How to make an optimizing compiler in a few months
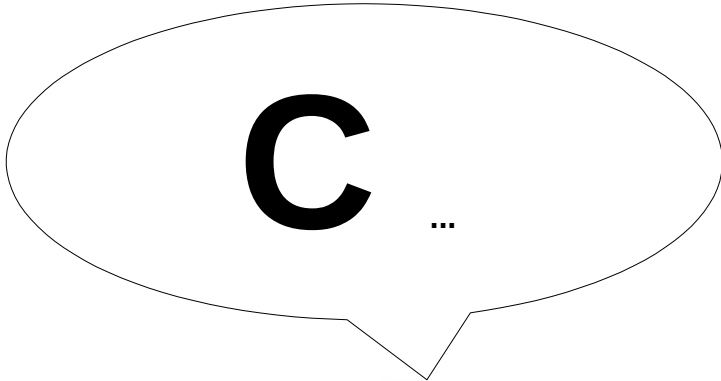
joe.armstrong@ericsson.com

You mean

```
define i32 @main() nounwind {
 ;;return register

 %tmp_1  = alloca i32 ,align 4
 %i  = alloca i32 ,align 4
 %max  = alloca i32 ,align 4
 %n  = alloca i32 ,align 4
 %tmp_2  = add i32 0,0
 store i32 %tmp_2 ,i32* %i
 %tmp_3  = add i32 0,100000000
 store i32 %tmp_3 ,i32* %max
 %tmp_4  = add i32 0,0
 store i32 %tmp_4 ,i32* %n
 br label %initfor_1

initfor_1:
 %tmp_5  = add i32 0,0
 store i32 %tmp_5 ,i32* %i
 br     %testfor_3

updatefor_2:
 %tmp_6  = load i32* %i
 %tmp_7  = add i32 0,1
 %tmp_8  = add i32 %tmp_6 ,%tmp_7
 store i32 %tmp_8 ,i32* %i
 br label %testfor_3

testfor_3:
 %tmp_9  = load i32* %i
 %tmp_10  = load i32* %max
 %tmp_11  = icmp slt i32 %tmp_9 ,%tmp_10
 br i1  %tmp_11 ,label %bodyfor_4,label %endfor_5

bodyfor_4:
 %tmp_12  = load i32* %n
 %tmp_13  = load i32* %i
 %tmp_14  = add i32 %tmp_12 ,%tmp_13
 store i32 %tmp_14 ,i32* %n
 br label %updatefor_2

endfor_5:
 %tmp_15  = getelementptr [6 x i8]* @main.str1, i32 0, i32 0
 %tmp_16  = load i32* %n
 %tmp_17  = call i32  (i8* , ...)* @printf(i8*  %tmp_15 , i32  %tmp_16 )
 %tmp_18  = add i32 0,0
 ret i32  %tmp_18
}
```

```
int main()
{
 int i=0, max=100000000,n=0;
 for(i = 0; i < max; i = i + 1){
  n  = n + i;
 }
 printf("n=%i\n", n);
 return(0);
}
```

ZAPBAR

- Front-end in Erlang

  Erlang → LLVM assembler

- Middle-end

  LLVM opt

- Back-end

  Gecode (Constraint logic programming)

Sum 1..1000000

```
> timer:tc(lists, sum, [lists:seq(1,1000000)]).
{74037,500000500000}
```

74 ms

C

```c
int printf(const char * format, ...);

int main()
{
  int i, max=1000000, sum=0;
  for(i = 0; i <= max; i = i + 1){
    sum = sum + i;
    }
  printf("sum 1..%i = %i\n", max, sum);
  return(0);
}
```

This shows
shows a run
of the compiler
(which is written in Erlang)

```
./wow.sh test2.c
Scanning:test2.c
Eshell V5.8  (abort with ^G)
1> Pass 1:ecc_normalize_function_heads
Pass 1:ecc_normalize_function_heads took:5524
us
Pass 2:ecc_make_prototypes
Type signatures are:
declare i32  @printf(i8* , ...)
declare i32  @main()
Pass 2:ecc_make_prototypes took:1153 us
Pass 3:ecc_lift_vars
Pass 3:ecc_lift_vars took:687 us
Pass 4:ecc_fix_scopes
Pass 4:ecc_fix_scopes took:631 us
Pass 5:ecc_fold_types
Pass 5:ecc_fold_types took:465 us
Pass 6:ecc_gen_llvm
Pass 6:ecc_gen_llvm took:773 us
Pass 7:ecc_asm_llvm
Pass 7:ecc_asm_llvm took:204 us
Created : (ok) test2.ll
```

test2.ll

```
; Compiled by the amazing Ericsson C->LLVM compiler
; Hand crafted in Erlang
; ModuleID = 'test2.c'

target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:...
target triple = "i386-pc-linux-gnu"

; Sock it to me baby

;; globals
declare i32  @printf(i8* , ...)

@main.str1 = private constant [16x i8] c"sum 1..%i =
%i\0A\00"

;; code
define i32 @main() nounwind {
 ;;return register

 %tmp_1  = alloca i32 ,align 4
 %i  = alloca i32 ,align 4
 %max  = alloca i32 ,align 4
 %sum  = alloca i32 ,align 4
 %tmp_2  = add i32 0,1000000
 store i32 %tmp_2 ,i32* %max
 %tmp_3  = add i32 0,0
 store i32 %tmp_3 ,i32* %sum
 br label %initfor_1

initfor_1:
 %tmp_4  = add i32 0,0
 store i32 %tmp_4 ,i32* %i
 br label %testfor_3
```

```
updatefor_2:
 %tmp_5  = load i32* %i
 %tmp_6  = add i32 0,1
 %tmp_7  = add i32 %tmp_5 ,%tmp_6
 store i32 %tmp_7 ,i32* %i
 br label %testfor_3

testfor_3:
 %tmp_8  = load i32* %i
 %tmp_9  = load i32* %max
 %tmp_10  = icmp sle i32 %tmp_8 ,%tmp_9
 br i1  %tmp_10 ,label %bodyfor_4,label %endfor_5

bodyfor_4:
 %tmp_11  = load i32* %sum
 %tmp_12  = load i32* %i
 %tmp_13  = add i32 %tmp_11 ,%tmp_12
 store i32 %tmp_13 ,i32* %sum
 br label %updatefor_2

endfor_5:
 %tmp_14  = getelementptr [16 x i8]* @main.str1, i32 0, i32 0
 %tmp_15  = load i32* %max
 %tmp_16  = load i32* %sum
 %tmp_17  = call i32  (i8* , ...)* @printf(i8*  %tmp_14 , i32  %tmp_15 ,
i32  %tmp_16 )
 %tmp_18  = add i32 0,0
 ret i32  %tmp_18
}
```
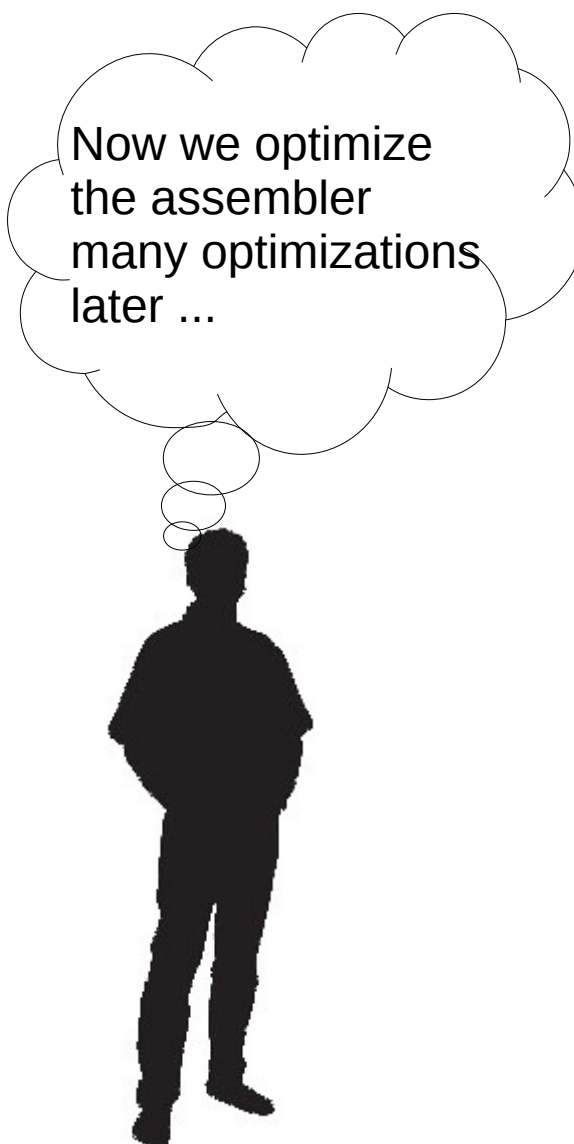
This is the output
of the complier
as you can see the code
is unoptimized
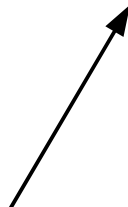
> opt -std-compile-opts -S $test2.ll > test2_opt.ll

Now we optimize
the assembler
many optimizations
later ...

```
; ModuleID = 'test2.ll'
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:...
target triple = "i386-pc-linux-gnu"

@main.str1 = private constant [16 x i8] c"sum 1..%i = %i\0A\00"

declare i32 @printf(i8* nocapture, ...) nounwind

define i32 @main() nounwind {
endfor_5:
  %tmp_17 = tail call i32 (i8*, ...)*
    @printf(i8* getelementptr inbounds ([16 x i8]* @main.str1, i32 0, i32 0),
        i32 1000000,
        i32 50005000) nounwind ;
  ret i32 0
}
```

The optimizer has
detected the loop in the
assembler program and
unrolled the loop
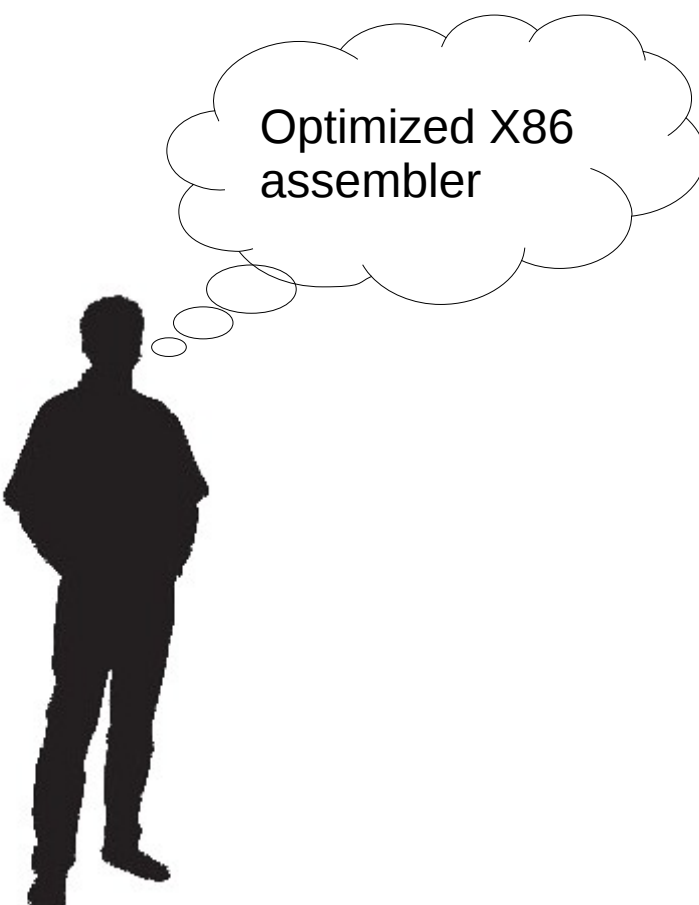
This is optimized assembler
It was produced by the program
"opt" which is part
of the LLVM distribution

"A large number of optimizations are perfomed on the assember code"

llvm-as < /dev/null | opt -std-compile-opts -disable-output -debug-pass=Arguments
Pass Arguments: -preverify -domtree -verify -lowersetjmp -globalopt -ipsccp
-deadargelim -instcombine -simplifycfg -basiccg -prune-eh -inline -functionattrs
-argpromotion -domtree -domfrontier -scalarrepl -simplify-libcalls -instcombine -jump-
threading -simplifycfg -instcombine -tailcallelim -simplifycfg -reassociate -domtree
-loops -loopsimplify -domfrontier -loopsimplify -lcssa -loop-rotate -licm -lcssa -loop-
unswitch -instcombine -scalar-evolution -loopsimplify -lcssa -iv-users -indvars -loop-
deletion -loopsimplify -lcssa -loop-unroll -instcombine -memdep -gvn -memdep
-memcpyopt -sccp -instcombine -jump-threading -domtree -memdep -dse -adce
-simplifycfg -strip-dead-prototypes -print-used-types -deadtypeelim -globaldce
-constmerge -preverify -domtree -verify

>llc test2_opt.ll -o test2_opt.s

Optimized X86 assembler

Now we use
one of the built-in LLVM
backends to produce
optimised X86 32 bit code

```
        .file    "test2_opt.ll"
        .text
        .globl        main
        .align        16, 0x90
        .type main,@function
main:                           # @main
# BB#0:                         # %endfor_5
        subl  $12, %esp
        movl $50005000, 8(%esp)     # imm = 0x2FB0408
        movl $10000, 4(%esp)        # imm = 0x2710
        movl $.Lmain.str1, (%esp)
        call   printf
        xorl   %eax, %eax
        addl  $12, %esp
        ret
        .size  main, .-main

        .type .Lmain.str1,@object
        .section     .rodata.str1.1,"aMS",@progbits,1
.Lmain.str1:                       # @main.str1
        .asciz         "sum 1..%i = %i\n"
        .size .Lmain.str1, 16

        .section     .note.GNU-stack,"",@progbits
```

ECC passes

- Parsing and type normalization
- Normalize heads
- Make prototypes
- Lift vars
- Fix scopes
- Fold Types
- Compile to LLVM
- Render assembled code

Parsing

- 955 lines of yrl - derived from a parser written by Tony Rogvall

Type normalisation

- Convert boustrophedonic types to Erlang terms

- int *p[10]

  {arrayOf, 10,{pointerTo, int}}

- int (*p)[10]

  {pointerTo, {arrayOf,10,int}}

Boustrophodonic

Of or relating to text written from left to right and right to left in alternate lines. [1]

Examples are the rongo-rongo script of Easter Island, some of those in the Etruscan language, a few early Latin inscriptions and some ancient Greek texts created in a transitional period at about 500BC before which writing ran from right to left but afterwards from left to right. The word is itself from the Greek meaning "as the ox ploughs". [1]

From βούς—bous, "ox" + στρέφειν—strephein, "to turn" (cf. strophe) [2]

[1] http://www.worldwidewords.org/weirdwords/ww-bou1.htm

[2] wikipedia

Normalize heads

- Merge K&R style heads with ANSI

  int foo(int i, int j) {...}

  and

  int foo(i, j)

    int i, j;

    {…}

  are identical after normalisation

make prototypes

- make prototypes for all functions

   Use the specified form if given

   If not supplied use the derived type signature

Lift vars

```
foo(){

   char *c="abc";

   bar("123");

   ...
```

```
foo(){

   int i = 6;

   ...
```

```
foo(){

   static int i = 6;

   …
```

```
static char *tmp1="abc";

static char *tmp2="123"

foo(){

   char *c;

   c = tmp1;

   bar(tmp2);

   ...
```

```
foo(){

int i;

i = 6;
```

```
int tmp23 = 6;

foo(){

   /* rename i as
      tmp23 */
```

Simple compiler

```
compile_statement(#'FOR'{init=Init,test=Test,update=Update,body=Body}) ->
      {_,InitCode}             = compile_statement(Init),
      {_,UpdateCode}           = compile_statement(Update),
      {{_Type,Reg}, TestCode}  = compile_rval(Test),
      BodyCode                 = compile_compound(Body),
      InitLabel                = new_label("initfor"),
      UpdateLabel              = new_label("updatefor"),
      TestLabel                = new_label("testfor"),
      BodyLabel                = new_label("bodyfor"),
      EndLabel                 = new_label("endfor"),
      Code = [{br,InitLabel},
            InitLabel,   InitCode,   {br,TestLabel},
            UpdateLabel, UpdateCode,{br,TestLabel},
            TestLabel,   TestCode,   {br,{eTYPE,i1}, Reg,
                                      BodyLabel, EndLabel},
            BodyLabel,   BodyCode,   {br,UpdateLabel},
            EndLabel],
      {void(),C};
```
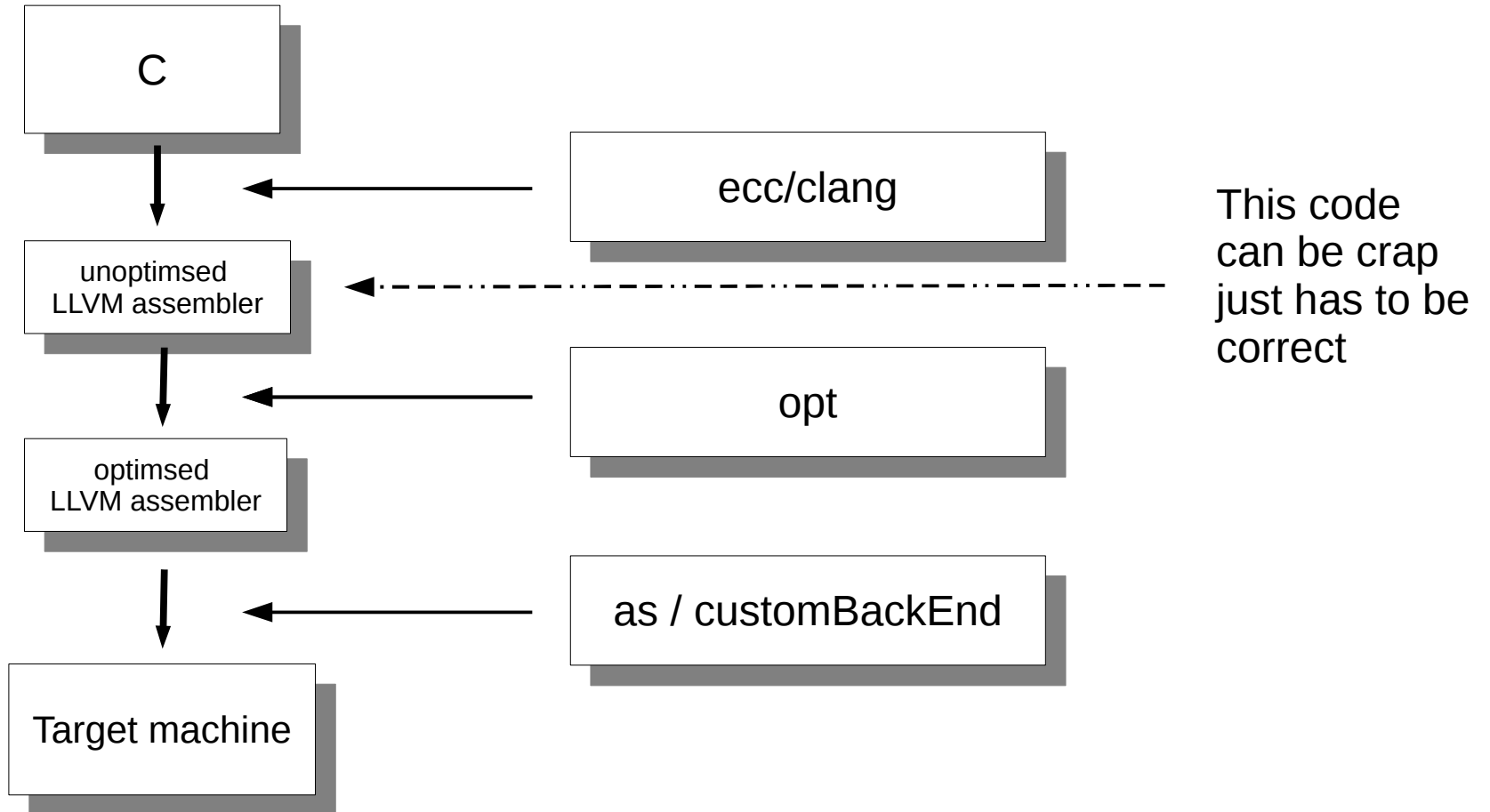
```
      br init
init:      ...
      br test
update:
      ..     br test
test:
    …
   br true body; false end
body:
    ...
    br update
end:
```

Compiling a for loop

Statistics

```
 186     531    5067 ecc_asm_llvm.erl
 112     322    2889 ecc_compile.erl
 803    2671   24296 ecc_cpp.erl
 108     330    2362 ecc_cpp_eval_expr.erl
  56     109     927 ecc_db.erl
  25      51     560 ecc.erl
 231     846    7366 ecc_fix_scopes.erl
  61     220    1746 ecc_fold_types.erl
 333    1168   10534 ecc_gen_llvm.erl
 223     621    5520 ecc_lib.erl
 107     302    3213 ecc_lift_vars.erl
  38      93     932 ecc_make_db.erl
  73     271    2400 ecc_make_prototypes.erl
  82     259    2746 ecc_normalize_function_heads.erl
 156     500    5129 ecc_parser.erl
 101     355    3478 ecc_typedefs.erl
  40      86    1044 record_compiler.erl
 955    3388   39131 ecc_gram.yrl
3690   12123  119340 total
```

Summary

C

ecc/clang

unoptimsed
LLVM assembler

This code
can be crap
just has to be
correct

opt

optimsed
LLVM assembler

as / customBackEnd

Target machine

Stuff I haven't mentioned

- JIT is easy

- Gecode

- "inline C" in Erlang

- "inline X" in Erlang

- Integration with Erlang (ie make bitcode in Erlang not using llvm-as, and integration with BEAM)

Status

- Will be released as open source
- Needs more work
- "tricky bits done"
- Probably should be rewritten (again - this will be the 5'th rewrite)

Questions