Bringing GNU Emacs to Native Code

Andrea Corallo, Luca Nassi, Nicola Manca

22-04-2020
Design

- Emacs is a Lisp implementation (Emacs Lisp).
- It’s made to sit on top of OS slurping and processing text to present it in uniform UIs.
- Most of Emacs core is written in Emacs Lisp (~80%).
- ~20% is C (~300 kloc) mainly for performance reason.
- Arguably the most deployed Lisp today?

Emacs Lisp Nowadays

- Sort of a small CL-ish Lisp.
- Has no standard and is still evolving (slowly).
- Elisp is byte-compiled.
- Byte interpreter is implemented in C.
- Emacs has an optimizing byte-compiler written in Elisp.
Elisp sucks (?)

- No lexical scope.
  Two coexisting Lisp dialects.
- Lacks multi-threading.
- Lack of true multi-threading.
- No name spaces.
- It’s slow.

Still not a general purpose Programming Language
Emacs Future
Emacs Future
Emacs Future

Guile
GNU extension language

Rust Programming Language
Emacs Future

- Guile (GNU extension language)
- EVIL
- The Rust Programming Language
Emacs Future

C as a base language is fine as long as is not abused
   ▶ "Lingua franca" ubiquitous programming language.
   ▶ High performance.

The big win is to have a better Lisp implementation
   ▶ Benefit all existing Elisp.
   ▶ Less C to maintain in long term.
   ▶ Emacs becomes more easily extensible.

Previous attempts:
   ▶ Elisp on top of Guile (Guile-emacs).
   ▶ Various attempt to target native code in the past: 3 jitters, 1 compiler targeting C (https://tromey.com).
Elisp byte-code

- Push and pop stack-based VM.
- Lisp expression:
  
  \[ (* (+ a 2) 3) \]

- Lisp Assembly Program LAP:
  
  (byte-varref a)
  (byte-constant 2)
  (byte-plus)
  (byte-constant 3)
  (byte-mult)
  (byte-return)
Elisp byte-code execution

0 (byte-varref a)
1 (byte-constant 2)
2 (byte-plus)
3 (byte-constant 3)
4 (byte-mult)
5 (byte-return)
Elisp byte-code execution

0 (byte-varref a) \leq
1 (byte-constant 2)
2 (byte-plus)
3 (byte-constant 3)
4 (byte-mult)
5 (byte-return)
Elisp byte-code execution

0 (byte-varref a)
1 (byte-constant 2) <=
2 (byte-plus)
3 (byte-constant 3)
4 (byte-mult)
5 (byte-return)
Elisp byte-code execution

0 (byte-varref a)
1 (byte-constant 2)
2 (byte-plus)  <=
3 (byte-constant 3)
4 (byte-mult)
5 (byte-return)
Elisp byte-code execution

0 (byte-varref a)
1 (byte-constant 2)
2 (byte-plus)
3 (byte-constant 3) <=
4 (byte-mult)
5 (byte-return)
Elisp byte-code execution

0 (byte-varref a)
1 (byte-constant 2)
2 (byte-plus)
3 (byte-constant 3)
4 (byte-mult) <=
5 (byte-return)
Elisp byte-code execution

0 (byte-varref a)
1 (byte-constant 2)
2 (byte-plus)
3 (byte-constant 3)
4 (byte-mult)
5 (byte-return) ≤

SP → Exec stack

306
Elisp byte-code execution

Byte compiled code
- Fetch
- Decode
- Execute:
  - stack manipulation.
  - real execution.

Native compiled code
- Better leverage CPU for fetch and decode.
- Nowadays CPU are not stack-based but register-based.
;; "No matter how hard you try, you can't make
;; a racehorse out of a pig.
;; You can, however, make a faster pig."

Jamie Zawinski byte-opt.el.
Object manipulation

Manipulating every object requires

- Checking its type.
- Handle the case where the type is wrong.
- Access the value (tag subtraction).
- Do something.
- "Box" the output object.
The plan
Native compiler requirements

- Perform Lisp specific optimizations.
- Allow GCC to optimize (exposing common operations).
- Produce re-loadable code.

Not a Jitter!
Emacs does not fit well with the conventional JIT model:
- Compile once runs many.
  Worth invesing in compile time.
- Don’t want to recompile the same code every new session.
Plugging into GCC

libgccjit

- Added by David Malcolm in GCC 5.
- The venerable GCC compiled as shared library.
- Drivable programmatically describing libgccjit IR describing a C-ish semantic.
- Despite the name, you can use it for Jitters or AOT.
- A programmable GCC front-end.
Basic byte-code compilation algorithm

- **Byte-code:**
  0 (byte-varref a)
  1 (byte-constant 2)
  2 (byte-plus)
  3 (byte-constant 3)
  4 (byte-mult)
  5 (byte-return)

- For every PC stack depth is known at compile time.

- **Compiled pseudo code:**

```plaintext
Lisp_Object local[2];
local[0] = varref (a);
local[1] = two;
local[0] = plus (local[0], local[1]);
local[1] = three;
local[0] = mult (local[0], local[1]);
```
Why optimizing outside GCC

- The GCC infrastructure has no knowledge of primitives return type.
- GCC has no knowledge of which Lisp functions are optimizable and in which conditions.
- GCC does not provide help for boxing and unboxing values.

The trick is to generate code using information on Lisp that GCC will be able to optimize.
The plan

**Stock byte-compiler pipeline**

```
elisp → LAP → byte-code (.elc)
```

**Native compiler pipeline**

```
elisp → LAP

LIMPLE → LIMPLE SSA → libgccjit IR → libgccjit → native code (.eln)
```
Native compiler implementation

Relies on **LIMPLE IR** and is divided in passes:

1. spill-lap
2. limplify
3. ssa
4. propagate
5. call-optim
6. dead-code
7. tail-recursion-elimination
8. propagate
9. final

**speed is back**

Optimizations like in CL are controlled by `comp-speed` ranging from 0 to 3.
Passes: spill-lap

- The input used for compiling is the internal representation created by the byte-compiler (LAP).
- It is used to get the byte-code before being assembled.
- This pass is responsible for running the byte-compiler and extracting the LAP IR.
Convert LAP into LIMPLE.

**LIMPLE**
- Named LIMPLE as tribute to GCC GIMPLE.
- Control Flow Graph (CFG) based.
- Each function is a collection of basic blocks.
- Each basic block is a list of *insn*. 
Passes: simplify

... cond branch
... cond branch
...
Passes: limplify
Passes: simplify
Passes: limplify
Static Single Assignment
Bring LIMPLE into SSA form

- Create edges connecting the various basic blocks.
- Compute dominator tree for each basic block.
- Compute the dominator frontiers.
- Place \( \phi \) functions.
- Rename variables to become uniquely assigned.
Passes: propagate

Iteratively propagates within the control flow graph for each variable value, type and ref-prop.

- Return types known for certain primitives are propagated.
- Pure functions and common integer operations are optimized out.

Done also by the byte-optimizer Propagate has greater chances to succeeds due to the CFG analysis.
Passes: call-optim - funcall trampoline

- Byte-compiled code calls directly functions that got a dedicated opcode.
- All the other has to use the `funcall` trampoline!

**A primitive that, when called, lets you call something else**

The most generic way to dispatch a function call.

- Primitives.
- Byte compiled.
- Interpreted.
- Advised functions...
Passes: call-optim - example
Passes: call-optim - example
Passes: call-optim - example
Passes: call-optim - example
Passes: call-optim - example

All primitives get the same dignity
Passes: call-optim - intra compilation unit

What about intra compilation unit functions?
Passes: call-optim - intra compilation unit

What about intra compilation unit functions?
What about intra compilation unit functions?

The system **should** be resilient to in flight function redefinition.

Really!?
Passes: call-optim - the dark side

BUT WE CAN'T OPTIMIZE!

...SPEED 3
Passes: call-optim - intra compilation unit (speed 3)

Allow GCC IPA passes to take effect.
int foo (int a, int b)
{
    ...
    ...

    return foo (d, c);
}
int
foo (int a, int b)
{
ininit:
    ...  
    ...
    ...
    a = d;
    b = c;
    goto init;
}

- Does not consume implementation stack.
- Better support functional programming style.
Drives LIMPLE into libgccjit IR and invokes the compilation.

Also responsible for:

- Defining the inline functions that give GCC visibility on the Lisp implementation.
- Suggesting to them the correct type if available while emitting the function call.

```c
static Lisp_Object car (Lisp_Object c, bool cert_cons)
```

Final is the only pass implemented in C.
The result of the compilation process for a compilation unit is a file with `.eln` extension (Emacs Lisp Native).

Technically speaking, this is a shared library where Emacs expects to find certain symbols to be used during load by the load machinery.
To allow the user to feed the propagation engine with type suggestions, two entry points have been implemented:

- comp-hint-fixnum
- comp-hint-cons

(comp-hint-fixnum x) to promise that this expression evaluates to a fixnum.
As in Common Lisp these are trusted when compiling optimizing and treated as assertion otherwise.
Integration

Unload
Through garbage collector integration.

Image Dump
Through portable dumper integration.

Build system
Native bootstrap and installation.

Documentation and source integration
Go to definition and documentation works as usual
disable disassemble disassemble native code.
Deferred compilation

Minimize compile-time impact:
➢ Byte-code load triggers an async compilation.
➢ Perform a "late load".
Deferred compilation

byte compilation

\[ t_0 \rightarrow t_1 \]

\[ .elc \]

File System
Deferred compilation
Deferred compilation

byte compilation

t0 \rightarrow t1

async native compilation

.elc

.eln

File System
Deferred compilation

byte compilation

async native compilation

Function defs automagically swapped

File System
Deferred compilation

- Works well for packages.
- Usable for Emacs compilation too.
Performance

Optimizing

okay but for what?

elisp-benchmarks

Up-streamed on GNU ELPA a package with a bunch of micro benchmarks.

https://elpa.gnu.org/packages/elisp-benchmarks.html

Some ported from CL some new.
Performance - results

- benchmarks compiled at speed 3.
- Emacs compiled at speed 2.

Results

<table>
<thead>
<tr>
<th>benchmark</th>
<th>byte-comp</th>
<th>native (s)</th>
<th>speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>inclist</td>
<td>19.54</td>
<td>2.12</td>
<td>9.2x</td>
</tr>
<tr>
<td>inclist-type-hints</td>
<td>19.71</td>
<td>1.43</td>
<td>13.8x</td>
</tr>
<tr>
<td>listlen-tc</td>
<td>18.51</td>
<td>0.44</td>
<td>42.1x</td>
</tr>
<tr>
<td>bubble</td>
<td>21.58</td>
<td>4.03</td>
<td>5.4x</td>
</tr>
<tr>
<td>bubble-no-cons</td>
<td>20.01</td>
<td>5.02</td>
<td>4.0x</td>
</tr>
<tr>
<td>fibn</td>
<td>20.04</td>
<td>8.79</td>
<td>2.3x</td>
</tr>
<tr>
<td>fibn-rec</td>
<td>20.34</td>
<td>7.13</td>
<td>2.9x</td>
</tr>
<tr>
<td>fibn-tc</td>
<td>21.22</td>
<td>5.67</td>
<td>3.7x</td>
</tr>
<tr>
<td>dhrystone</td>
<td>18.45</td>
<td>7.22</td>
<td>2.6x</td>
</tr>
<tr>
<td>nbody</td>
<td>19.79</td>
<td>3.31</td>
<td>6.0x</td>
</tr>
</tbody>
</table>
Performance - analysis

Looking at CPU performance events (PMUs)

- Big reduction in instruction executed.
- Instruction mix shows less load/store.
- CPU misprediction decrease (easier code to digest for the prediction unit).
State of the project

Sufficiently stable to be used in production

- Bootstrap clean compiling all lexically scoped Emacs files plus external packages.
- Fairly stable (weeks of up-time at speed 2).
- GNU/Linux X86_64, X86_32 (also wide-int), AArch64.

Further development

- Inter Procedural Analysis.
- Unboxing.
- Exposing more primitives to GCC.
- Providing warning and errors using the propagation engine.
State of the project - upstream

- Approached in November.
- Since January landed on emacs.git as feature branch feature/native-comp!
- Currently rounding (lasts?) edges.
Conclusions

Wanna help the pig fly!?
Conclusions

Wanna help the pig fly!?
Conclusions

Wanna help the pig fly!?

Other info:
http://akrl.sdf.org/gccemacs.html
https://debbugs.gnu.org/Emacs.html
akrl@sdf.org
emacs-devel@gnu.org