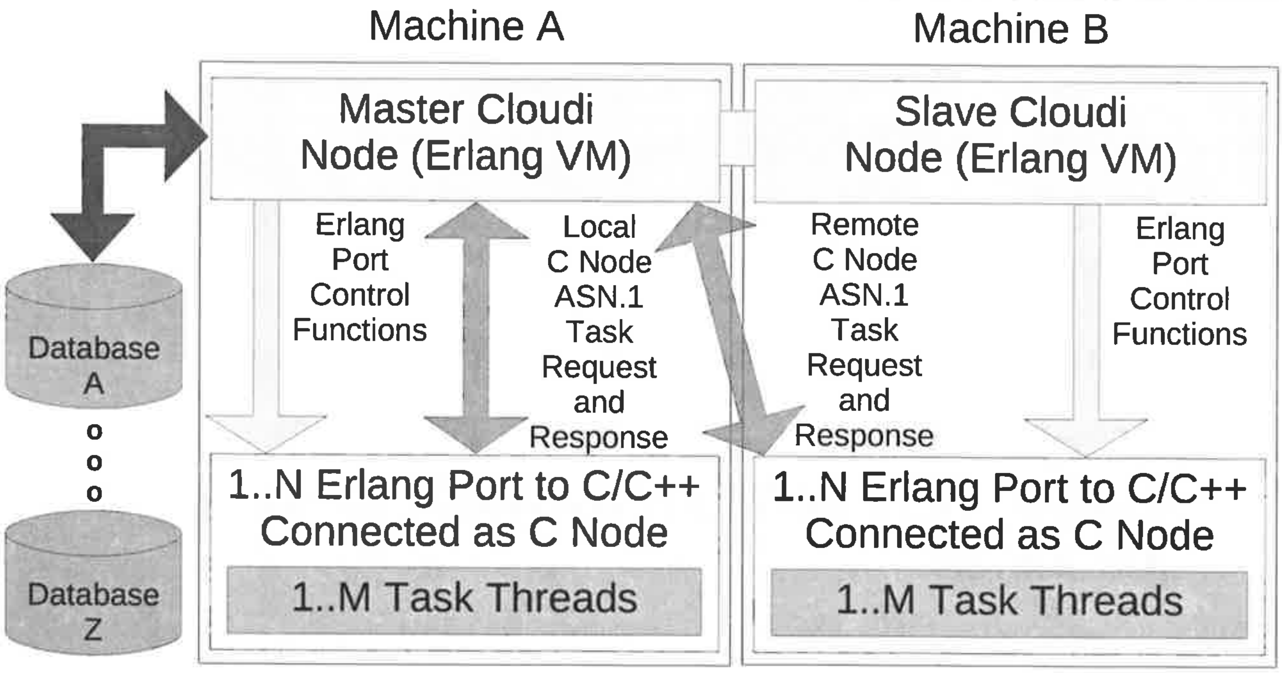
- Provides an open-source cloud
  - BSD License
- An alternative to paying for a black-box commercial cloud
  - Internal processing is secure processing
- Creates a stable distributed processing environment from any available Linux machines



Fault-tolerant Work Processing

- Erlang/OTP coordinates all work allocation, execution, and data output
- Any crash of C/C++ code is handled
  - Any signals, including uncatchable signals
- Uses Erlang Port processes subscribing to the cloud as Erlang C Nodes
  - Fault-tolerance overhead is currently 0.172 ms/task locally and 0.347 ms/task remotely (based on cloud_job_latency test results)

Fault-tolerant Work Processing . . .



Machine A

Machine B

Master Cloudi Node (Erlang VM)

Slave Cloudi Node (Erlang VM)

Database A

o o o

Database Z

Erlang Port Control Functions

Local C Node ASN.1 Task Request and Response

Remote C Node ASN.1 Task Request and Response

Erlang Port Control Functions

1..N Erlang Port to C/C++ Connected as C Node

1..M Task Threads

1..N Erlang Port to C/C++ Connected as C Node

1..M Task Threads

Dynamic Load Balancing and Scheduling

- Workers are ideally stateless and form a pool of workers in the cloud
- Cloudi adjusts the task size based on the task execution time that is requested
  - Convergence is slow to avoid problems with unstable work processing
- Cloudi verifies that work is loaded
  - During work allocation
  - After node reconnection

Ordered Work Input/Output

- The Erlang work module enforces an order on the work task input
- Cloudi maintains the task input order when collecting output so data is stored in the same order
- Work processing is paused when excessive data accumulation occurs

Distributed Execution of C/C++ Work

- One “do_work” function is required in a dynamic library for the C/C++ work
  - Loaded when Cloudi requests it
- Six Erlang functions within the work module provide work task specification
  - The functions define the task size (float value in range (0..1)) and task data (binary data)
- Any Erlang data module can handle output
  - PostgreSQL, MySQL, memcached, Tokyo Tyrant

The Future

- Replicated Cloudi instances can be used for failover (needs management application)
  - Failover uses separate epmd processes for local name registration
- More databases will be supported
- More fault-tolerance testing
- Download Cloudi @ http://cloudi.org/
  - Version 0.0.8 alpha is now available!

Questions?

Discodex: intuitive data indexing

Disco combines the strengths of Erlang and Python to enable rapid development of massively parallel computational pipelines. Disco implements the MapReduce framework, making it a powerful platform for doing distributed computing on immense datasets. The first step to building a system driven by data, is indexing the data in such a way that it is accessible in logarithmic or constant time. Such random access is crucial for building online systems, but also valuable in optimizing many other applications which rely upon lookups into the data. `Discodex` builds on top of Disco,abstracting away some of the most common operations for organizing piles of raw data into distributed, append-only indices and querying them. By adopting erlang-style immutability of data structures, itis possible to index and query billions of data items efficiently. Discodex adopts a similar strategy to Disco in achieving this goal: making the interface so embarrassingly simple and intuitive, that development time is never an excuse for not building an index. In this talk we discuss the architecture of this awesome, open-source tool (with Erlang at its heart), and how to use it. We also provide a real-world example of using Discodex for data insight at Nokia, and the reason we built it in the first place.

Ville Tuulos

Erlang hacker at Nokia Research and initiator of the Disco Project

Ville Tuulos is a researcher with Nokia Research in Palo Alto. He has been working with large data sets since 1999, building solutions for statistical information retrieval. After several misguided attempts to orchestrate highly distributed systems in C and Python, he found Erlang in 2006. He is also co-author of the book "Mobile Python - Rapid application development on the mobile platform". In 2007 he started to build Disco, an Erlang / Python implementation of the Map/Reduce framework for distributed computing. Disco is now used by Nokia and others for quick prototyping of data-intensive software, using hundreds of gigabytes of real-world data.

Jared Flatow

Erlang Hacker at the Nokia Research Center

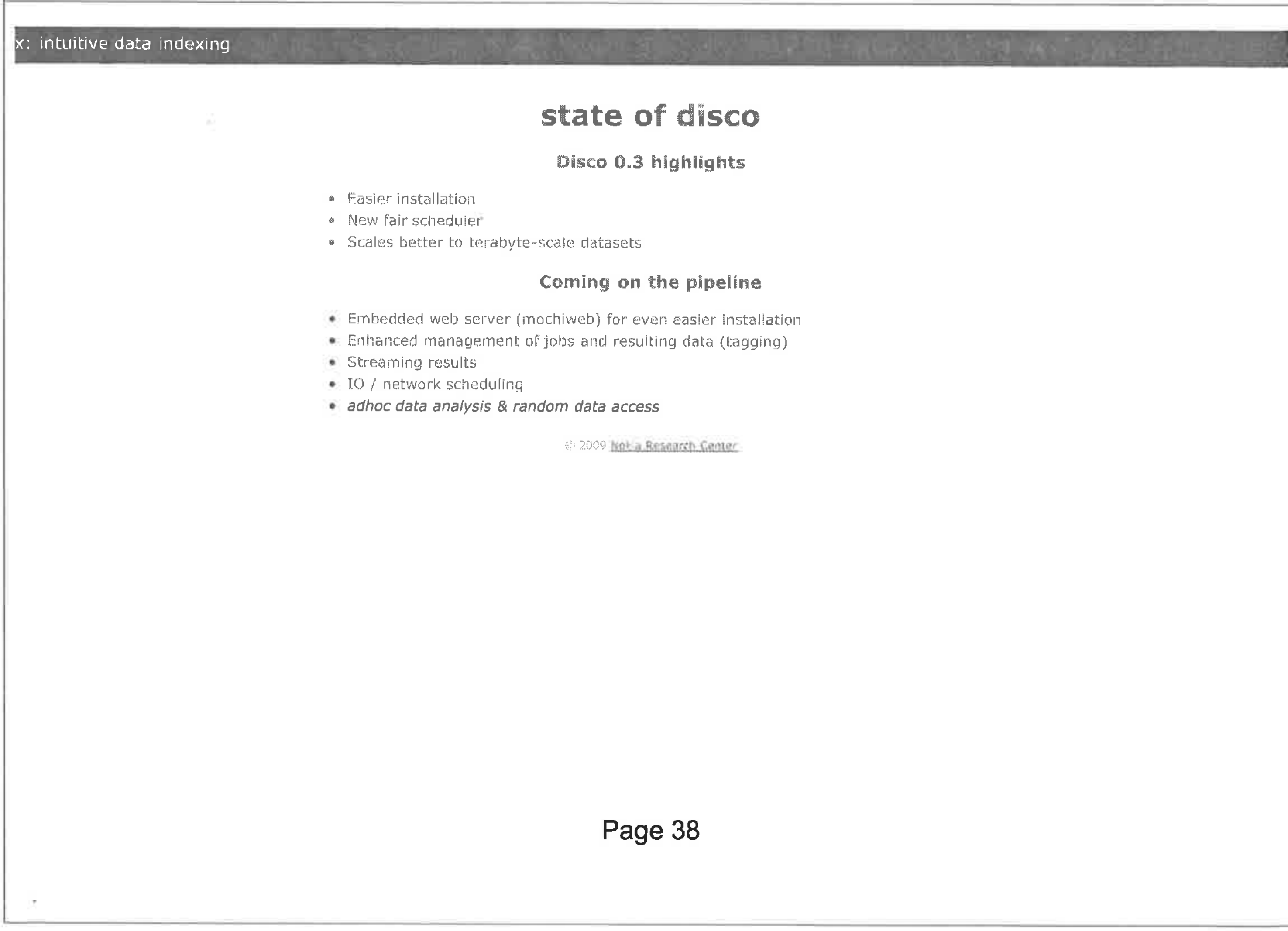
Jared Flatow is an engineer at the Nokia Research Center in Palo Alto. Prior to joining the NRC in 2009, Jared was tackling large-data problems in the field of bioinformatics at the Northwestern University Biomedical Informatics Center in Chicago. Jared was among the first to apply Map/Reduce to problems in bioinformatics, presenting his work using Hadoop at the Next-Generation Sequencing Data Analysis conference in 2008. Jared long-dreamed of rewriting a Map/Reduce framework in Python, but it was not until discovering Disco that he realized the elegance of combining the strengths of both Erlang and Python towards that end. Since then, Jared has become a contributor to Disco, and co-architect of the simple and scalable Discodex data- indexing pipeline. His recent efforts have been aimed at using Discodex to achieve a full-stack massive-scale data visualization pipeline with his colleagues at the Nokia Research Center.

x: intuitive data indexing

discodex: intuitive data indexing

Erlang User Conference, Stockholm, 2009

Jared Flatow
Ville Tuulos



x: intuitive data indexing

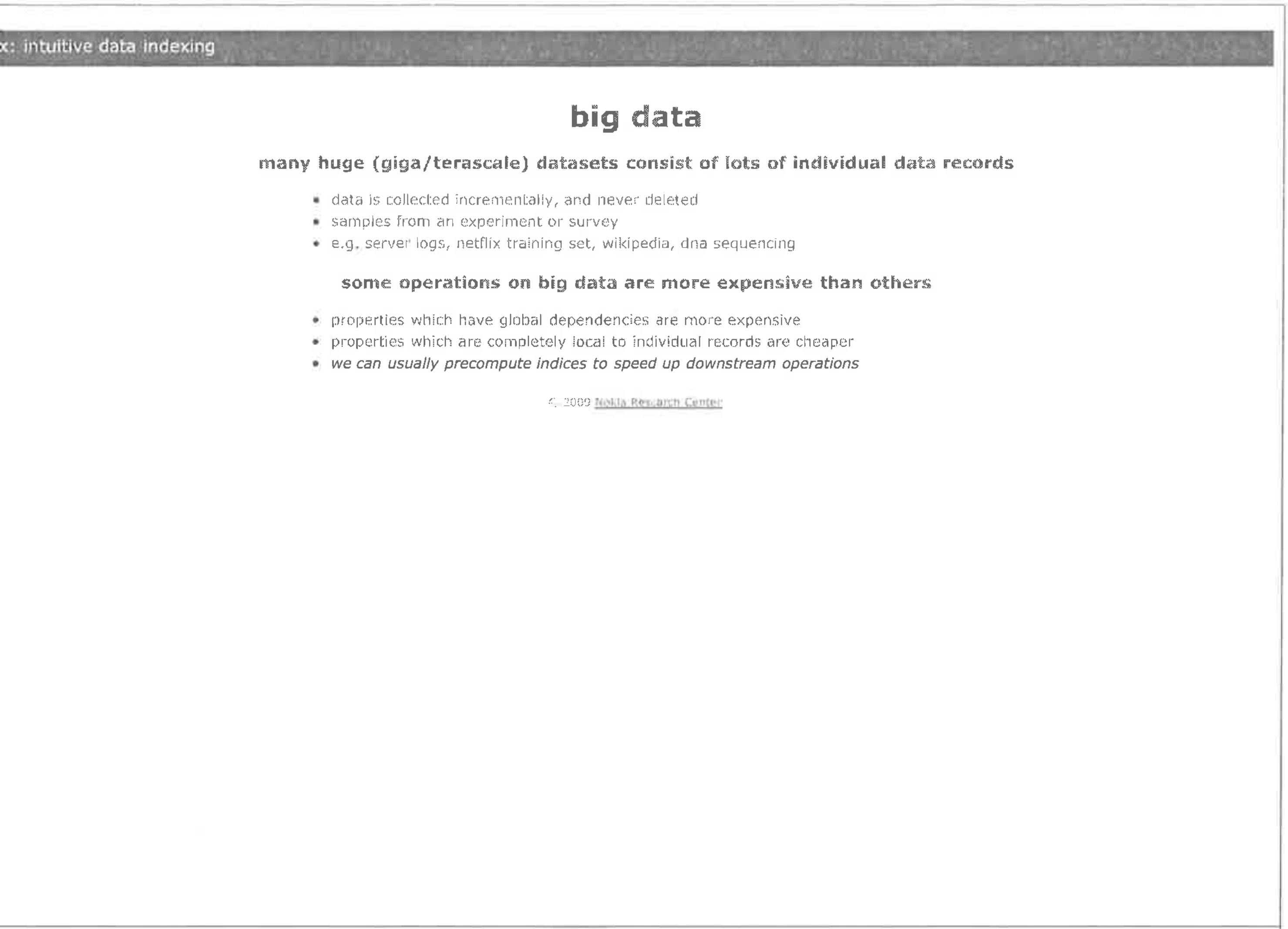
state of disco

Disco 0.3 highlights

- Easier installation
- New fair scheduler
- Scales better to terabyte-scale datasets

Coming on the pipeline

- Embedded web server (mochiweb) for even easier installation
- Enhanced management of jobs and resulting data (tagging)
- Streaming results
- IO / network scheduling
- *adhoc data analysis & random data access*



x: intuitive data indexing

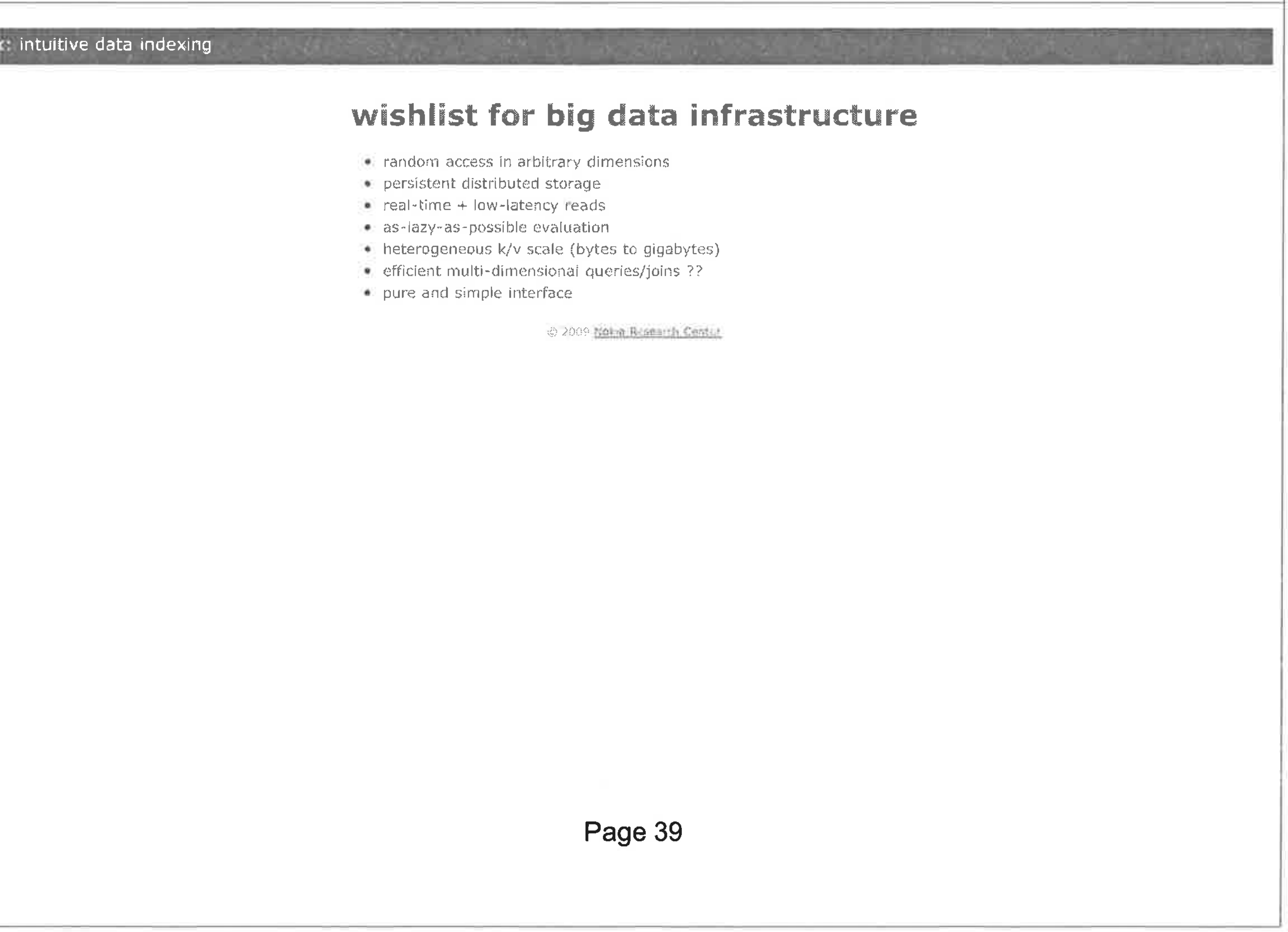
big data

many huge (giga/terascale) datasets consist of lots of individual data records

- data is collected incrementally, and never deleted
- samples from an experiment or survey
- e.g. server logs, netflix training set, wikipedia, dna sequencing

some operations on big data are more expensive than others

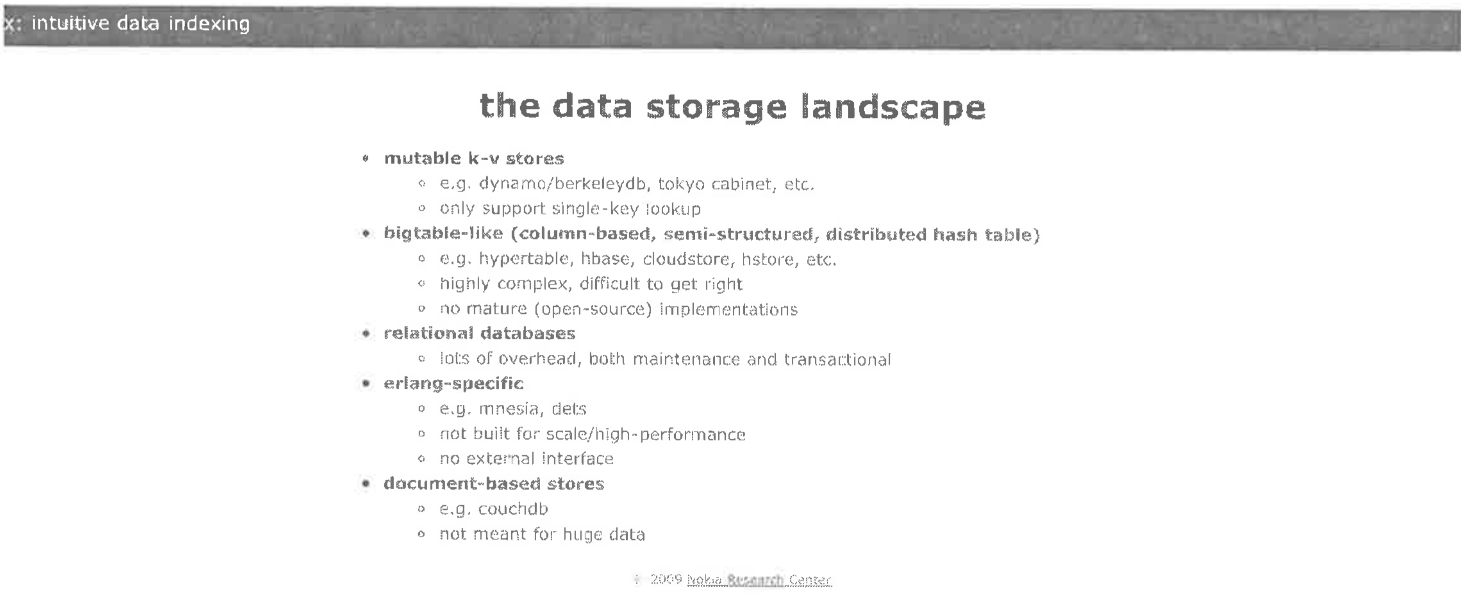
- properties which have global dependencies are more expensive
- properties which are completely local to individual records are cheaper
- *we can usually precompute indices to speed up downstream operations*



: intuitive data indexing

wishlist for big data infrastructure

- random access in arbitrary dimensions
- persistent distributed storage
- real-time + low-latency reads
- as-lazy-as-possible evaluation
- heterogeneous k/v scale (bytes to gigabytes)
- efficient multi-dimensional queries/joins ??
- pure and simple interface

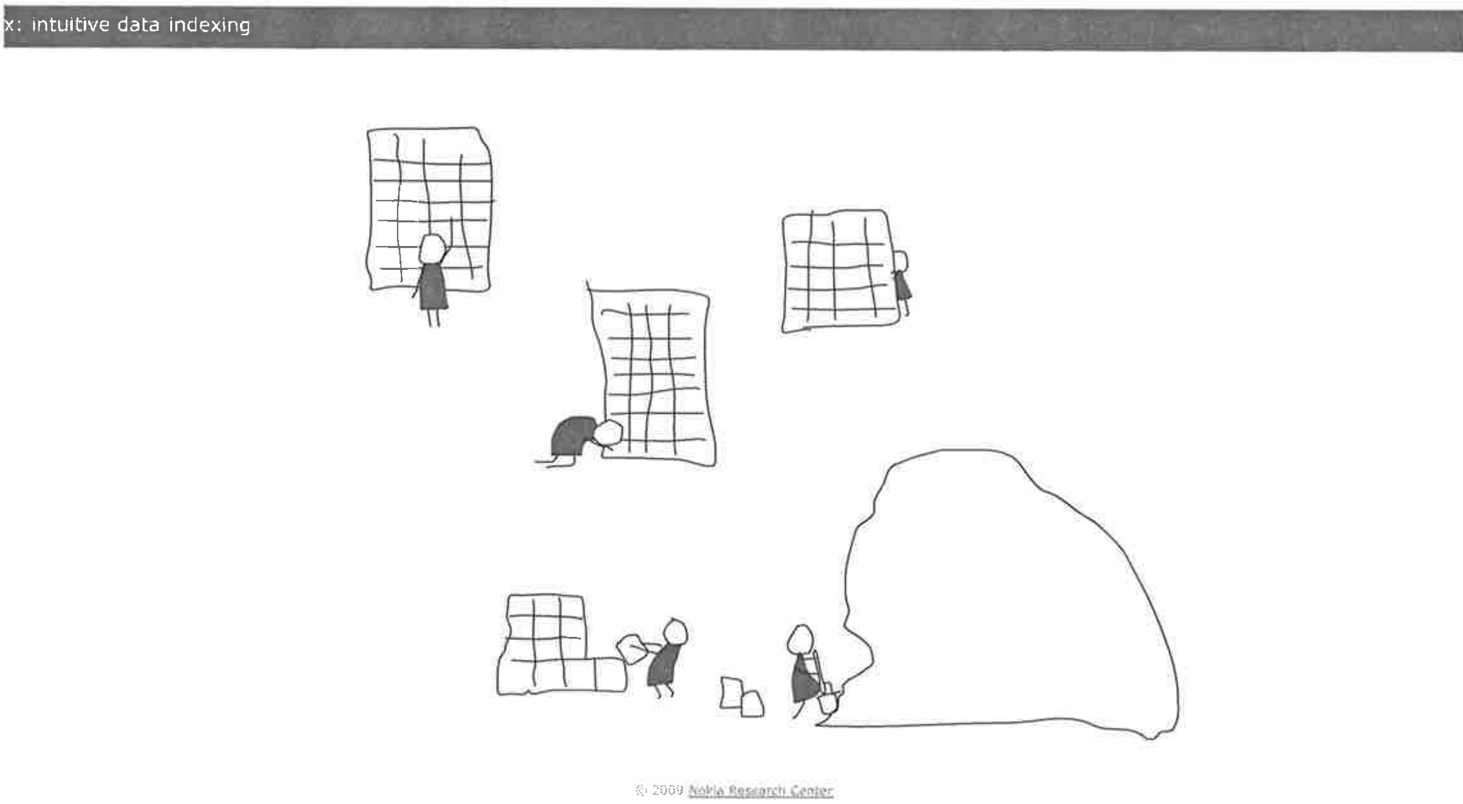


x: intuitive data indexing

the data storage landscape

- **mutable k-v stores**
  - e.g. dynamo/berkeleydb, tokyo cabinet, etc.
  - only support single-key lookup
- **bigtable-like (column-based, semi-structured, distributed hash table)**
  - e.g. hypertable, hbase, cloudstore, hstore, etc.
  - highly complex, difficult to get right
  - no mature (open-source) implementations
- **relational databases**
  - lots of overhead, both maintenance and transactional
- **erlang-specific**
  - e.g. mnesia, dets
  - not built for scale/high-performance
  - no external interface
- **document-based stores**
  - e.g. couchdb
  - not meant for huge data

© 2009 Nokia Research Center



x: intuitive data indexing

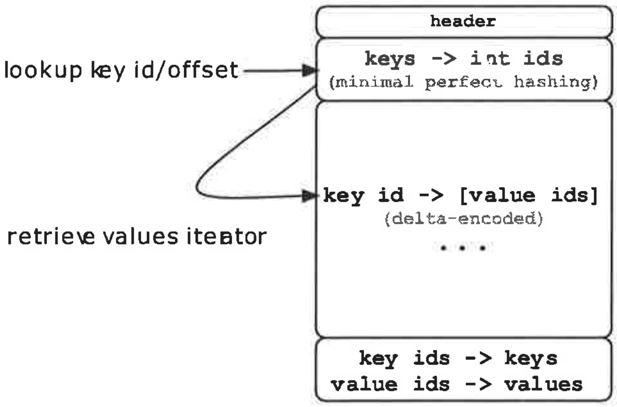
© 2009 Nokia Research Center

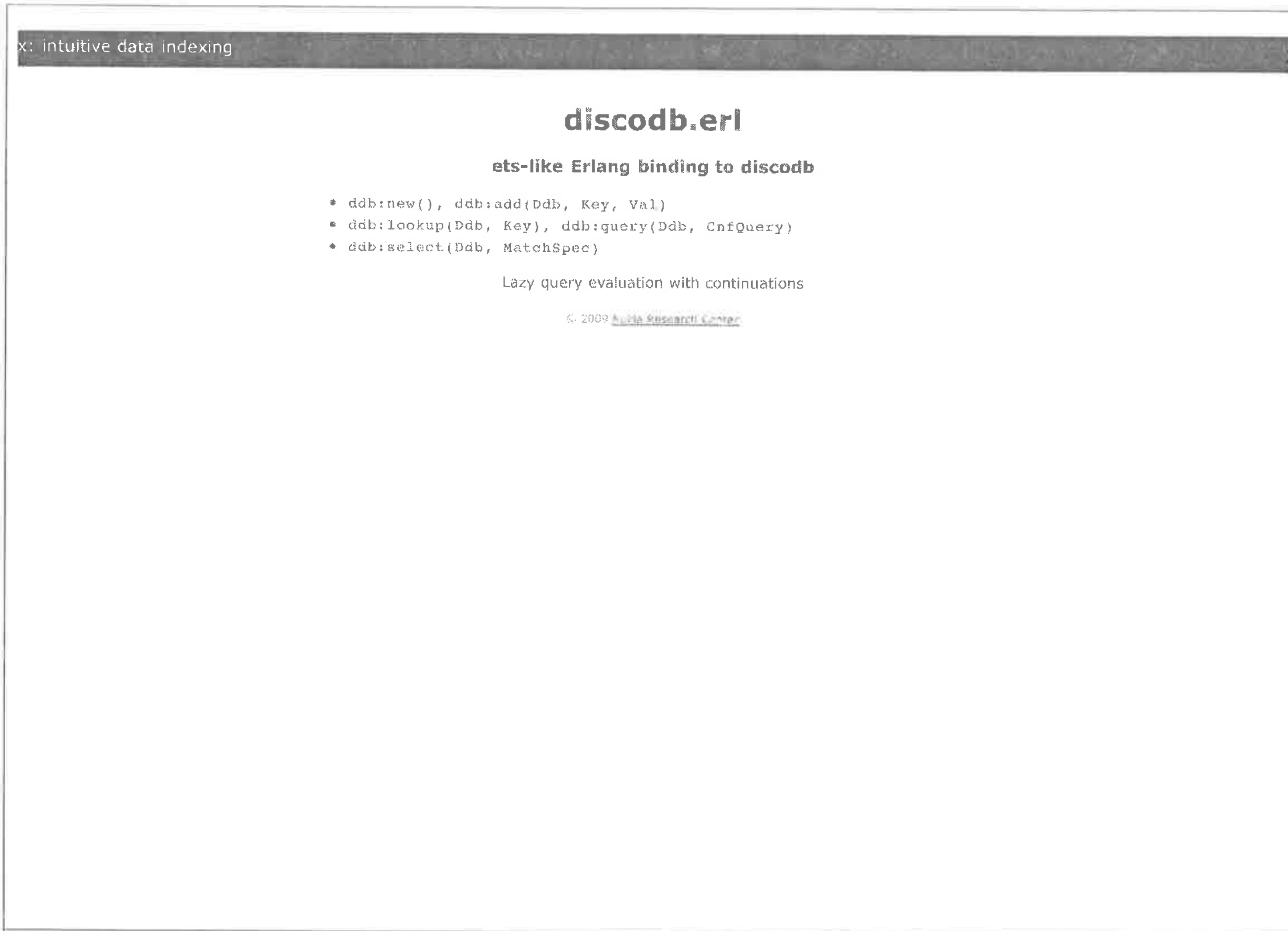
discodb

- low-level C data structure
- maps key -> multiset(values)
- immutable + persistent: write once to a file
- Python/erlang wrappers: api = dictionary + cnf

discodb format

designed for lightning fast random-acces





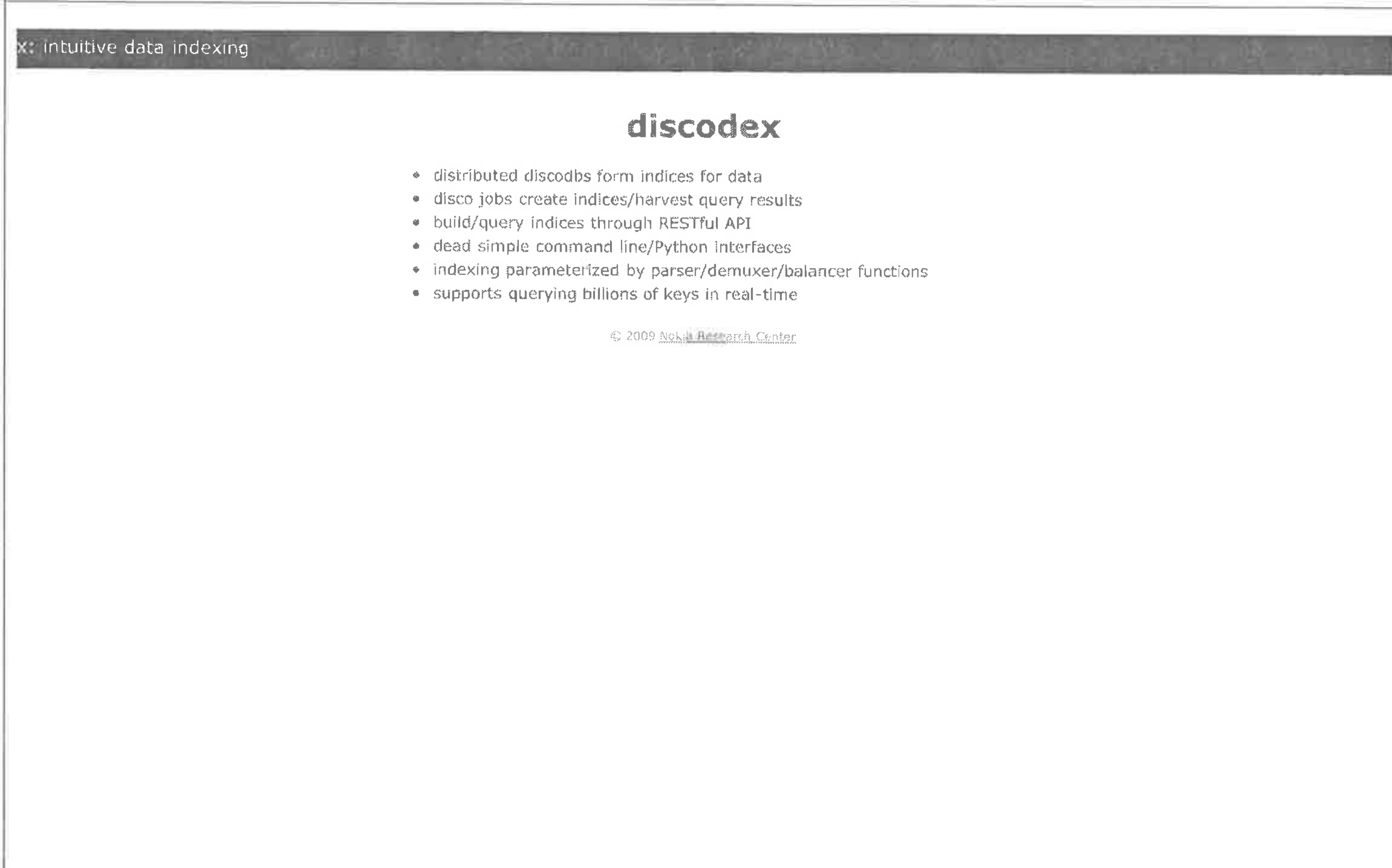
x: intuitive data indexing

discodb.erl

ets-like Erlang binding to discodb

- `ddb:new(), ddb:add(Ddb, Key, Val)`
- `ddb:lookup(Ddb, Key), ddb:query(Ddb, CnfQuery)`
- `ddb:select(Ddb, MatchSpec)`

Lazy query evaluation with continuations

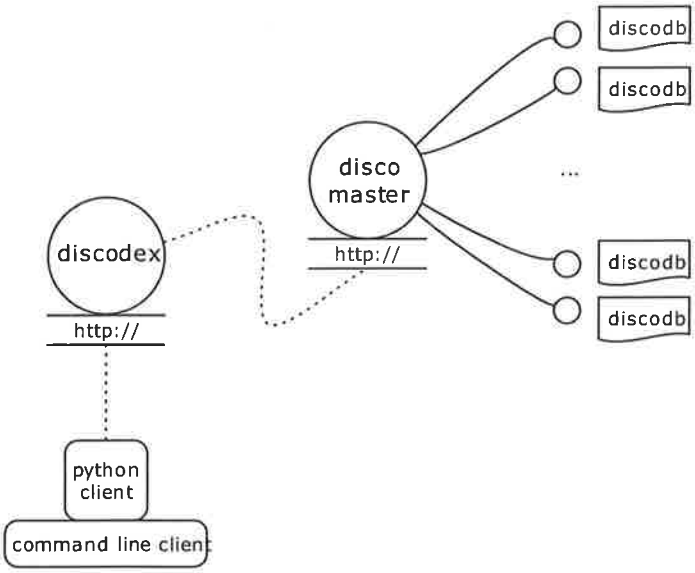


x: intuitive data indexing

discodex

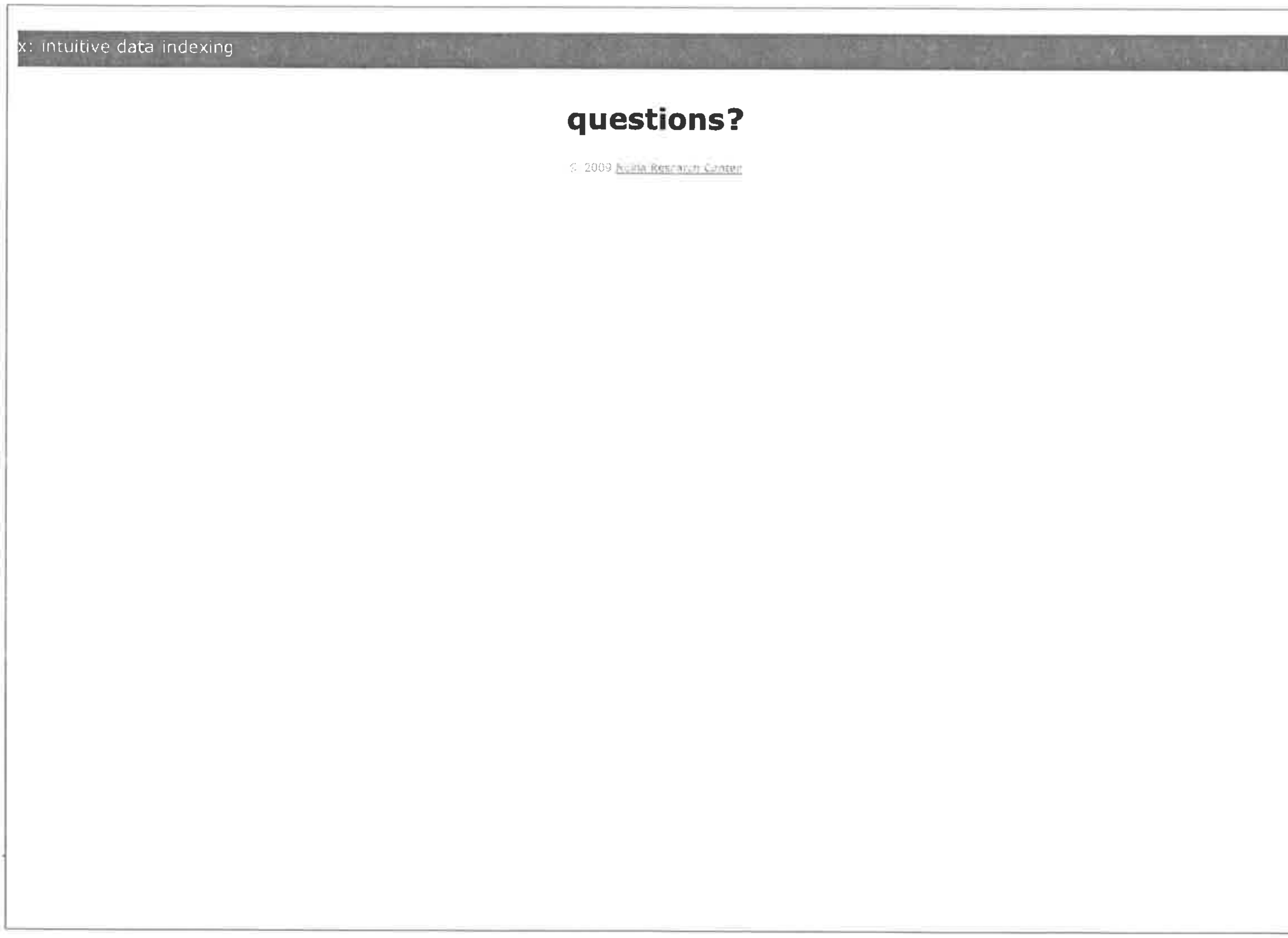
- distributed discodbs form indices for data
- disco jobs create indices/harvest query results
- build/query indices through RESTful API
- dead simple command line/Python interfaces
- indexing parameterized by parser/demuxer/balancer functions
- supports querying billions of keys in real-time

discodex design



```
> cat dataset | discodex index
> discodex get <index> | discodex query <query>
```

live demo!



x: intuitive data indexing

questions?

Ulf Wiger
Uber Erlang programmer and CTO

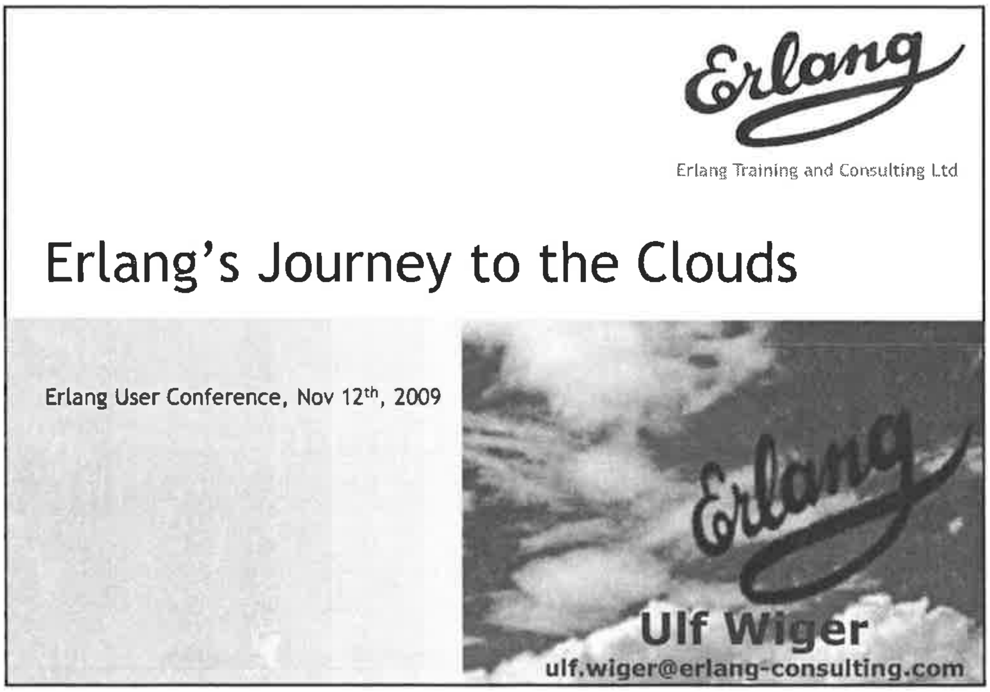
Erlang's Journey to the Clouds

Abstract

Erlang was invented in the 90s to address rapid development of non-stop scalable telecoms systems. Initial requirements included massive concurrency, distribution transparency, in-service upgrades, plug-and-play expansion and high programmer productivity. A rapidly growing Open Source community is now using Erlang for scalable web services, messaging systems and cloud computing services. In this talk we will look at how Erlang is breaking out of the clusters of last century and entering today's cloud computing environments.

Biography

Ulf Wiger became one of the first commercial users of Erlang (certainly the first in North America) when he bought a license in 1993. At the time, he was busy designing disaster response systems in Alaska. In 1996, he joined Ericsson and became Chief Designer of the AXD 301 development. At nearly 2 million lines of Erlang code, AXD 301 is the most complex system ever built in Erlang, and probably the most complex commercial system built in any functional language. In recent years, Ulf has been involved in several products based on the AXD 301 architecture, and has been an active member of the Open Source Erlang community. In February 2009, Ulf began his new job as CTO of Erlang Training and Consulting Ltd.

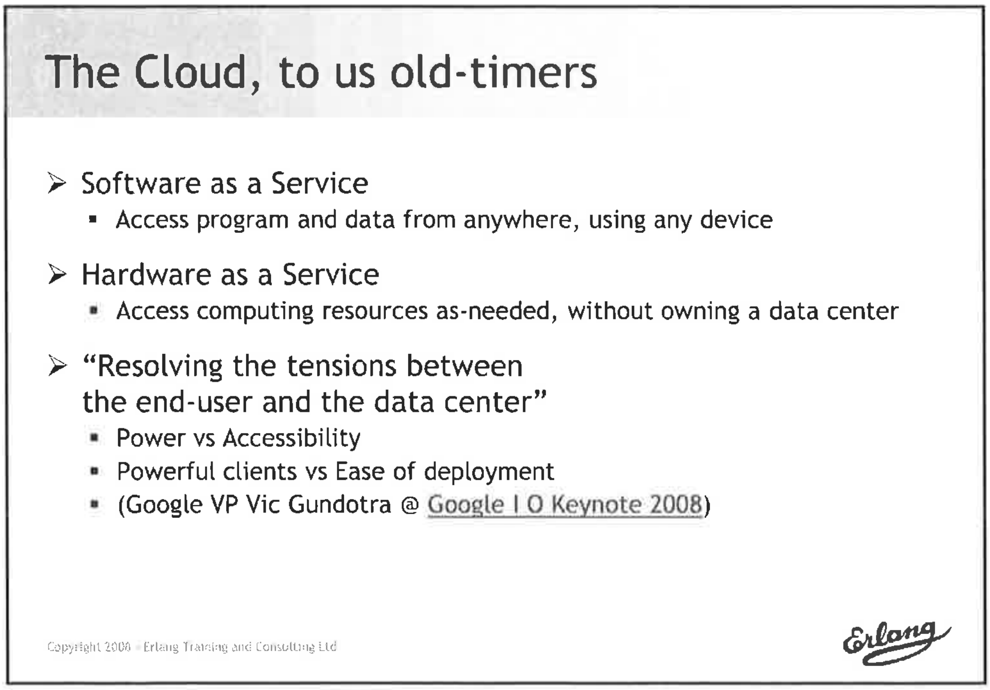


Erlang Training and Consulting Ltd

Erlang's Journey to the Clouds

Erlang User Conference, Nov 12th, 2009

Ulf Wiger
ulf.wiger@erlang-consulting.com



The Cloud, to us old-timers

- Software as a Service
  - Access program and data from anywhere, using any device
- Hardware as a Service
  - Access computing resources as-needed, without owning a data center
- "Resolving the tensions between the end-user and the data center"
  - Power vs Accessibility
  - Powerful clients vs Ease of deployment
  - (Google VP Vic Gundotra @ Google I O Keynote 2008)

Telecom in the 90s

'Stovepipe model' on its way out

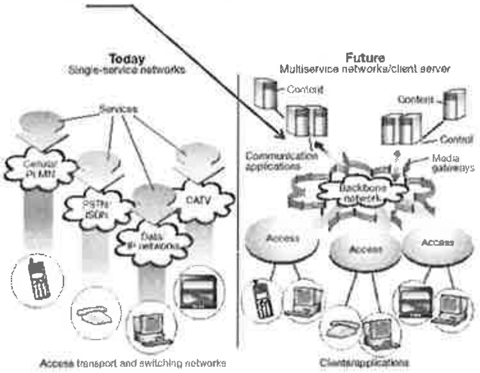
The network as a communications 'cloud'

Broadband ISDN -> Voice over ATM -> Voice over IP

Today -> Mobile IP (IPv6)

"Conversational services"

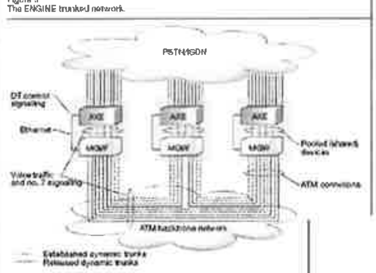
Erlang Problem Statement (paraphrased):
"How can we program telephony
in THIS environment?"



Source: Ericsson Review No 1, 1998

Bridging the Legacy

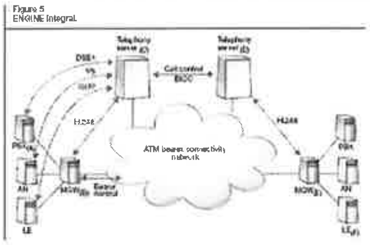
Figure 3
The ENGINE trunked network.



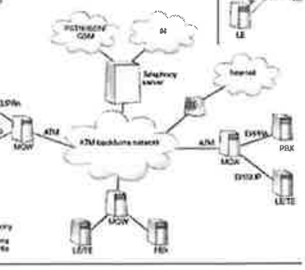
Virtual voice trunks

Source: Ericsson Review No 3, 2000

Figure 5
ENGINE Integral.



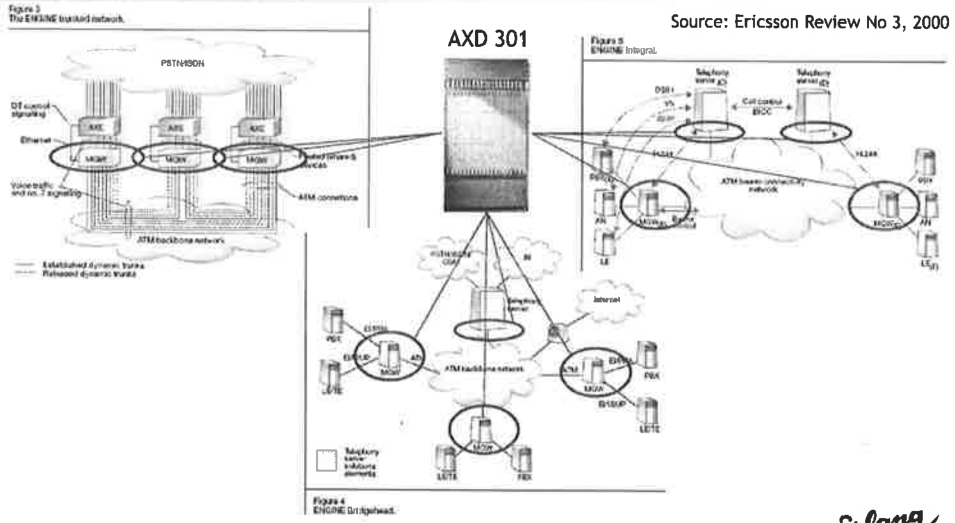
Cross-domain
Voice-over-Packet



Single-Domain
Voice-over-Packet

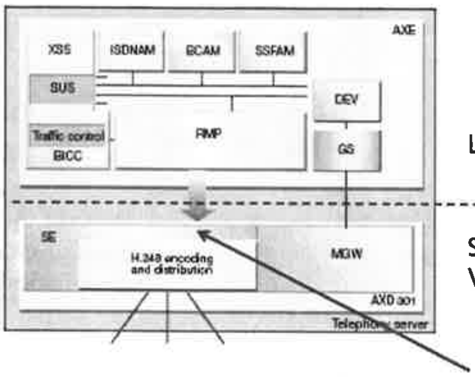
Figure 4
ENGINE Bridgehead.

Erlang the Enabler



Source: Ericsson Review No 3, 2000

Erlang the Enabler



Legacy Phone Switch

PLEX

Switch Emulator and
Voice-over-ATM Controller

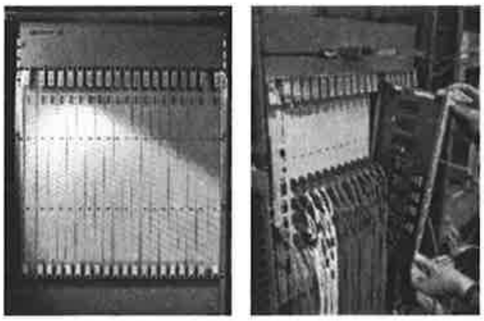
Erlang

Extremely complex state machines
Scalability and redundancy required
> 99.999% uptime, including maintenance

So What Next...?

- The first big project worked out
- Erlang proven ready for the Big Time
- Erlang released as Open Source 1998
  - No fanfare, no marketing...
  - 1-2 messages/day on the mailing list first year
  - < 1000 downloads/month
- New initiatives needed...

Idea: A Scalable Web Server...?

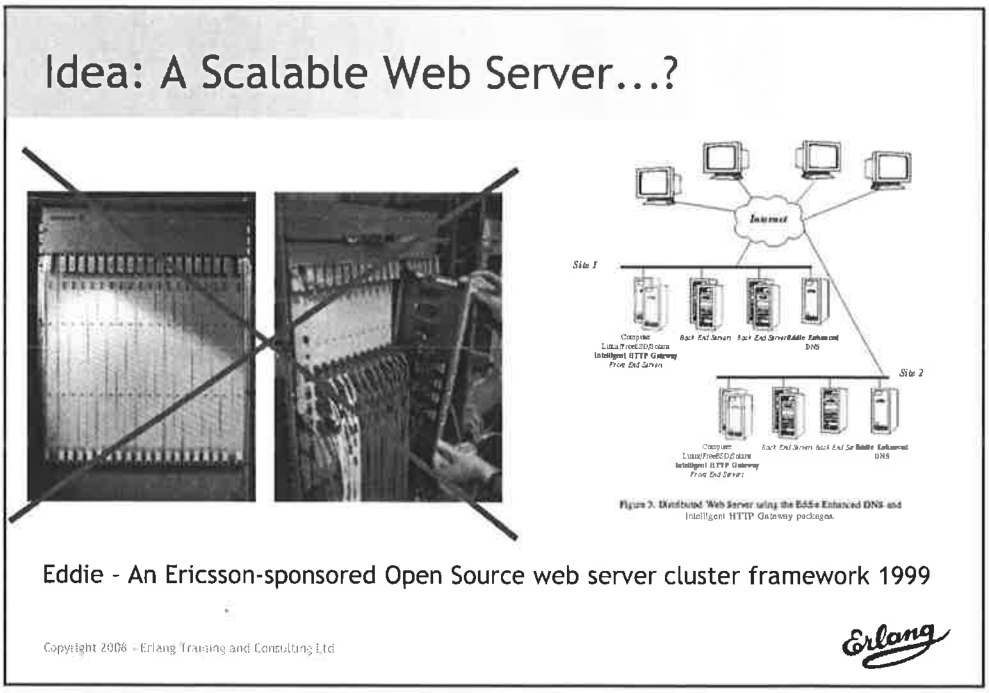


Build a 5-nines scalable web server based on AXD 301

- 256 processor boards on a non-blocking, redundant 160 Gbps backbone!

Two erlang-related Innovation Cell proposals presented at the same time

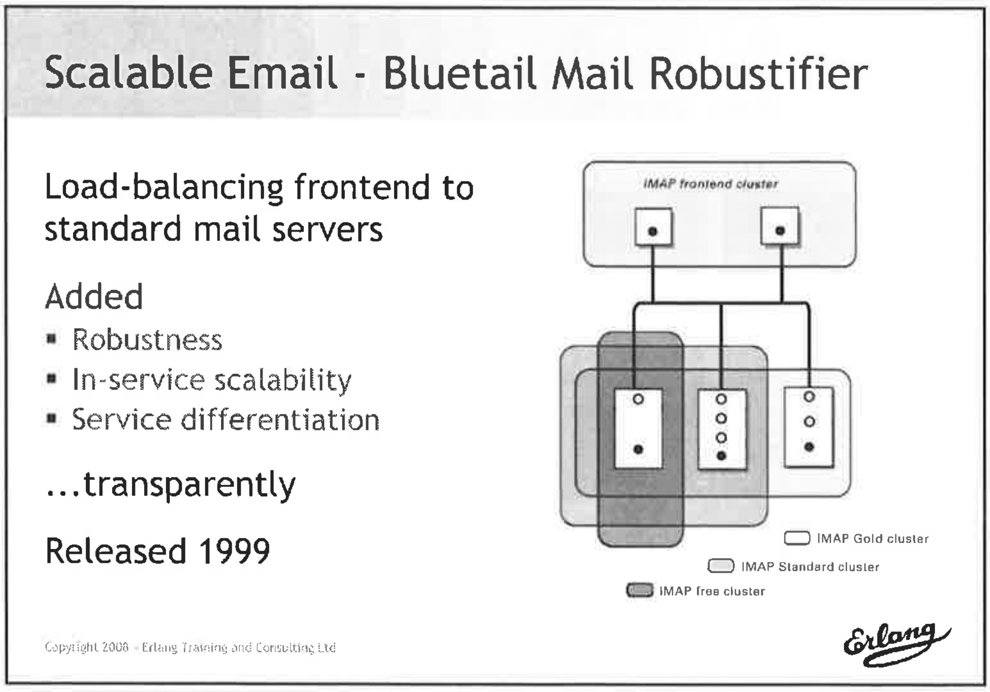
The AXD 301 track rejected - Ericsson doesn't sell web servers



Idea: A Scalable Web Server...?

Eddie - An Ericsson-sponsored Open Source web server cluster framework 1999

Copyright 2008 - Erlang Training and Consulting Ltd



Scalable Email - Bluetail Mail Robustifier

Load-balancing frontend to standard mail servers

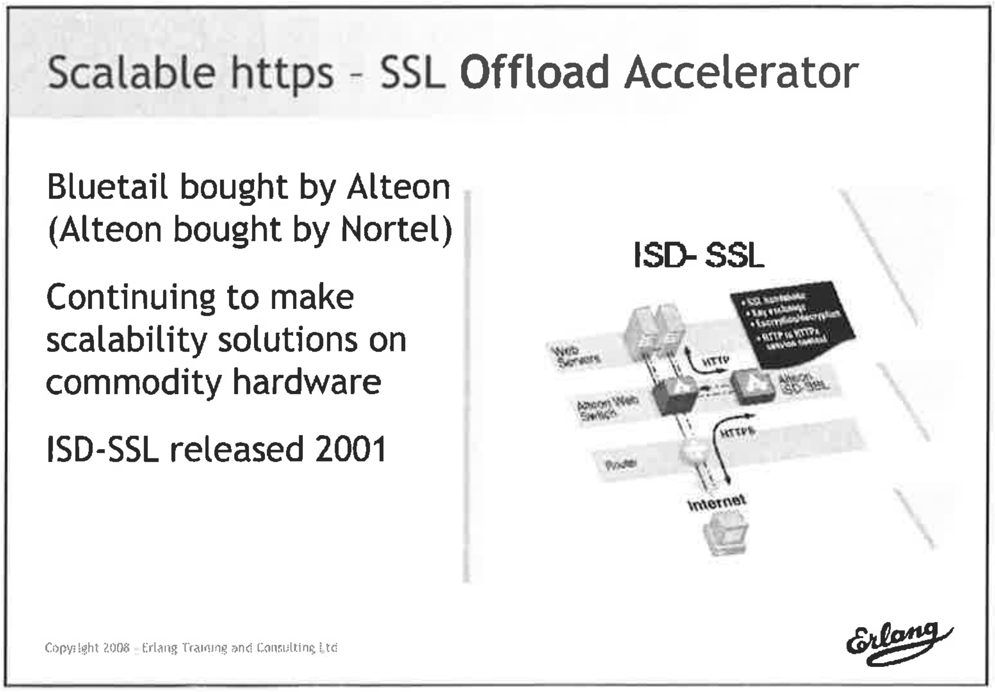
Added

- Robustness
- In-service scalability
- Service differentiation

...transparently

Released 1999

Copyright 2008 - Erlang Training and Consulting Ltd



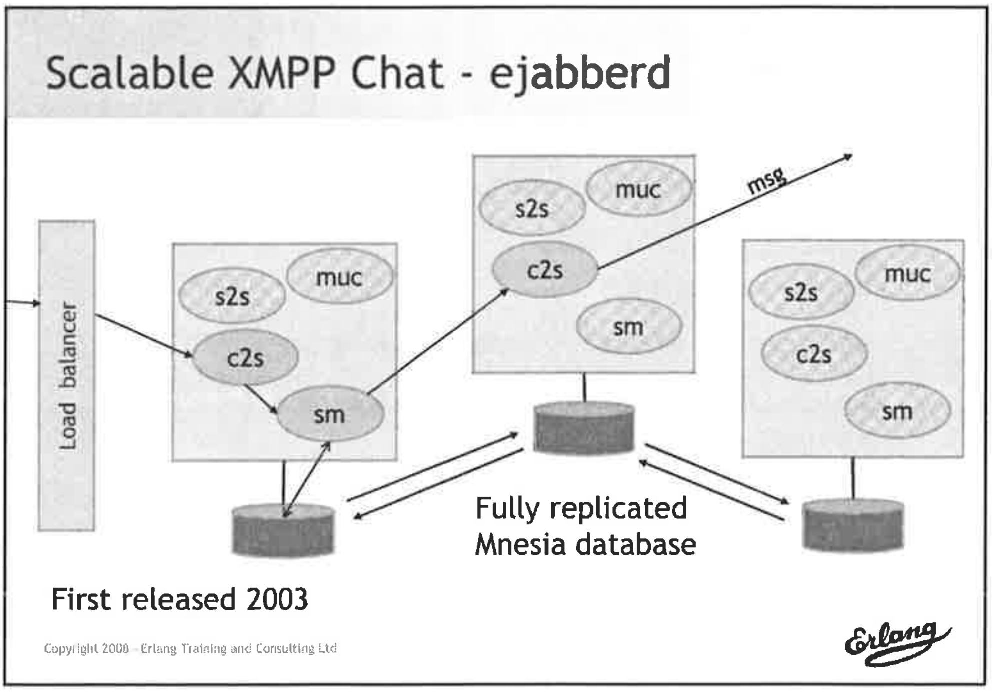
Scalable https - SSL Offload Accelerator

Bluetail bought by Alteon (Alteon bought by Nortel)

Continuing to make scalability solutions on commodity hardware

ISD-SSL released 2001

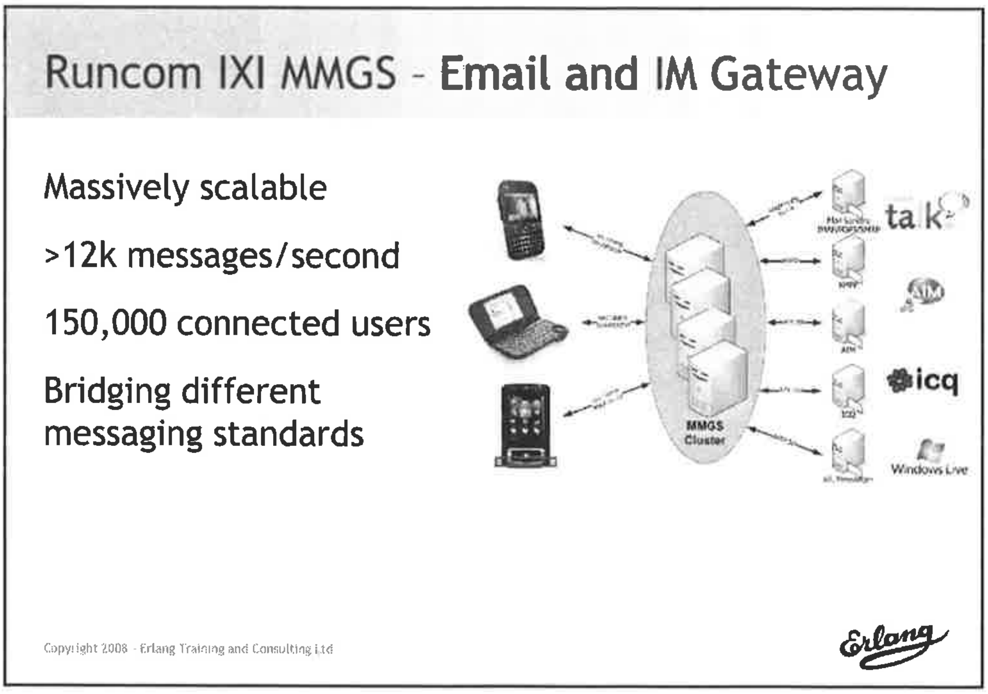
ISD- SSL
• SSL handshake
• Key exchange
• Encryption/decryption
• HTTP to HTTPs session control
Web Servers
HTTP
Alteon ISD-SSL
Alteon Web Switch
HTTPS
Router
Internet



Scalable XMPP Chat - ejabberd

Fully replicated Mnesia database

First released 2003



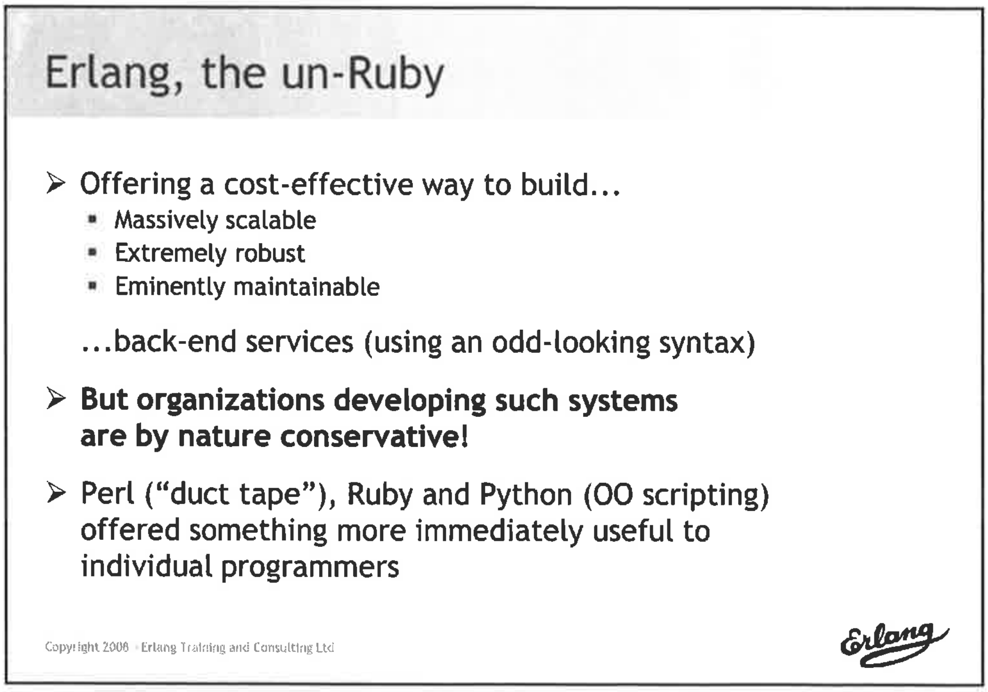
Runcom IXI MMGS - **Email and IM Gateway**

Massively scalable

>12k messages/second

150,000 connected users

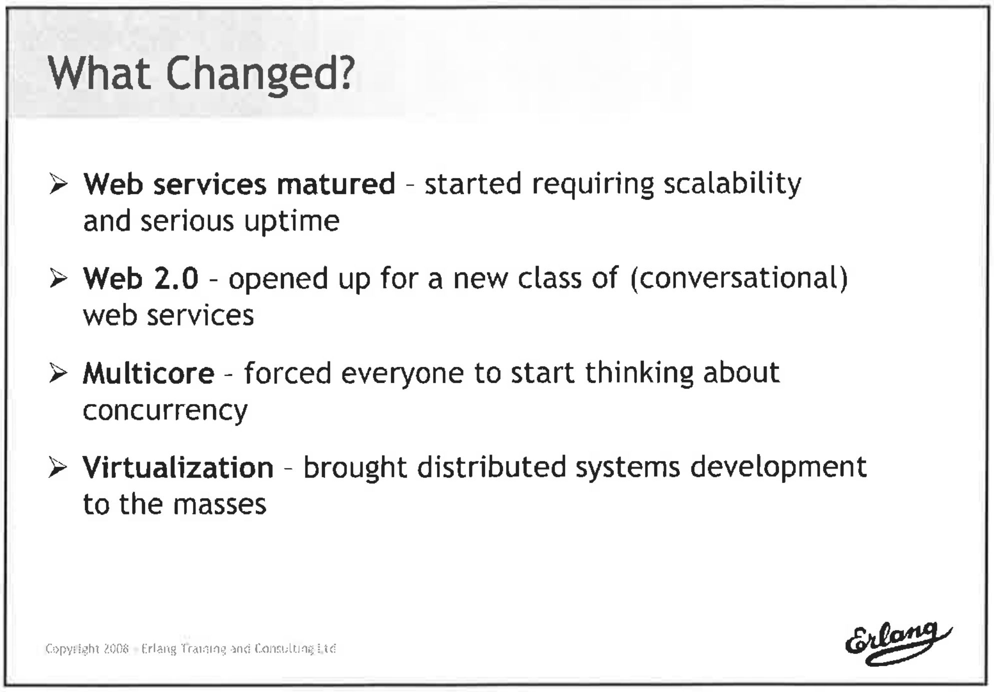
Bridging different messaging standards



Erlang, the un-Ruby

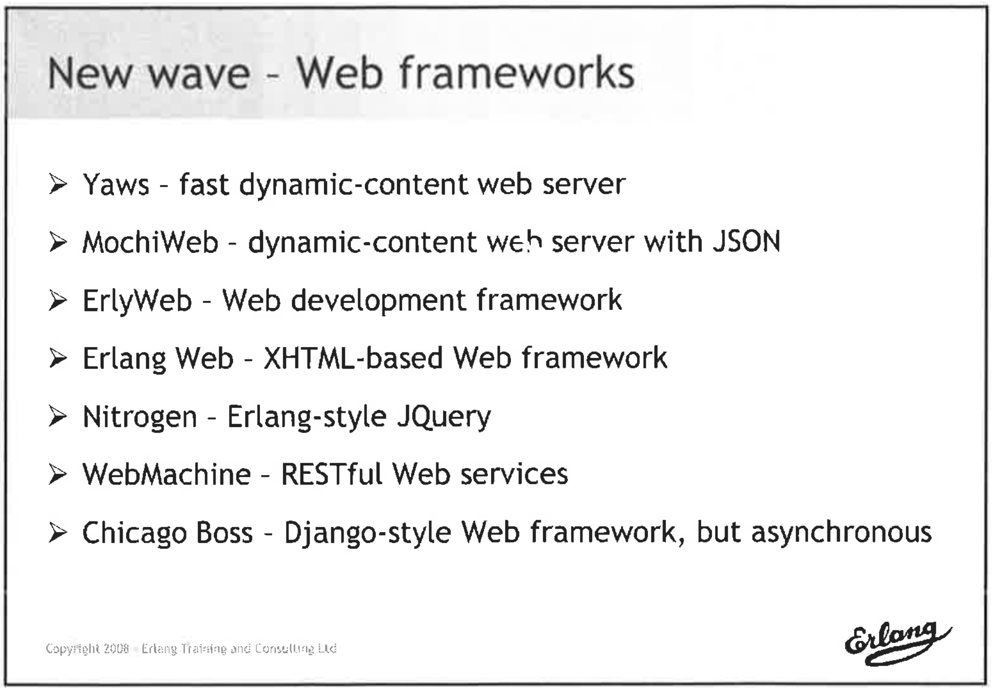
- Offering a cost-effective way to build...
  - Massively scalable
  - Extremely robust
  - Eminently maintainable

  ...back-end services (using an odd-looking syntax)
- **But organizations developing such systems are by nature conservative!**
- Perl ("duct tape"), Ruby and Python (OO scripting) offered something more immediately useful to individual programmers



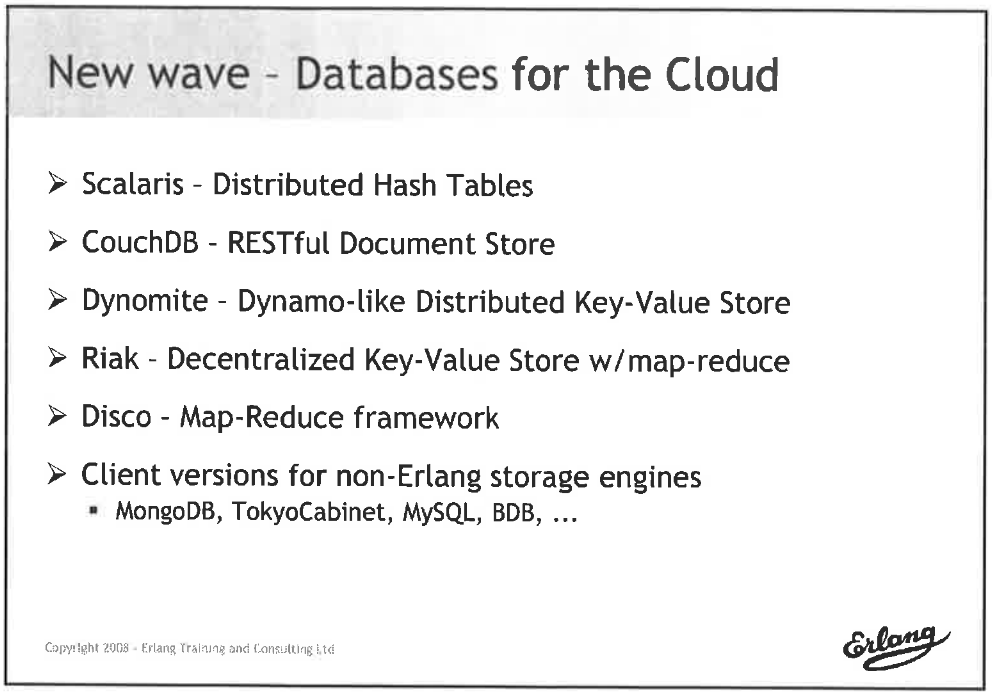
What Changed?

- **Web services matured** - started requiring scalability and serious uptime
- **Web 2.0** - opened up for a new class of (conversational) web services
- **Multicore** - forced everyone to start thinking about concurrency
- **Virtualization** - brought distributed systems development to the masses



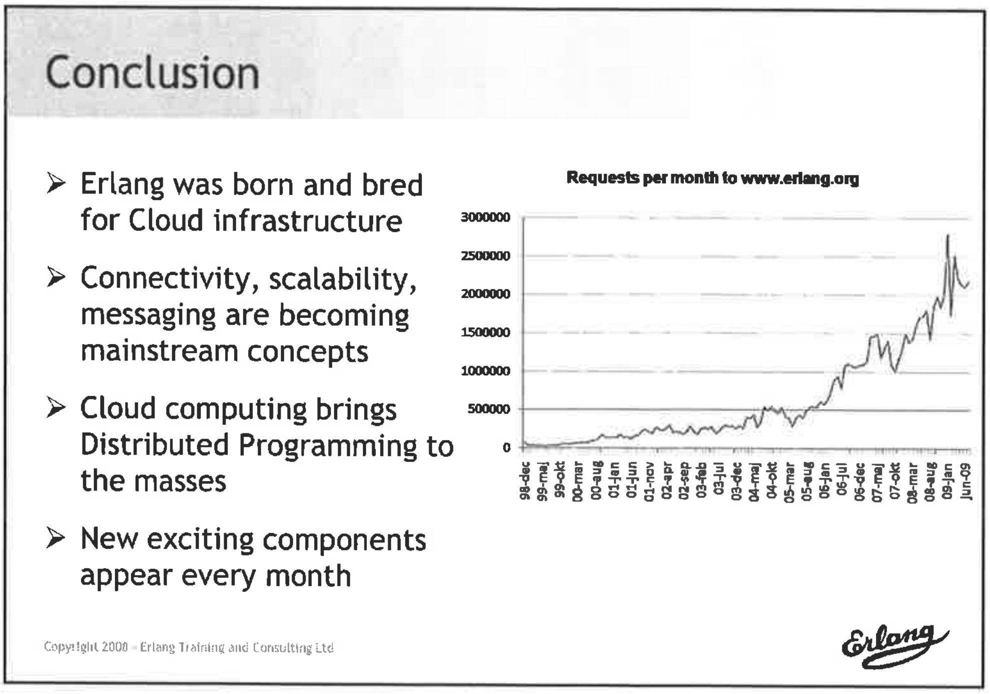
New wave - Web frameworks

- Yaws - fast dynamic-content web server
- MochiWeb - dynamic-content web server with JSON
- ErlyWeb - Web development framework
- Erlang Web - XHTML-based Web framework
- Nitrogen - Erlang-style JQuery
- WebMachine - RESTful Web services
- Chicago Boss - Django-style Web framework, but asynchronous



New wave - Databases for the Cloud

- ➢ Scalaris – Distributed Hash Tables
- ➢ CouchDB - RESTful Document Store
- ➢ Dynomite – Dynamo-like Distributed Key-Value Store
- ➢ Riak - Decentralized Key-Value Store w/map-reduce
- ➢ Disco - Map-Reduce framework
- ➢ Client versions for non-Erlang storage engines
  - MongoDB, TokyoCabinet, MySQL, BDB, ...



Conclusion

- ➢ Erlang was born and bred for Cloud infrastructure
- ➢ Connectivity, scalability, messaging are becoming mainstream concepts
- ➢ Cloud computing brings Distributed Programming to the masses
- ➢ New exciting components appear every month

Rickard Cardell

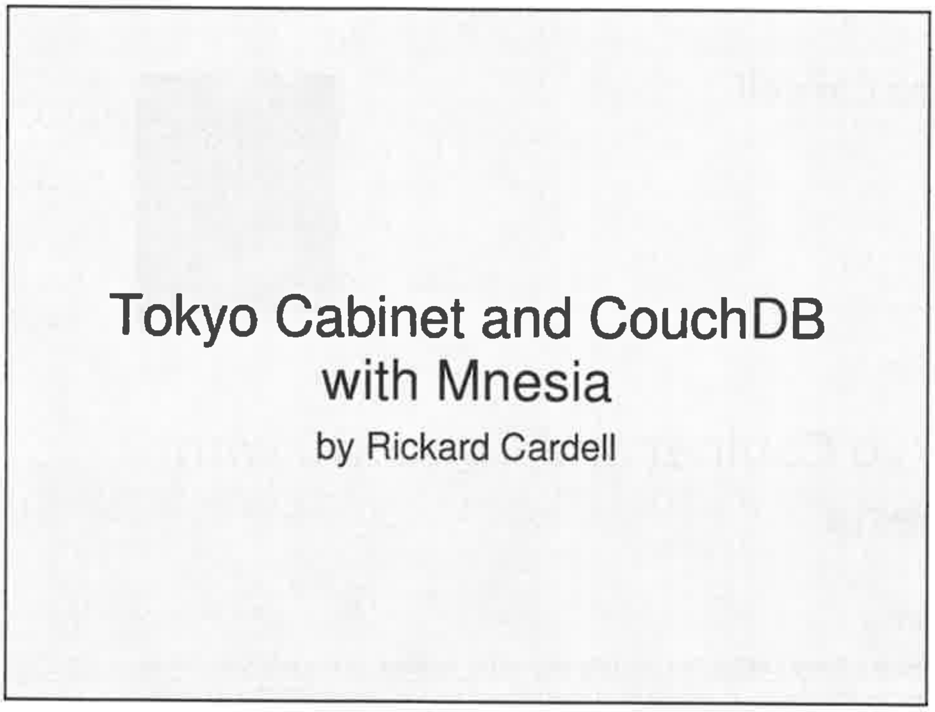
Tokyo Cabinet and CouchDB with Mnesia

Abstract

Couch DB and Tokyo Cabinet are two very interesting database managers. CouchDB is famous for its robustness, its simple document storage model, and its RESTful interface, and also for the fact that it is written in Erlang. TokyoCabinet, on the other hand, is written in C, is blazingly fast, and an interface to Mnesia already exists (tcerl via mnesiaex). In this talk I will discuss how I used Mnesia as a frontend to these database managers and the problems I encountered while integrating it with a legacy Erlang system based on Mnesia. I will also present the results of some transaction benchmarks, and discuss some interesting features of CouchDB and TokyoCabinet.

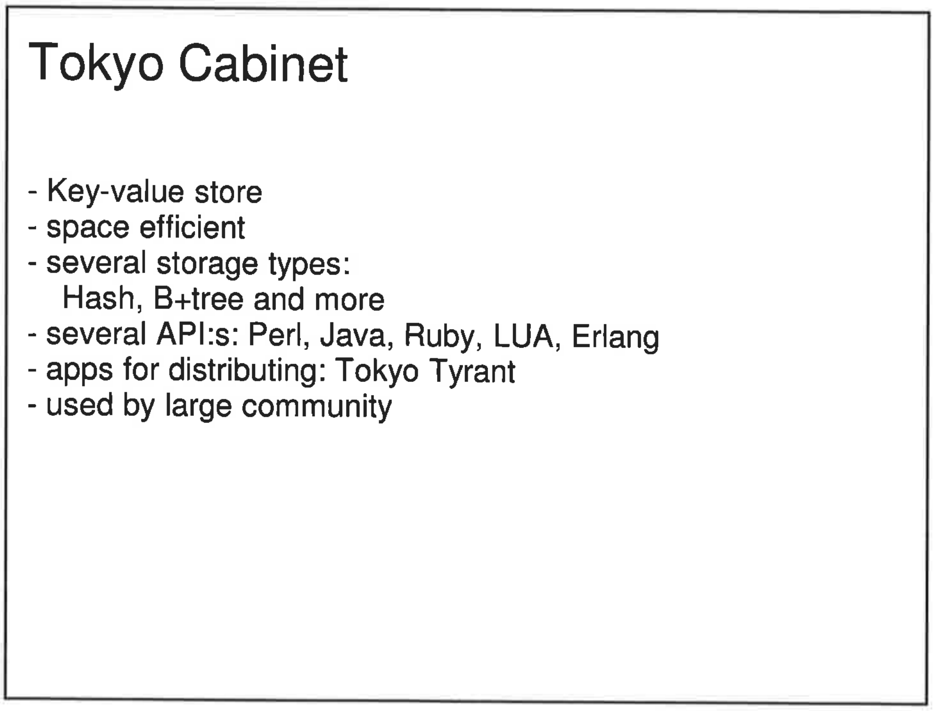
Biography

Rickard is doing a Master's thesis at Kreditor, where he evaluates two different database managers, Tokyo Cabinet and CouchDB, for possible use as a backend for Mnesia. He got his first experience with Erlang during a course in Distributed Systems at Uppsala University.



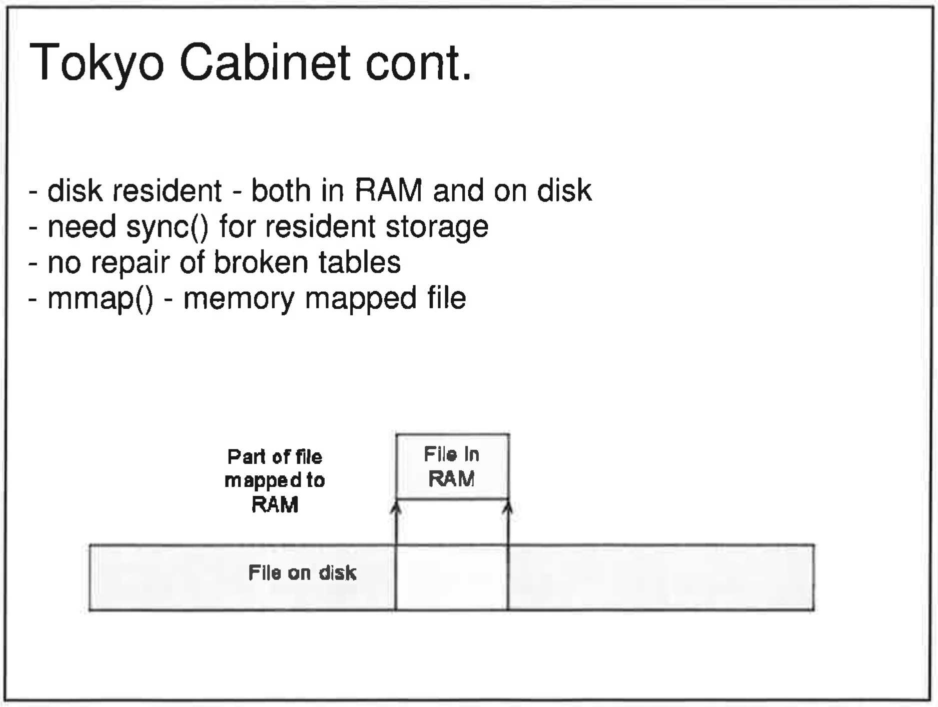
Tokyo Cabinet and CouchDB with Mnesia

by Rickard Cardell



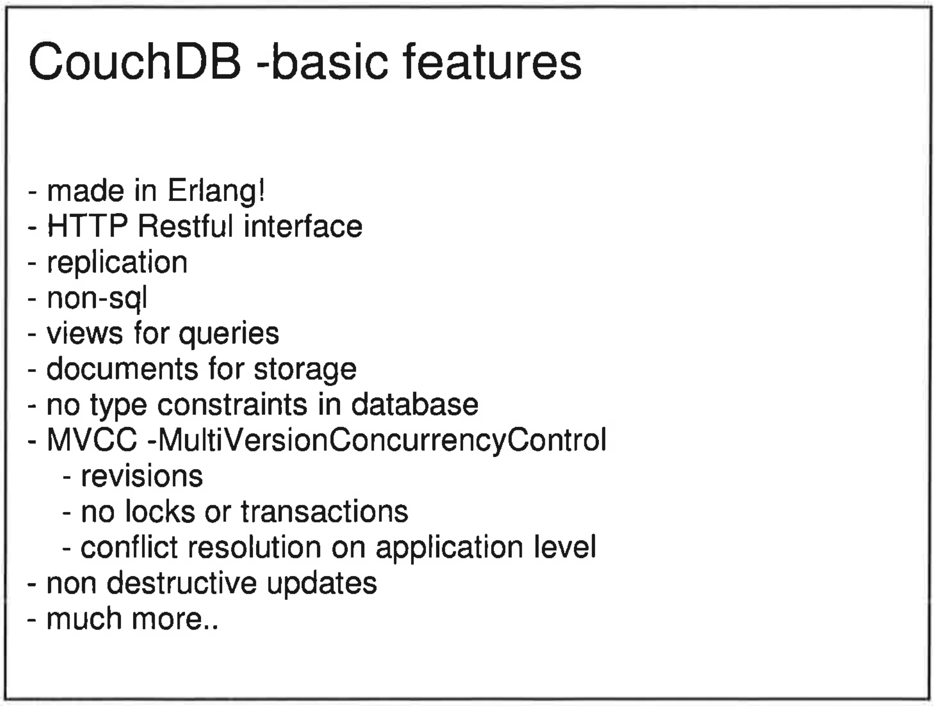
Tokyo Cabinet

- Key-value store
- space efficient
- several storage types:
  Hash, B+tree and more
- several API:s: Perl, Java, Ruby, LUA, Erlang
- apps for distributing: Tokyo Tyrant
- used by large community



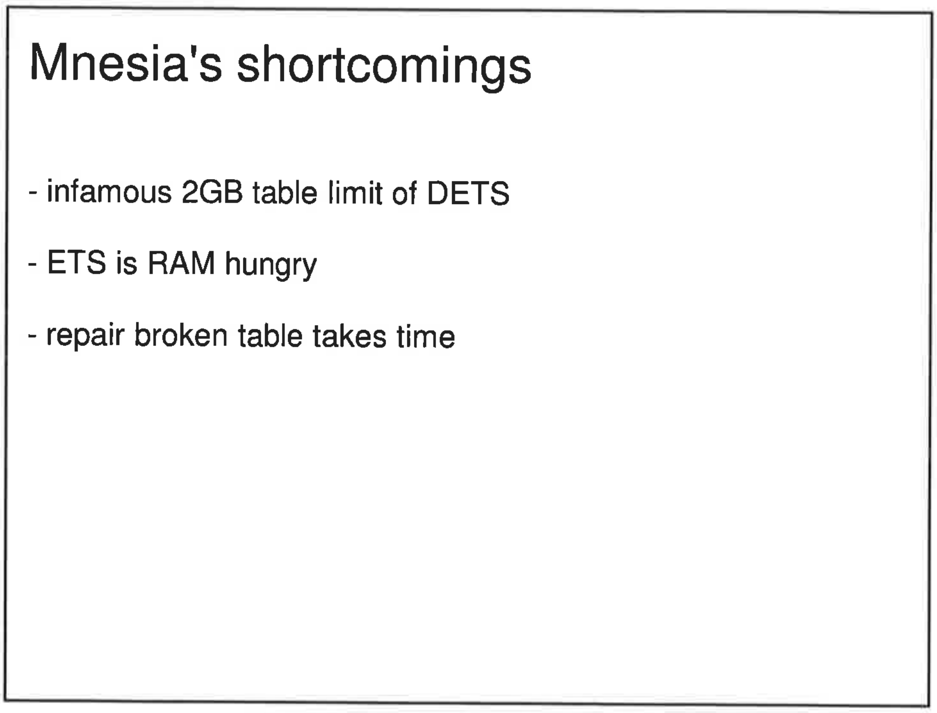
Tokyo Cabinet cont.

- disk resident - both in RAM and on disk
- need sync() for resident storage
- no repair of broken tables
- mmap() - memory mapped file



CouchDB -basic features

- made in Erlang!
- HTTP Restful interface
- replication
- non-sql
- views for queries
- documents for storage
- no type constraints in database
- MVCC -MultiVersionConcurrencyControl
    - revisions
    - no locks or transactions
    - conflict resolution on application level
- non destructive updates
- much more..

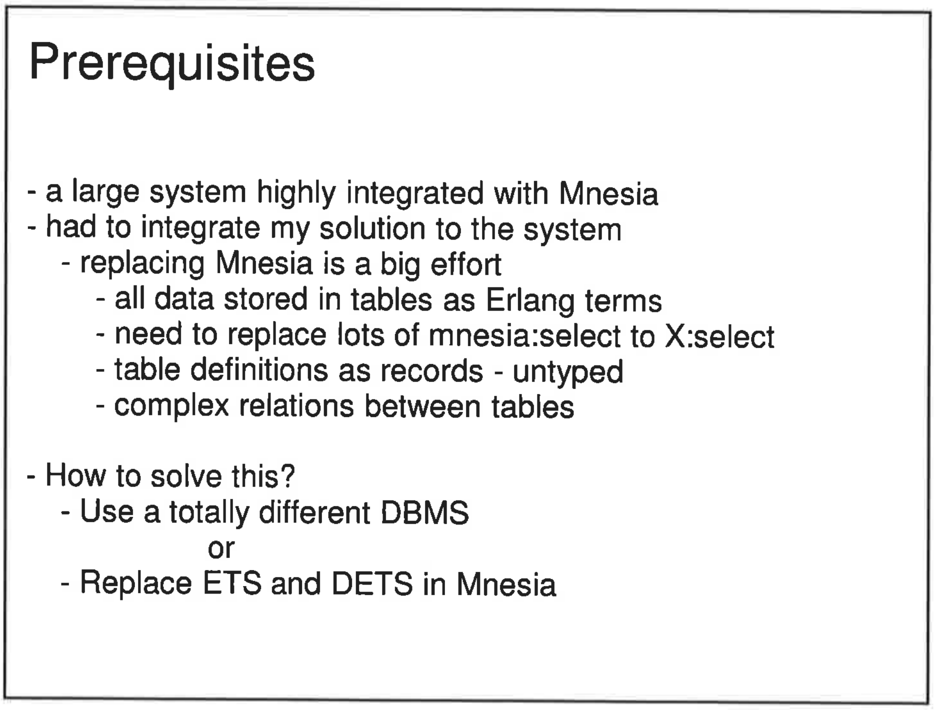


Mnesia's shortcomings

- infamous 2GB table limit of DETS

- ETS is RAM hungry

- repair broken table takes time



Prerequisites

- a large system highly integrated with Mnesia
- had to integrate my solution to the system
    - replacing Mnesia is a big effort
        - all data stored in tables as Erlang terms
        - need to replace lots of mnesia:select to X:select
        - table definitions as records - untyped
        - complex relations between tables

- How to solve this?
    - Use a totally different DBMS

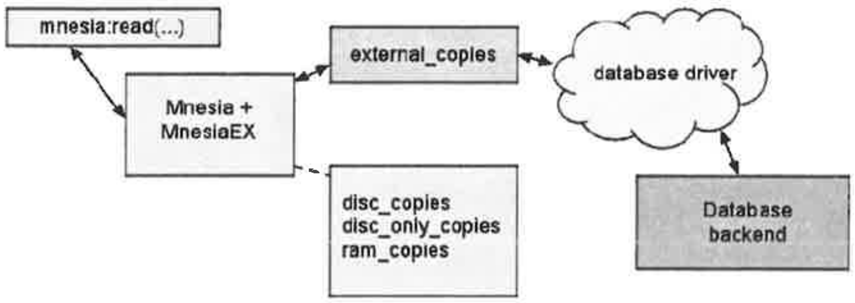
      or
    - Replace ETS and DETS in Mnesia

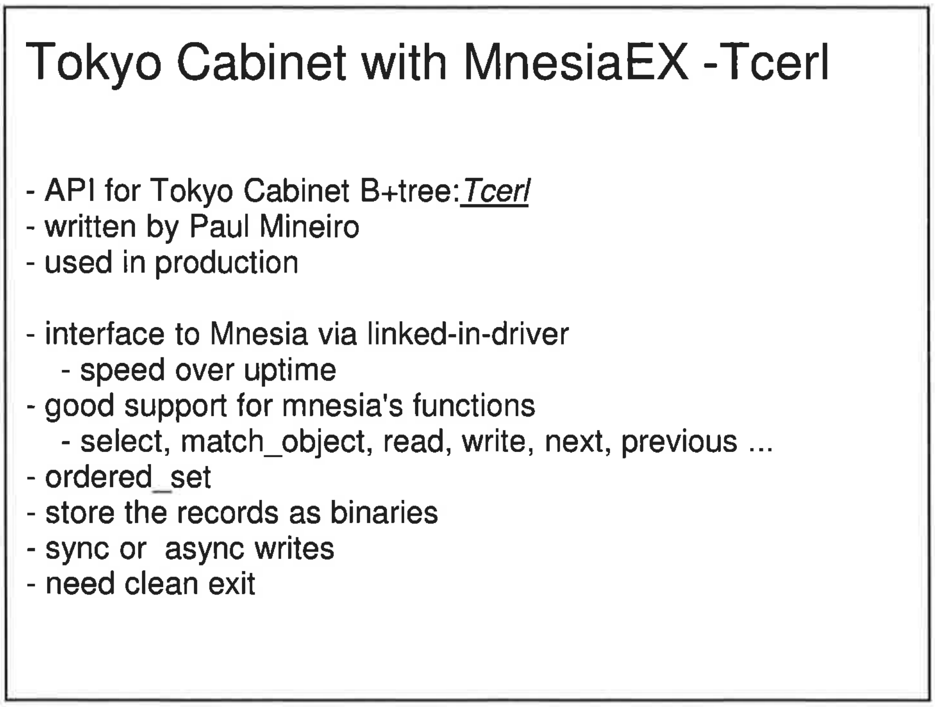
My solution

- make backends of Tokyo Cabinet and CouchDB
  - less code changes to the system
  - transparent to the user
  - will make use of Mnesia locks and transactions
- already an extension to Mnesia: MnesiaEX

MnesiaEX - Mnesia extension

- ability to apply arbitrary storage to Mnesia
- almost transparent to the user
- adds a new storage type external_copies
  - works together with current storage types
- ACID issues
- Tokyo Cabinet has already API: Tcerl

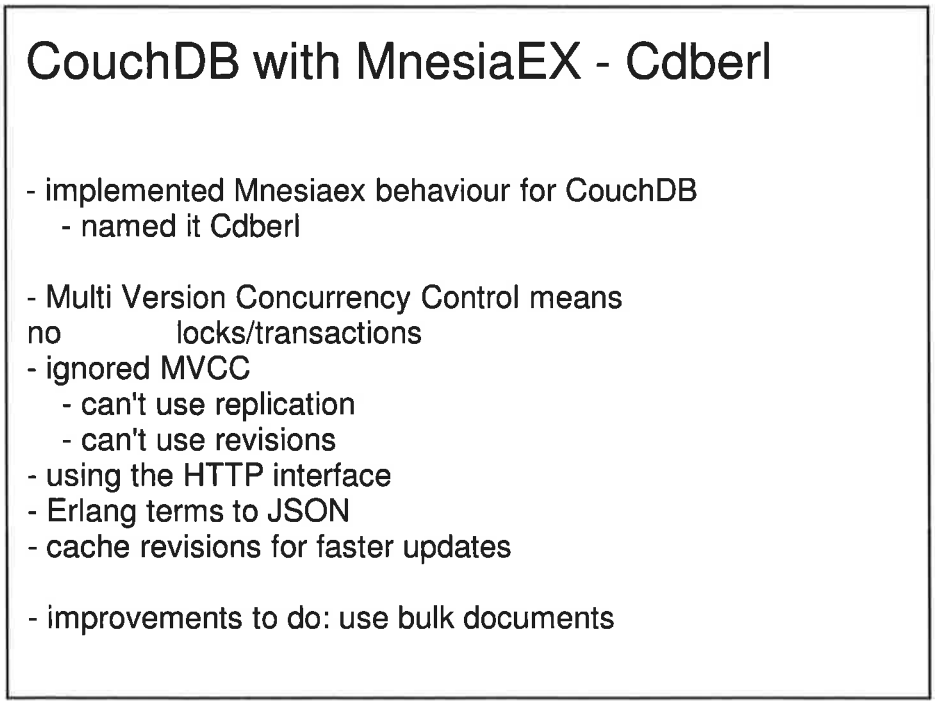




Tokyo Cabinet with MnesiaEX -Tcerl

- API for Tokyo Cabinet B+tree:*Tcerl*
- written by Paul Mineiro
- used in production

- interface to Mnesia via linked-in-driver
  - speed over uptime
- good support for mnesia's functions
  - select, match_object, read, write, next, previous ...
- ordered_set
- store the records as binaries
- sync or async writes
- need clean exit

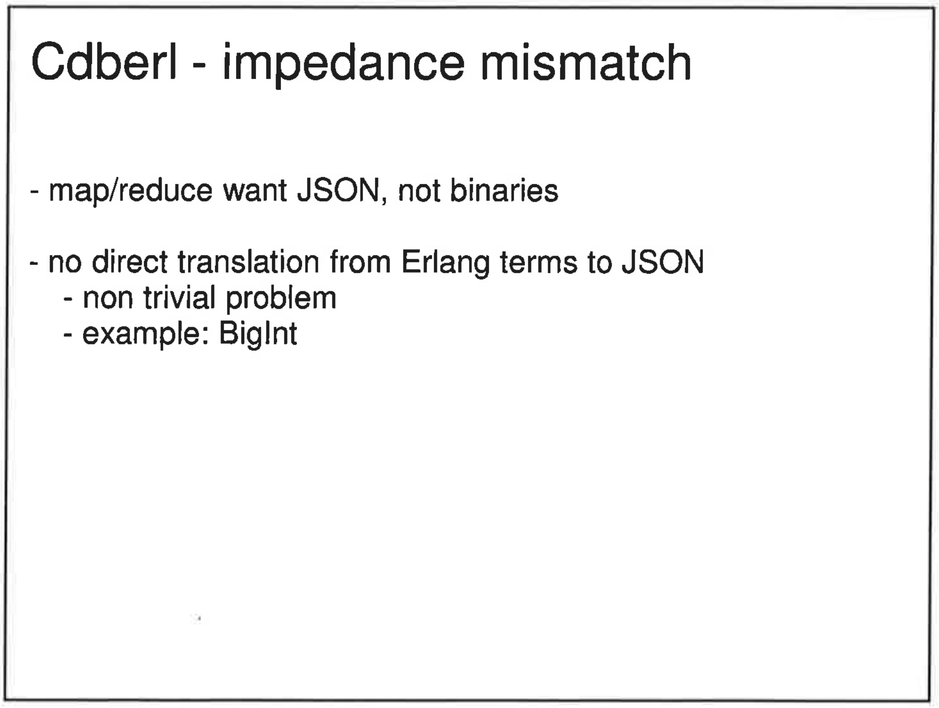


CouchDB with MnesiaEX - Cdberl

- implemented Mnesiaex behaviour for CouchDB
  - named it Cdberl

- Multi Version Concurrency Control means
no locks/transactions
- ignored MVCC
  - can't use replication
  - can't use revisions
- using the HTTP interface
- Erlang terms to JSON
- cache revisions for faster updates

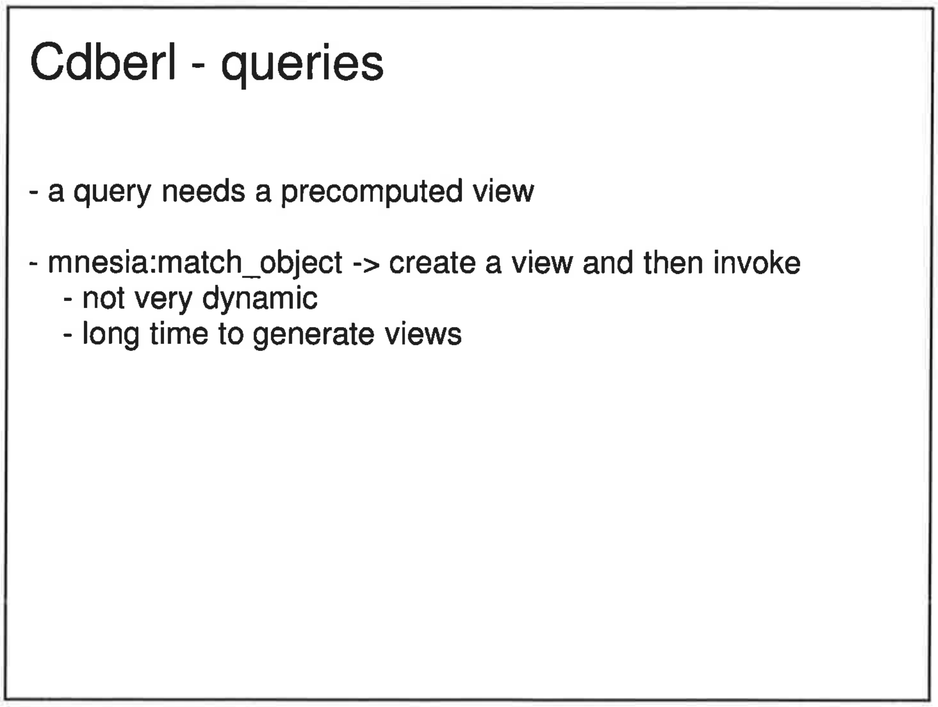
- improvements to do: use bulk documents



Cdberl - impedance mismatch

- map/reduce want JSON, not binaries

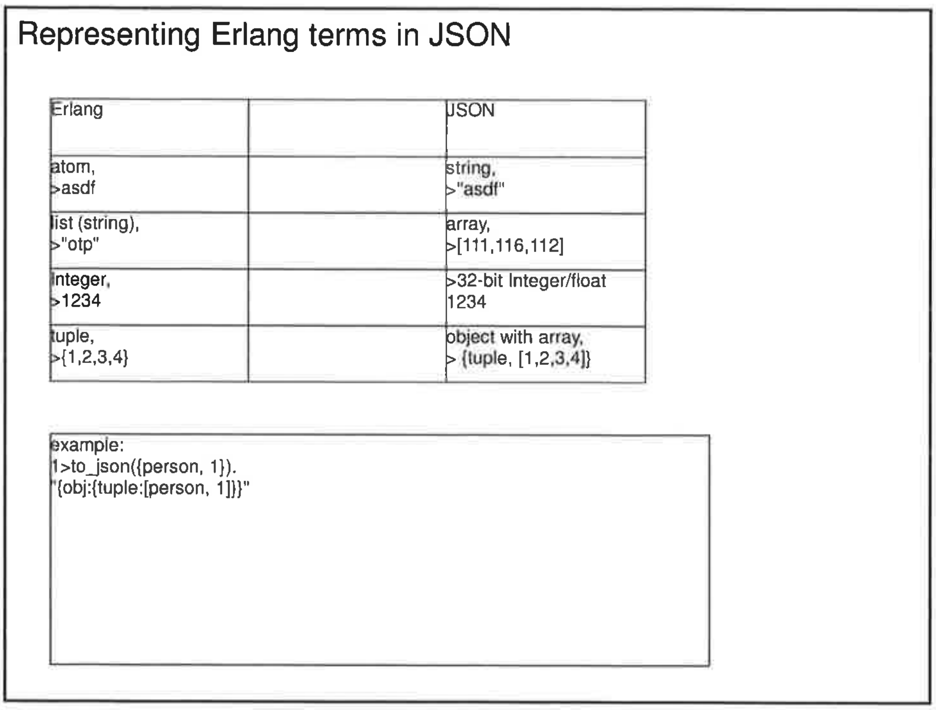
- no direct translation from Erlang terms to JSON
    - non trivial problem
    - example: BigInt



Cdberl - queries

- a query needs a precomputed view

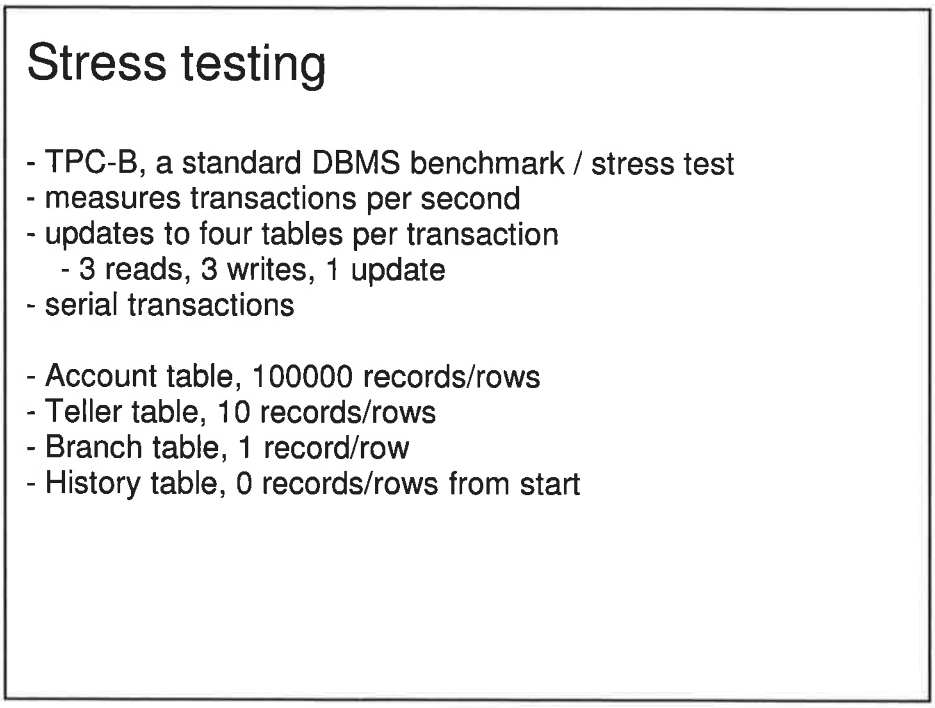
- mnesia:match_object -> create a view and then invoke
    - not very dynamic
    - long time to generate views



Representing Erlang terms in JSON

Erlang		JSON
atom, >asdf		string, >"asdf"
list (string), >"otp"		array, >[111,116,112]
Integer, >1234		>32-bit Integer/float 1234
tuple, >{1,2,3,4}		object with array, > {tuple, [1,2,3,4]}

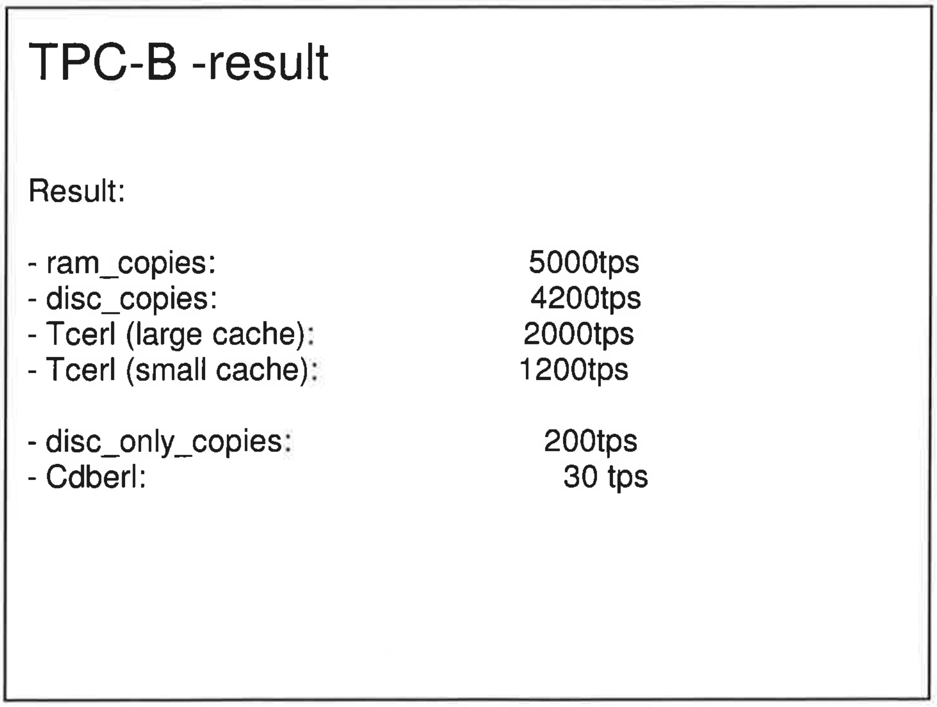
```
example:
1>to_json({person, 1}).
"{obj:{tuple:[person, 1]}}"
```



Stress testing

- TPC-B, a standard DBMS benchmark / stress test
- measures transactions per second
- updates to four tables per transaction
  - 3 reads, 3 writes, 1 update
- serial transactions

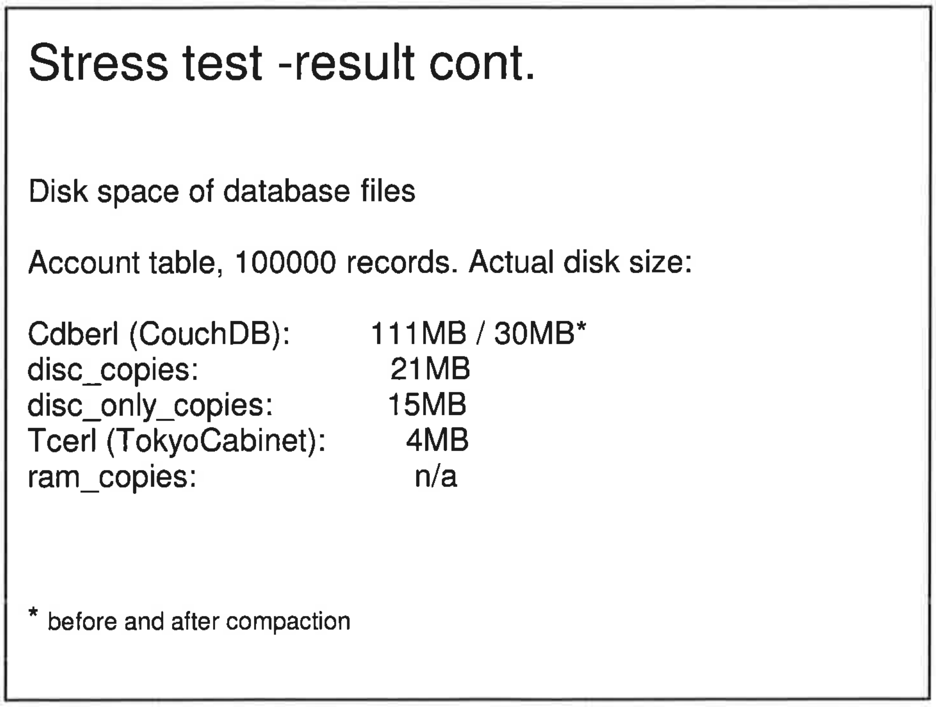
- Account table, 100000 records/rows
- Teller table, 10 records/rows
- Branch table, 1 record/row
- History table, 0 records/rows from start



TPC-B -result

Result:

- ram_copies:	5000tps
- disc_copies:	4200tps
- Tcerl (large cache):	2000tps
- Tcerl (small cache):	1200tps
- disc_only_copies:	200tps
- Cdberl:	30 tps



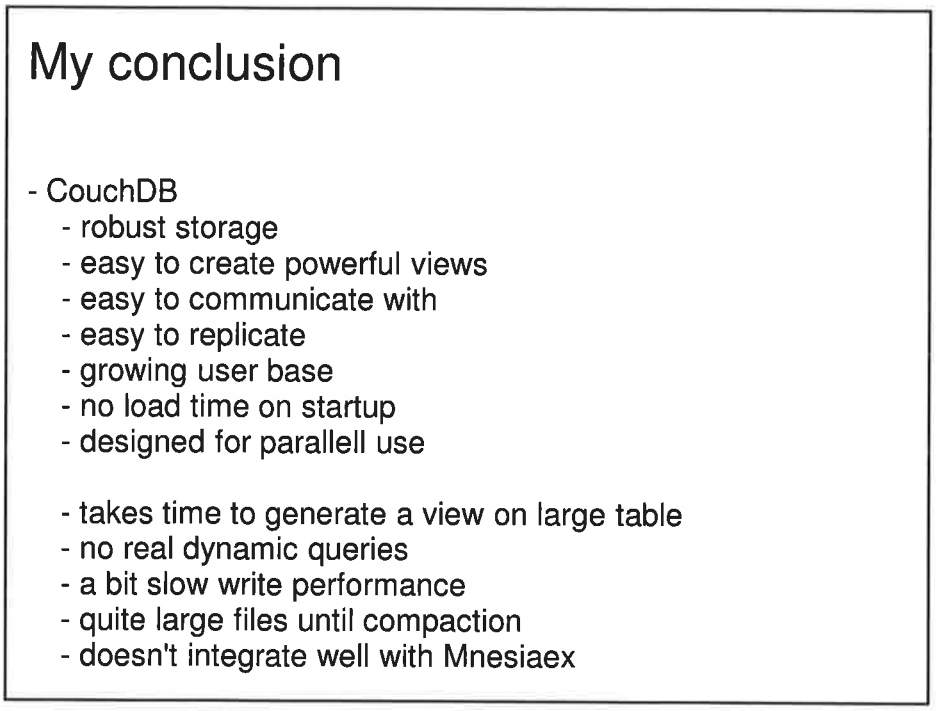
Stress test -result cont.

Disk space of database files

Account table, 100000 records. Actual disk size:

Cdberl (CouchDB):	111MB / 30MB*
disc_copies:	21MB
disc_only_copies:	15MB
Tcerl (TokyoCabinet):	4MB
ram_copies:	n/a

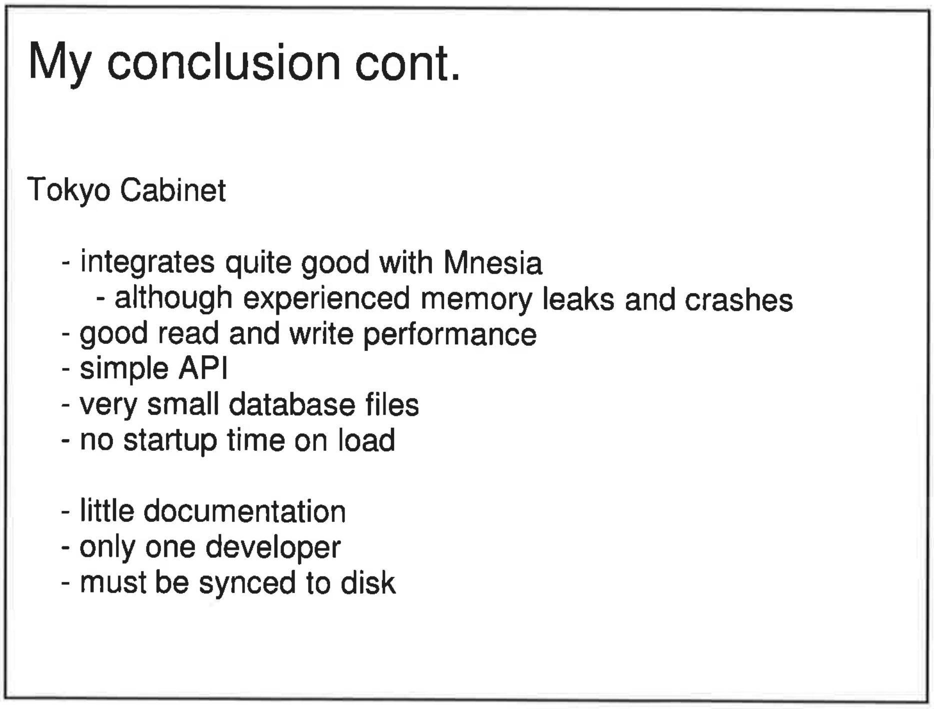
* before and after compaction



My conclusion

- CouchDB
  - robust storage
  - easy to create powerful views
  - easy to communicate with
  - easy to replicate
  - growing user base
  - no load time on startup
  - designed for parallell use

  - takes time to generate a view on large table
  - no real dynamic queries
  - a bit slow write performance
  - quite large files until compaction
  - doesn't integrate well with Mnesiaex

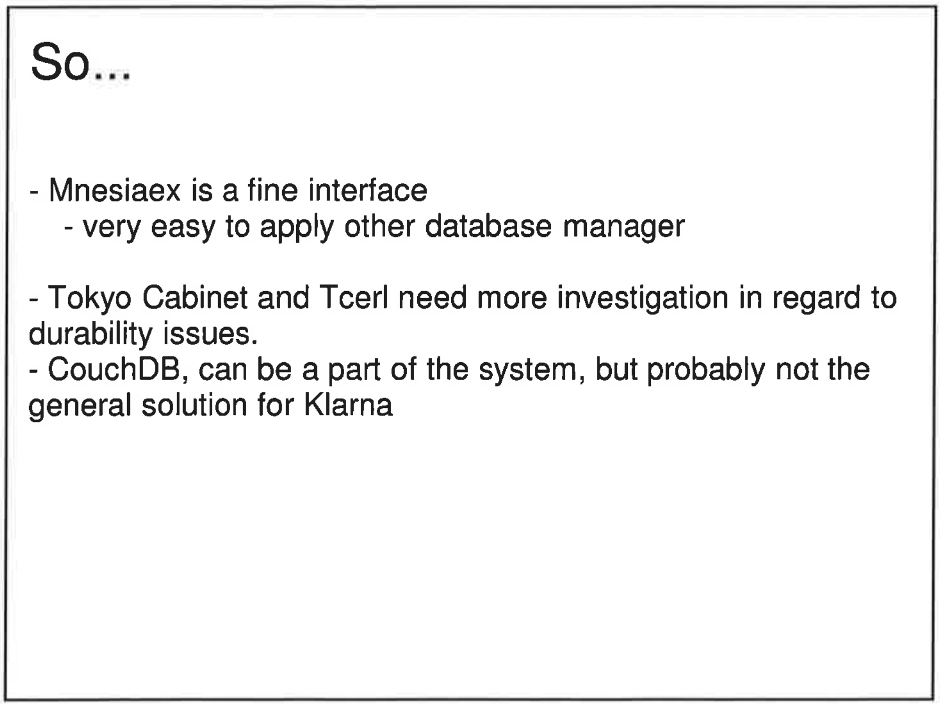


My conclusion cont.

Tokyo Cabinet

- integrates quite good with Mnesia
  - although experienced memory leaks and crashes
- good read and write performance
- simple API
- very small database files
- no startup time on load

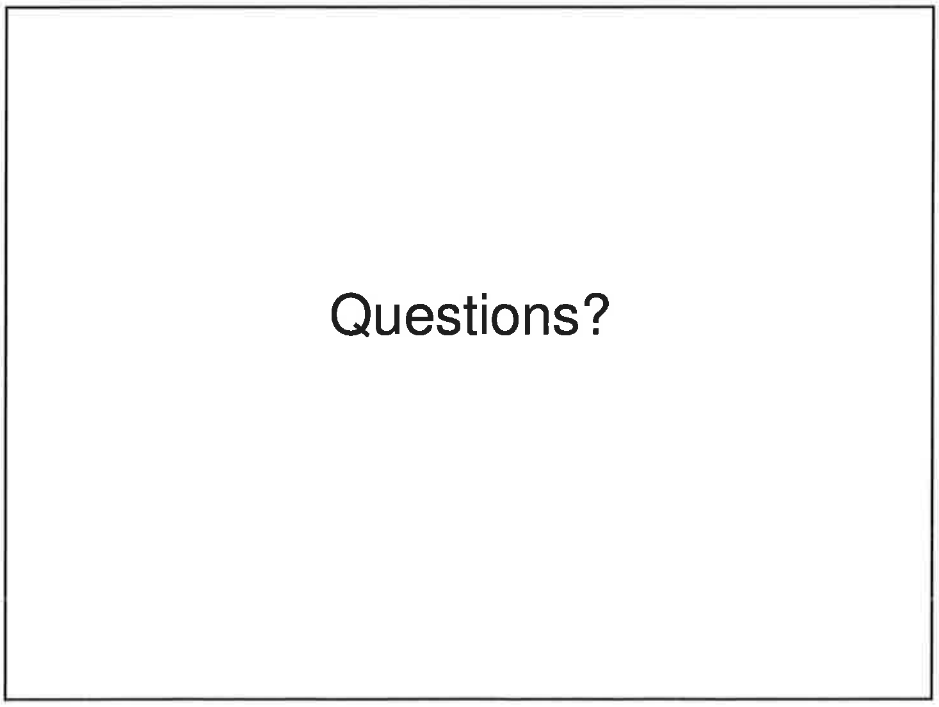
- little documentation
- only one developer
- must be synced to disk



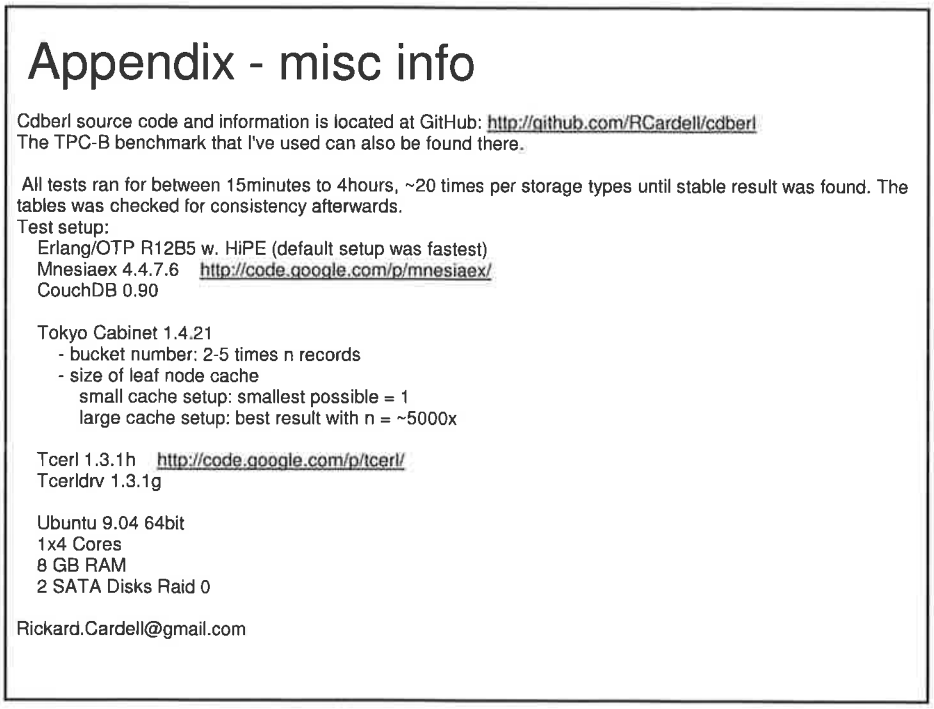
So...

- Mnesiaex is a fine interface
    - very easy to apply other database manager

- Tokyo Cabinet and Tcerl need more investigation in regard to durability issues.
- CouchDB, can be a part of the system, but probably not the general solution for Klarna



Questions?



Appendix - misc info

Cdberl source code and information is located at GitHub: http://github.com/RCardell/cdberl
The TPC-B benchmark that I've used can also be found there.

All tests ran for between 15minutes to 4hours, ~20 times per storage types until stable result was found. The tables was checked for consistency afterwards.

Test setup:
Erlang/OTP R12B5 w. HiPE (default setup was fastest)
Mnesiaex 4.4.7.6 http://code.google.com/p/mnesiaex/
CouchDB 0.90

Tokyo Cabinet 1.4.21
- bucket number: 2-5 times n records
- size of leaf node cache
  small cache setup: smallest possible = 1
  large cache setup: best result with n = ~5000x

Tcerl 1.3.1h http://code.google.com/p/tcerl/
Tcerldrv 1.3.1g

Ubuntu 9.04 64bit
1x4 Cores
8 GB RAM
2 SATA Disks Raid 0

Rickard.Cardell@gmail.com

Page 66

Anonymity in Erlang

Generally speaking, servers and clients in Erlang are implemented as named functions in named modules. Similarly, processes communicate via messages that have a statically-known structure, and specifically, with static tags, that serve as the \names" of the messages. This exposes a great deal of information about an Erlang application: The names of the modules, the name of the entry-point functions within the module, the \names" of the messages between the server and the client, etc. In this work, we show how higher-order functions, and some well-studied techniques from functional programming, can be used to obtain anonymity of servers and messages.

Guy Wiener

Guy Wiener is a Ph.D. student at Ben-Gurion University, studying software engineering and programming languages.

Mayer Goldberg

Mayer Goldberg did his PhD research in programming languages under the direction of Daniel P Friedman of Indiana University, and Olivier Danvy of the University of Aarhus. Mayer is currently a senior lecturer of computer science at Ben-Gurion University, Beer Sheva, Israel.

Anonymity in Erlang

Mayer Goldberg & Guy Wiener

Abstract

Generally speaking, servers and clients in Erlang are implemented as named functions in named modules. Similarly, processes communicate via messages that have a statically-known structure, and specifically, with static tags, that serve as the "names" of the messages. This exposes a great deal of information about an Erlang application: The names of the modules, the name of the entry-point functions within the module, the "names" of the messages between the server and the client, etc.

In this work, we show how higher-order functions, and some well-studied techniques from functional programming, can be used to obtain anonymity of servers and messages.

1 Introduction

The basic looping construct in Erlang is the tail-recursive function call. Erlang compiles these into code that is as efficient as `while`-loops in other programming languages.

Servers in Erlang are typically written using recursive functions [2, Chapter 8.2] and [3, Example 4-1]. The server is implemented as a function that receives a message, and as long as the server *continues* after receiving the message, the server will be re-invoked by calling the respective function tail-recursively.

Invoking a function on an Erlang node involves calling one of the `spawn` functions with the node (optional), the module name, the function name, and the list of its arguments. It is also possible to call `spawn` by passing it a closure as an argument. Any global name that is evaluated by this function must be defined on the node on which the spawned process will be running. If the name is undefined a run-time error will occur.

The fact that the server is recursive exposes several things about the underlying architecture:

- The name of the module in which the server appears needs to be known to the client.
- The client must also know the name of the server function and its arity.
- There must be, on the server, a mounted file system. If the module file is located on that file system, then the Erlang system must have read privileges to it. If the module file is located only on the client, the Erlang system must have write privileges to it

There are situations in which these constraints are undesirable. Erlang is currently available for many platforms, including thin clients and mobile devices. Forcing the client and the server to share files in advance prevents users from taking advantage of available Erlang nodes on resource-restricted platforms. Even if accessing the file system is permitted, it places a constraint on general-purpose servers. Instead of having the client hold the software that it wants to send to the general-purpose server, it forces the client to first update the server and only then run the new software on the server. Exposing the structure of the server module is also not always desired, since that makes the server writer more focused on the public functions of the module, rather then on the message protocol.

2 Replacing recursion with self-application

Any recursive call can be replaced by a call to a function reference that is passed as an argument. Consider the following example of the factorial function, given in the ubiquitous C programming language:

Recursive version:

```
int fact(int n) {
  return (n == 0) ?
         1 :
         n * fact(n - 1); }
```

Used as:

```
int n_fact = fact(n);
```

Self-applicative version:

```
int fact(void *f, int n) {
  reutrn (n == 0) ?
         1 :
         n * ((int (*)(void *, int))f)(f, n - 1); }
```

Used as:

```
int n_fact = fact(&fact, n);
```

Notice that the self-applicative version computes the factorial function only when the function is applied to itself, along with an integer argument. If we were to code the self-applicative version of factorial in Erlang, we would get:

```
fact(F, 0) -> 1;
fact(F, N) -> N * (F(F, N - 1)).
```

which can be invoked as follows:

```
fact(fun fact/2, N).
```

Note that the self-applicative function fact is not recursive. It can be written entirely using anonymous functions:

```
Fact =
fun (N) ->
  (fun (F) -> F(F, N) end)
  (fun (F, N) ->
    case N of
     0 -> 1;
     _ -> N * F(F, N - 1)
    end
   end)
end.
```

and this is how it can be invoked:

```
> Fact(5).
120
> Fact(7).
5040
```

Writing such functions can be simplified by slightly changing the interface of such functions, and abstracting out a general-purpose “recursion-maker”, known in the functional programming community as a “fixed-point combinator” [4, Page 178]:

```
Y1 =
fun (F) ->
 (fun (X) -> X(X) end)
 (fun (M) ->
   F (fun (Arg) ->
        (M(M))(Arg) end)
  end)
end.
```

And here is how Y1 could be used:

To define factorial:

```
Factorial =
Y1 (fun (Fact) ->
      fun (N) ->
        case N of
          0 -> 1;
          _ -> N * Fact(N - 1)
        end
      end
    end).
```

To define Fibonacci:

```
Fibonacci =
Y1 (fun (Fib) ->
      fun (N) ->
        case N of
          0 -> 0;
          1 -> 1;
          _ -> Fib(N - 1) +
               Fib(N - 2)
        end
      end
    end).
```

We can now use our functions as follows:

```
> Factorial(5).
120
> Factorial(7).
5040
```

```
> Fibonacci(10).
55
> Fibonacci(20).
6765
```

In languages that support variadic functions (functions that take any number of arguments), for example, Scheme, Python, etc., it is possible to extend Y1 so that it can be used to define any number of mutually-recursive functions, each taking any number of arguments [6, 7]. Because Erlang lacks support for defining variadic functions, we shall not pursue this direction here, but rather encode our functions using self-application directly.

3 Anonymous servers

As we have seen, it is possible to rewrite recursive functions, so that the recursive call is replaced by self-application. This allows recursive functions to be written anonymously, that is without using a globally-defined name for the function. Since a server in Erlang is a specific kind of recursive function, this same rewriting strategy can be used to create anonymous Erlang servers.

Here is a toy server for returning successive Fibonacci numbers:

```
FibonacciServer =
(fun (X) -> (X(X))(0, 1) end)
(fun (M) ->
  fun (N1, N2) ->
   fun () ->
    receive
     {fib, Pid} ->
       Pid ! {fib, N1},
       ((M(M))(N2, N1+N2))() ;
     restart -> ((M(M))(0, 1))() ;
     done -> ok
     end
   end
  end
 end).
```

The server answers 3 messages:

- {fib, *Pid*}, to have the next number in the Fibonacci sequence sent to the process ID *Pid*.
- restart, to restart the Fibonacci server.
- done, to stop the Fibonacci server.

This is the entire code for the server. Any client can pass this code onto any server in its cluster, via the **spawn** command. Here is how it can be used:

```
(one@erlang.edu)26> P = spawn('two@erlang.edu', FibonacciServer).
<5574.42.0>
(one@erlang.edu)27> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)28> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)29> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)30> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)31> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)32> flush().
Shell got {fib,0}
Shell got {fib,1}
Shell got {fib,1}
Shell got {fib,2}
Shell got {fib,3}
ok
(one@erlang.edu)33> P ! restart.
restart
(one@erlang.edu)34> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)35> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)36> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)37> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)38> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)39> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)40> P ! done.
done
(one@erlang.edu)41> flush().
Shell got {fib,0}
Shell got {fib,1}
Shell got {fib,1}
Shell got {fib,2}
Shell got {fib,3}
Shell got {fib,5}
ok
```

Notice that we do not know anything about the node on which the server is running other than its address: No paths, module names, function names, etc.

4 Anonymous messages

Now that we have anonymous functions, we move on to anonymous messages. Messages typically have a static structure to them, they are usually ordered tuples the first tuple of which is an atom that serves as a tag that identifies the type of message.

For many purposes it is desirable that message tags be privately shared between a client and a server, and not accessible otherwise. This is quite easily achieved as an extension to our anonymous server. For private tags in the following examples, we chose 20-digit long pseudo-randomly generated large integers. In a realistic application, we might use larger integers, or might prefer pseudo-randomly generated atoms of great length.

In the following example, we modified the Fibonacci server, to define a server-client pair in which the client asks for the next Fibonacci number using a message that is tagged via a secret tag, generated at run-time, and shared only between this specific server and client. The code could be used to create any number of such server-client pairs, each pair sharing its own secret pseudo-random tag. We left out the definition for the pseudo-random number generator Random/0, which just calls random:uniform/0 several times, and creates an integer of the right length.

```
ServerAndClient  =
fun () ->
 (fun (Mfib) ->
   {(fun (X) -> (X(X)) (0, 1) end)
      (fun (M) ->
        fun (N1, N2) ->
         fun () ->
          receive
           {Mfib, Pid} ->
             Pid ! {fib, N1},
             ((M(M))(N2, N1+N2))() ;
           restart -> ((M(M))(0, 1))() ;
           done -> ok
         end
        end
       end
     end),
    (fun (Pid) -> Pid ! {Mfib, self()}, ok end)}
   end)
 (Random())
end.
```

The following interaction shows how to define a server-client pair, how to spawn the server, how the client communicates with it, and what messages are received in response.

```
6> {Server, Client} = ServerAndClient().
{#Fun<erl_eval.20.117942162>,#Fun<erl_eval.6.35866844>}
7> Pid = spawn(Server).
<0.39.0>
8> Client(Pid).
ok
9> Client(Pid).
ok
   ...
14> Client(Pid).
ok
15> flush().
Shell got {fib,0}
Shell got {fib,1}
Shell got {fib,1}
Shell got {fib,2}
Shell got {fib,3}
Shell got {fib,5}
Shell got {fib,8}
ok
```

The value of Mfib is a 20-digit integer, and is unique to the client-server pair. We could just have easily have picked a 40-digit, or 100-digit integer, rendering impractical any attempt to arrive at the tag, either by guessing, or by systematically trying every integer.

In fact, we can make the tag even more secure, by having the client and server re-select a new tag every so often. An unauthorized client that might try to connect to the server by trying out numbers sequentially would be able to conclude nothing from past failures.

Here is a function for creating server-client pairs that re-select the tag after each use. Each call to **ServerClientAndInitialTag** would return a new triple of server, client, and an initial tag by which both the server and the client are synchronized:

```
ServerClientAndInitialTag =
fun () ->
 (fun (Mfib) ->
   {(fun (X) -> (X(X)) (0, 1, Mfib) end)
      (fun (M) ->
        fun (N1, N2, MfibMsg) ->
         fun () ->
          receive
           {MfibMsg, Pid} ->
             (fun (NewMfibMsg) ->
               Pid ! {fib, N1, NewMfibMsg},
               ((M(M))(N2, N1+N2, NewMfibMsg))()
              end)
             (Random());
           restart -> ((M(M))(0, 1))() ;
           done -> ok
         end
        end
       end
     end),
    (fun (Pid, Msg) -> Pid ! {Msg, self()}, ok end),
    Mfib}
  end)
  (Random())
end.
```

The following interaction shows how to define a server-client pair, what is the value of the first tag, how to spawn the server, how the client communicates with it, and what messages are received in response, and how the values of subsequent tags change according to the value of the responses from the server.

```
5> {Server, Client, InitialTag} = ServerClientAndInitialTag().
{#Fun<erl_eval.20.117942162>,#Fun<erl_eval.12.35291978>,
 684685090982776576}
6> Pid = spawn(Server).
<0.38.0>
7> Client(Pid, 684685090982776576).
ok
8> flush().
Shell got {fib,0,9230532871182784512}
ok
9> Client(Pid, 9230532871182784512).
ok
10> flush().
```

```
Shell got {fib,1,31133272937120710656}
ok
11> Client(Pid, 31133272937120710656).
ok
12> flush().
Shell got {fib,1,59651150244340162560}
ok
13> Client(Pid, 59651150244340162560).
ok
14> flush().
Shell got {fib,2,55825795815156547584}
ok
15> Client(Pid, 55825795815156547584).
ok
16> flush().
Shell got {fib,3,56212452643441508352}
ok
17> Client(Pid, 56212452643441508352).
ok
18> flush().
Shell got {fib,5,57882298363287674880}
ok
```

As can be seen, each message sent to the server has its own message tag that is synchronized between the server and the client. Each message sent from the server to the client contains, in addition to the `fib` tag and the next value in the Fibonacci sequence, the message tag to be used by the client in the subsequent request.

Any realistic client would need, of course, to receive the messages sent back by the server, and especially in this last example, where the client would be unable to communicate with the server unless it obtained the name of the subsequent message tag. For brevity, however, we presented only the simplest server-client pairs that use anonymous messages to communicate.

5 Discussion

Self-application has its origins in the λ-calculus and combinatory logic, where it is central to defining recursive functions. Functional programming languages, and especially those that are dynamically-typed, can use self-application in place of recursion. Exercises related to self-application and recursion are common in functional programming courses. For example, *Structure and Interpretation of Computer Programs* [1, Section 4.1.7, Page 393], and *The Little LISPer* [5, Chapter 9, Page 171].

In Erlang, self-application is more than just a programming exercise. While recursive functions cannot be passed between nodes, it is possible to pass functions that use self-application in place of recursion. We have thus been able to pass complete [albeit, small] servers among nodes. This has been done without requiring any access to the file system on the server host.

Once we have an entire server as a higher-order, non-recursive function that can be passed between nodes, it is straightforward to abstract over the message tags, and create server-client pairs that have their own, private message tags. For some added privacy, we can even have the message tags change between message calls.

In the servers we demonstrated, the message tags are selected pseudo-randomly during run-time. This is significant, because at no point does the source code contain the message tags. The message tags are private even if the source code is available.

If the pseudo-random number generator in these examples is replaced by a true, hardware-based, random number generator, a formidable level of privacy should be achieved.

References

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, McGraw-Hill Book Company, Second edition, 1996.

[2] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

[3] Francesco Cesarini and Simon Thompson. *Erlang Programming.* O'Reilly, June 2009.

[4] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*, volume I. North-Holland Publishing Company, 1958.

[5] Daniel P. Friedman and Matthias Felleisen. *The Little LISPer.* Science Research Associates, Inc, 1986.

[6] Mayer Goldberg. A Variadic Extension of Curry's Fixed-Point Combinator. In Olin Shivers, editor, *Proceedings of the 2002 ACM SIGPLAN Workshop on Scheme and Functional Programming*, pages 69–78, October 2002.

[7] Christian Queinnec. *LISP In Small Pieces.* Cambridge University Press, 1996.

Kostis Sagonas
HiPE compiler writer and tool developer

Opaque Data Types in Erlang

Abstract

Many languages provide mechanisms for programmers to declare abstract data types (ADTs), hide the details of their implementation, and allow manipulation of these ADTs only by controlled interfaces. This information hiding strategy allows the implementation of the ADT module to be changed without disturbing the client programs. In Erlang programs structural information about ADTs is exposed by pattern matching and type inspecting built-ins, making it very hard to guarantee that changes in the ADT's implementation will not have devastating effects on client's code. We have recently extended Erlang with the ability to declare opaque terms (i.e., terms whose structure should not be inspected outside their defining module) and detect violations of their opaqueness using Dialyzer. In this talk we will present this addition to the language and its capabilities, and will show interesting examples of code that (erroneously) depended on implementation details of commonly used library modules (ETS tables, gb_sets, gb_trees, etc.).

Biography

Kostis Sagonas is an academic who has been involved in the development of the Erlang language and its implementation since 1999. At Uppsala University, he has led the development team of the HiPE native code compiler. Together with his students he has contributed directly to various changes and additions to the Erlang language (e.g., bit-level pattern matching and bit-stream comprehensions, type and spec declarations, etc.) its libraries and runtime system support for concurrency. A hacker at heart, he has created widely used software development tools for Erlang, like dialyzer which by now has located literally thousands of bugs and software defects in Erlang programs. At NTUA in Athens, he has turned his obsession for clean designs and programs into yet another slick software tool for Erlang, called tidier, whose details are going to be unveiled for the first time at the Erlang Factory.

Improving your test code with Wrangler

In this talk we show the 'similar code' detection facilities of Wrangler, combined with its portfolio of refactorings, allow test code to be shrunk dramatically, under the guidance of the test engineer. The talk is illustrated with examples from Open Source and commercial Erlang development projects.

Huiqing Li

Inventor of Wrangler

Huiqing Li got her PhD at Kent University in September 2006 and works as a post doc in the EU project ProTest to further develop the refactoring tool Wrangler.

Simon Thompson

Creator of Wrangler and co-author of Erlang Programming

Simon Thompson is Professor of Logic and Computation in the Computing Laboratory of the University of Kent, where he has taught computing at undergraduate and postgraduate levels for the past twenty five years, and where he has been department head for the last six. His research work has centred on functional programming: program verification, type systems, and most recently development of software tools for functional programming languages. His team has built the HaRe tool for refactoring Haskell programs, and is currently developing Wrangler to do the same for Erlang. His research has been funded by various agencies including EPSRC and the European Framework programme. His training is as a mathematician: he has an MA in Mathematics from Cambridge and a D.Phil. in mathematical logic from Oxford. He has written four books in his field of interest; Type Theory and Functional Programming published in 1991; Miranda: The Craft of Functional Programming (1995), Haskell: The Craft of Functional Programming (2nd ed. 1999) and Erlang Programming (with Francesco Cesarini, 2009). Apart from the last, which is published by O'Reilly, these are all published by Addison Wesley.

Improving your test code with Wrangler

Huiqing Li[1], Adam Lindberg[2], Andreas Schumacher[3] and Simon Thompson[1]

[1] School of Computing, University of Kent, UK
{H.Li,S.J.Thompson}@kent.ac.uk
[2] Erlang Training and Consulting Ltd.
adam.lindberg@erlang-consulting.com
[3] Ericsson SW Research, Ericsson AB
andreas.schumacher@ericsson.com

Abstract. In this paper we show how the 'similar code' detection facilities of Wrangler, combined with its portfolio of refactorings, allow test code to be shrunk dramatically, under the guidance of the test engineer. This is illustrated by a sequence of refactorings of a test suite taken from an Erlang project at Ericsson AB.

Key words: Erlang, similar code, refactoring, testing, clone detection, generalisation, strategies

1 Introduction

Wrangler [1,2] is a tool that supports interactive refactoring of Erlang programs[4]. It is integrated with Emacs and now also with Eclipse. Wrangler itself is implemented in Erlang. Wrangler supports a variety of refactorings, as well as a set of "code smell" inspection functionalities, and facilities to detect and eliminate code clones. In this paper we explore a case study of test code provided by Ericsson SW Research[5]

Why is test code particularly prone to clone proliferation? One reason is that many people touch the code: a first few tests are written, and then others author more tests. The quickest way to write these is to copy, paste and modify an existing test, even if this is not the best way to structure the code, it can be done with a minimal understanding of the code.This observation applies equally well to long-standing projects, particul ırly with a large element of legacy code.

What comes out very clearly from thւ case study is the fact that refactoring or clone detection cannot be completely automated. In a preliminary experiment by one of the Wrangler developers (Thompson) the code was reduced by some 20% using a "slash and burn" approach, simply eliminating clones one by one. The result of this was – unsurprisingly – completely unreadable. It is only with the collaboration of project engineers, Lindberg and Schumacher, that we were

[4] We acknowledge the support of the 7th Framework Programme of the European Commission for the ProTest[3] project to which the work reported here contributes.

[5] We are grateful to Ericsson SW Research for permission to include portions of the code in this paper.

able to identify which clone candidates should be removed, how they could be named and parameterised and so forth. Moreover it requires domain insight to decide on which clones might not be removed. These questions are discussed at length here.

The remainder of the paper is organised as follows. Section 2 describes the Wrangler refactoring tool, and in particular the support that it provides for clone detection and removal. Section 3 describes the case study itself, and Section 4 highlights some lessons coming from the case study (cross-referenced with the stages of the case study). We then describe future work and conclude the paper.

2 Clone detection and removal with Wrangler

Duplicated code, or the existence of code clones, is one of the well-known bad code smells when refactoring and software maintenance is concerned. 'Duplicated code', in general, refers to a program fragment that is identical or similar to another, though the exact meaning of 'similar' might vary slightly between different application contexts.

While some code clones might have a sound reason for their existence [4], most clones are considered harmful to the quality of software, as code duplication increases the probability of bug propagation, the size of both the source code and the executable, compile time, and more importantly the maintenance cost [5,6].

The most obvious reason for code duplication is the reuse of existing code (by *copy*, *paste* and *modify* for example), logic or design. Duplicated code introduced for this reason often indicates program design problems such as the lack of encapsulation or abstraction. This kind of design problem can be corrected by refactoring out the existing clones in a later stage [7,8,9], it could also be avoided by first refactoring the existing code to make it more reusable, then reuse it without duplicating the code [8]. In the last decade, substantial research effort has been put into the detection and removal of clones from software systems; however, few such tools are available for functional programs, and there is a particular lack of tools that are integrated with existing programming environments.

Wrangler Wrangler [1,2] is a tool built in the School of Computing at The University of Kent, with support from the European Commission. Wrangler supports interactive refactoring of Erlang programs. It is integrated with Emacs and now also with Eclipse. Wrangler itself is implemented in Erlang. Wrangler supports a variety of refactorings, as well as a set of "code smell" inspection functionalities, and facilities to detect and eliminate code clones.

Wrangler provides a 'similar' code detection approach based on the notion of *anti-unification* [10,11] to detecting code clones in Erlang programs, as well as a mechanism for automatic clone elimination under the user's control. The *anti-unifier* of two terms denotes their *least-general common abstraction*, therefore captures the common syntactical structure of the two terms.

In general, we say two expression/expression sequences, `A` and `B`, are *similar* if there exists a 'non-trivial' least-general common abstraction, `C`, and two sets of substitutions σ_1 and σ_2 which take `C` to `A` and `B` respectively. By 'non-trivial' we mean mainly that the size of the least-general common abstraction should satisfy some threshold, but certainly other conditions could be added.

This clone detection approach is able, for example, to spot that the two expressions `((X+3)+4)` and `(4+(5-(3*X)))` are similar as they are both instances of the expression `(Y+Z)`, and so both instances of the function

```
add(Y,Z) -> Y+Z.
```

Our approach uses the Abstract Syntax Tree (AST) annotated with static semantic information as the internal representation of Erlang programs. Scalability, one of the major challenges faced by AST-based clone detection approaches, is achieved by a two-phase clone detection technique. More details of the process can be found in [12].

Wrangler clone detection Wrangler provides facilities for finding both identical and similar code. Two pieces of code are said to be *identical* if they are the same when the values of literals and the names of variables are ignored, while the binding structures are the same. The Wrangler definition of *similarity* is given above. For both identical and similar code, two operations are possible:

Detection This operation will identify all code clones (up to identity or similarity) in a module or across a project. For each clone the common generalisation for the clone is generated in the report, and can be cut and pasted into the module prior to clone elimination.

Search This operation allows the identification of all the code that is identical or similar to a *particular selection*, so is directed rather than speculative.

Examples of both will be seen in the case study.

3 The Case Study

The case study examined part of an Erlang implementation of the SIP (Session Initiation Protocol) [13]. As a part of SIP message processing it is possible to transform messages by applying rewriting rules to messages. This SIP message manipulation (SMM) is tested by a test suite contained in the file `smm_SUITE.erl`, which is our subject here.

The size of the sequence of versions of the files – in lines of code – is indicated in Figure 1, which shows that the code has been reduced by about 25% through these transformations. As we discuss at the end of this section there is still considerable scope for clone detection and elimination, and this might well reduce the code by a further few hundred lines of code.

Version	LOC	Version	LOC	Version	LOC
1	2658	6	2218	10	2149
2	2342	7	2203	11	2131
3	2231	8	2201	12	2097
4	2217	9	2183	13	2042
5	2216				

Fig. 1: The size of the refactored files

The sequence of transformations

In this section we give an overview of the particular steps taken in refactoring the SMM test code.

Step 1 We begin by generating a report on similar code in the module. 31 clones are detected, with the most common on being cloned 15 times. The generalisation suggested in the report is

```
new_fun() ->
    SetResult = ?SMM_IMPORT_FILE_BASIC(?SMM_RULESET_FILE_1, no),
    ?TRIAL(ok, SetResult),
    %% AmountOfRuleSets should correspond to the amount of rule sets in File.
    AmountOfRuleSets = ?SMM_RULESET_FILE_1_COUNT,
    ?OM_CHECK(AmountOfRuleSets, ?MP_BS, ets, info, [sbgRuleSetTable, size]),
    ?OM_CHECK(AmountOfRuleSets, ?SGC_BS, ets, info, [smmRuleSet, size]),
    AmountOfRuleSets.
```

which shows that the code is literally *repeated* sixteen times. This function definition can be cut and pasted into the test file, and all the clones folded against it. Of course, it needs to be renamed: we choose to call it `import_rule_set_file_1` since the role of this function is to import rule sets which determine the actions taken by the SMM processor, and this import is a part of the common setup in a number of different test cases. The function can be renamed when it is first introduced, or after folding against it, using the *Rename Function* refactoring.

Step 2 Looking again at similar code detection we find a twelve line code block that is repeated six times. This code creates two SMM filters, and returns a tuple containing names and keys for the two filters. This is a common pattern, under which the extracted clone returns a tuple of values, which are assigned to a tuple of variables on function invocation, thus:

```
{FilterKey1, FilterName1, FilterState, FilterKey2, FilterName2}
      = create_filter_12()
```

We have named the function create_filter_12; this reflects a general policy of not trying to anticipate general names for functions when they are introduced. Rather, we choose the most specific name, generalising it – or indeed the functionality itself — at a later stage if necessary, using Wrangler.

Step 3 At this step a 21 line clone is detected:

```
new_fun() ->
    {FilterKey1, FilterName1, FilterState, FilterKey2, FilterName2}
        = create_filter_12(),
    ?OM_CHECK([#smmFilter{key=FilterKey1,
              filterName=FilterName1,
              filterState=FilterState,
              module=undefined}],
          ?SGC_BS, ets, lookup, [smmFilter, FilterKey1]),
    ?OM_CHECK([#smmFilter{key=FilterKey2,
              filterName=FilterName2,
              filterState=FilterState,
              module=undefined}],
          ?SGC_BS, ets, lookup, [smmFilter, FilterKey2]),
    ?OM_CHECK([#sbgFilterTable{key=FilterKey1,
                   sbgFilterName=FilterName1,
                   sbgFilterState=FilterState}],
          ?MP_BS, ets, lookup, [sbgFilterTable, FilterKey1]),
    ?OM_CHECK([#sbgFilterTable{key=FilterKey2,
                   sbgFilterName=FilterName2,
                   sbgFilterState=FilterState}],
          ?MP_BS, ets, lookup, [sbgFilterTable, FilterKey2]),
    {FilterName2, FilterKey2, FilterKey1, FilterName1, FilterState}.
```

Inspecting this shows up a smaller clone, encapsulated in the suggested function

```
new_fun(FilterState, FilterKey2, FilterName2) ->
    ?OM_CHECK([#sbgFilterTable{key=FilterKey2,
                   sbgFilterName=FilterName2,
                   sbgFilterState=FilterState}],
          ?MP_BS, ets, lookup, [sbgFilterTable, FilterKey2]).
```

which we choose to replace first: as a general principle we found it more useful to replace clones *bottom up*. The clone was replaced by the function

```
check_filter_exists_in_sbgFilterTable(FilterKey, FilterName,FilterState) ->
    ?OM_CHECK([#sbgFilterTable{key=FilterKey,
                   sbgFilterName=FilterName,
                   sbgFilterState=FilterState}],
          ?MP_BS, ets, lookup, [sbgFilterTable, FilterKey]).
```

where as well as renaming the function and variable names, the order of the variables is changed. This can be done simply by editing the list of arguments, because before folding against the function there are no calls to it, since it is newly introduced.

Steps 4–5 Introduce two variants of `check_filter_exists_in_sbgFilterTable`:

- In the first the check is for the filter occurring only once in the table, so that a call to `ets:tab2list` replaces the earlier call to `ets:lookup`.
- In the second the call is to a different table, `sbgFilterTable` being replaced by `smmFilter`.

Arguably these three alternatives could have been abstracted into a common generalisation, but it was felt by the engineers that each of the three functions each encapsulated a meaningful activity, whereas a generalisation would have had an unwieldy number of parameters as well as being harder to name appropriately.

Step 6 Erlang provides two mechanisms for finding out whether the code for a module `M` is loaded:

```
    erlang:module_loaded(M) -> true | false
    code:is_loaded(M) -> {file, Loaded} | false
```

Use of the former is deprecated outside the code server, but both are used in this file. We want to *remove the deprecated calls*, all of which are symbolic calls in contexts like:

```
    ?OM_CHECK(false, ?SGC_BS, erlang, module_loaded, [FilterAtom1])
```

So what we do is to define a new function, in which we abstract over module name, the type of blade and the expected result of the call to `erlang:module_loaded`.

```
code_is_loaded(ModuleName, BS, Result) ->
    ?OM_CHECK(Result, BS, erlang, module_loaded, [ModuleName]).
```

We then fold against this definition to remove all calls to `erlang:module_loaded`, expect for that in the definition of `code_is_loaded` itself. We can then write a *different* definition of this function, which implements the same functionality using the other primitive:

```
code_is_loaded(BS, ModuleName, false) ->
    ?OM_CHECK(false, BS, code, is_loaded, [ModuleName]).
code_is_loaded(BS, ModuleName, true) ->
    ?OM_CHECK({file,atom_to_list(ModuleName)}, BS,
              code, is_loaded, [ModuleName]).
```

At this point it is possible to stop, having introduced the `code_is_loaded` function. Alternatively, in order to keep the code as close as possible to its previous version, we can *inline* this function definition. In the next step we will see another reason for doing this inlining.

Step 7 We note that as well as finding symbolic calls to `code:is_loaded` within the `OM_CHECK` macro call, it is also called within `CH_CHECK`. We are unable to replace a macro call by a variable, and so we write – by hand – a generalisation in which the macro call is determined by an atom parameter

```
code_is_loaded(BS, om, ModuleName, false) ->
      ?OM_CHECK(false, BS, code, is_loaded, [ModuleName]);
code_is_loaded(BS, om, ModuleName, true) ->
      ?OM_CHECK({file, atom_to_list(ModuleName)},
             BS, code, is_loaded, [ModuleName]);
code_is_loaded(BS, ch, ModuleName, false) ->
      ?CH_CHECK(false, BS, code, is_loaded, [ModuleName]);
code_is_loaded(BS, ch, ModuleName, true) ->
      ?CH_CHECK({file, atom_to_list(ModuleName)},
             BS, code, is_loaded, [ModuleName]).
```

It is here that inlining of the `code_is_loaded` function in step 6 is valuable: it allows us to deal with *premature generalisation*, under which we find that we want further to generalise a function without layering a number of intermediate calls: we inline the earlier generalisations and then build the more general function in a single step.

Steps 8,9 In this step a ten line clone was identified found, but rather than replacing that – which combines a number of operations – it was decided to look at sub-clones, and this indicated code used 22 times in the module, extracted as

```
check_add_rule_set_to_filter(FilterKey, FilterName, RuleSetName,
                 FilterRuleSetPosition, Result) ->
    AddResult =
        ?SMM_ADD_RULE_SET_TO_FILTER(FilterKey, FilterName,
                    RuleSetName, FilterRuleSetPosition),
    ?TRIAL(Result, AddResult).
```

This gives the ninth version of the code, and two similar sub-clones are extracted thus:

```
check_ruleset_name_in_filter(FilterName, RuleSetName) ->
    {ok, RuleSetKey} = ?SMM_NAME_TO_KEY(sbgRuleSetTable, RuleSetName),
    check_ruleset_key_in_filter(RuleSetKey, [[FilterName]]),
    RuleSetKey.

check_ruleset_key_in_filter(RuleSetKey, Result) ->
    ?OM_CHECK(Result,
          ?MP_BS, ets, match, [sbgIsmFilterRuleSetUsageTable,
                   {'_', {RuleSetKey, '_'}, '_', '$1'}]).
```

which gives the tenth version of the code.

Step 10 Clone detection now gives this clone candidate:

```
new_fun(FilterName, NewVar_1) ->
    FilterKey = ?SMM_CREATE_FILTER_CHECK(FilterName),
    %%Add rulests to filter
    RuleSetNameA = "a",
    RuleSetNameB = "b",
    RuleSetNameC = "c",
    RuleSetNameD = "d",
    ... 16 lines which handle the rules sets are elided ...
    %%Remove rulesets
    NewVar_1,
    {RuleSetNameA, RuleSetNameB, RuleSetNameC, RuleSetNameD, FilterKey}.
```

The main body of the function sets up four rule sets and adds them to a filter, but the function identified contains extraneous functionality at the start and end:

- the filter key is created as the first action: `FilterKey = ...`, and
- (in at least one of the clones) the rulesets are removed thus: `NewVar_1` .

Instead, the body is extracted thus:

```
add_four_rulesets_to_filter(FilterName, FilterKey) ->
    RuleSetNameA = "a",
    RuleSetNameB = "b",
    RuleSetNameC = "c",
    RuleSetNameD = "d",
    ... 16 lines which handle the rules sets are elided ...
    {RuleSetNameA, RuleSetNameB, RuleSetNameC, RuleSetNameD}.
```

in which the final action doesn't need to be passed in as a parameter, and the `FilterKey` becomes a parameter rather than a component of the result.

Step 11 In a similar way to the previous step, a lengthy clone is identified, but in the abstraction the first line is omitted, making it an abstraction without parameters: `setup_rulesets_and_filters`.

Step 12 This final step consists of a sequence of stages concerned with refactoring the form of data representation in dealing with

- sets of attributes, which are transformed into named lists, such as

  ```
  Attributes_1=[LineW1, ColumnW1, TypeAtom, FailingRuleSetW1, ResasonW1],
  ```

- other sets of attributes are represented by 0-ary functions:

  ```
  ruleset_error_attributes() -> [?sbgRuleSetErrorLineNumber, ..., ...
  ```

Finally, we can replace an explicit zipping together of lists by the uses of function lists:zip/2, and this gives us a much better structured function:

```
import_warning_rule_set(main) ->
    %% Since the rule set file contains errors, no rule sets will be imported.
    ...
    AttributeNames = [line_number, column, type, failing_rule_set, error_reason],
    Key1 = 1,
    ?ACTION("Do SNMP Get operations on the"
        "sbgRuleSetErrorTable with key ~p.~n", [Key1]),
    GetResult1 = ?SMM_SMF_GET(rule_set_error, [{key, Key1}], AttributeNames),
    LineW1 = 4,
    ColumnW1 = ?SMM_RULE_SET_WARN_1_COL,
    TypeAtom = warning,
    TypeMibVal = ?SMM_REPORT_TYPE(TypeAtom),
    FailingRuleSetW1 = " ",
    ResasonW1 = ?SMM_RULE_SET_WARN_1_REPORT(LineW1),
    Attributes_1 = [LineW1, ColumnW1, TypeAtom, FailingRuleSetW1, ResasonW1],
    ?TRIAL(lists:zip(AttributeNames, Attributes_1), GetResult1),
    ... 40 lines elided ... .
```

an incidental benefit of the inspection was to reveal that the lines

```
TypeAtom = warning,
TypeMibVal = ?SMM_REPORT_TYPE(TypeAtom),
```

were repeated, doubtless a cut-and-paste error, which went undetected because the pattern matches succeeded at the second occurrence.

Continuing the case study: further clone detection

The work reported here produced a sequence of twelve revisions of the code, but it is possible to make further revisions. In this section we look at a selection of reports from similar and identical code search, and comment on some of the potential clones identified.

Similar code The similar code detection facility with the default parameter values reports 16 further clones, each duplicating code once. The total number of duplicated lines here is 193, and so a reduction of some 145 lines could be made by replacing each clone into a function definition plus two function calls.

Looking for similar code with the similarity parameter reduced to 0.5 rather than the default of 0.8 reports 47 clones, almost three times as many. Of these, eight duplicate the code twice (that is, there are *three* instances of the code clone) and some of these provide potential clients for replacement. However, not all of them appear to be good candidates for replacement. Take the example of

```
/Users/simonthompson/Downloads/smm_SUITE13.erl:1755.4-1761.50:
This code has been cloned twice:
/Users/simonthompson/Downloads/smm_SUITE13.erl:1772.4-1778.50:
/Users/simonthompson/Downloads/smm_SUITE13.erl:1789.4-1795.50:

The cloned expression/function after generalisation:

new_fun(FilterAtom1, FilterAtom2, NewVar_1, NewVar_2, NewVar_3) ->
    NewVar_1,
    code_is_loaded(?SGC_BS, om, FilterAtom1, NewVar_2),
    code_is_loaded(?MP_BS, om, FilterAtom1, false),
    code_is_loaded(?SGC_BS, om, FilterAtom2, NewVar_3),
    code_is_loaded(?MP_BS, om, FilterAtom2, false).
```

This code has three parameters, and the first is an arbitrary expression, `NewVar_1`, evaluated first in the function body. A more appropriate candidate is given by omitting this expression, and giving the generalisation

```
new_fun(FilterAtom1, FilterAtom2, NewVar_1, NewVar_2) ->
    code_is_loaded(?SGC_BS, om, FilterAtom1, NewVar_1),
    code_is_loaded(?MP_BS, om, FilterAtom1, false),
    code_is_loaded(?SGC_BS, om, FilterAtom2, NewVar_2),
    code_is_loaded(?MP_BS, om, FilterAtom2, false).
```

This particular generalisation has a similarity score of more than 0.8, but does not appear in the default report because it involves only 4 expressions, and the default cut-off is a sequence of at least 5.

Identical code The standard report to detect identical code reports 87; the number is larger here because the default threshold for reporting here is to consist of at least 20 tokens rather than 5 expressions or more. However, a number of *over-generalisations* result from this report; for example, the function

```
new_fun(ModuleName1, NewVar_1, NewVar_2, NewVar_3, NewVar_4) ->
    code_is_loaded(?SGC_BS, NewVar_1, ModuleName1, NewVar_2),
    code_is_loaded(?SGC_BS, NewVar_3, ModuleName1, NewVar_4).
```

is reported as occurring 23 times. Arguably, generalising to replace these two expressions will not result in code that is more readable than it is already: it clearly states that it checks for two pieces of code being loaded.

Carrying on There are clearly some more clones that might be detected, but as the work progresses the effort involved in identifying and replacing code clones becomes more than the value of transforming the code in this way, and so the engineers performing the refactoring will need to decide when it is time to put the work aside.

4 Lessons learned

This section highlights the lessons learned during the activity reported in the previous section, cross-referring to the steps of that process when appropriate.

Inlining is a useful refactoring There is a clear use case for function *inlining* or *unfolding* when performing a series of refactorings based on clone elimination [Step 7]. The scenario is one of *premature generalisation* thus:

- identify common code, and generalise this, introducing a function for this generalisation;
- subsequently identify that there is a further generalisation of the original code, which could benefit from being generalised;
- the problem is that some of the original code disappeared in the first stage of generalisation, and so it needs to be inlined in order to generalise it further.

Of course, it would be possible to keep the intermediate generalisation as well as the final one, but in general that makes for less readable code, requiring the reader to understand two function definitions and interfaces rather than one.

Inlining is also useful to support a limited form of API migration [Step 6].

Bottom-up is better than top-down We looked at ways in which clones might be removed, and two approaches seem appropriate: bottom-up and top-down. In the latter case we would remove the largest clones first, while in the other approach we would look for small clones first, particularly those which are the most common. We decided to use the *bottom-up* approach for two reasons [Steps 3, 8, 9].

- Using this it is much easier to identify pieces of functionality which can easily be *named* because they have an identifiable purpose.
- it is also likely that these will not have a huge number of parameters, and in general we look for code which is not over-general.

Finally, there is the argument that – to a large extent, at least – it should not matter about the order in which clone removal takes place, since a large clone will remain after small clones are removed, and *vice versa.*

Clone removal cannot be fully automated What we have achieved in this example is clearly *semi-automated*: we have the Wrangler support for identifying candidates for clones but they may well need further analysis and insight from uses to identify what should be done. For example,

- Is there a spurious last action which belongs to the next part of the code, but which just happens to follow the clone when it is used? If so, it should not be included in the clone [Steps 10,11]. This can also apply to actions at the start of the identified code segment.

- Another related reason for this might be that this last operation adds another component to the return tuple of an extracted function; we should aim to keep these small.
- We might find that we identify behaviour of interest which occurs close to an identified clone; then use similar expression search to explore further.
- An identified clone may contain two pieces of separate functionality which are used together in many cases, but not in all cases of interest. Because of the thresholding of clone parameters it might be that we only see the larger clone because the smaller one is below the threshold chosen for the similar code report [Steps 3, 8, 9].

Self-documenting code Is it always useful to name values, as in

```
BlahMeaning = ... blah_exp ... ,
FooMeaning  = ... foo_exp ... ,
Result = f(BlahMeaning, FooMeaning),
```

rather than

```
Result = f(... blah_exp ... , ... foo_exp ...)
```

as the former case is *self documenting* in a way that the latter is not. On the other hand, is it the responsibility of the client of an API, here for the function `f`, to document this API at its calling points? If it is to be documented at a calling point it can be done by choice of variable names, or by suitably placed comments.

Naming values How should constants best be named in Erlang code. Three options are possible: namely definition by

- a macro definition
- a function of arity 0
- a local variable

The code at hand does the first and last: the second was added during the refactoring process.

Another option presents itself: instead of worrying about what is contained in a set of variables, should a `record` be used instead? This has some advantages, but records have drawbacks in Erlang. Names are not first class, so cannot be passed as parameters or values, which is something that can be used in practice, such as passing a list of field names to the `zip/2` function [Step 13].

Improvements to Wrangler The case study also brought to light a number of improvements that might be made to Wrangler. Inlining was the most important, and is now included in the latest release of the system.

It was also suggested that a number of options could be added to the *Code Inspector*: this highlights "bad smells" and other notable code features, such as:

variables used only once, variables not used, and rebinding of variables (that it, bound variables occurring in a pattern match).

Finally, the question was raised about how much refactoring sequences used on one module could be *reused* in refactoring another. This question of memoisation or scripting merits further work.

5 Conclusions and Future Work

As we have reported, the exercise here was only possible as a collaboration between the developers of Wrangler and engineers engaged in developing and testing the target system. Together it was possible substantially to re-engineer the test code to make it more compact and more structured. As well as illustrating the way in which Wrangler can be used, we were able to provide guidelines on refactoring test code in Erlang which can also be applied to systems written in other languages and paradigms.

Wrangler is under active development as a part of the ProTest project, and the insights gained here will feed into its further development.

References

1. Li, H., Thompson, S., Lövei, L., Horváth, Z., Kozsik, T., Víg, A., Nagy, T.: Refactoring Erlang Programs. In: EUC'06, Stockholm, Sweden (November 2006)
2. Li, H., Thompson, S., Orosz, G., T"oth, M.: Refactoring with Wrangler, updated. In: ACM SIGPLAN Erlang Workshop 2008, Victoria, British Columbia, Canada
3. ProTest: Property based testing. `http://www.protest-project.eu/`
4. Kapser, C., Godfrey, M.W.: "Clones Considered Harmful" Considered Harmful. In: Proc. Working Conf. Reverse Engineering (WCRE). (2006)
5. Roy, C.H., Cordy, R.: A Survey on Software Clone Detection Research. Technical report, School of Computing, Queen's University at Kingston, Ontario, Candada
6. Monden, A., Nakae, D., Kamiya, T., Sato, S., Matsumoto, K.: Software Quality Analysis by Code Clones in Industrial Legacy Software. In: METRICS '02, Washington, DC, USA (2002)
7. Balazinska, M., Merlo, E., Dagenais, M., Lague, B., Kontogiannis, K.: Partial Redesign of Java Software Systems Based on Clone Analysis. In: Working Conference on Reverse Engineering. (1999) 326–336
8. M. Fowler: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
9. Higo, Y., Kamiya, T., Kusumoto, S., Inoue, K.: ARIES: Refactoring Support Environment Based on Code Clone Analysis. In: IASTED Conf. on Software Engineering and Applications. (2004) 222–229
10. Plotkin, G.D.: A note on inductive generalization. Machine Intelligence **5** (1970) 153–163
11. Reynolds, J.C.: Transformational systems and the algebraic structure of atomic formulas. Machine Intelligence **5** (1970) 135–151
12. Li, H., Thompson, S.: Similar Code Detection and Elimination for Erlang Programs. In: PADL10. (2010) to appear.
13. SIP: Session Initiation Protocol. `http://tools.ietf.org/html/rfc3261`

Patrik Nyblom
Erlang virtual machine developer

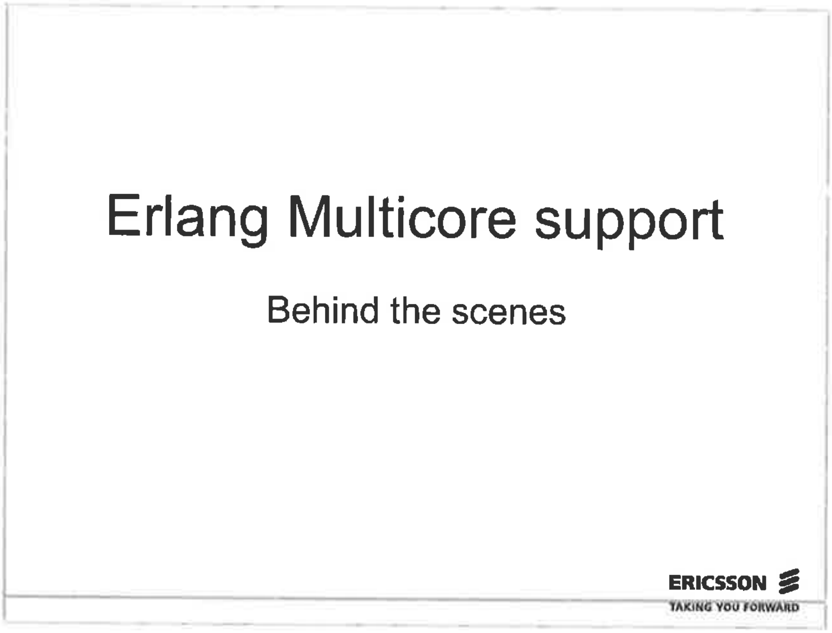
Erlang SMP support - behind the scenes

Abstract

This talk will tell you the story about how Erlang got multicore support and will give you all the gory details about utilizing multicore processors in a conventional programming language. I'll tell you what we've done at OTP so that you, as an Erlang programmer, can sit back and enjoy the fact that you don't have to bother with such things!

Biography

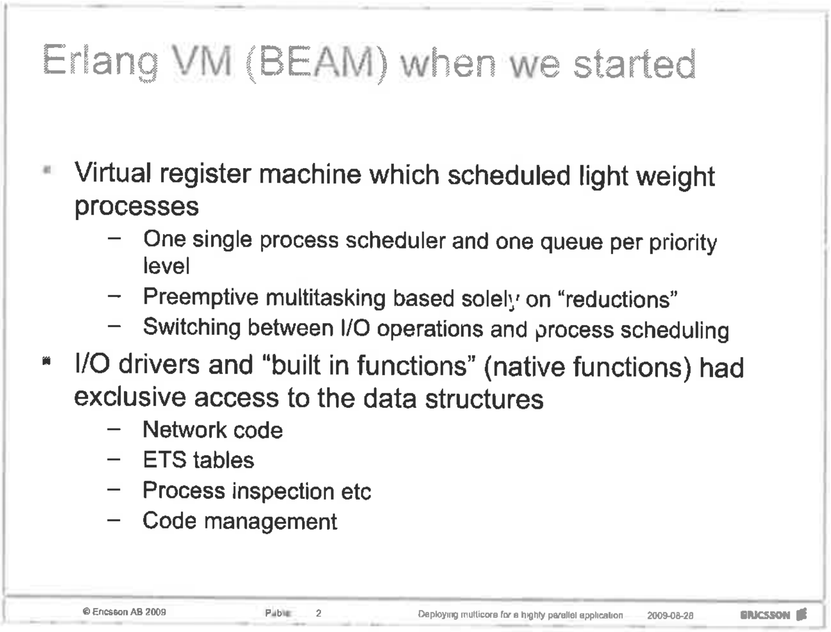
Patrik Nyblom works in the OTP project developing the Erlang virtual machine and has done so for the past ten years.



Erlang Multicore support

Behind the scenes

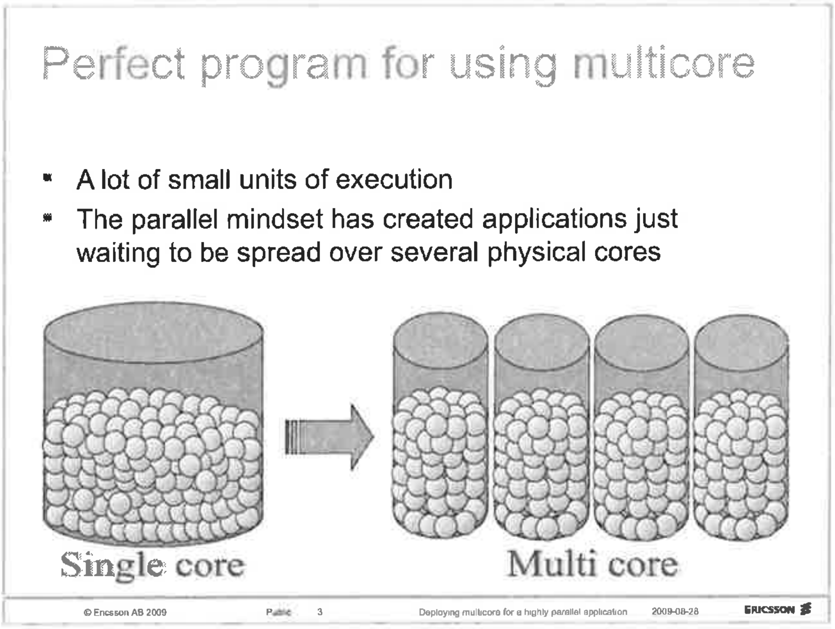
ERICSSON
TAKING YOU FORWARD



Erlang VM (BEAM) when we started

- Virtual register machine which scheduled light weight processes
  - One single process scheduler and one queue per priority level
  - Preemptive multitasking based solely on “reductions”
  - Switching between I/O operations and process scheduling
- I/O drivers and “built in functions” (native functions) had exclusive access to the data structures
  - Network code
  - ETS tables
  - Process inspection etc
  - Code management

© Ericsson AB 2009 Public 2 Deploying multicore for a highly parallel application 2009-08-28 ERICSSON



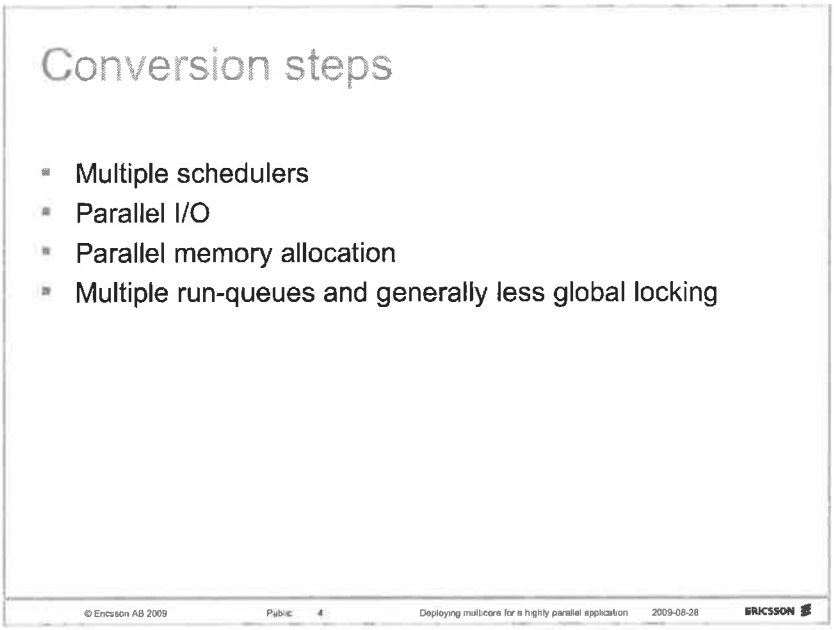
Perfect program for using multicore

- A lot of small units of execution
- The parallel mindset has created applications just waiting to be spread over several physical cores

Single core

Multi core

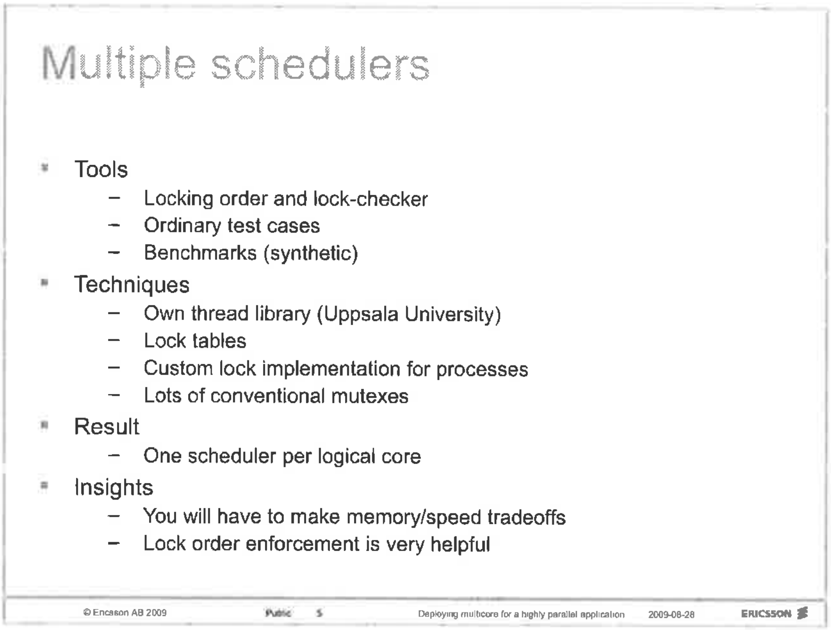
© Ericsson AB 2009 Public 3 Deploying multicore for a highly parallel application 2009-08-28 ERICSSON



Conversion steps

- Multiple schedulers
- Parallel I/O
- Parallel memory allocation
- Multiple run-queues and generally less global locking

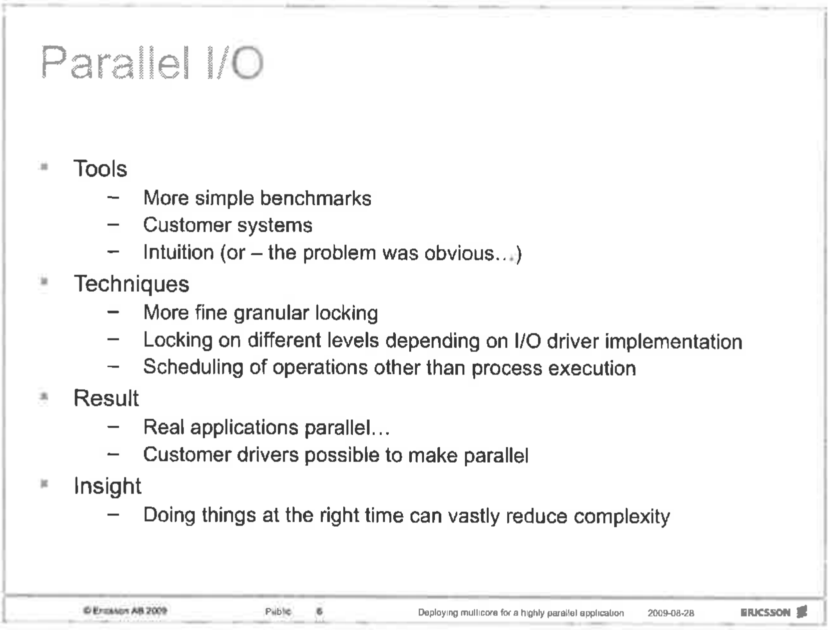
© Ericsson AB 2009 Public 4 Deploying multi-core for a highly parallel application 2009-08-28 ERICSSON



Multiple schedulers

- Tools
  - Locking order and lock-checker
  - Ordinary test cases
  - Benchmarks (synthetic)
- Techniques
  - Own thread library (Uppsala University)
  - Lock tables
  - Custom lock implementation for processes
  - Lots of conventional mutexes
- Result
  - One scheduler per logical core
- Insights
  - You will have to make memory/speed tradeoffs
  - Lock order enforcement is very helpful

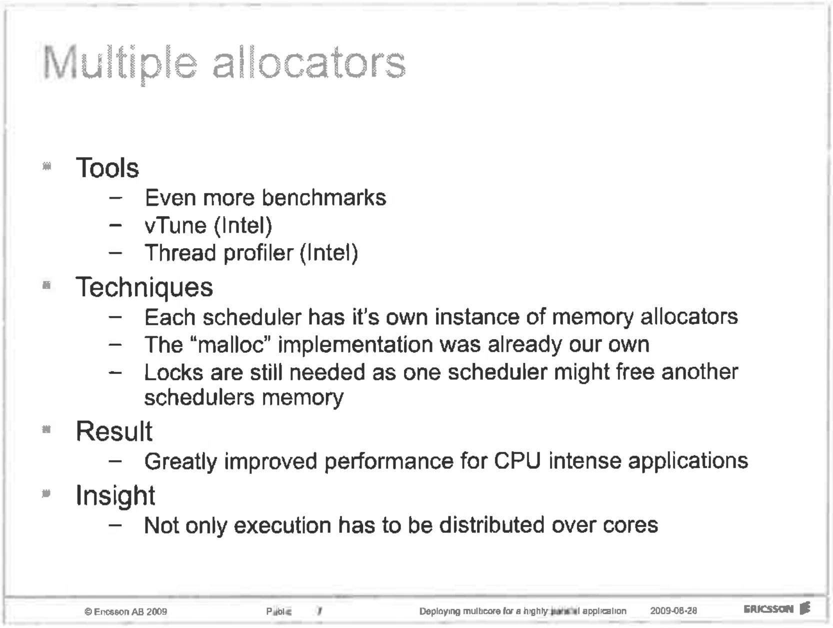
© Ericsson AB 2009 Public 5 Deploying multicore for a highly parallel application 2009-08-28 ERICSSON



Parallel I/O

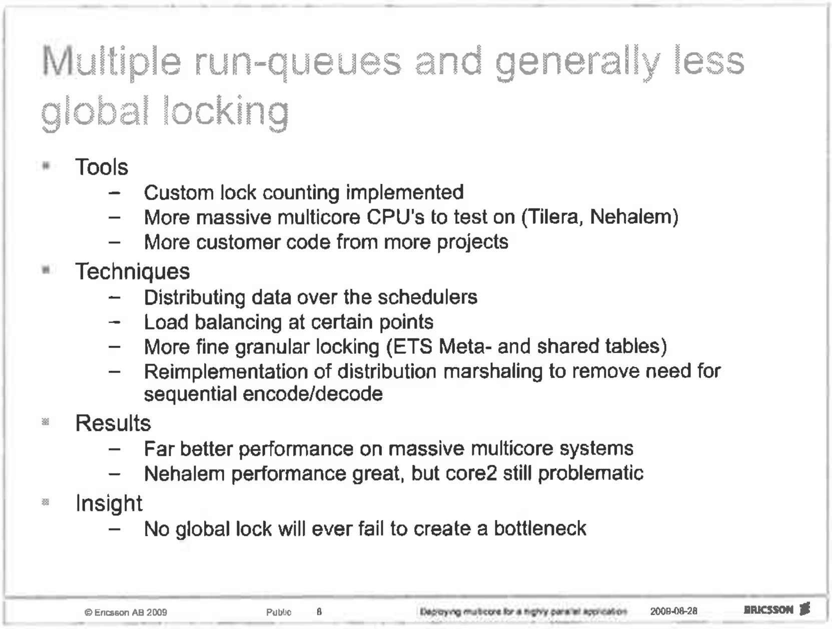
- Tools
  - More simple benchmarks
  - Customer systems
  - Intuition (or – the problem was obvious…)
- Techniques
  - More fine granular locking
  - Locking on different levels depending on I/O driver implementation
  - Scheduling of operations other than process execution
- Result
  - Real applications parallel…
  - Customer drivers possible to make parallel
- Insight
  - Doing things at the right time can vastly reduce complexity

© Ericsson AB 2009 Public 6 Deploying multicore for a highly parallel application 2009-08-28 ERICSSON



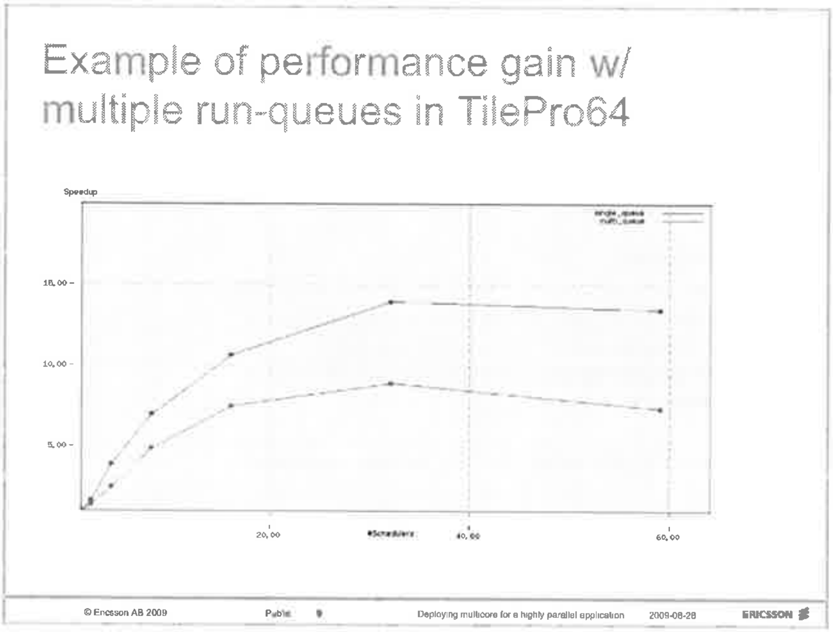
Multiple allocators

- Tools
  - Even more benchmarks
  - vTune (Intel)
  - Thread profiler (Intel)
- Techniques
  - Each scheduler has it's own instance of memory allocators
  - The "malloc" implementation was already our own
  - Locks are still needed as one scheduler might free another schedulers memory
- Result
  - Greatly improved performance for CPU intense applications
- Insight
  - Not only execution has to be distributed over cores



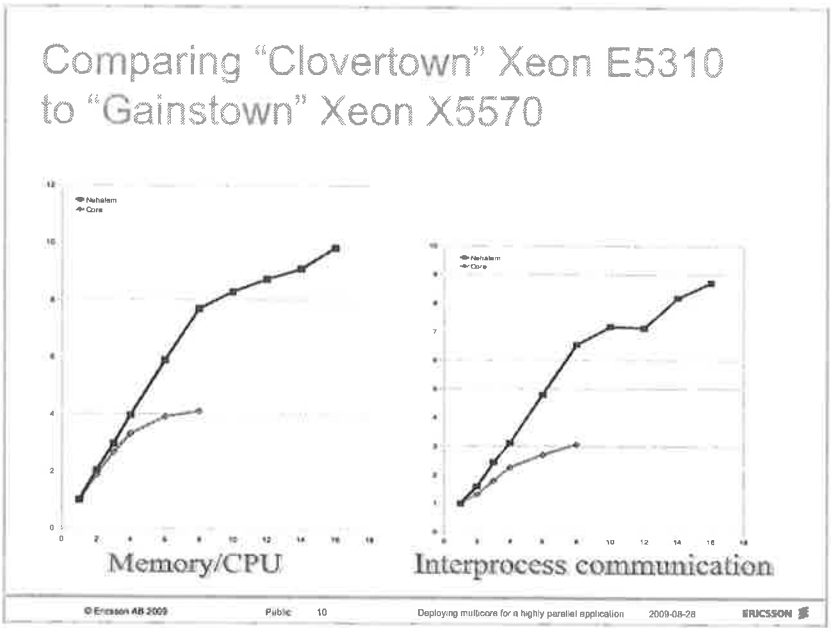
Multiple run-queues and generally less global locking

- Tools
  - Custom lock counting implemented
  - More massive multicore CPU's to test on (Tilera, Nehalem)
  - More customer code from more projects
- Techniques
  - Distributing data over the schedulers
  - Load balancing at certain points
  - More fine granular locking (ETS Meta- and shared tables)
  - Reimplementation of distribution marshaling to remove need for sequential encode/decode
- Results
  - Far better performance on massive multicore systems
  - Nehalem performance great, but core2 still problematic
- Insight
  - No global lock will ever fail to create a bottleneck



Example of performance gain w/ multiple run-queues in TilePro64

© Ericsson AB 2009 Public 9 Deploying multicore for a highly parallel application 2009-08-28 ERICSSON

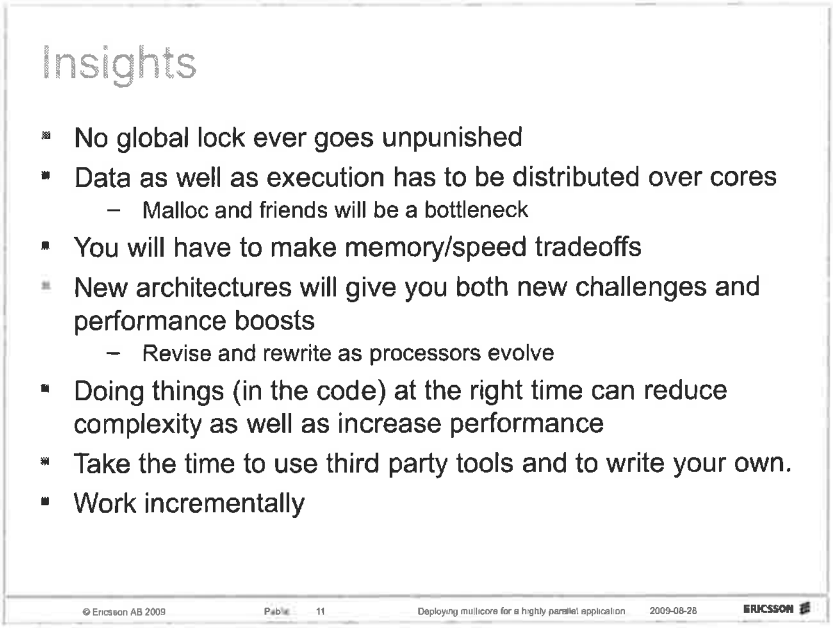


Comparing "Clovertown" Xeon E5310 to "Gainstown" Xeon X5570

Memory/CPU

Interprocess communication

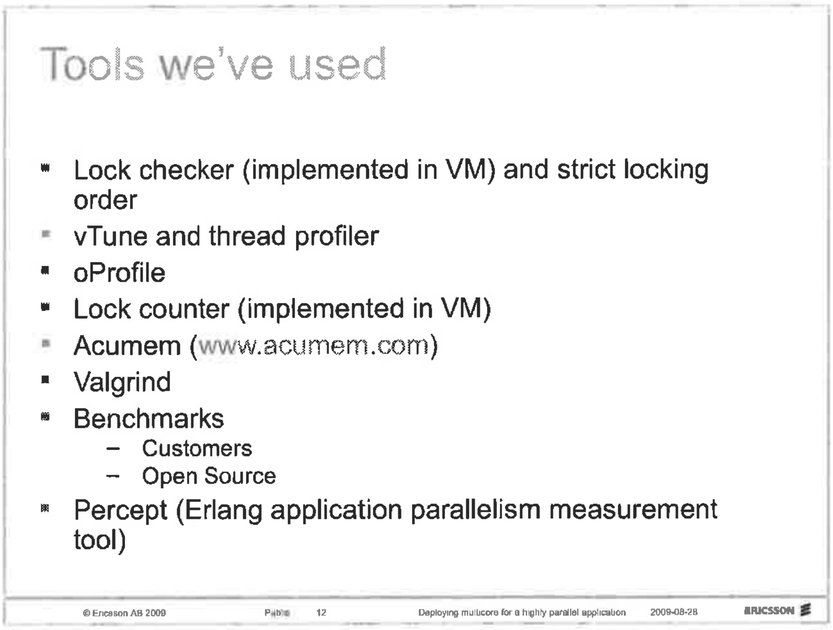
© Ericsson AB 2009 Public 10 Deploying multicore for a highly parallel application 2009-08-28 ERICSSON



Insights

- No global lock ever goes unpunished
- Data as well as execution has to be distributed over cores
  - Malloc and friends will be a bottleneck
- You will have to make memory/speed tradeoffs
- New architectures will give you both new challenges and performance boosts
  - Revise and rewrite as processors evolve
- Doing things (in the code) at the right time can reduce complexity as well as increase performance
- Take the time to use third party tools and to write your own.
- Work incrementally

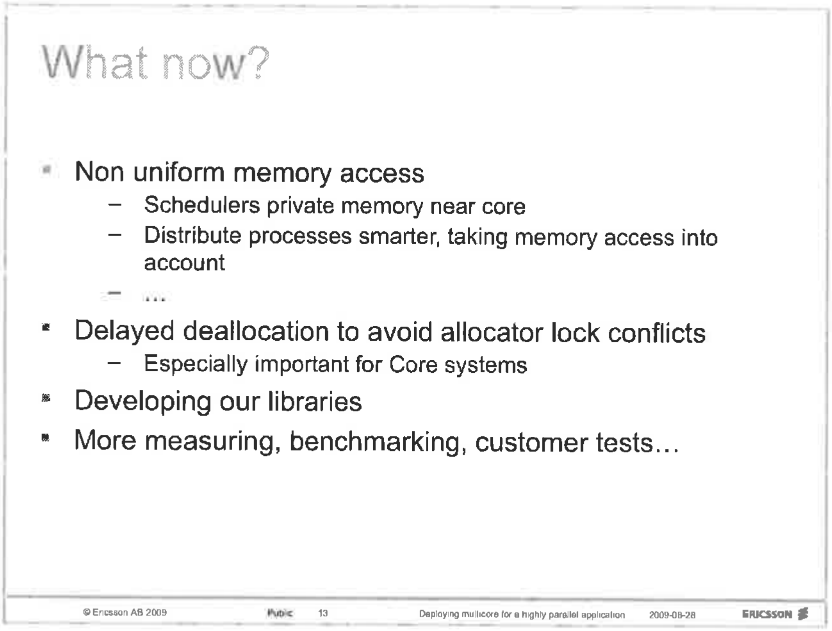
© Ericsson AB 2009 Public 11 Deploying multicore for a highly parallel application 2009-08-28 ERICSSON



Tools we've used

- Lock checker (implemented in VM) and strict locking order
- vTune and thread profiler
- oProfile
- Lock counter (implemented in VM)
- Acumem (www.acumem.com)
- Valgrind
- Benchmarks
  - Customers
  - Open Source
- Percept (Erlang application parallelism measurement tool)

© Ericsson AB 2009 Public 12 Deploying multicore for a highly parallel application 2009-08-28 ERICSSON



What now?

- Non uniform memory access
  - Schedulers private memory near core
  - Distribute processes smarter, taking memory access into account
  - …
- Delayed deallocation to avoid allocator lock conflicts
  - Especially important for Core systems
- Developing our libraries
- More measuring, benchmarking, customer tests…

ERICSSON

TAKING YOU FORWARD

Eric Merritt
Erlang author and Erlware Committer

A continuous build system for Erlang

Abstract

Eric will talk about how to use a new continuous build system for Erlang. He will also provide an intro into working with the continuous build system to automatically detect changes in source, build, test and publish OTP applications and releases. This will allow you to start getting the benefits of easy and straightforward continuous build in your Erlang projects.

Biography

Eric Merritt is a Software Engineer who specializes in concurrent languages and distributed systems. For the last eight years he has been coding Erlang and has also been heavily involved in the Erlang community. Currently, Eric is a core developer for the Erlware family of open source products and is the primary developer for the Sinan build system. Eric has been involved in both professional and Open Source development for the last ten years. He started his career developing applications in C and Java on IBM Mainframe and Midrange hardware. He also provided training and consulting in object-oriented principles and concepts. However, his interest in languages, concurrency and distributed systems quickly drove him to more interesting and challenging work at companies like Amazon.com. Currently Eric brings his expertise to PEAK6 Investments, LP, a successful proprietary trading firm in Chicago.

BERT is to Erlang as JSON is to JavaScript (plus a mini Git tutorial!)

We'll start off with a mini Git tutorial to help conceptualize the problems we had at GitHub that were addressed in the crafting of BERT and BERT-RPC. If you're unfamiliar with Git, this may open your eyes to the power, flexibility, and speed of this distributed version control system. Your language is great at dealing with distributed systems, shouldn't your SCM be just as adept? BERT (Binary ERlang Term) is a new serialization format based on Erlang's external term format. It supports rich data types such as atoms, heterogenous lists, tuples, binary data, booleans, dictionaries, and more. Just as JSON acts as an excellent inter-process data format for web-based technologies, BERT acts as an efficient inter-process data format for low-latency server technologies. Built on top of BERT is BERT-RPC, a simple, dynamic RPC protocol providing both synchronous and asynchronous requests, caching directives, streaming, and even callbacks. Tom created these technologies to help us scale GitHub. Tom needed a fast, robust way for one process to make low-latency calls to another. Tom looked at Thrift and Protocol Buffers, but those solutions were too complex and not flexible enough to hang with Ruby. Tom also wrote Ernie, an Erlang/Ruby hybrid BERT-RPC server that makes it dead simple to write your RPC functions in Ruby (or other languages). Together, all these technologies power GitHub's new federated architecture and allow us to independently and horizontally scale both frontend and backend layers.

Tom Preston-Werner

Cofounder of GitHub and Erlectricity maintainer

As cofounder of GitHub, the world's largest and most active Git hosting website, Tom Preston-Werner possesses inside knowledge of the Git ecosystem and how it is changing the way in which code collaboration is done. Tom is responsible for both low-level system architecture scaling and user interface design/usability. He's currently obsessed with finding ways to marry the productivity of Ruby with the high concurrency prowess of Erlang.

Scott Chacon

Git evangelist and Ruby developer

Scott Chacon is a Git evangelist and Ruby developer employed at Logical Awesome working on GitHub. He is the author of the Git Internals PDF book by Peepcode as well as the maintainer of http://git-scm.com and the Git Community Book. Scott has presented at RailsConf, RubyConf, Google, and a number of local groups in addition to teaching corporate training on Git across the country.

Putting UBF to work (*and Getting the outside world to talk to Erlang*)

Many protocols have formal specifications: ASN.1, ONC-RPC, CORBA, AMQP, Thrift, Protocol Buffers, zillions more.... We believe there are at least three reasons why industry uses such specifications. However, most industrial uses take advantage of the first mostly and only occasionally the second: #1 Specify bits "on the wire" in a way all parties agree. #2 API documentation: how the protocol's API works. We have found that there's a very important third reason: #3 Protocol meta-data: input for other tools (development, testing, etc.). My company is one of several that is collectively building a custom Webmail system for a large carrier in Asia. (The initial deployment will be used by well over 1 million customers.) We definitely take advantage of reasons #1 and #2 to facilitate the multi-way project planning with several development groups and the carrier/customer. Their value in communicating both with developers and with project managers is quite large.

We would like to share with other Erlang developers our experiences of using and enhancing UBF for reason #3. Much of this work is slowly making its way into the wider world, using an MIT license and distributed via GitHub (http://github.com/norton/ubf/tree/master).

Joseph Wayne Norton
Erlang Enthusiast

Joe Norton is a technical manager, system architect, developer, and Erlang enthusiast working in the mobile industry.

Scott Lystig Fritchie
Gemini Mobile Technologies

Scott Lystig Fritchie met his first UNIX system in 1986 and has almost never met one since that he didn't like. A career detour as a UNIX systems administ rator got him neck-deep in messaging systems, e-mail and Usenet News. He rediscovered full-time programming while at Sendmail, Inc., where a colleague introduced him to Erlang in 2000. His world hasn't been the same since then. Now at Gemini Mobile Technologies, Inc., he's back in the messaging world. He's finishing a distributed key-value database with strong consistency semantics, micro transactions, and on-the-fly resizing, all in Erlang. In addition to hacking Erlang code and occasionally the Erlang virtual machine, he's had papers published by USENIX, the Erlang User Conference, and the ACM.

Putting UBF to Work

(and Getting the Outside World to Talk to Erlang)

Joseph Wayne Norton / Scott Lystig Fritchie
norton@geminimobile.com / fritchie@geminimobile.com

Gemini Mobile Technologies, Inc.

November 12, 2009

A Quick Survey

Universal Binary Format

Who has heard of UBF ...

- inside the Erlang community?
- outside the Erlang community?

Who has tinkered with UBF ...

- at home?
- at work?

Who has deployed UBF ...

- as part of a commercial service?
- as part of a commercial product?

Introduction

Protocols and Specifications

Many protocols have formal specifications

- ASN.1, ONC-RPC, Corba, AMQP, Thrift, Protocol Buffers, zillion more ...

Why does industry use such specifications?

- Specify bits "on the wire" in a way all parties agree
- API documentation: how the protocol's API works

Introduction

continued...

We have found that there are very important other reasons ...

- System design and architecture: input for humans
- Protocol meta-data: input for tools

UBF has proved to be very handy in helping us with the above items ...

What is UBF?

in a nutshell

- UBF(A) is a protocol above a stream transport (e.g. TCP/IP), for encoding structured data roughly equivalent to well-formed XML.
- UBF(B) is a programming langauge for describing types in UBF(A) and protocols between clients and servers. UBF(B) is roughly equivalent to to Verified XML, XML-schemas, SOAP and WDSL.
- UBF(C) is a meta-level protocol used between UBF client and servers.

Many, many thanks to Joe Armstrong, UBF's designer and original implementor.

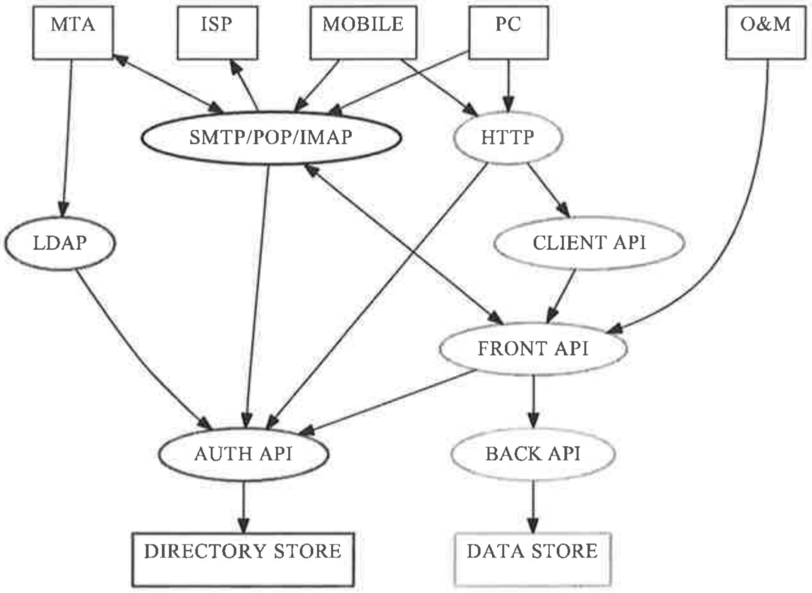
Why UBF?

in a nutshell

- RPC with a formal, precise specification
- Erlang server implementation
- Erlang and non-Erlang client implementations
- Simple yet elegant, concise yet expressive
- And most importantly ... easy to extend and to customize to our needs

UBF Case Study

A Custom Webmail System



UBF Case Study

Contract Statistics

API	Contracts	Methods	Types	Leaf Types	Records
Auth	2	26	96	53	4
Client	5	28	288	231	13
Front	11	61	469	358	32
Back	10	29	186	136	5
Total	*28*	*85*	*628*	*443*	*35*

These code snippets show how to obtain the methods, types, leaf types, and records of an UBF contract.

```
Methods = [ {Req,Res} || {Req,Res} <- Mod:contract_anystate() ].
Types = [ {T,Mod:contract_type(T)} || T <- Mod:contract_types() ].
LeafTypes = [ {T,Mod:contract_type(T)} || T <- Mod:contract_leaftypes() ].
Records = [ {R,Mod:contract_record(R)} || R <- Mod:contract_records() ].
```

UBF Contracts

Changes & New Features

Predefined primitives

- Renamed constant() to atom()
- Renamed int() to integer()
- New float(), tuple(), list(), and proplist() primitives
- Optional type()? - 'undefined' or type()
- Optional attributes (e.g. binary(ascii,nonempty))
    - ascii: only ASCII values
    - asciiprintable: only printable ASCII values
    - nonempty: not equal to '', [], <<>>, or {}
    - nonundefined: not equal to 'undefined'

UBF Contracts

continued ...

User-defined primitives

- New binary and float constants
- Support Erlang syntax for integer constants
- New integer ranges (..integer(), integer()..)
- #record{...} syntax with automatic generation of Erlang record defines
- [type()]? for optional lists
- [type()]+ for mandatory lists
- [type()]{N}, [type()]{N,}, and [type()]{M,N} for length-constrained lists

UBF Contracts

continued ...

New feature "type importing" ...

- Permit type only contracts (i.e. STATE and ANYSTATE contract blocks are now optional)
- Import UBF types from other UBF contracts
- Check and permit duplicate UBF types only having the same definition

UBF Contracts

continued ...

New feature "type importing" ...

- Import ABNF-based types from ABNF specifications
  - ABNF-based types are formal specifications for binary() types.
  - Ander Nygren's abnfc module is used to parse ABNF specifications into an internal abstract syntax tree (AST).
  - Implemented new ABNF parser for UBF's contract checker to verify binaries against ABNF-based types.
- Import EEP8-based types from any Erlang module
  - Using a parse transformation, UBF contract types are automatically added to an existing Erlang module having type defines.
  - Not all EEP8-based types are supported (pid(), fun(), ...)
  - Support for EEP8-based records is in progress

UBF Plugin Callbacks

Changes & New Features

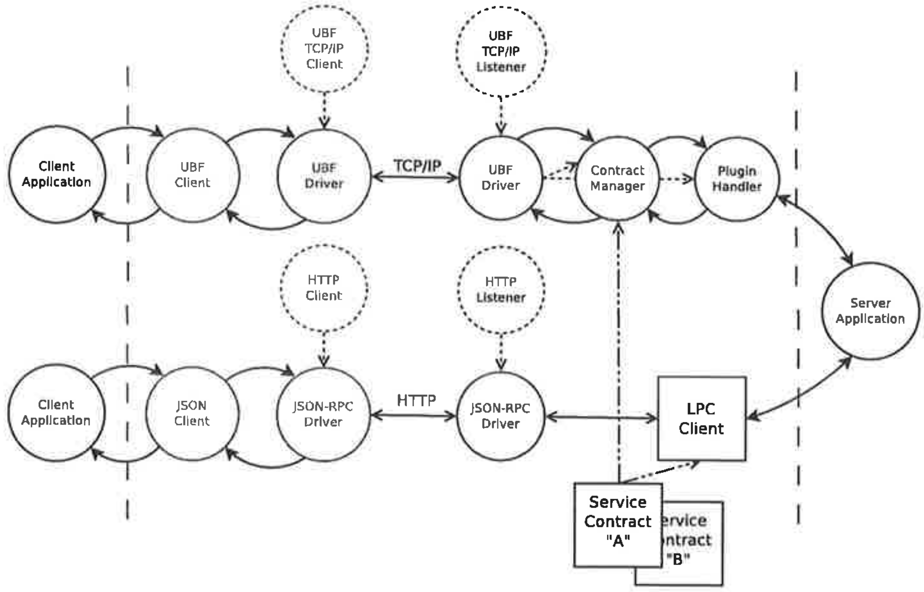
Stateful is the original callback API implementing a shared plugin manager process, a per-session contract manager process, and a per-session plugin process.

Stateless is a new callback API implementing only a per-session contract manager process and per-session plugin process. The implementation callback function API is a bit less complex.

LPC is a new callback API implementation that has no side-effects. LPC stands for Local Procedure Call.

The implementor of a UBF plugin can choose to implement one, two, or all three of the above callbacks.

UBF Transports



Key Points

- Same contract with multiple transports
- Same application with multiple contracts

UBF Transports

Changes & New Features

EBF is “Erlang Binary Format”, a simple TCP/IP protocol that uses Erlang-style conventions.

- Uses Erlang BIFs term_to_binary() and binary_to_term() to serlialize terms.
- Terms are framed using the 'gen_tcp' {packet, 4} format: a 32-bit unsigned integer specifies packet length.

ETF is “Erlang Term Format”, a simple protocol that relies on Erlang’s native distribution. This approach can be useful for Erlang-only deployments.

UBF Transports

continued ...

JSF is “JavaScript Format”, a simple TCP/IP protocol that uses JSON (RFC 4627).

- Uses LShift’s Erlang-rfc4627 to serialize terms.
- A few extra conventions are layered on top of LShift’s implementation to help distinguish between atoms, tuples, records, and ubf strings.

UBF-JSONRPC is a framework for integrating UBF and JSON-RPC over HTTP.

- Relies on JSF and provides new helper utilities to encode and to decode JSON-RPC requests and responses.
- Includes a simple inets-based HTTP client and HTTP server module that demonstrates how to use the LPC callback API.

UBF Meta-Data

Documentation

Client API - add a draft mail (UBF, EBF, and ETF style)

```
{ mail_add_draft, authinfo(), maildraft_olduid()?, mailheaders(), draftbody_parsed()
   , [rfc2396_url()], maildraft_options()?, timeout_or_expires() } =>
   { ok, uid(), [mimepart_url()] } | res_err();
```

Client API - add a draft mail (JSF and JSON-RPC style)

```
request {
        "version" : "1.1",
        "id"      : binary(),
        "method"  : "mail_add_draft",
        "params"  : [ maildraft_olduid()?, mailheaders(), draftbody_parsed()
                    , [rfc2396_url()], maildraft_options()?, timeout_or_expires() ]
 }
response {
        "version" : "1.1",
        "id"      : binary(),
        "result"  : {"$T" : [ {"$A" : "ok"}, uid(), [mimepart_url()] ]}
                    | res_err() | null,
        "error"   : error()?
 }
}
```

UBF Meta-Data

Testing

Functional Testing

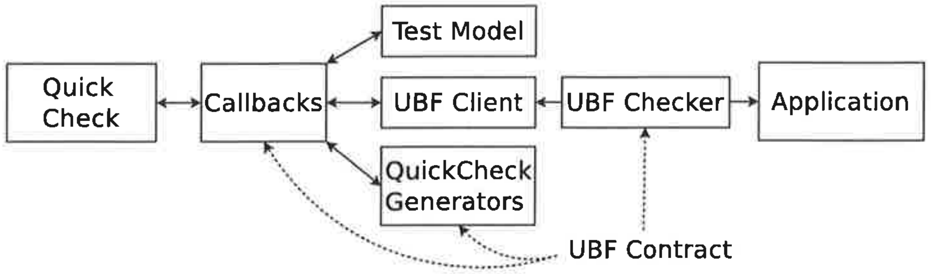
- Integration - external clients and external servers
- Dialyzer - re-use type definitions
- EUnit - automatic test input generation
- QuickCheck - automatic generation of QuickCheck generators and abstract state machine property tests

Performance Testing

- Load client generators
- Load server stubs
- Transaction logs and statistics

UBF and QuickCheck

Basic Strategy



- Boilerplate generators and properties *(with only one push of a button)*
- Custom generators
- Custom properties
- Non-UBF APIs
- Top-down and bottom-up testing
- Low-level transport and high-level application layers

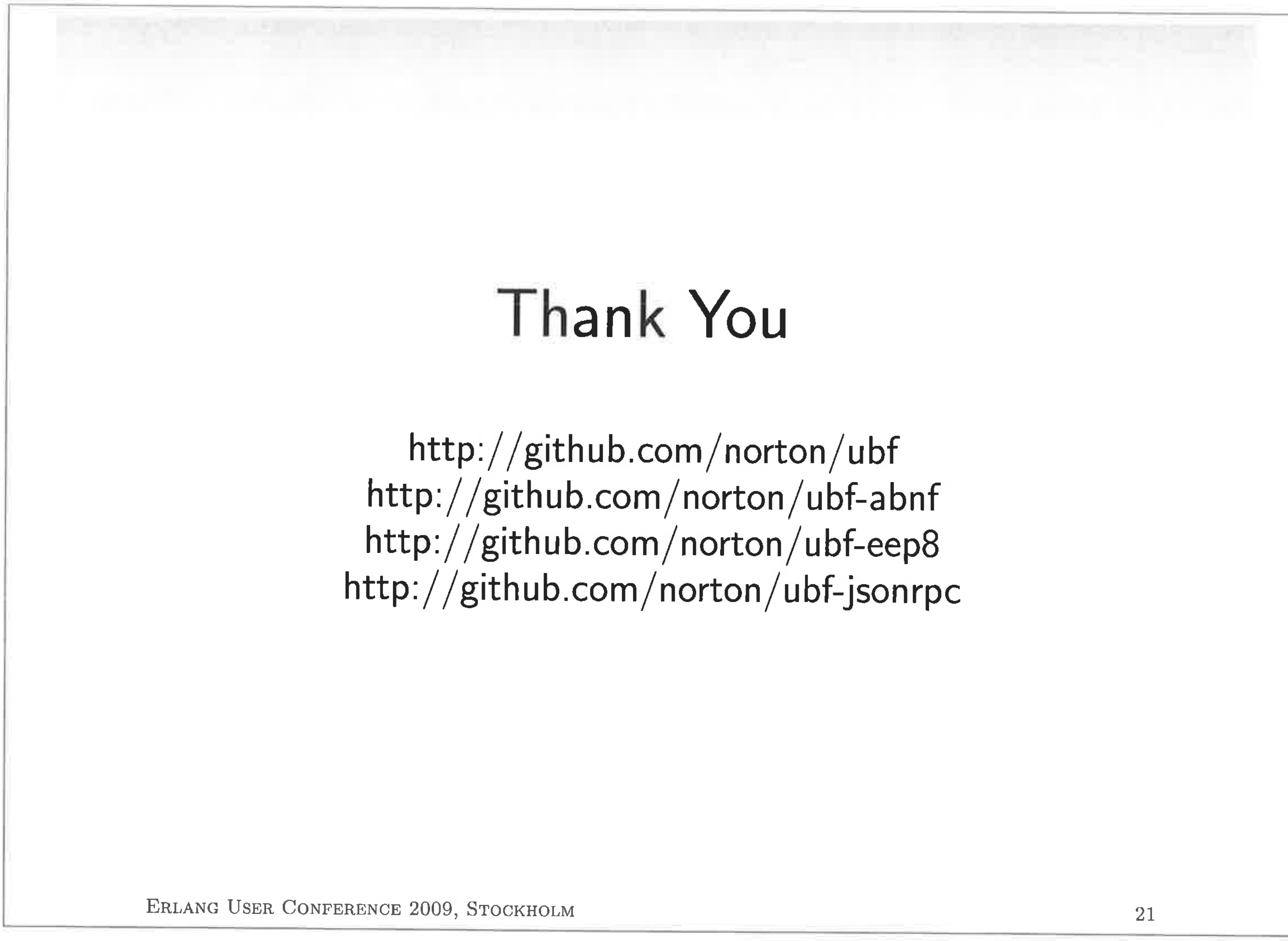
What's Next?

Documentation

- Complete edocs and examples for the UBF code repositories
- ABNF specification for UBF(A), UBF(B), and UBF(C)
- ABNF specification for EBF, JSF, and JSON-RPC

Interoperability

- EUnit: open-source the input generators for UBF
- QuickCheck: open-source the generators and the framework for UBF
- EEP8: better integration with Erlang specs, types, and records
- FFI: Erlang ports and drivers based on UBF for other languages (e.g. C/C++, Haskell)
- Other tools and approaches (e.g. Protocol Buffers, Thrift, BERT)



Thank You

http://github.com/norton/ubf
http://github.com/norton/ubf-abnf
http://github.com/norton/ubf-eep8
http://github.com/norton/ubf-jsonrpc

Erlang User Conference 2009, Stockholm 21

Kenneth Lundin
Manager of the Erlang/OTP development team

Latest news from the Erlang/OTP team at Ericsson

Abstract

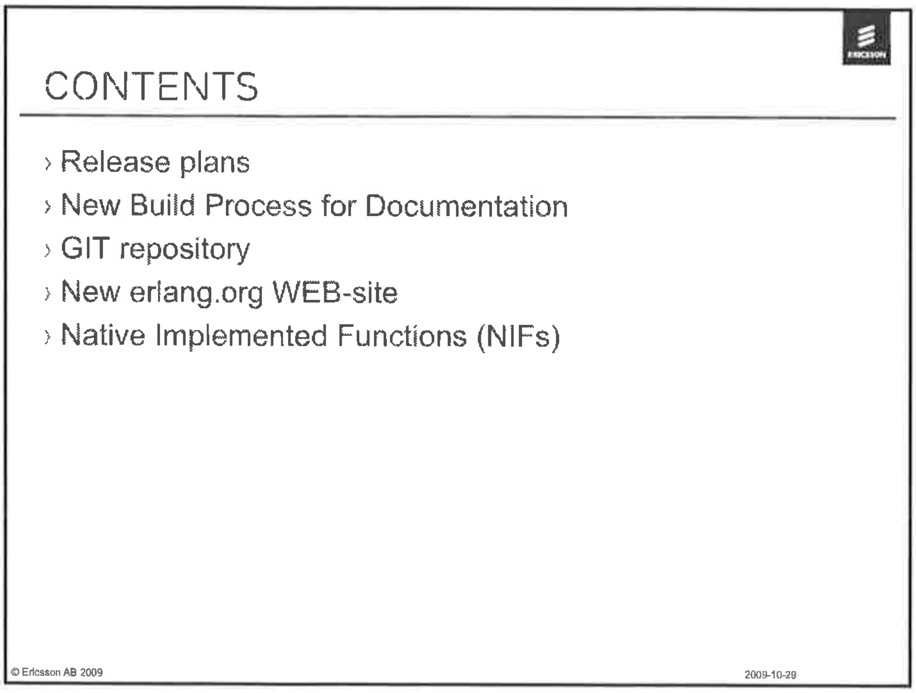
Kenneth gives an update of the Erlang/OTP team's work at Ericsson - their current projects and plans for future.

Biography

Kenneth Lundin has been working with SW development since the late 70s. As a curiousity it can be mentioned that Kenneth was one of the pioneers in the use of C++ at Ericsson. Unsurprisingly Kenneth's interest for OO languages has been slighty revised since then. He joined the Erlang/OTP project in it's early stages 1996 and has been working both with application components and the runtime system since then. Has been managing the team for about 10 years now.

ERLANG/OTP LATEST NEWS

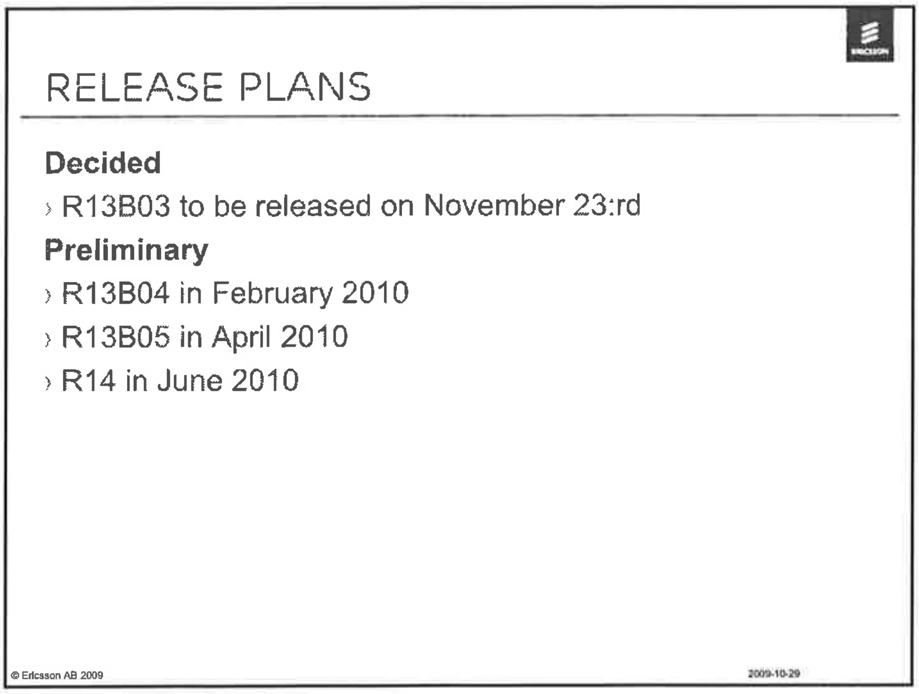
ERLANG USER CONFERENCE 2009
Kenneth Lundin



CONTENTS

- Release plans
- New Build Process for Documentation
- GIT repository
- New erlang.org WEB-site
- Native Implemented Functions (NIFs)

2009-10-29



RELEASE PLANS

Decided

› R13B03 to be released on November 23:rd

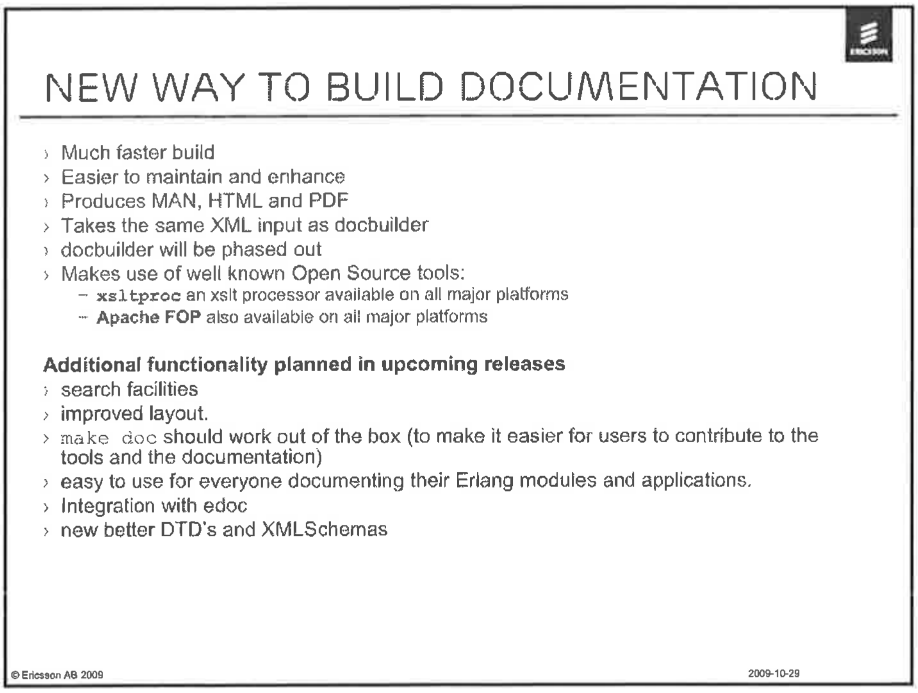
Preliminary

› R13B04 in February 2010

› R13B05 in April 2010

› R14 in June 2010

2009-10-29



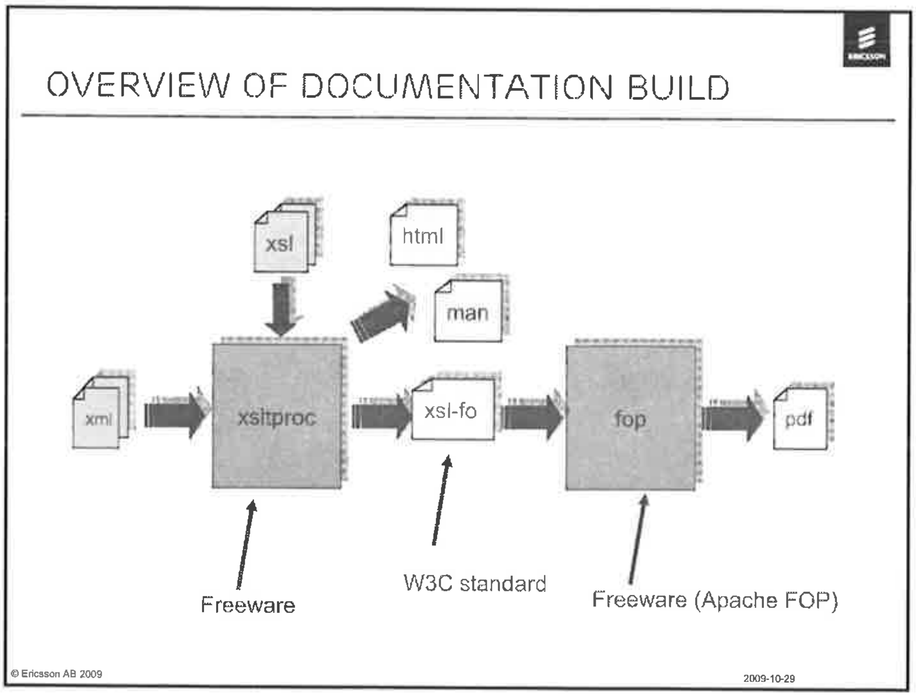
NEW WAY TO BUILD DOCUMENTATION

- Much faster build
- Easier to maintain and enhance
- Produces MAN, HTML and PDF
- Takes the same XML input as docbuilder
- docbuilder will be phased out
- Makes use of well known Open Source tools:
  - `xsltproc` an xslt processor available on all major platforms
  - **Apache FOP** also available on all major platforms

Additional functionality planned in upcoming releases

- search facilities
- improved layout.
- `make doc` should work out of the box (to make it easier for users to contribute to the tools and the documentation)
- easy to use for everyone documenting their Erlang modules and applications.
- Integration with edoc
- new better DTD's and XMLSchemas

2009-10-29



OVERVIEW OF DOCUMENTATION BUILD

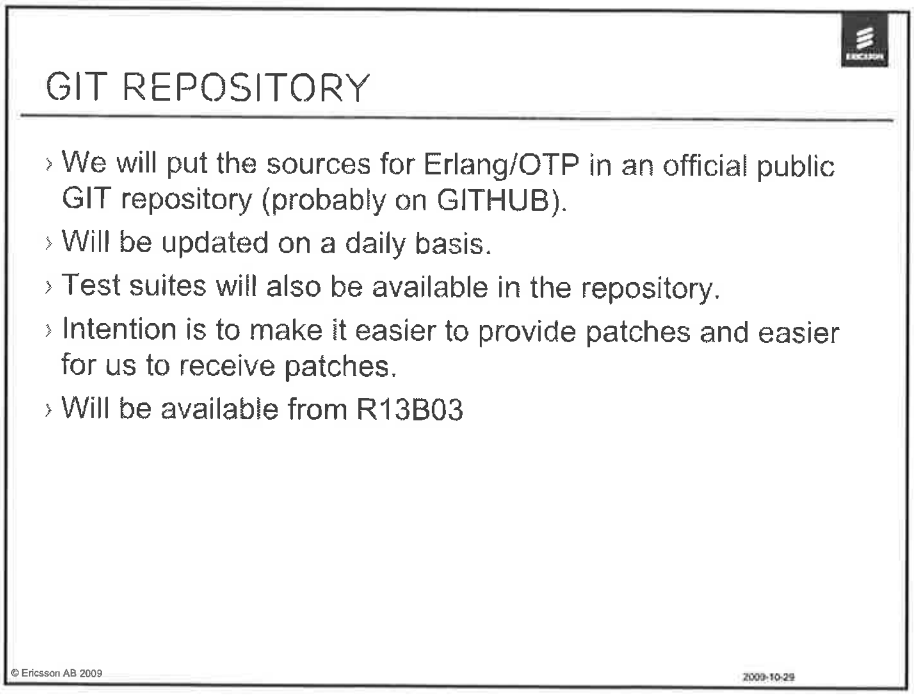
Freeware

W3C standard

Freeware (Apache FOP)

© Ericsson AB 2009

2009-10-29



GIT REPOSITORY

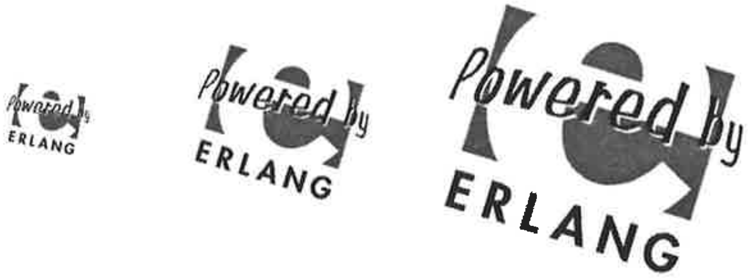
- We will put the sources for Erlang/OTP in an official public GIT repository (probably on GITHUB).
- Will be updated on a daily basis.
- Test suites will also be available in the repository.
- Intention is to make it easier to provide patches and easier for us to receive patches.
- Will be available from R13B03

© Ericsson AB 2009

2009-10-29

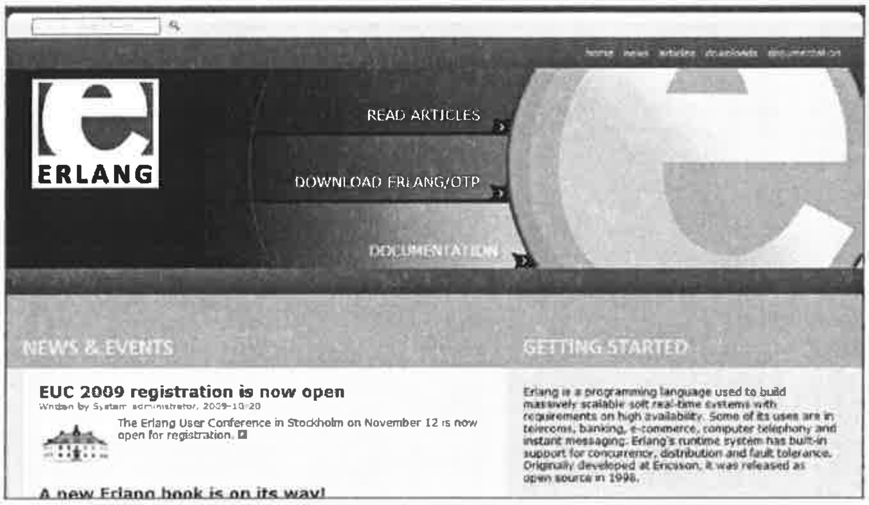
NEW ERLANG.ORG WEB-SITE

- erlang.org with new layout and technology
- Easier to update news and articles
- The goal is to make the site more alive and up to date.



© Ericsson AB 2009 2009-10-29

NEW ERLANG.ORG WEB-SITE (SNAPSHOT)



NEWS & EVENTS

EUC 2009 registration is now open

Written by System administrator, 2009-10-20

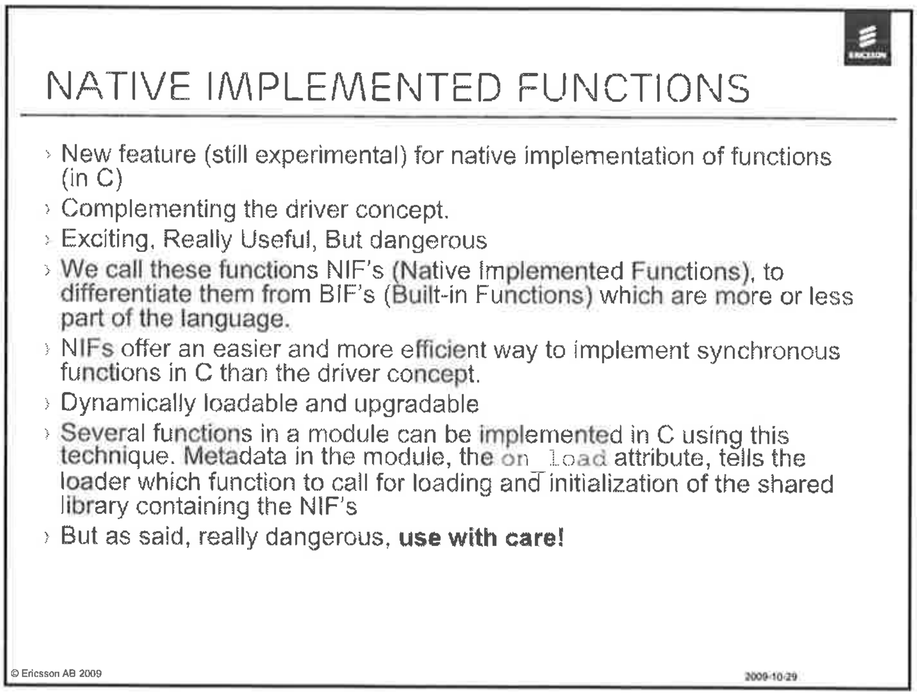
The Erlang User Conference in Stockholm on November 12 is now open for registration.

A new Erlang book is on its way!

GETTING STARTED

Erlang is a programming language used to build massively scalable soft real-time systems with requirements on high availability. Some of its uses are in telecoms, banking, e-commerce, computer telephony and instant messaging. Erlang's runtime system has built-in support for concurrency, distribution and fault tolerance. Originally developed at Ericsson, it was released as open source in 1998.

© Ericsson AB 2009 2009-10-29

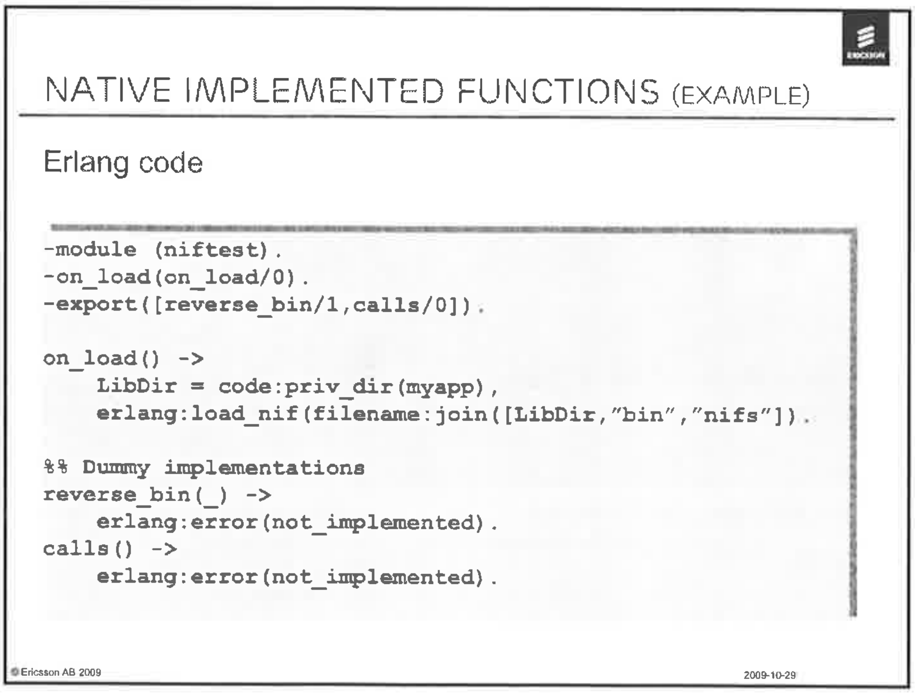


NATIVE IMPLEMENTED FUNCTIONS

- New feature (still experimental) for native implementation of functions (in C)
- Complementing the driver concept.
- Exciting, Really Useful, But dangerous
- We call these functions NIF's (Native Implemented Functions), to differentiate them from BIF's (Built-in Functions) which are more or less part of the language.
- NIFs offer an easier and more efficient way to implement synchronous functions in C than the driver concept.
- Dynamically loadable and upgradable
- Several functions in a module can be implemented in C using this technique. Metadata in the module, the `on_load` attribute, tells the loader which function to call for loading and initialization of the shared library containing the NIF's
- But as said, really dangerous, **use with care!**

© Ericsson AB 2009

2009-10-29



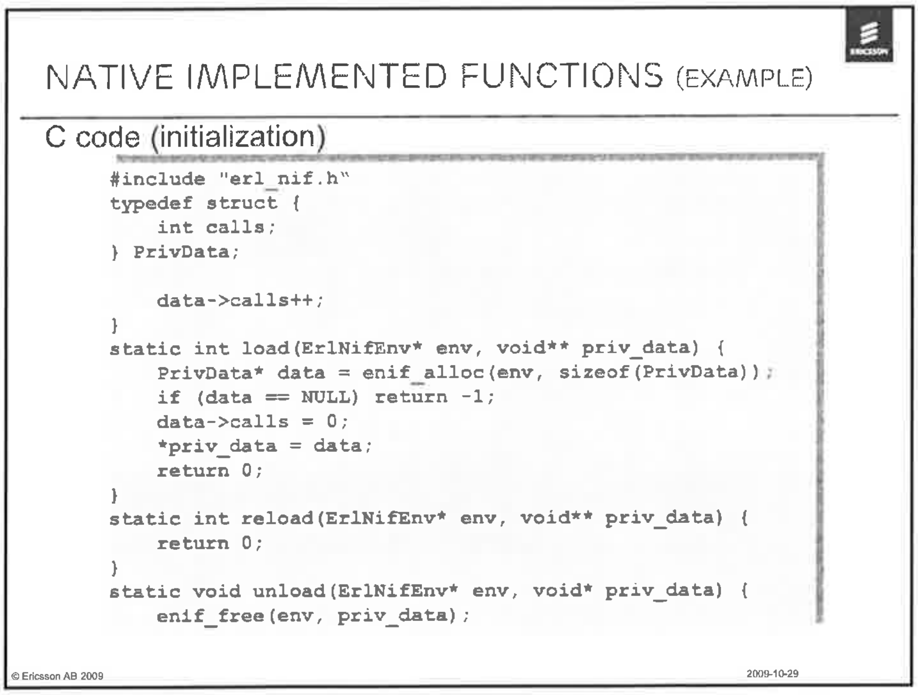
NATIVE IMPLEMENTED FUNCTIONS (EXAMPLE)

Erlang code

```
-module (niftest).
-on_load(on_load/0).
-export([reverse_bin/1,calls/0]).

on_load() ->
    LibDir = code:priv_dir(myapp),
    erlang:load_nif(filename:join([LibDir,”bin”,”nifs”]).

%% Dummy implementations
reverse_bin(_) ->
    erlang:error(not_implemented).
calls() ->
    erlang:error(not_implemented).
```

© Ericsson AB 2009

2009-10-29

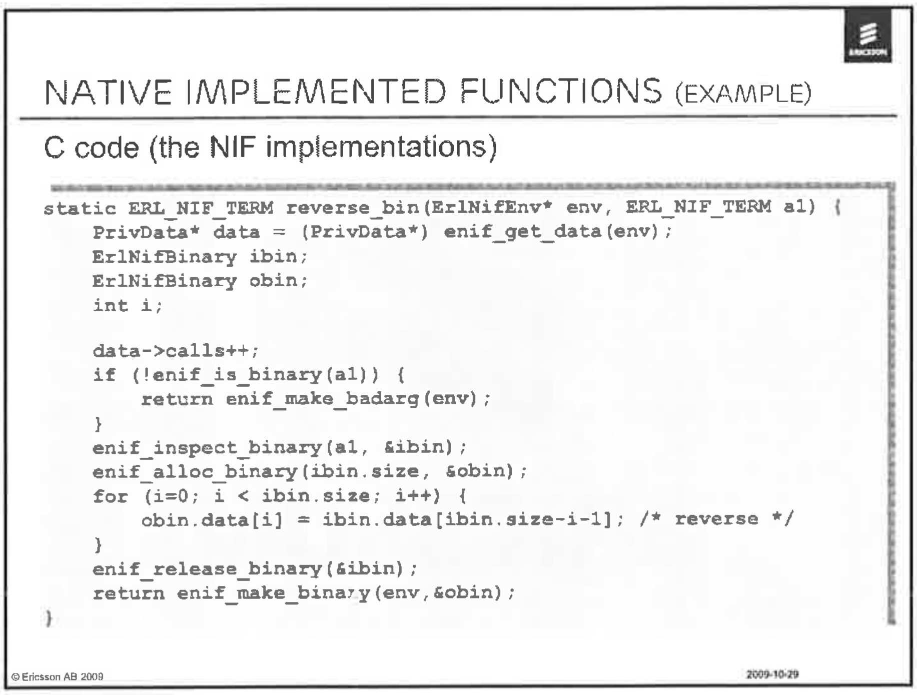


NATIVE IMPLEMENTED FUNCTIONS (EXAMPLE)

C code (initialization)

```
#include "erl_nif.h"
typedef struct {
    int calls;
} PrivData;

    data->calls++;
}
static int load(ErlNifEnv* env, void** priv_data) {
    PrivData* data = enif_alloc(env, sizeof(PrivData));
    if (data == NULL) return -1;
    data->calls = 0;
    *priv_data = data;
    return 0;
}
static int reload(ErlNifEnv* env, void** priv_data) {
    return 0;
}
static void unload(ErlNifEnv* env, void* priv_data) {
    enif_free(env, priv_data);
```

© Ericsson AB 2009 2009-10-29

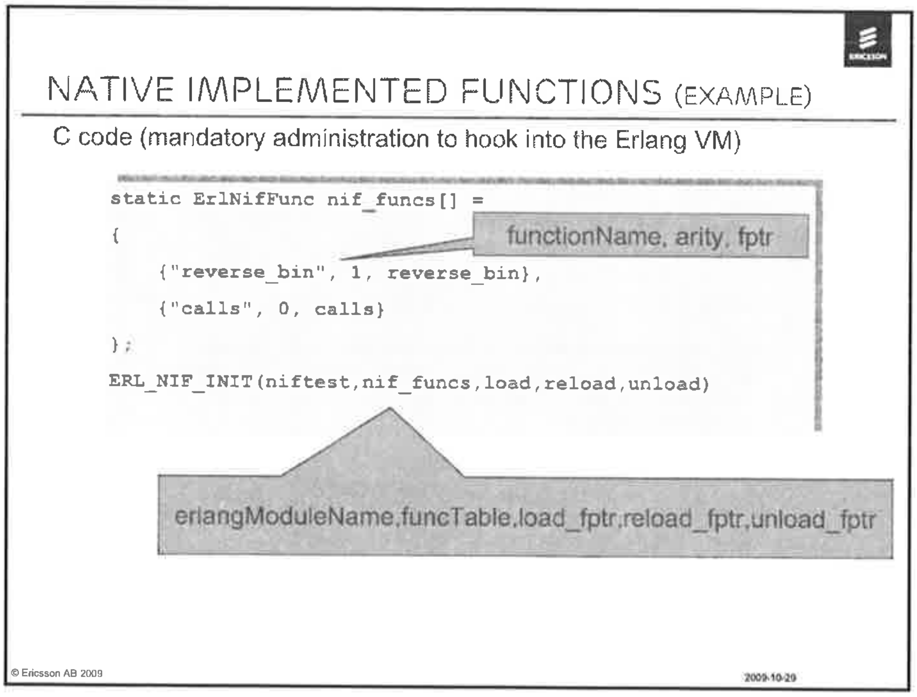


NATIVE IMPLEMENTED FUNCTIONS (EXAMPLE)

C code (the NIF implementations)

```
static ERL_NIF_TERM reverse_bin(ErlNifEnv* env, ERL_NIF_TERM a1) {
    PrivData* data = (PrivData*) enif_get_data(env);
    ErlNifBinary ibin;
    ErlNifBinary obin;
    int i;

    data->calls++;
    if (!enif_is_binary(a1)) {
        return enif_make_badarg(env);
    }
    enif_inspect_binary(a1, &ibin);
    enif_alloc_binary(ibin.size, &obin);
    for (i=0; i < ibin.size; i++) {
        obin.data[i] = ibin.data[ibin.size-i-1]; /* reverse */
    }
    enif_release_binary(&ibin);
    return enif_make_binary(env,&obin);
}
```

© Ericsson AB 2009 2009-10-29



NATIVE IMPLEMENTED FUNCTIONS (EXAMPLE)

C code (mandatory administration to hook into the Erlang VM)

```
static ErlNifFunc nif_funcs[] =
{
    {"reverse_bin", 1, reverse_bin},
    {"calls", 0, calls}
};
ERL_NIF_INIT(niftest,nif_funcs,load,reload,unload)
```

functionName, arity, fptr

erlangModuleName,funcTable,load_fptr,reload_fptr,unload_fptr

© Ericsson AB 2009

2009-10-29