Gotta Prove Fast

Building an Ecosystem for Effortless Native Compilation of Tactics

Sebastian Ullrich | 2022/01/22
The Lean 4 Project [de Moura & Ullrich 2021]

Provide a fully extensible theorem proving frontend

Erase the boundary between built-in and custom syntax/tactics/... by reimplementing >75% of Lean in Lean itself

Make Lean an efficient, general-purpose programming language
Compiling Tactics – *How?*
Compiling Tactics – How?

- A Just-In-Time Compiler? LLVM JIT?
  + run tactic with native performance in the same file
  - re-compile tactic in every importing file...?
  - would first need a true LLVM backend
Compiling Tactics – How?

- A Just-In-Time Compiler? LLVM JIT?
  + run tactic with native performance in the same file
  - re-compile tactic in every importing file...?
  - would first need a true LLVM backend

- reuse Ahead-Of-Time toolchain for stand-alone Lean programs
  + much simpler... hopefully!
  + benefits stand-alone use case as well
  - should probably use the interpreter in the same file
Lean 4 Compilation Pipeline

- **elaborator**
  - core term
  - tactic
  - exec
  - IR
  - interpreter
  - C emitter
  - IR
  - file: .c
  - file: .exe

- **kernel**
  - core term

- **compiler**
  - core term

NB: The Trusted Code Base is unaffected!
Lean 4 Compilation Pipeline

- **elaborator**
  - core term
  - tactic exec
  - IR
- **kernel**
- **compiler**
  - core term
  - IR
- **interpreter**
- **C emitter**
  - IR
- **clang**
  - .dll
  - .c
  - .exe

NB: The Trusted Code Base is unaffected!
Lean 4 Compilation Pipeline

NB: The Trusted Code Base is unaffected!
So You Want to Build a Native Binary

**Installing rustup on Windows**

On Windows, go to [https://www.rust-lang.org/tools/install](https://www.rust-lang.org/tools/install) and follow the instructions for installing Rust. At some point in the installation, you'll receive a message explaining that you'll also need the C++ build tools for Visual Studio 2013 or later. The easiest way to acquire the build tools is to install Build Tools for Visual Studio 2019. When asked which workloads to install make sure "C++ build tools" is selected and that the **Windows 10 SDK** and the English language pack components are included.
So You Want to Build a Native Binary

Installing rustup on Windows

On Windows, go to https://www.rust-lang.org/tools/install and follow the instructions for installing Rust. At some point in the installation, you'll receive a message explaining that you'll also need the C++ build tools for Visual Studio 2013 or later. The easiest way to acquire the build tools is to install Build Tools for Visual Studio 2019. When asked which workloads to install make sure "C++ build tools" is selected and that the Windows 10 SDK and the English language pack components are included.

- Windows 10 SDK: 1.69 GB download
So You Want to Build a Native Binary

Installing `rustup` on Windows

On Windows, go to https://www.rust-lang.org/tools/install and follow the instructions for installing Rust. At some point in the installation, you’ll receive a message explaining that you’ll also need the C++ build tools for Visual Studio 2013 or later. The easiest way to acquire the build tools is to install Build Tools for Visual Studio 2019. When asked which workloads to install make sure "C++ build tools" is selected and that the **Windows 10 SDK** and the English language pack components are included.

- Windows 10 SDK: 1.69 GB download
- Xcode Command Line Tools: ~580 MB download
**So You Want to Build a Native Binary**

**Installing rustup on Windows**

On Windows, go to [https://www.rust-lang.org/tools/install](https://www.rust-lang.org/tools/install) and follow the instructions for installing Rust. At some point in the installation, you’ll receive a message explaining that you’ll also need the C++ build tools for Visual Studio 2013 or later. The easiest way to acquire the build tools is to install Build Tools for Visual Studio 2019. When asked which workloads to install make sure "C++ build tools" is selected and that the [Windows 10 SDK](https://developer.microsoft.com/en-us/windows/downloads/sdk/) and the English language pack components are included.

- Windows 10 SDK: 1.69 GB download
- Xcode Command Line Tools: ~580 MB download

We need effortless setup *not requiring root*!
So You Want to Build a Native Binary

**Installing rustup on Windows**

On Windows, go to [https://www.rust-lang.org/tools/install](https://www.rust-lang.org/tools/install) and follow the instructions for installing Rust. At some point in the installation, you’ll receive a message explaining that you’ll also need the C++ build tools for Visual Studio 2013 or later. The easiest way to acquire the build tools is to install Build Tools for Visual Studio 2019. When asked which workloads to install make sure "C++ build tools" is selected and that the **Windows 10 SDK** and the English language pack components are included.

The **Zig** language manages to provide small, self-contained toolchains for many platforms
Assembling a Native Compilation Pipeline

LLVM comes with many necessary parts:

- a C compiler
- basic C headers
- a runtime library
- a linker (good macOS support since LLVM 13)
- a static archiver
Assembling a Native Compilation Pipeline

LLVM comes with many necessary parts:
- a C compiler
- basic C headers
- a runtime library
- a linker (good macOS support since LLVM 13)
- a static archiver

It does not help at all with assembling
- the above with *reasonable* runtime dependencies
LLVM comes with many necessary parts:
- a C compiler
- basic C headers
- a runtime library
- a linker (good macOS support since LLVM 13)
- a static archiver

It does not help at all with assembling
- the above with *reasonable* runtime dependencies
- a libc
  - Linux: bundle older glibc for compatibility
  - macOS: bundle libSystem.tbd from SDK/Nixpkgs
  - Windows:
Assembling a Native Compilation Pipeline

LLVM comes with many necessary parts:

- a C compiler
- basic C headers
- a runtime library
- a linker (good macOS support since LLVM 13)
- a static archiver

It does not help at all with assembling

- the above with *reasonable* runtime dependencies
- a libc
  - Linux: bundle older glibc for compatibility
  - macOS: bundle `libSystem.tbd` from SDK/Nixpkgs
  - Windows:
    ```
    cp /clang64/x86_64-w64-mingw32/lib/lib{m,bcrypt,mingw32,moldname,mingwex,msvcr71,msvcr90,msvcr90kb,msvcr100,pthread,advapi32,shell32,user32,kernel32,ucrtbase}.*
    ```
Compiling Tactics – *Where?*

```
elab "my_tac" : tactic => ...
```

... by my_tac

... by my_tac

Must partition modules into shared libraries to avoid duplicate symbols or simply generate one library per module (& package).

Compiling Tactics – Where?
Compiling Tactics – *Where?*

```plaintext
elab "my_tac" : tactic => ...
...
... by my_tac
... by my_tac
```

Must partition modules into shared libraries to avoid duplicate symbols or simply generate one library per module (& package).
Compiling Tactics – Where?

```
... by my_tac
```

```
elab "my_tac" : tactic => ...
```

```
elab "my_tac2" ...
```

```
... by my_tac
```

Must partition modules into shared libraries to avoid duplicate symbols or simply generate one library per module (package).
Compiling Tactics – Where?

Must *partition* modules into shared libraries to avoid *duplicate symbols*
Compiling Tactics – *Where?*

Must *partition* modules into shared libraries to avoid *duplicate symbols*
Compiling Tactics – Where?

Must *partition* modules into shared libraries to avoid *duplicate symbols* … or simply generate one library per module (& package)
Compiling Tactics – *Unless?*

Compiling definitions that are never executed is wasteful

```
@tactic def myTac ...
def helper ...
def somethingElse ...
def tacHelper ...
import ...
theorem ...
```
Compiling Tactics – *Unless*?

Compiling definitions that are never executed is wasteful – *phase separation* [Flatt 2002] could tell us where the metaprograms are:

```
def helper ...
@tactic] meta def myTac ...
def somethingElse ...
meta def tacHelper ...
import def ...
import def ...
```
Compiling Tactics – *Unless?*

Compiling definitions that are never executed is wasteful – *phase separation* [Flatt 2002] could tell us where the metaprograms are

Bonus points if changing `helper` does not recompile `myTac` – *separate compilation!*
Summary

Today:

- Effortless native tactic compilation by coordinating the compiler, build system, and interpreter
- A stand-alone LLVM toolchain is possible

For the future:

- Scalability questions such as to number and size of shared libraries remain to be seen
- We want a better module/compilation unit system

1 https://github.com/leanprover/lean4/issues/416
Relevant Pull Requests

- *Build & distribute release builds with Zig as stand-alone C compiler* (abandoned)
  https://github.com/leanprover/lean4/pull/659

- *Bundle LLVM on all platforms* (merged)
  https://github.com/leanprover/lean4/pull/795

- *Simple, opt-in precompilation scheme*
  https://github.com/leanprover/lean4/pull/949
### Early mathlib4 Benchmarks (Warm ccache)

<table>
<thead>
<tr>
<th>diff [s]</th>
<th>drv</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.0976</td>
<td>-12.4%</td>
</tr>
<tr>
<td>-0.0882</td>
<td>-3.6%</td>
</tr>
<tr>
<td>-0.0812</td>
<td>-6.6%</td>
</tr>
<tr>
<td>+0.0745</td>
<td>+100.0%</td>
</tr>
<tr>
<td>+0.0697</td>
<td>+100.0%</td>
</tr>
<tr>
<td>+0.069</td>
<td>+100.0%</td>
</tr>
<tr>
<td>+0.0687</td>
<td>+100.0%</td>
</tr>
<tr>
<td>+0.0686</td>
<td>+100.0%</td>
</tr>
<tr>
<td>+0.0684</td>
<td>+100.0%</td>
</tr>
<tr>
<td>+0.0682</td>
<td>+100.0%</td>
</tr>
<tr>
<td>+0.0681</td>
<td>+100.0%</td>
</tr>
<tr>
<td>+0.068</td>
<td>+100.0%</td>
</tr>
</tbody>
</table>

... 

+7.84 +16.5% total