



MLIR Actions



Credits

- Chia Hung Duan
- Jacques Pienaar
- Jeff Niu
- Tomás Longeri
- Axel Villanueva (internship)

“Action”?

- Is a way to encapsulate any IR transformation
- Provides a mechanism to instrument the compiler

What can you do with it:

- Tracing the compiler: think “pass instrumentation” but finer grain.
- Implement compiler “fuel” / bisection techniques to find bugs.
- Interactive debugging of the compiler (“mlir-gdb”)

“Action” vs Pass

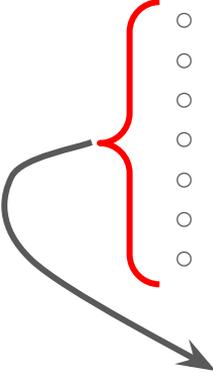
Pass:

- Well defined contract: “takes valid IR in, produce valid IR”
- Define a “phase” of compilation, intended to be composable (you can reorder optimization passes) => “basic block” to build the compiler.
- Is organized in a “pipeline” and scheduled by a “pass manager”
- Can be instrumented:
 - `--mlir-print-ir-after-all` (Print IR after each pass)
 - `--mlir-print-ir-after-change` (When printing the IR after a pass, only print if the IR changed)
 - `--mlir-print-ir-after-failure` (When printing the IR after a pass, only print if the pass failed)
 - `--mlir-print-ir-before=<pass-arg>` (Print IR before specified passes)
 - `--mlir-print-ir-before-all`
 - `--mlir-timing` (Display execution times)
 - *PassInstrumentation*: inject C++ code that runs before/after every pass.

“Action” vs Pass

Pass:

- Well defined contract: “takes valid IR in, produce valid IR”
- Define a “phase” of compilation, intended to be composable (you can reorder optimization passes) => “basic block” to build the compiler.
- Is organized in a “pipeline” and scheduled by a “pass manager”
- Can be instrumented:
 - `--mlir-print-ir-after-all` (Print IR after each pass)
 - `--mlir-print-ir-after-change` (When printing the IR after a pass, only print if the IR changed)
 - `--mlir-print-ir-after-failure` (When printing the IR after a pass, only print if the pass failed)
 - `--mlir-print-ir-before=<pass-arg>` (Print IR before specified passes)
 - `--mlir-print-ir-before-all`
 - `--mlir-timing` (Display execution times)
 - *PassInstrumentation*: inject C++ code that runs before/after every pass.



Why should this be specific to passes? Action is an opportunity to make the underlying infra and the feature set orthogonal and more generally applicable

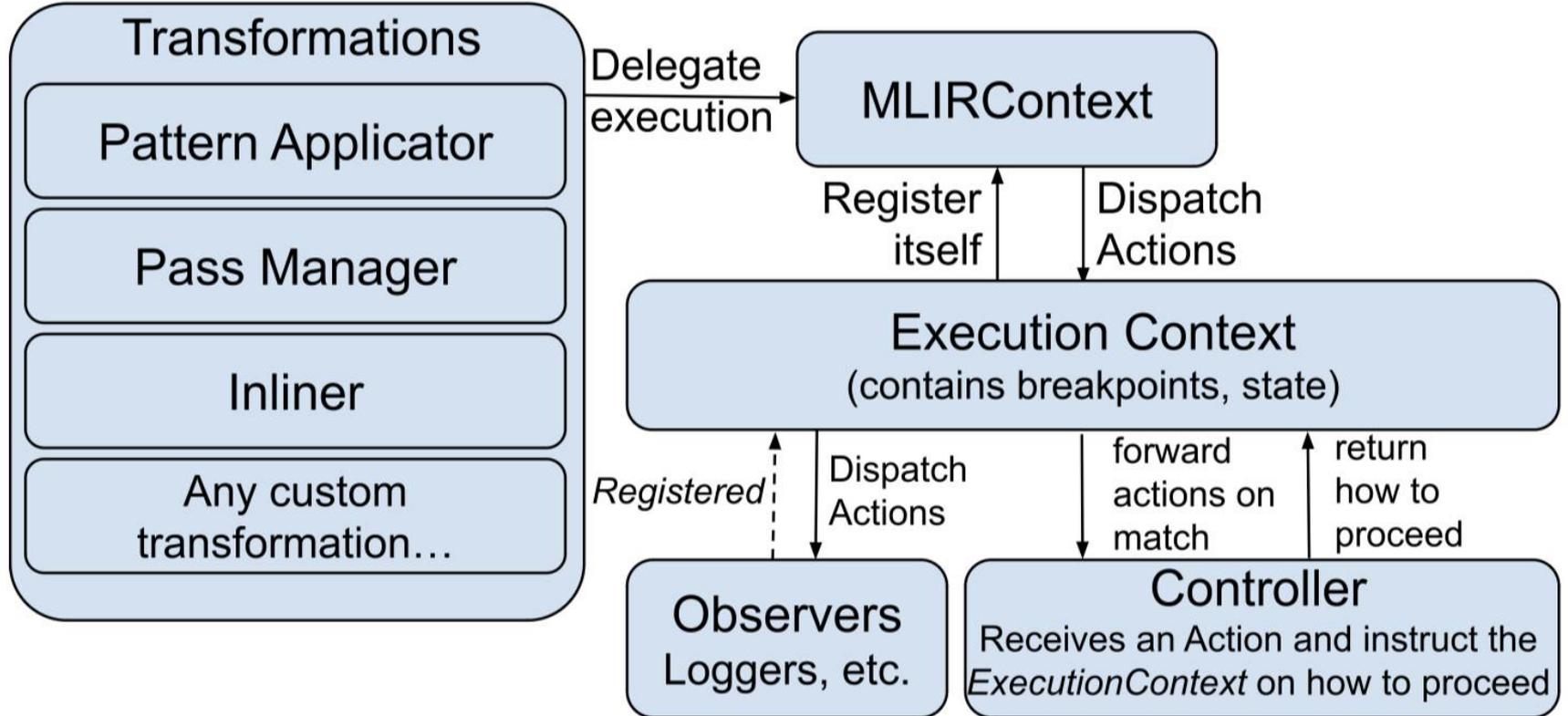
Demo

Demo

These are just examples to showcase the framework, the goal is to inspire you to build more of these :)

- Add more *Actions* and *dispatch()* in MLIR, and your own project!
- Snapshot the complete IR after every transforms touching a location
- Collect statistics on timing
- Build breakpoints for your own project (ML Graph node name?)
- Tools to skip some patterns at runtime?
- Can implement a “selective -O0”: “do not optimize this range of lines in the source code”.

Implementation: Overview



Performance overhead

- Cost of the dispatch: when not enabled it costs a pointer check on the MLIRContext: ~free.
- Cost of the instrumentation: full rich traces with locations are very expensive to produce (10x-20x slowdown with very naive implementation).
 - > Won't replace a profiler (actually could be used to emit tracy-annotations?)
 - > Complementary: much more tweakable / pluggable.

More

- DebugCounter / “Compiler fuel”
 - Bisection of a miscompile by skipping actions selectively.
 - Action framework exposes all the hooks, need to be scripted now!
Note: actions are “skippable” by default, but it can be overridden per action.

- Richer returned value:

```
struct ActionResult {  
    IRUnit op; // handle to the update IRUnit (can be a new one)  
    bool changed; // whether the IR was changed.  
    LogicalResult status; // whether the transforms succeeded or  
    not.  
}
```

- Multiple tags: add per-dispatch tags on top of the static one. For example instead of just dispatching “pass-execution-action”, we could also include the pass name.