Bringing GNU Emacs to Native Code

Andrea Corallo, Luca Nassi, Nicola Manca

22-04-2020

Outline

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 (?)

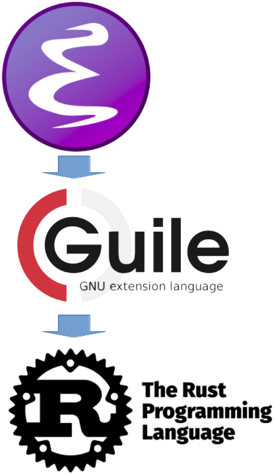
- ~~No lexical scope~~.
  Two coexisting Lisp dialects.
- ~~Lacks multi threading~~.
- Lack of true multi-threading.
- No name spaces.
- It's slow.

Still not a general purpose Programming Language

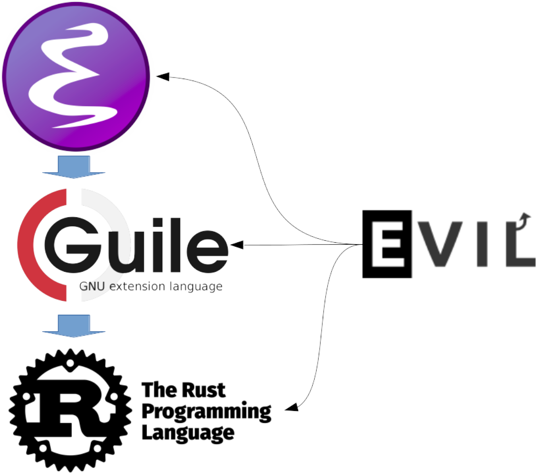
Emacs Future

Emacs Future

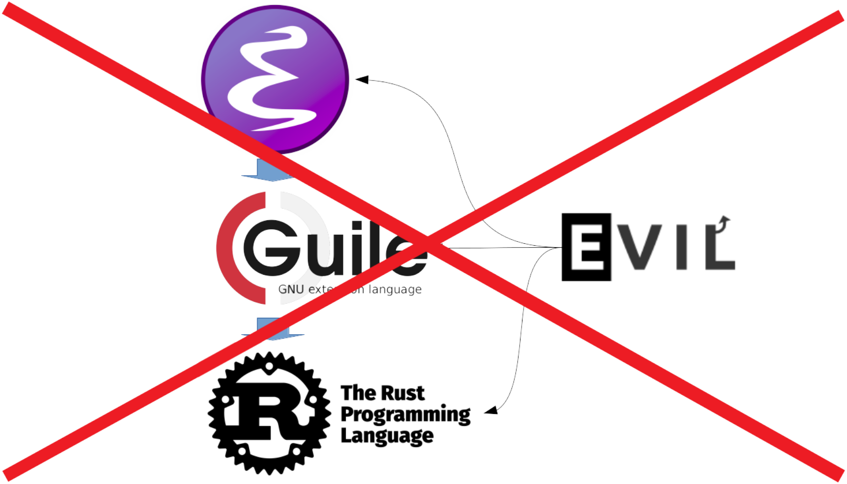
Emacs Future



Emacs Future



Emacs Future



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:

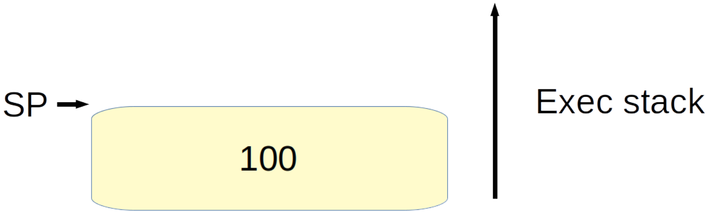
  ```
  (byte-varref a)
  (byte-constant 2)
  (byte-plus)
  (byte-constant 3)
  (byte-mult)
  (byte-return)
  ```

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)
```

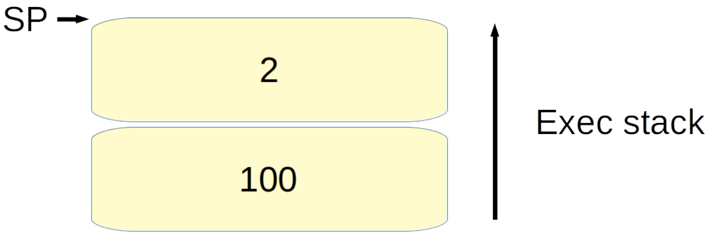
Elisp byte-code execution

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



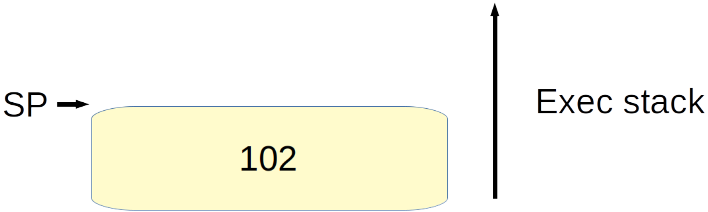
Elisp byte-code execution

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



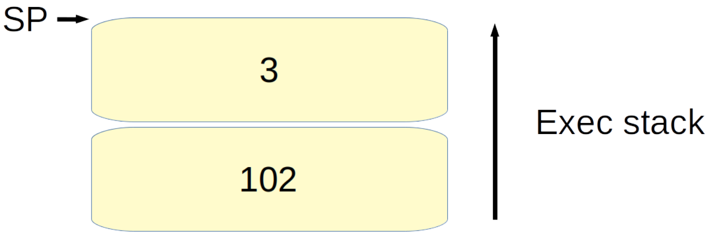
Elisp byte-code execution

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



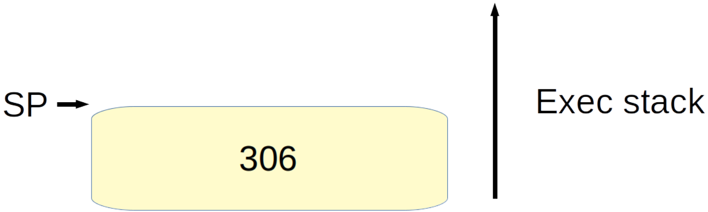
Elisp byte-code execution

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



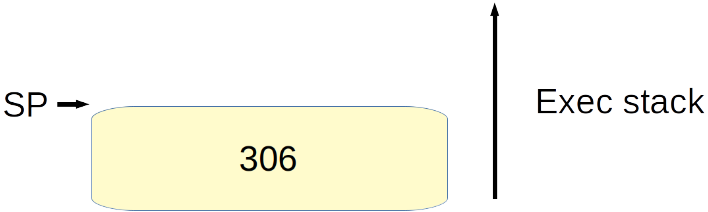
Elisp byte-code execution

```
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)     <=
```



Elisp byte-code execution

Byte compiled code

- Fetch
- Decode
- Execute:
  - stack manipulation.
  - real execution.

Native compiled code

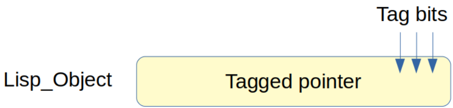
- Better leverage CPU for fetch and decode.
- Nowadays CPU are not stack-based but register-based.

Elisp byte-code 2

```
;; "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`.

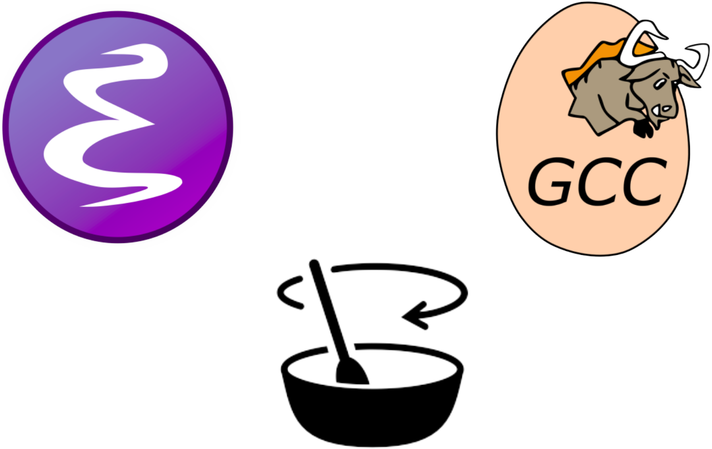
Object manipulation



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:

```
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

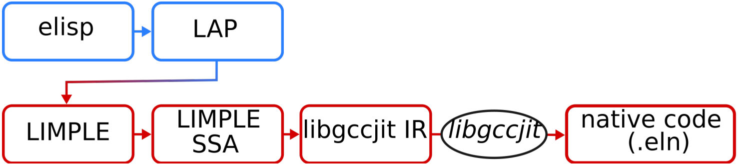
- The GCC infrastructure has no knowledge of primitives return type.
- GCC has no knowledge of which Lisp functions are optimizable and in which conditions.
- GCC does not provide help for boxing and unboxing values.

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

The plan

Stock byte-compiler pipeline

Native compiler pipeline



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.

Passes: limplify

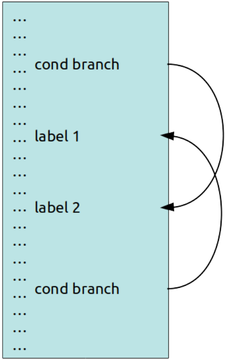
Convert LAP into LIMPLE.

LIMPLE

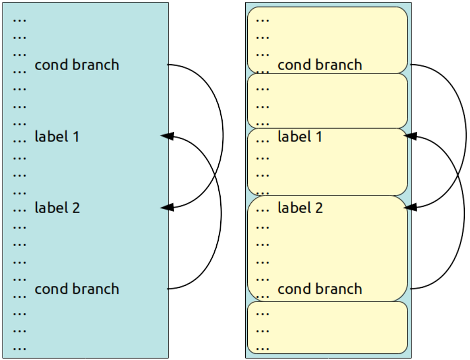
- Named LIMPLE as tribute to GCC GIMPLE.
- Control Flow Graph (CFG) based.
- Each function is a collection of basic blocks.
- Each basic block is a list of `insn`.

Passes: limplify

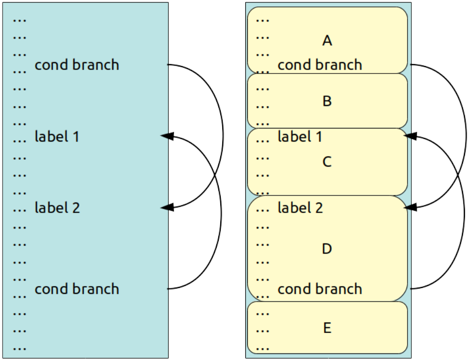
Passes: limplify



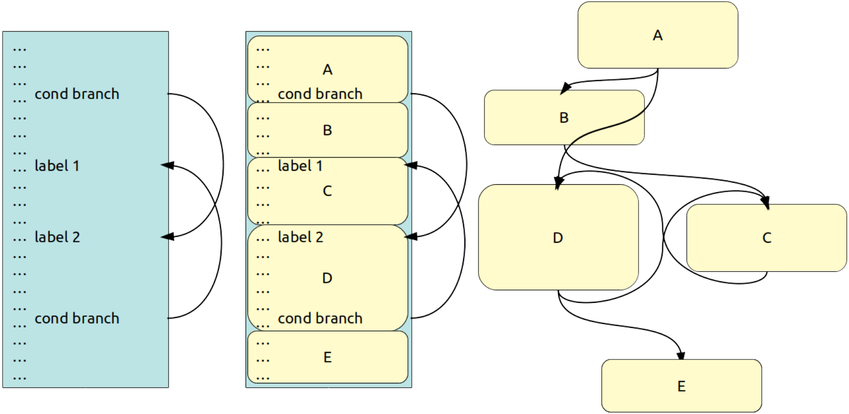
Passes: limplify



Passes: limplify



Passes: limplify



Passes: ssa

Static Single Assignment

Bring LIMPLE into SSA form

`http://ssabook.gforge.inria.fr/latest/book.pdf`

- 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

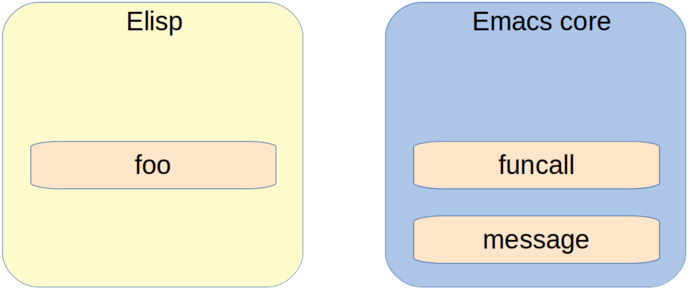
- Byte-compiled code calls directly functions that got a dedicated opcode.
- All the other has to use the `funcall` trampoline!

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

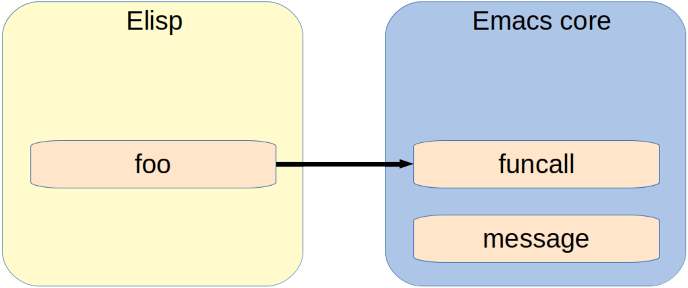
The most generic way to dispatch a function call.

- Primitives.
- Byte compiled.
- Interpreted.
- Advised functions. . .

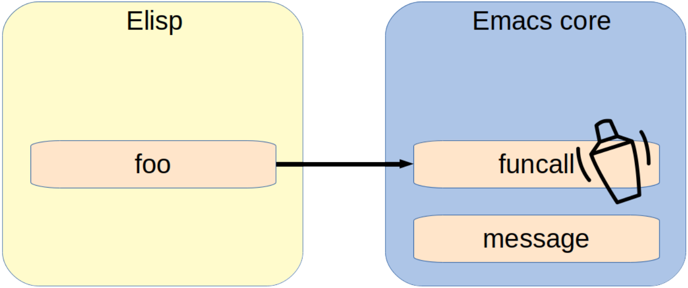
Passes: call-optim - example



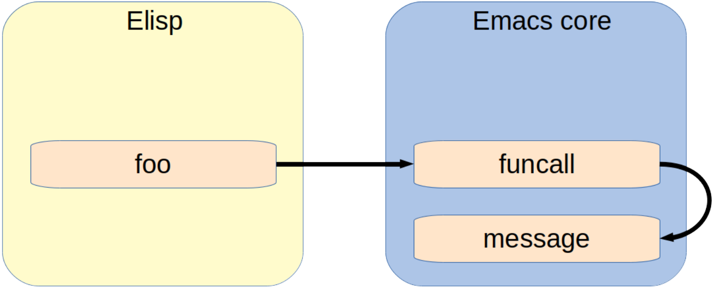
Passes: call-optim - example



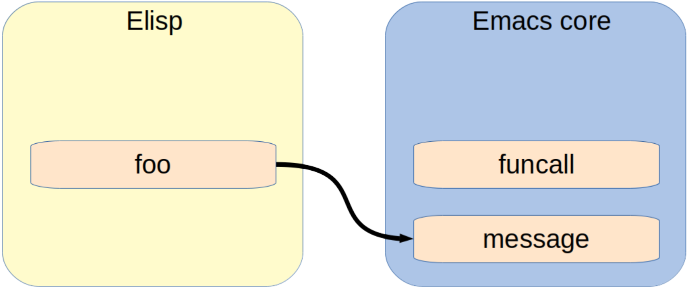
Passes: call-optim - example



Passes: call-optim - example



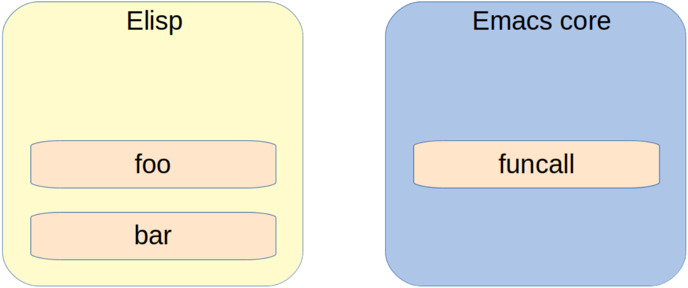
Passes: call-optim - example



All primitives get the same dignity

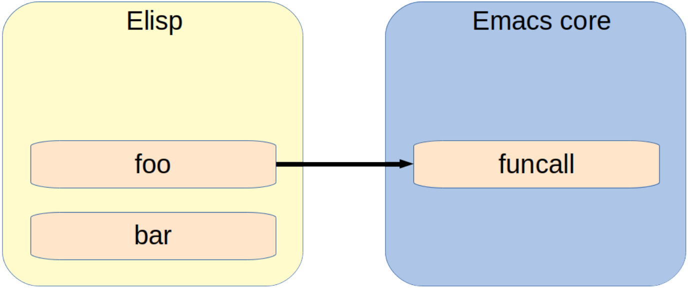
Passes: call-optim - intra compilation unit

What about intra compilation unit functions?



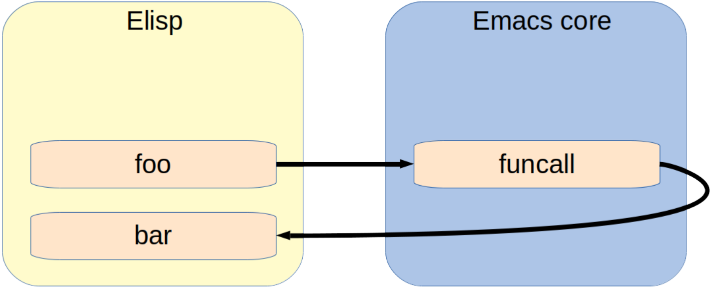
Passes: call-optim - intra compilation unit

What about intra compilation unit functions?



Passes: call-optim - intra compilation unit

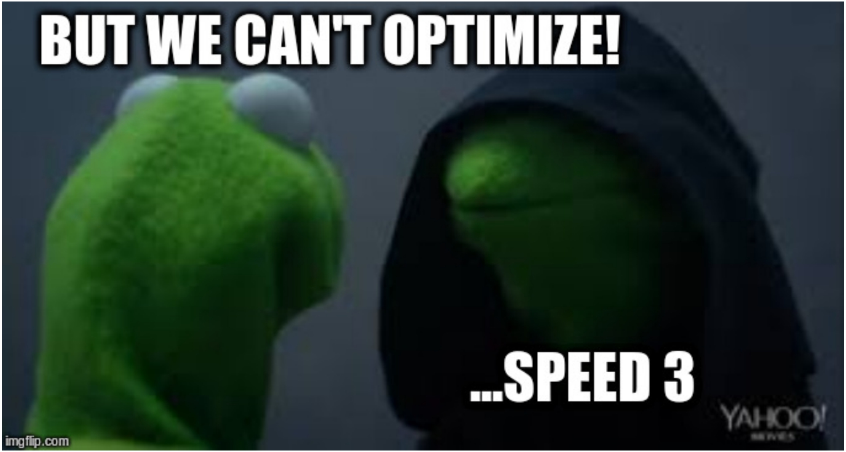
What about intra compilation unit functions?



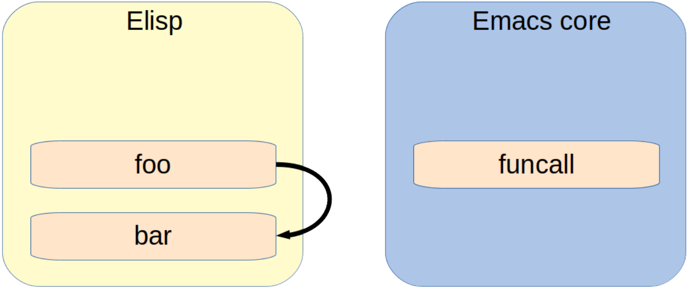
The system <u>should</u> be resilient to in flight function redefinition.

Really!?

Passes: call-optim - the dark side



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



Allow GCC IPA passes to take effect.

Passes: tail-recursion-elimination

```
int
foo (int a, int b)
{
  ...
  ...

  return foo (d, c);
}
```

Passes: tail-recursion-elimination

```
int
foo (int a, int b)
{
 init:
   ...
   ...

   a = d;
   b = c;
   goto init;
}
```

- Does not consume implementation stack.
- Better support functional programming style.

Passes: final - interface libgccjit

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.

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

Final is the only pass implemented in C.

Passes: final - `.eln`

- 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.

Extending the language - Compiler type hints

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
`disassemble` disassemble native code.

Integration

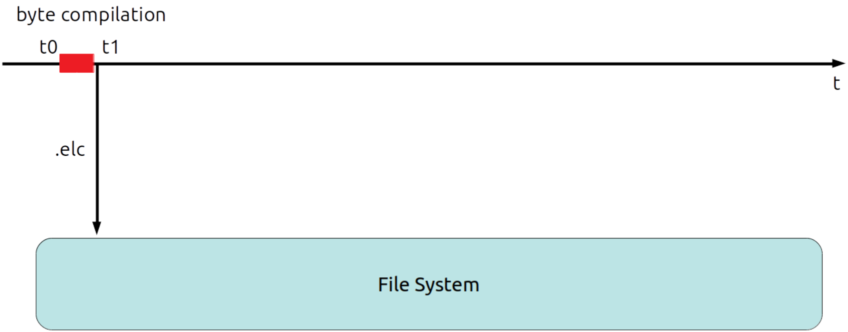
```
00000000001781a0 <F6f72672d6d6f6465_org_mode>:
  1781a0:       41 55                   push   %r13
  1781a2:       41 54                   push   %r12
  1781a4:       55                      push   %rbp
  1781a5:       53                      push   %rbx
  1781a6:       48 81 ec 28 03 00 00    sub    $0x328,%rsp
  1781ad:       48 8b 2d 34 1e 22 00    mov    0x221e34(%rip),%rbp        # 399fe8 <fr
eloc_link_table@@Base-0x7e900>
  1781b4:       48 8b 1d 25 1e 22 00    mov    0x221e25(%rip),%rbx        # 399fe0 <d_
reloc@@Base-0x77ba0>
  1781bb:       48 8b 45 00             mov    0x0(%rbp),%rax
  1781bf:       48 8b bb d8 65 00 00    mov    0x65d8(%rbx),%rdi
  1781c6:       ff 90 38 24 00 00       callq  *0x2438(%rax)
  1781cc:       48 8b 45 00             mov    0x0(%rbp),%rax
  1781d0:       48 8b b3 18 02 00 00    mov    0x218(%rbx),%rsi
  1781d7:       48 8b bb d8 65 00 00    mov    0x65d8(%rbx),%rdi
  1781de:       ff 50 58                callq  *0x58(%rax)
  1781e1:       48 8b 83 e0 65 00 00    mov    0x65e0(%rbx),%rax
  1781e8:       48 8d 74 24 10          lea    0x10(%rsp),%rsi
  1781ed:       bf 01 00 00 00          mov    $0x1,%edi
  1781f2:       48 89 44 24 10          mov    %rax,0x10(%rsp)
  1781f7:       48 8b 45 00             mov    0x0(%rbp),%rax
  1781fb:       ff 90 58 19 00 00       callq  *0x1958(%rax)
  178201:       48 8b 45 00             mov    0x0(%rbp),%rax
  178205:       31 c9                   xor    %ecx,%ecx
  178207:       31 d2                   xor    %edx,%edx
  178209:       48 8b 73 30             mov    0x30(%rbx),%rsi
  17820d:       48 8b 7b 18             mov    0x18(%rbx),%rdi
  178211:       ff 50 48                callq  *0x48(%rax)
  178214:       48 8b 45 00             mov    0x0(%rbp),%rax
```

Deferred compilation

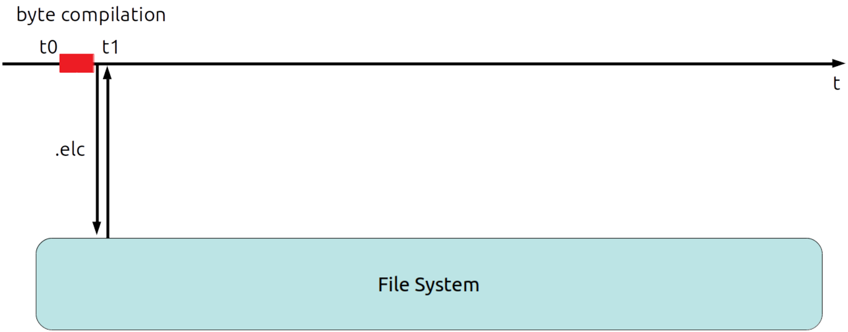
Minimize compile-time impact:

- Byte-code load triggers an async compilation.
- Perform a "late load".

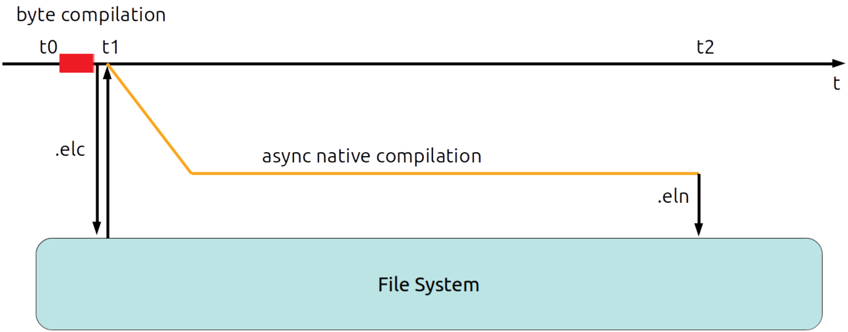
Deferred compilation



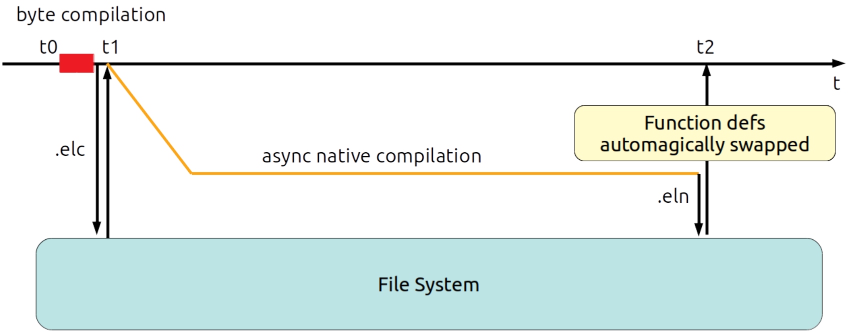
Deferred compilation



Deferred compilation



Deferred compilation

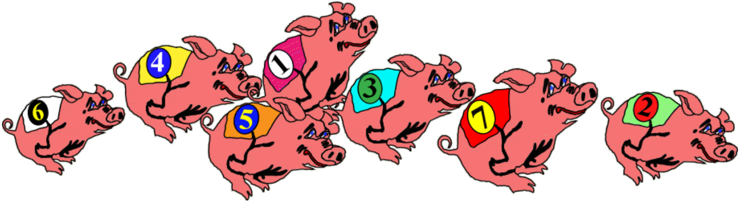


Deferred compilation

- 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

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

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