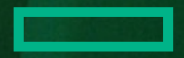


Hewlett Packard
Enterprise

CHAPEL 1.31/1.32 RELEASE NOTES: DYNO UPDATES

Chapel Team

June 22, 2023 / September 28, 2023

OUTLINE

- Background and Goals
- Summary of Progress since 1.30
- Details of Progress since 1.30
- Goals for 1.33 and 1.34
- Other Dyno Improvements

BACKGROUND AND GOALS

COMPILER REWORK EFFORT

- *Dyno* is an ongoing effort to address problems with the Chapel compiler
- Focused on improving:
  - Speed
  - Error messages
  - Compiler architecture and program representation
  - Compiler development

- Recent work has focused on:
  - Supporting the Chapel 2.0 effort
  - Replacing the early compilation passes with incremental versions, including an incremental resolver
  - Building better IDE support
  - Factoring the compiler into multiple processes coordinated by a compiler driver

COMPILER REWORK DELIVERABLES (1/2)

Faster Compilation with an Incremental Compilation Front-end

- Only re-parse and do type resolution based on files that were edited
  - Could result in reducing compilation time
  - Type resolution is one of the most time-consuming parts of compilation today
- Will still have the whole-program optimization and code-generation back-end

Faster Compilation with Separate Compilation

- Make most of the optimizations happen per-file
- Will need a linking step for optimizations like function inlining that span files
- Should result in significantly faster compilation times

COMPILER REWORK DELIVERABLES (2/2)

Dynamic Compilation and Evaluation

- Enable Chapel code snippets to be written and run interactively
  - e.g., in Jupyter notebooks

Reduced Memory Usage

- Using a compiler driver approach allows all compilation memory to be reclaimed before the link phase
- Should address out-of-memory errors when compiling large Chapel programs

SUMMARY OF PROGRESS SINCE 1.30

SUMMARY OF PROGRESS TOWARDS 1.31 AND 1.32 GOALS

1. Frontend Integration & Improving New Resolver
   - Goal: able to disable production scope resolver by 1.31
     - Enabled Dyno scope resolver in 1.31, though still relying on the production resolver for a few corner cases
   - Goal: new type resolver works in opt-in mode for end-to-end compilation for most tests by 1.32
     - Slipped due to reallocation of resources towards the Chapel 2.0 effort
2. Separate Compilation
   - Goal: Demonstrate saving generated code for a concrete function in the library files by 1.32
     - Slipped due to reallocation of resources towards the Chapel 2.0 effort
3. Incremental Compilation
   - Goal: Demonstrate live scope resolution from an editor by 1.31
     - Achieved for 1.32 with the Language Server Protocol effort
4. Compiler Driver
   - Goal: Get opt-in compiler driver support merged for 1.31
     - Achieved in 1.32 and available with '--compiler-driver'

SUMMARY OF OTHER PROGRESS

- Created a Python interface to the compiler library to help with Chapel 2.0 efforts
- Demonstrated it with two prototype tools:
  - a code rewriter (to help migrate existing tests and applications)
  - a linter (designed to check that the standard modules follow the style guide)

Note: Many of the changes discussed in the Language deck were motivated by Dyno compiler efforts

- Generally, sought to reduce the complexity both for the compiler implementation and for users

DETAILS OF PROGRESS SINCE 1.30

- Scope Resolution
- Type and Call Resolution
- Chapel Language Server
- Compiler Driver Mode
- Using the Compiler Library in Python

SCOPE RESOLUTION

SCOPE RESOLUTION: BACKGROUND

- *Scope resolution* is the process of matching identifiers with declared symbols
  - For example, in the following code, the ‘arg’ being printed refers to the ‘arg: string’ formal

```
proc printArg(arg: string) {
  writeln(arg);
}
```

- In 1.30, the Dyno scope resolver was functional but not yet enabled in production

- ‘extern’ blocks enable working with C code in a streamlined manner
  - they also interact with scope resolution, e.g., to figure out what ‘g’ refers to in the following snippet

```
extern {
  int g;
}
writeln(g);
```

SCOPE RESOLUTION: THIS EFFORT AND NEXT STEPS

This Effort:

- Enabled the Dyno scope resolver's use in production
  - caveat: still leaning on production scope resolver to handle gaps in implementation
- Fixed a few bugs uncovered by production use of the new scope resolver
- Added reasoning about extern blocks to the Dyno scope resolver
  - works with 'clang' precompiled header files
  - an interesting first case of using an external tool to support a Dyno query

Next Steps:

- Identify and fix gaps in the Dyno scope-resolver that are currently handled by the production scope resolver
- Disable the production scope resolver in favor of the Dyno scope resolver

TYPE AND CALL RESOLUTION

RESOLVING TYPES AND CALLS: BACKGROUND

- *Resolving* includes resolving types and resolving calls

```
var x = "hello";        // resolving a type: determine that 'x' has the type 'string'
f(1);                   // resolving a call: determine that 'f(1)' calls 'f' below
proc f(arg: int) { }
```

- Resolution implements a large part of Chapel's semantics
  - It is also one of the major bottlenecks in the production compiler

- A new incremental resolver is part of the Dyno effort

- Past approach: get a draft of each major component in order to:
  1. Raise language design issues before language stabilization
  2. Demonstrate integration of all resolver components in the new resolver effort

- Goal: replace production resolver and scope resolver with new Dyno resolver

RESOLVING TYPES AND CALLS: STATUS

- Currently have draft implementations for the major features required for the resolver:
  - progress since April 2023 in **bold** — many of these need more work

- generic instantiation
- implicit conversions
- tuple types
- type construction
- varargs functions
- loop index variables
- param loops
- enums
- method calls
- function disambiguation
- ‘?t’ in formals
- caching of instantiations
- compiler-generated functions

- fields
- parenless methods
- split init
- copy elision
- task/loop intents
- initializer bodies
- split init and copy elision
- operator overloads
- reductions
- task/loop intents
- const checking
- return intent overloading
- ref-if-modified for e.g. arrays

- generating calls e.g. ‘deinit’
- error types for ‘catch’
- arrays & domains
- try / throws checking
- reflection

- **arrays and domains**
- **‘new R( )’ runs ’R.init( )’**
- **‘forwarding’**
- **param folding**
- **return type inference**
- **opaque ‘extern’ types**
- **‘class’ typeclass**

CHAPEL LANGUAGE SERVER

CHAPEL LANGUAGE SERVER

Background: A language server enables editors to reason about code written in that language

- powers code completion, error reporting, 'go to declaration', as well as other features
- many implement the Microsoft Language Server Protocol

This Effort: Started development of a Chapel language server

- works with the Language Server Protocol
- demonstrates the usefulness of the compiler library

Status: An initial implementation is included in the 1.32 source release

- The 'go to declaration' feature is implemented

Next Steps: Further develop the language server

- Add tests and new features such as error reporting
- Get feedback from early adopters

COMPILER DRIVER MODE

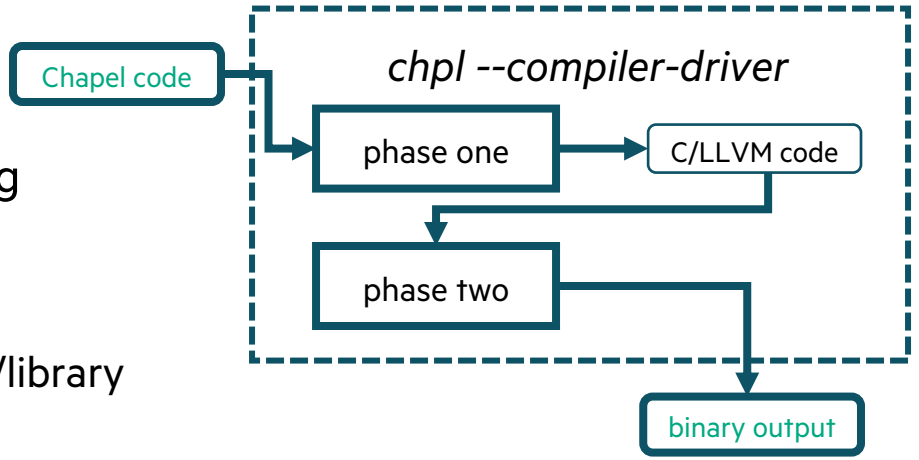
COMPILER DRIVER MODE: BACKGROUND AND THIS EFFORT

Background: The Chapel compiler mostly runs as a single process responsible for all compilation stages

- Memory allocated early in compilation (e.g., the AST) unnecessarily remains during later stages
- Previously, we began prototype work on a *compiler driver* mode
  - In this mode, 'chpl' acts as a thin wrapper running different compilation stages as subprocesses
- Potential benefits include:
  - Reduced memory pressure
  - Convenient method of running or debugging just some parts of compilation
  - Looser coupling of compiler code

This Effort: Completed prototype

- Added opt-in compiler driver mode via '--compiler-driver' flag
- Driver work is organized into two phases (diagram)
  - 'Phase one' does compilation through code generation
  - 'Phase two' does assembly and linking to generate an executable/library



COMPILER DRIVER MODE: STATUS AND NEXT STEPS

Status: Compiler driver mode is considered experimental at this point

- '--compiler-driver' is usable and passes tests with C or LLVM backend
- Works with GPU codegen, but does everything in first compilation stage
- No performance testing results yet

Next Steps:

- Get nightly testing for compiler driver mode on par with default mode
- Refactor implementation to improve code quality
- Integrate properly with GPU backend
- Gather performance data, particularly memory usage of 'chpl'
- Eventually, switch to using compiler driver mode by default, with a transitional opt-out flag

USING THE COMPILER LIBRARY IN PYTHON SCRIPTS

COMPILER LIBRARY IN PYTHON: BACKGROUND AND THIS EFFORT

Background:

- The Dyno effort exposes the front-end's functionality as a C++ library
- This is intended for building language tools (chpldoc, language server)
- However, using the C++ API comes with costs and barriers to starting a new project
  - Need to add source files to build system, configure include paths, etc.
  - Requires lower-level code (memory management, etc.)
  - C++'s library ecosystem is not as rich as Python's; there's no standard package manager

This Effort:

- Wrap the front-end library in a CPython module to expose some functionality to Python programs
  - Currently, only AST information (not scope resolution or function resolution)
- Simple tools can be made to work with very little effort
- Created two proof-of-concept tools — a linter and a code replacer

COMPILER LIBRARY IN PYTHON: IMPACT — LINTER

- Using Python, wrote a linter in ~100 lines of code
- Linter supports warnings that may be stylistic or non-universal
  - Incorrect capitalization of types and variables
  - ‘for’ loop with both ’do’ and ‘{‘
  - nested ‘coforall’ statements

- Example rule for nested ‘coforall’s:
  - For each ‘coforall’ node, searches upward to see if it has another ‘coforall’ node parent

```
def check_nested_coforall(node):
    parent = node.parent()
    while parent is not None:
        if isinstance(parent, Coforall):
            return False
        parent = parent.parent()
    return True
```

COMPILER LIBRARY IN PYTHON: IMPACT — CODE REWRITER

- Using Python, implemented a tool for syntax-aware code modification
  - Can be used to help migrate deprecated features to their new equivalents
  - Applicable in complex cases not amenable to naïve search–and-replace

- Used this tool to add ‘serializable’, ‘writeSerializable’, etc. to ~150 tests automatically
- Python code rewriting scripts are short, reproducible, and can be shared with users to aid migration

COMPILER LIBRARY IN PYTHON: STATUS AND NEXT STEPS

Status:

- Python wrapper merged into Chapel codebase
- Can be installed from source to develop new tools

Next Steps:

- Improve how the Python module is distributed
- Include the Python-based tools (linter, rewriter) in the next Chapel release
- Expose more of the Dyno library through the Python bindings to allow more powerful tools
- Consider providing code rewriter scripts in future releases to help with updates to user codes

DYNO GOALS FOR 1.33 AND 1.34

SUMMARY OF DYNO GOALS FOR 1.33 AND 1.34

Dyno development will work toward these goals:

1. Integrating New Resolver (supporting faster compilation)
   - Goal: replace the production type resolver by 1.34

2. Separate Compilation (supporting faster compilation)
   - Goal: Demonstrate saving generated code for a concrete function in the library files by 1.34
   - Goal: Demonstrate adjustments to an existing production compiler pass to support separate compilation by 1.34

3. Compiler Driver (reducing memory requirements when compiling)
   - Goal: Move compiler driver to production default instead of opt-in only by 1.33

4. Demonstrate the Compiler Library (improved interactivity)
   - Goal: Get the Language Server Protocol support ready for users by 1.34
   - Goal: Support library stabilization with Python tooling and a linter by 1.33

OTHER DYNO IMPROVEMENTS

OTHER DYNO IMPROVEMENTS

For a more complete list of Dyno changes and improvements in the 1.31 and 1.32 releases, refer to the following sections in the CHANGES.md file:

- Developer-oriented changes: 'dyno' Compiler improvements/changes

THANK YOU

https://chapel-lang.org
@ChapelLanguage