CHAPEL 1.27.0/1.28.0 RELEASE NOTES: ONGOING EFFORTS—DYNO UPDATES

Chapel Team
June 30, 2022 / September 15, 2022
DYNO UPDATES
OUTLINE

• Introduction and Motivation
• Separate Compilation
• Parsing to uAST
• Scope Resolution
• Resolving Types and Calls
• Dyno-chpIdoc
• Performance
• Goals
INTRODUCTION AND MOTIVATION
COMPILER REWORK EFFORT

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

- Recent work has focused on:
  - Creating a documented compiler library suitable for use in the compiler and other tools
  - Rewriting ‘chpldoc’ to use this new compiler library
  - Replacing the early compilation passes with incremental versions
  - Adding features to the incremental resolver
PROBLEMS FACED BY THE CHAPEL COMPILER

**Speed:**
- The current compiler is generally slow, and extremely so for large programs (~7s to 15 minutes)
- Large programs require complete recompilation whenever a change is made

**Errors:**
- For incorrect programs, the compiler frequently displays only some of the errors at a time
- Compilation errors can be hard for users to understand and address

**Structure and Program Representation:**
- The compiler is structured only for whole-program analysis, preventing separate/incremental compilation
- Unclear how to integrate an interpreter, provide IDE support, or ‘eval’ Chapel snippets
- Compilation passes are highly coupled

**Development:**
- The modularity of the compiler implementation needs improvement
- There is a steep learning curve to become familiar with the compiler implementation
**COMPILER REWORK DELIVERABLES**

**Incremental Compilation Front-end**
- Only re-parse and do type resolution on files that were edited
  - Could result in reducing compilation time
- Will still have the whole-program optimization and code-generation back-end

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

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

Throughout the effort, working towards improving the learning curve and error messages.
**CURRENT STATUS**

- Using dyno parser in production since 1.27
- Using dyno ‘chpldoc’ in production as of 1.28
- Close to replacing scope resolution
SEPARATE COMPILATION
We would like to support separate compilation
  - Challenging because there are generic functions and no equivalent to C++ header files
  - Compiled libraries will store AST or source code for generic functions in case new instantiations are needed

In a separate compilation scenario, both compile and link steps need a more flexible pass structure
  - compile: need to be able to compile a library without also re-compiling all dependencies
  - link: do not want to go through entire compilation process
    - rather, link should be limited to:
      - instantiating generics as necessary
      - connecting invocations of concrete functions to their implementations

Neither of these are possible with the current rigid whole-program pass structure
  - Each pass is run in turn on the entire AST
  - Passes make whole-program assumptions and modify global variables
SEPARATE COMPILATION: NEXT STEPS

- Develop a prototype file format for a Chapel library
  - Requires serialized uAST to instantiate new calls
  - Need to discuss a suitable file extension

- Update ‘chpl’ to produce this new file format as a library file
  - Initially, produce a very simple table of symbols and uAST
  - Eventually, begin to cache information in this file, such as resolved functions and types

- Update ‘chpl’ to read this new file format
  - Initially, substitute a Chapel library file for a ‘.chpl’ file to skip the parsing stage
    - In this way, early prototypes will provide support similar to precompiled headers in C/C++
  - Then, begin exploring how to leverage cached information to skip steps of compilation
PARSING TO UAST
- Parsing is the process of reading source code and generating an abstract syntax tree (AST).

- Since 1.27, 'chpl' uses the new dyno parser that generates uAST from source code:
  - uAST (untyped AST) is more faithful to the source code than the old AST.

- A new pass in the compiler translates uAST to the old AST.

- The old parser has been removed in 1.28 to reduce maintenance.

```chapel
module Mod { 
  var x = 8;
  proc f(arg: int) {} 
  f(x); 
}
```
**SCOPE RESOLUTION: BACKGROUND**

- **Scope resolution** is the process of matching identifiers with declared symbols
  - That is, the process of recognizing that, in ‘arg = 0’ below, ‘arg’ refers to ‘ref arg: int’
    ```plaintext
    proc zero(ref arg: int) {
        arg = 0;
    }
    ```

- In 1.26, dyno scope resolution support was:
  - inextricable from type resolution
    - Ultimately the model we will pursue long-term, but not ideal for incremental progress
    - No way to use scope resolution results elsewhere, like the old compiler
  - minimal
    - Basic functionality for simple variables and use-statements
    - Other essential features missing, e.g., could not scope-resolve fields from inside methods
SCOPE RESOLUTION: THIS EFFORT

- Added support for scope resolution API queries to dyno library
  - Allows for some coarse-grained queries, like scope-resolving a function body
- Improved support for various features
  - fields, ‘include’ statements, task intents, catch statements, and more
- ‘--dyno’ flag now activates new scope resolution queries
  - A pass at the beginning of the old compiler invokes these queries when translating uAST to old AST

```
proc zero(ref arg: int) {
  arg = 0;
}
```
**SCOPE RESOLUTION: IMPACT AND NEXT STEPS**

**Impact:**
- Dyno scope resolution can now be used with ‘--dyno’
- Significantly improved the implementation and implemented missing pieces
- Can begin to run test suite with ‘--dyno’ to measure progress

**Next Steps:**
- Expand ‘--dyno’ support for test suite
  - current status: 13,626/14,020 tests pass (97%)
  - Note: ‘--dyno’ still leans on old compiler for some unhandled cases, so true progress is difficult to quantify
- Begin to disable parts of old compiler as new functionality becomes stable
- Improve support for language features
  - E.g., ‘except’/‘only’ lists, transitive properties of use/import statements
- Preserve error detection and messages
  - E.g., use-before-definition errors
RESOLVING TYPES AND CALLS
**RESOLVING TYPES AND CALLS: BACKGROUND**

- *Resolving* includes resolving types and resolving calls

- Resolving types is the process of assigning types to symbols
  
  ```
  var x = "hello";  // 'x' has the type 'string'
  var y = 1;       // 'y' has the type 'int'
  type t = x.type; // 't' refers to the type 'string'
  ```

- Resolving calls is the process of determining which function a call refers to
  
  - Instantiations are determined if the function is generic
  - If there are multiple overloads, determines which overload is called

  ```
  proc f(arg) {}      // #1
  proc f(arg: numeric) {} // #2
  proc f(arg: int) {}    // #3
  f(1);                // Calls 'f' #3
  f("hello");         // Calls 'f' #1 with 'arg' instantiated with type 'string'
  f(2.0);              // Calls 'f' #2 with 'arg' instantiated with type 'real'
  ```
RESOLVING TYPES AND CALLS: STATUS

• In 1.26, the resolver was improved to support many language features
  • Tuples, recursive types, type queries, function disambiguation, and more

• In 1.28, the resolver also supports:
  • Ranges, if-expressions, ‘enum’s
  • Vararg functions
  • Loop index variables and ‘param’ for-loops
  • Improved function return type inference
  • Better inference for generic-with-defaults records and classes
  • Rejecting invalid calls to dependently typed functions
# Resolving Types and Calls: Status

Areas of progress since April 2022 are in **bold**

<table>
<thead>
<tr>
<th>Done</th>
<th>In Progress</th>
<th>To Do</th>
</tr>
</thead>
<tbody>
<tr>
<td>• generic instantiation</td>
<td>• scope resolution</td>
<td>• initializers set types</td>
</tr>
<tr>
<td>• param folding</td>
<td>• resolve method calls</td>
<td>• check initializers</td>
</tr>
<tr>
<td>• implicit conversions</td>
<td>• function disambiguation</td>
<td>• split init</td>
</tr>
<tr>
<td>• read a file on use/import</td>
<td>• resolve ‘new’ to initializers</td>
<td>• copy init &amp; copy elision</td>
</tr>
<tr>
<td>• resolve tuple types</td>
<td>• resolve ‘?t’ in formals</td>
<td>• deinit</td>
</tr>
<tr>
<td>• type construction</td>
<td>• caching of instantiations</td>
<td>• forwarding</td>
</tr>
<tr>
<td>• <strong>varargs functions</strong></td>
<td>• default functions</td>
<td>• const checking</td>
</tr>
<tr>
<td>• <strong>resolve loop index variables</strong></td>
<td>• <strong>casts and other operators</strong></td>
<td>• try/catch</td>
</tr>
<tr>
<td>• <strong>param loops</strong></td>
<td>• <strong>arrays &amp; domains</strong></td>
<td>• task/loop intents</td>
</tr>
<tr>
<td>• <strong>resolve enums</strong></td>
<td>• <strong>resolve fields</strong></td>
<td>• reflection</td>
</tr>
<tr>
<td></td>
<td>• <strong>resolve parenless methods</strong></td>
<td>• reductions</td>
</tr>
</tbody>
</table>
RESOLVING TYPES AND CALLS: IMPACT

- Ranges can now be resolved
- Some of the recent work was motivated by resolving ranges
- Ranges use some tricky language features as shown below

```chapel
record range
{
    type idxType = int; // element type
    param boundedType: BoundedRangeType = BoundedRangeType.bounded; // bounded or not
    // ...
    var _low : chpl__idxTypeToIntIdxType(idxType); // lower bound
    // ...
}

proc range.init(type idxType, low: idxType, high: idxType)

initializer is a dependently-typed function

'range' is generic with defaults

'range' uses param enums
```
RESOLVING TYPES AND CALLS: NEXT STEPS

- Complete the implementation of large language features missing from the resolver
  - Arrays and domains
  - Initialization and split initialization
  - Copy initialization and copy elision
  - Reflection
As of 1.28, ‘chpldoc’ has been replaced by a dyno-based implementation

‘chpldoc’ now demonstrates a tool using the dyno library
  - Previous versions of ‘chpldoc’ were a pass within the compiler

Community development of linters or code formatting tools is now possible
  - ‘chpldoc’ can serve as an example

Provides better formatting for syntax that is rendered in the documentation
  - Including range expressions, array expressions, nilable class types, and numeric literals
  - A result of the uAST more closely representing the source code than the legacy compiler’s AST

See the ‘chpldoc’ section in the Compiler, Performance, and Tools deck for more information
PERFORMANCE
**Background:** Efforts are underway to study and improve dyno performance
- Wall-clock performance (how long does the resolver run?)
- Query performance (how many queries are needed to first resolve something? re-resolve?)

**This Effort:** Substitute specialized data structures in key places
- The LLVM project provides such data structures (e.g., ‘SmallPtrSet’) – Can perform better when used appropriately

**Impact:** Use of LLVM data structures led to performance improvements
- Exact magnitude is unclear while dyno scope resolving is still under development
- Measured a 17% performance improvement with partial scope resolution

**Next Steps:** Continue tracking resolver performance
- Investigate overhead of query framework itself
- Identify functions to turn into queries
NEXT STEPS
These are the planned ‘dyno’ goals for the 1.29 and 1.30 releases:

1. Work towards replacing the scope resolver in the production compiler
   – Goal: use the ‘dyno’ scope resolver in production by 1.29 or 1.30

2. Build more components of the dyno resolver to reach feature-completeness
   – Goal: have initial implementations of remaining major components of the new resolver by 1.30

3. Design and implement file formats and commands for separate compilation
   – Goal: implement uAST serialization/deserialization by 1.29
   – Goal: implement draft library file format and demonstrate separate compilation commands by 1.30

4. Begin work on improving error messages
   – Goal: migrate errors from parsing and dyno scope resolution to a more user-friendly format by 1.29
OTHER DYNO IMPROVEMENTS
OTHER DYNO IMPROVEMENTS

For a more complete list of dyno changes and improvements in the 1.27.0 and 1.28.0 releases, refer to the following section in the CHANGES.md file:

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

https://chapel-lang.org
@ChapelLanguage