Metaprogramming for the Masses

Richard Carlsson

Klarna

Metaprogramming

Writing programs that create or manipulate data structures that represent programs

Homoiconic languages

"...in that their internal and external representations are essentially the same" - Alan Kay

```
(ADD 2 3)            ; LISP code
'(ADD 2 3)           ; LISP data
(EVAL '(ADD 2 3))    ; interpreting data as code
```

Erlang is not one of them

```
case X + 1 of ...      % Erlang code

{foo, 42, [...]}       % Erlang data
```

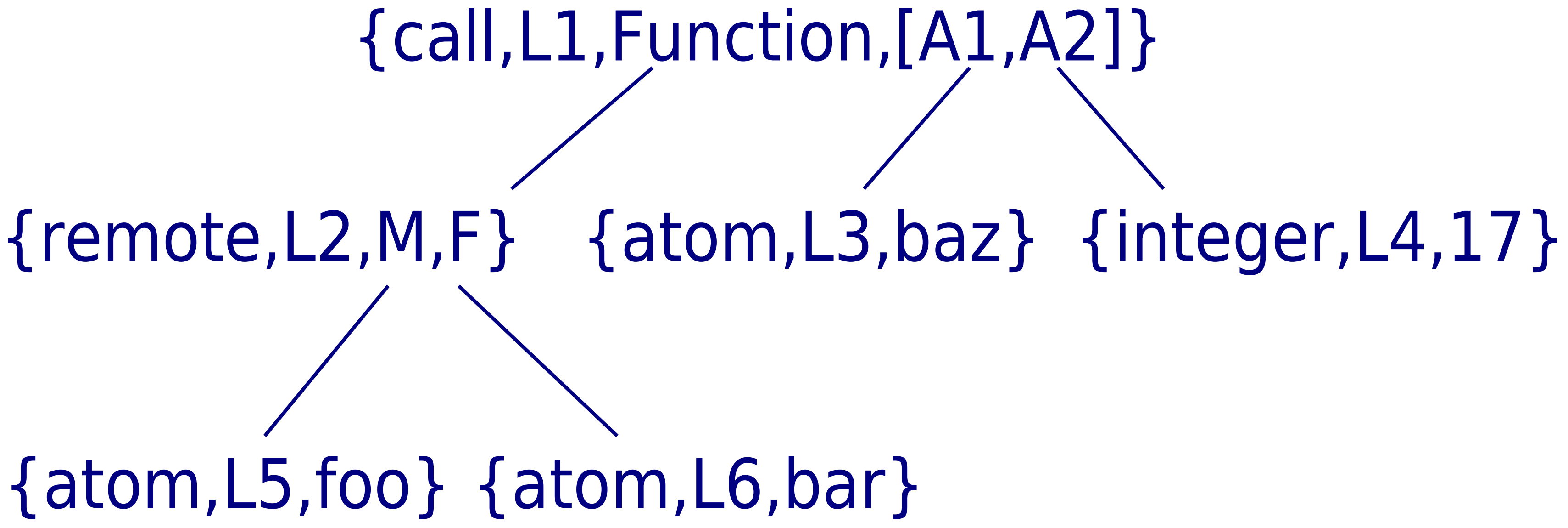
Scanning and parsing

```
Text = "foo:bar(baz,17)."
{ok, Toks, _Line} = erl_scan:string(Text, L0).
{ok, Exprs} = erl_parse:parse_exprs(Toks).
Exprs = [{call,1,{remote,1,{atom,1,foo},
{atom,1,bar}},[{atom,1,baz},{integer,1,17}]}]
```

erl_parse:parse_form/1

erl_parse:parse_term/1

The abstract format

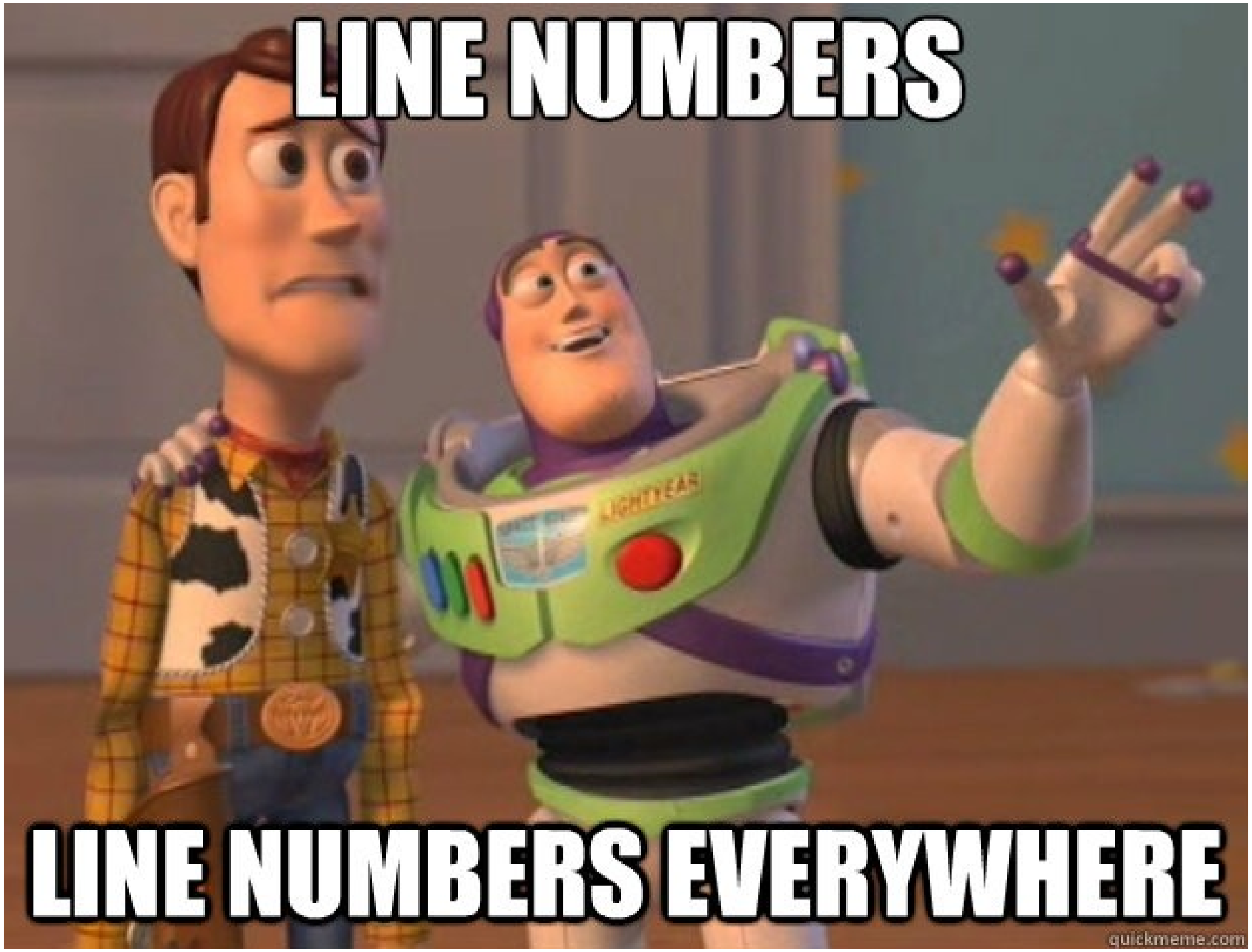


{call,L1,Function,[A1,A2]}

{remote,L2,M,F} {atom,L3,baz} {integer,L4,17}

{atom,L5,foo} {atom,L6,bar}

http://www.erlang.org/doc/apps/erts/absform.html


LINE NUMBERS
LINE NUMBERS EVERYWHERE
quickmeme.com

Not nearly abstract enough

- Explicit tuple representation
- Unnecessary details (line numbers)
- Ad hoc, context dependent
  - Should it be 'foo' or {atom,L,'foo'}?
  - {record_field,...} used for multiple things
- New format changes break existing code
- No room for additional info/annotations

Syntax Tools

- **erl_syntax** module provides proper abstract datatype for Erlang syntax trees
- Hides details, adds annotations, comments
- Not context dependent
- Can take “abstract format” trees as input
- Generic functions for traversal etc.
- Must revert to standard abstract format before passing the AST to the compiler

Still rather too verbose

```
1> Tree =
erl_syntax:application(erl_syntax:module_qualifier
(erl_syntax:atom(foo), erl_syntax:atom(bar)),
[erl_syntax:atom(baz),erl_syntax:integer(17)]).

2> erl_prettypr:format(Tree).
"foo:bar(baz, 17)"
```

Step-by-step decomposition

```
case erl_syntax:type(Tree) of
    application ->
        Op = erl_syntax:application_operator(Tree),
        case erl_syntax:type(Op) of
            module_qualifier ->
                M = erl_syntax:module_qualifier_argument(Op),
                F = erl_syntax:module_qualifier_body(Op),
                ...
```

Plain tuples allow matching

```
case Tree of
    {call,_,{remote,_,{atom,_,foo},{atom,_,bar}}, [A1, A2]} ->
        %% found a call to foo:bar/2!
        ... ;
    _ ->
        %% something else
```

What if erl_syntax had patterns?

Me, around 1997

15 years later

A simple DSL for business logic

- Once, as a very young company, Klarna had all the business logic in Erlang code
  - Management/Finance could not read it
  - Developers required both to change logic **and** to explain the current logic
  - Code upgrade necessary for all changes
  - No trace of **how** decisions were made

Tobbe's first draft

- Simple decision engine by Tobbe, using Erlang tuples & lists to express rules:
  - **{first, […]}** % orelse operator
  - **{all, […]}** % andalso operator
  - **{equal, X, Y}**, **{plus, X, Y}**, …
  - “input variables” (dict as input to engine)
- Still in Erlang (though in a single place)
  - Still mostly unreadable to non-developers

From the mouths of babes

Bumped into CEO in the corridor

“Can't you visualize the rules for us like in the Wiki, with labels and bullet points?”

Why not use Wiki syntax as DSL

The things we wanted to express seemed to match the basic MediaWiki notation well

```
== RuleName ==
Blah blah comments blah.
* person.age > 18
* person.country = "SE"
```

- JavaScript semantics for values, names, and operators
- Input environment defined as a JSON structure

Easily nested conditionals

```
== Allowed to Purchase ==
* first of
** person.is_vip
** person.income >= limits.min_income
** all of
*** person.country = "FI"
*** [[#Finnish Special Cases]]
```

Rules can be pasted into MediaWiki, no translation needed

Disclaimer: above example completely hypothetical

Calls become clickable links

[[#Name Of Rule]]

- No worse than any other syntax for calls
- Rules can take parameters

```
== Some Rule ==
* input(x)
* x > 42
```

- Passing parameters: **[[#Some Rule]](99)**

The good

- Non-developers can read and mostly understand the rules
  - Could start writing new rules pretty quickly
- Rules updated separately from code
- Rules engine can save evaluation traces for later analysis or debugging
- All rules in one place, not mixed up with system implementation details

The bad

- As in Prolog, negative rules become tricky
- “Make a yes or no decision” soon changed to “...and also compute an output value”
  - “...Actually, we want you to compute two output values ...or in fact, dozens of them”
- Language extended to manipulate state
- People didn't quite “get” backtracking that rolls back the state to the choice point

Where do we go now?

- We now have thousands of lines of rules
- It has served us well for a few years
- Would like to take lessons learned and rework the entire language
    - Will probably not have time for that
- Switch to a “real” business rules engine
    - Eresye? Or some “enterprise” system?

Implementation

- First version: interpreter (in Erlang)
  - Pretty easy to write
  - Fairly easy to tweak and debug
  - Non-Erlangish semantics of the actual DSL is not a big problem when interpreting
- Hard to share a large data structure (the rules) between processes in Erlang
- Single evaluation server holding the current rule set

Compiling for parallellism

- As our system load got heavier, we saw more need for running rules in parallel
- Beam modules are shared (read-only) between Erlang processes – no execution bottleneck
- Compile one “rule namespace” to a single Erlang module
  - Planned for compilation from the start
  - If you create a DSL, start by interpreting, but think about how you intend compilation to work

Code generation

- Generate Erlang code (not Core Erlang) to ensure complete safety and sanity checks
  - Compile and load directly to memory
- Different semantics of DSL (working on JSON structures) causes verbose code
  - From an input file of 5 K lines of rules
  - To 50 K lines of (prettyprinted) Erlang
  - Compiles to 600+ KB beam image in 10 s
  - The DSL is very compact

Writing the code generator got me thinking...

...maybe I should try out that old idea...

Merl

or

Why the hell didn't I do this years ago?

Smart parsing

```
1> merl:quote("X+1").
{op,1,'+',{var,1,'X'},{integer,1,1}}
2> merl:quote("X + 1, Y - 1").
[{op,1,'+',{var,1,'X'},{integer,1,1}},
 {op,1,'-',{var,1,'Y'},{integer,1,1}}]
3> merl:quote("foo -> bar").
{clause,1,[{atom,1,foo}],[],[{atom,1,bar}]}
4> merl:quote("f(X) -> X+1.").
{function,1,f,1,[{clause,1,[{var,1,'X'}],...}
```

Multiline quotes

```
merl:quote(["-module(foo).",
            "-export([f/1]).",
            "f(X) -> {ok, X}."])

[{attribute,1,module,foo},
 {attribute,2,export,[{f,1}]},
 {function,3,f,1,
       [{clause,3,[{var,3,'X'}],[],
          [{tuple,3,[{atom,3,ok},{var,3,'X'}]}]}]}]
```

Metavariable substitution

```
B = merl:term([1,2,3]),
T = merl:qquote("{foo, _@bar}",
                        [{'bar', B}])

erl_prettypr:format(T).
"{foo, [1, 2, 3]}"
```

- “Quasi-quote”: a phrase containing meta-variables

Metavariables for all occasions

- Variables: **_@foo**

```
merl:qquote("{ok, _@foo}", ...)
```

- Atoms: **'@bar'**

```
merl:qquote(" '@bar'(X) -> X + 1. ", ...)
```

- Integers: **909NN**

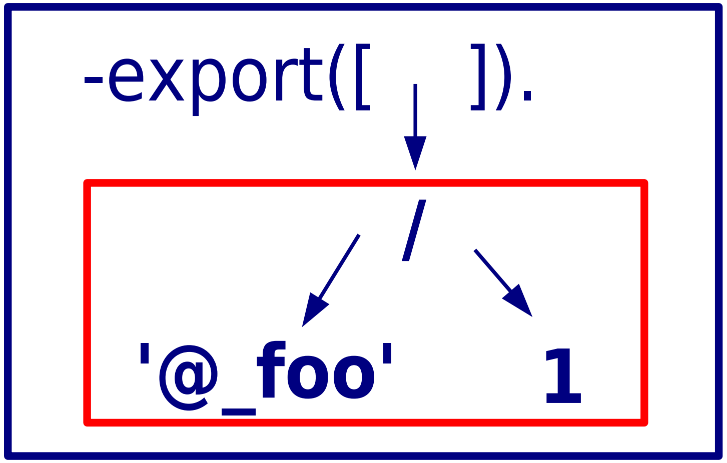
```
merl:qquote(" -export([foo/9091]). ", ...)
```

- Strings: **" '@xyz "**

```
merl:qquote(" -file(\"'@path\", 1). ", ...)
```

Lifted metavariables

- _@_foo, '@_foo'



- T = merl:qquote("-export([**'@_foo'/1**])",

  [{foo, **merl:term(42)**}]).

  erl_prettypr:format(T).

  "**-export([42])**"

Macros FTW

```
-include("merl.hrl").

T1 = ?Q("{baz, 42}"),
T2 = ?Q("{foo, 17, _@bar},", [{bar, T1}])
```

- Short and sweet
- ?Q with either 1 argument or 2
- Passes on line number from source file to provide useful parse errors

Matching

- Pat = ?Q("**{_@x**, **_@y}**")
- {ok, **[{x,First}, {y,Second}]**} =

  merl:match(Pat, ?Q("**{1,2}**"))
- **error** = merl:match(Pat, ?Q("**{1,2,3}**"))
- Anonymous metavariables: **@_**

  {ok, [{y,Second}]} =

  merl:match(?Q("{**_@_**, _@y}"),?Q("{1,2}"))

Synchronicity

- Showed early version to Simon Thompson in London 2011
  - “Oh, that looks a lot like what we just did for writing refactorings in Wrangler!”
- Upped the ante
- Conference-driven development!
  - Agreeing to talk about it in SF provided motivation to work on improvements

Glob metavariables in matches

- @@foo

```
Pat = ?Q("f(_@@args)"),
{ok, [{args, As}]} =
  merl:match(Pat, ?Q("f(1,2,3)"))
```

- Combines with lifting: @_@foo

```
Pat = ?Q("-export(['@_@x'/1])."),
{ok, [{x,[F,G]}]} =
  merl:match(Pat,?Q("-export([f/1,g/2])."))
```

Globs with static prefix/suffix

- Pat = ?Q("f(**_@a**, **_@b**, **_@@rest**, **_@c**)"),

  merl:match(Pat, merl:quote("f(1,2,3,4,5)")).

  {ok, [{**a**, {integer,_,1}}, {**b**, {integer,_,2}},

  {**c**, {integer,_,5}},

  {**rest**, [{integer,_,3},{integer,_,4}]}]}

- Result from successful match is always ordered on the metavariable names

Template data structures

- The result from quote/1 or qquote/1 is an abstract syntax tree (erl_syntax)
- To do variable substitution or matching, trees are converted to a more efficient form called *templates*
- qquote/2 calls the subst/2 function, which accepts both trees and templates as input
- If you are going to do multiple matches or substitutions, call template/1 once for all

Parse transform magic

- Including merl.hrl enables the transform

  Define MERL_NO_TRANSFORM to disable

- Evaluates constant merl calls and parses quoted strings to templates at compile time

  T = merl:term(**[1,2,3]**)

  ?Q(**"f() -> _@x."**, [{x, **X**}])

- Avoids runtime overhead of parsing and tree-to-template conversion

- Uses itself to compile itself

Inline metavariables

- Metavariables looking like normal Erlang variables are lifted to the Erlang level by the parse transform

```
Foo = ?Q("{foo, [1,2,3]}"),
Bar = ?Q("{bar, _@Foo}")
```

- No need for a list of tagged tuples
  - Faster substitution
- But the code needs the transform to work

Auto-abstracting inline variables

- Very common pattern:

```
TmpFoo = merl:term(Foo),
Bar = ?Q("{bar, _@TmpFoo}")
```

- Naming convention for automatically abstracting a constant term to a syntax tree

```
Bar = ?Q("{bar, _@Foo@}")
```

- No need for intermediate variable names
- Eliminated most calls to merl:term/1

Case switches

```
merl:switch(Tree,
            [{?Q("{bar, _@x}"),
                fun ([{x, X}]) -> X end},
             {?Q("{foo, _@x}"),
                fun ([{x, X}]) -> X end},
             fun () -> ?Q("undefined") end
            ]))
```

- Clause={Pattern,Body}|{Pattern,Guard,Body}
- Future: make parse transform expand inline

Module building API

- init_module/1
- add_function/4
- add_record/3
- add_import/3
- add_attribute/3
- set_file/2
- module_forms/1

Future directions

- Will be on GitHub soon

  https://github.com/richcarl
- Submit for inclusion in OTP
  - Part of Syntax Tools or separate app?
- Decomposition still a little messy
  - Inline metavariables in matches/switches?

Example time