Try, Not Halt: An Error Handling Strategy for Chapel

Preston Sahabu* Michael Ferguson Greg Titus Kyle Brady

Cray Inc., Seattle WA
`chapel_info@cray.com`

Previous to this effort, Chapel lacked a formal strategy for handling errors. Over time, the Chapel standard library settled on two ad-hoc strategies:

- Call `halt()` and stop the entire program.
- Create optional `out` arguments for the caller to handle.

Chapel has outgrown these approaches as it has matured. Library designers want to avoid halting the execution of user programs while enforcing the library's requirements, and neither strategy provides that capability. For more general users, few errors are so catastrophic that they require a halt, but the second strategy has several problems as well. It is difficult to track errors through the call stack without formal enforcement and the existing error codes in the `syserr` module cannot be extended with new codes or types of errors. Motivated by these shortcomings, this effort explored, designed, and is currently implementing a new error handling strategy for Chapel at the language level.

The initial design was modeled closely after that of the Swift language because it had an appealing compromise between error codes and exceptions. While error codes are less expressive than exceptions, traditional exceptions make it very difficult to reason about control flow. Swift's design aims for the best of both worlds by allowing several constructs to be used as an error (`struct`, `object`, `enum`, etc.) while eliminating any control flow ambiguity with a strict syntax. The initial Chapel error handling strategy followed suit:

- All functions that can throw an error are marked `throws`.
- When a function should throw an error, use `throw`.
- Errors are caught and handled by `do`/`catch` blocks.
- Calls to functions that throw must be decorated by `try` or `try!`
  - `try` forwards any errors to the enclosing `do`/`catch` block
  - `try!` halts the program on an error

The most controversial aspect of this strategy was its insistence on marking every throwing call with `try` and enclosing it with a `do` block. Ideally this allows users to reason about control flow using only local information, addressing the problem raised by traditional exceptions. However several members of the community saw this as overly verbose and most of the core team agreed, so the design had to be adjusted.

In this talk we will investigate the proposals to renovate the Swift model for Chapel use, weighing the benefits and drawbacks of each. We will then introduce the Chapel error handling strategy, which drew on the strengths of each proposal. The current design strips away the verbosity of the previous approach while maintaining the ability to easily reason about control flow.

The talk will conclude with a preview of future error handling features.

* presenting author