Chapel 1.33 / 2.0 Release Notes: Compiler and Tool Updates

Chapel Team
December 14, 2023 / March 21, 2024
Outline

• Separate and Incremental Compilation
  • Separate Compilation
  • Incremental Compilation
• Editor Tooling
• Debugging Chapel Programs
• Other Compiler and Tool Updates
Separate and Incremental Compilation
Separate and Incremental Compilation

- Separate Compilation
- Incremental Compilation
Separate and Incremental Compilation: Background

- We have been working towards significantly improving compile times in the Dyno effort
- We are also working towards supporting separate compilation and incremental compilation
- Both strategies may reduce the amount of time needed to compile a program
- The two strategies are related but not the same
- This presentation will discuss both and describe their status
Separate Compilation
What is Separate Compilation?

- In separate compilation, *library files* record information for use across multiple compilations
  - a *compiler* takes some unit of source code and generates a *library file*
  - a *linker* combines library files to create an executable program

- The user could initiate this process by telling the compiler to create a *library file*, e.g.,
  
  `chpl -c lib.chpl  # This could produce 'lib.chlib'

- The following slides show how library files are created with the ‘gcc’ C compiler
  - Note the ‘gcc’ commands because they demonstrate a user interface
What is Separate Compilation?

Separately Compiling in C

Library Code
lib.c

Compiler Command
gcc -c lib.c

Library File
lib.o

Program Code
prg.c

Compiler Command
gcc -c prg.c

Program Library File
prg.o

Header files are involved in this process, but are not illustrated here.
What is Separate Compilation?

Linking Separately Compiled Files in C

Library File
lib.o

Program Library File
prg.o

Linker Command
gcc lib.o prg.o -o prg

Executable
./prg
What is Separate Compilation?

Separately Compiling in Chapel can be Similar (planned, exact details TBD)

- **Library Code**
  - File: `lib.chpl`
  - Compiler Command: `chpl -c lib.chpl`
  - Result: `lib.chlib`

- **Program Code**
  - File: `prg.chpl`
  - Compiler Command: `chpl -c prg.chpl`
  - Result: `prg.chlib`
What is Separate Compilation?

Linking Separately Compiled Files in Chapel can be Similar (planned, details TBD)

```
chpl lib.chlib prg.chlib -o prg
```

Executable
./prg
Challenges to Separate Compilation for Chapel

- Chapel doesn't have an equivalent to C header files
  - The Chapel compiler *could* generate a header file from source code if necessary
  - However, library files themselves can store details needed to support separate compilation
  - Some details work like a precompiled header file (.pch) does for C compilers

- Generic functions are a challenge
  - Instantiation details for a generic function might not be known until link-time
    - So, we need the ability to instantiate at link-time

- The current whole-program compiler must be adjusted to support separate compilation
  - Adjustments are planned on a case-by-case basis for passes in the production compiler
  - Several approaches can be considered for each pass:
    - Compute the information the pass needs during link-time
    - Or compute it during separate compilation, storing it in the library file for use at link-time
    - Sometimes the pass can be rewritten so it is not whole-program
Separate Compilation for Chapel: Status and Next Steps

Status:

- Can generate a library file with a prototype flag:
  ```
  chpl --dyno-gen-lib lib.dyno lib.chpl
  ```

- Resulting 'lib.dyno' contains:
  - Serialized uAST, which provides capabilities like a precompiled header in C
  - LLVM IR for some non-generic functions

- Library files can be used when compiling a program:
  ```
  chpl lib.dyno prog.chpl
  ```

- The compiler will skip parsing 'lib.chpl' and use the uAST stored in 'lib.dyno' instead
  - Provides a modest speed improvement (around 0.1 seconds for the standard library)

- The compiler can skip code-generation for non-generic library functions
  - Saves time spent code-generating
  - However, resolution (the most expensive compilation phase) still occurs

Next Steps: Store more information in library files and identify more potentially redundant work
Incremental Compilation
What is Incremental Compilation?

- In incremental compilation, the compiler transparently reuses information to save time
  - The user doesn't need to be aware of the process

- The compiler detects source code changes and can recompile only newly changed portions
  - Recompilation can be finer-grained than e.g., 'gcc', which handles one source file at a time

- Some existing tools that leverage incremental compilation:
  - Many implementations of the Language Server Protocol (LSP)
  - The ‘ccache’ program
  - The Rust compiler

- Incremental compilation information can be stored:
  - Only in memory (typical for LSP implementations)
  - In the filesystem (‘ccache’)
The Chapel compiler launches a long-lived 'chpl' server that stores incremental compilation state:

```
chpl program.chpl # The first compiler invocation launches a compilation server and feeds it information
<edit program.chpl>
chpl program.chpl # The second invocation uses info from the server to speed up compilation
```

The Chapel language server runs continually and updates its state for live results:

- The server provides end-to-end commands such as "compile and run"
- The updates are incremental to maximize responsiveness
Incremental Compilation for Chapel: Status and Impact

- A fully incremental type and call resolver is available for a growing subset of Chapel

- Incremental resolution is up to 25x faster than initial resolution
  - With a simple 'proc trace' experiment program containing one function and calling it:
    - Initial type and call resolution = 0.5 s
    - After changing the input = 0.02 s

- The Chapel language server currently leverages incremental compilation
  - Allows for type and call resolution at interactive speeds:
    - Notice that the type of 'result' is updated in real time as the right-hand side is edited

```chapel
3 proc trace(x) {
4   writeln("The value of x is: ", x);
5   return x;
6 }
7 var result: int(64) = trace(x = 1);
```
Incremental Compilation for Chapel: Next Steps

- The *Dyno* team will complete the incremental type and call resolver

- After that, investigate end-to-end incremental compilation
  - Adjust passes that rely on whole-program info, similarly to incremental compilation
  - Gauge interest in a 'chpl' server or end-to-end compilation support via the language server
Editor Tooling
Editor Tooling: chpl-language-server and chplcheck

Background and This Effort

Background:
- the Language Server Protocol is an editor-agnostic way of adding code intelligence for programming languages
- language authors (or community) provide a server for a language, editors use the server for code intelligence

This Effort: provide two tools based on LSP
- **chpl-language-server**
  - go-to-definition, renaming, hover information
  - advanced features: type inference, param inlays, call graphs
  - more!
- **chplcheck**
  - detection of common errors that aren’t disallowed per se
  - report unconventional capitalization for records, classes, etc.
  - unused variables
  - extraneous ‘do’ blocks
  - more!
Editor Tooling: VSCode
Background and This Effort

Background:
• VSCode requires a full extension to support an LSP client

This Effort:
• Created a Chapel Language VSCode extension
  – supports chpl-language-server and chplcheck
  – improved syntax highlighting over previous community-contributed extensions
  – added other user improvements (e.g., GUI breakpoints, autofill code snippets)
• Used the extension to create a 2-click Chapel demo in the browser
Debugging Chapel Programs
Debugging Chapel Programs

Background

- Debugging Chapel programs has traditionally been fairly bare-bones
  - The main approach has been to use a typical C-style command-line debugger on the generated code / binary:
    $ chpl --savec output -g myProg.chpl
    $ ./myProg --lldb # or --gdb
    (lldb) b myProg.chpl:2
    (lldb) run
    * thread #2, stop reason = breakpoint 2.1
      frame #0: 0x100075d8c myProg.chpl__init_myProg (_ln..., _fn=...) at myProg.chpl:2
      1     for i in 1..10 {
      -> 2       writeln("i is ", i);
      3     }
    (lldb) p i
    (long) $0 = 1
  - Ease-of-use can vary by program, depending on the degree to which the code was transformed during compilation
  - Additional flags and tips are available in https://chapel-lang.org/docs/usingchapel/debugging.html
- For multi-locale runs, we have had some success running a gdb session per locale or using HPE’s ‘gdb4hpc’ tool
Debugging Chapel Programs
This Effort, Status, and Next Steps

This Effort:

- As our team has grown, so has the desire to provide better debugging experiences
  - both for ourselves and for users
- During Chapel 1.33 and 2.0, we made some improvements:
  - added a new ‘Debugger’ module providing a ‘breakpoint;’ pseudo-statement (see “Library Improvements” release notes)
  - improved portability of debugging when using the LLVM back-end on Mac OS X
  - prototyped configuration files that enable debugging Chapel within VSCode, similar to the previous slide’s example

Status: Debugging support has improved in modest ways as a result of these efforts

Next Steps: Continue improving support for debugging Chapel programs:

- Improve preservation of user identifiers and code structures during compilation
- Teach debuggers more about Chapel-specific types
- Continue improving support for debugging with IDEs and tools of interest to users and developers
- Extend debugging support to include evaluation of Chapel expressions (e.g., ‘p [a in A] sqrt(a)’)

24
Other Compiler and Tool Updates
Other Compiler and Tool Updates

For a more complete list of compiler and tool changes and improvements in the 1.33 and 2.0 releases, refer to the following sections in the CHANGES.md file:

- Improvements toCompilation Times / Generated Code
- Tool Improvements
- Compiler Improvements
- Compiler Flags
- Bug Fixes for Tools
- Developer-oriented changes: Compiler Flags
- Developer-oriented changes: Compiler improvements / changes
- Developer-oriented changes: 'dyno' Compiler improvements / changes
- Developer-oriented changes: Tool Improvements
- Developer-oriented changes: Utilities
Thank you

https://chapel-lang.org
@ChapelLanguage