Try, Not Halt:
An Error Handling Strategy for Chapel

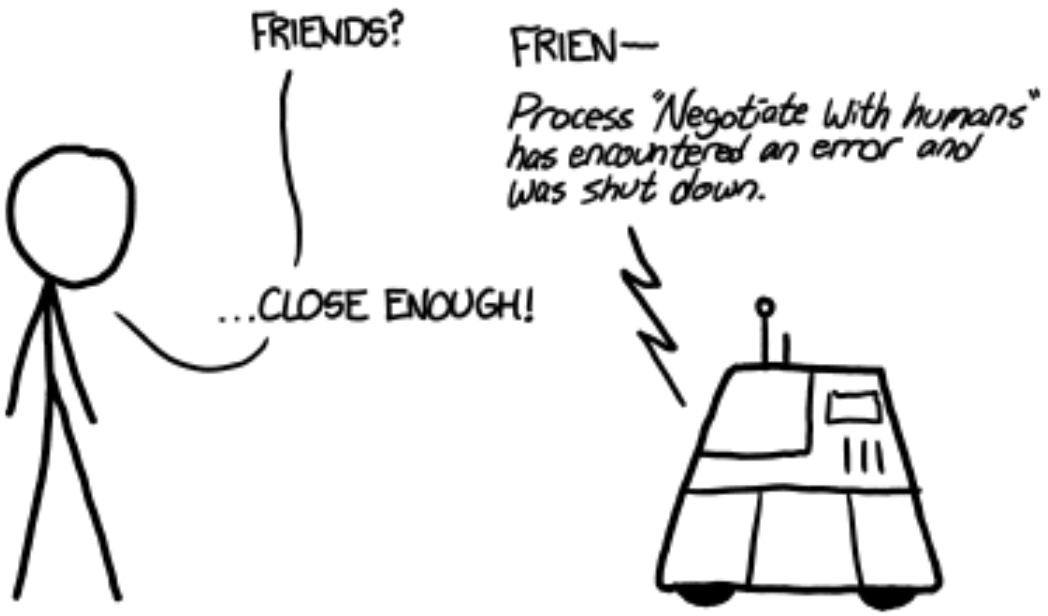
Preston Sahabu*, Michael Ferguson, Greg Titus, Kyle Brady
Chapel Team, Cray Inc.
June 2, 2017

Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

Motivation

- **Chapel lacked a general strategy for handling errors**
  - 'halt()' and "error" out arguments are used in practice, but insufficient



credit xkcd

- **First cut at language-level design was modeled after Swift**

Swift

- **All calls that might throw must be marked with 'try'**
  - Makes control flow clear with only local information

```
func canThrowErrors() throws { ... }

do {
  try canThrowErrors();
  try! canThrowErrors(); // will halt on failure
} catch {
  writeln("first call failed!");
}
```

- **Community feedback was that this is verbose**
  - (read: annoying)
- **How can this be streamlined?**

Swifter (Better, Faster, Stronger)

CHIP #8 investigated a few different options:

1. **Make 'try' optional.**
   - Inside 'do' blocks, users should already be aware of throwing calls
   - Outside 'do' blocks, calling fn signature can determine 'throw' or halt
   - Downside: errors silently passing through functions
2. **Make 'try' optional, with a compiler flag.**
   - Hardens code like Swift with the flag, otherwise the same as (1)
3. **Eliminate 'do' from the syntax.**
   - 'do' is already a keyword in Chapel
   - Replace it with a 'try' defined on compound statements

Current design and implementation

- **Combination of (2) and (3) was chosen for the design**
  - Full detail in CHIP #8
- **Chapel 1.15 contains a draft implementation**
- **Offers basic functionality, but with significant limitations**
  - Handling cannot yet span ‘begin’ / ‘cobegin’ / ‘coforall’ / ‘forall’ / ‘on’…
  - Virtual methods cannot yet throw
  - Halting errors do not yet print their type or message
  - Errors cannot yet be generic classes
- **Seeking feedback on design and implementation**
  - 'try' it out today!

Errors as classes

- **Base class 'Error' is provided**
  - For now, the initializer accepts a string argument

```
class Error {
  var msg: string;
}
```

- **'Error' may be used directly, or as the root of a hierarchy**
  - Standard set of 'Error' subclasses not currently included

```
class MyError: Error {}

class MyIntError: Error {
  var i: int;
}
```

Throwing errors

- **Throw an error with 'throw'**

```
// throwing a newly created error
throw new Error("error message here");

// throwing an error stored in a variable
var e = new Error("test error");
throw e;
```

- **Mark procedures that can throw with 'throws'**

```
proc mayThrowErrors() throws { ... }

proc mayThrowErrorsAlso(): A throws where { ... }

proc mayNotThrowErrors() { ... }
```

try/catch

- **'try' and 'try!' are used to handle thrown errors**
  - { } blocks try to match to an associated 'catch' clause
  - Single statements will not match any 'catch' clauses

```
try {
  mayThrowErrors();
  mayNotThrowErrors();       // non-throwing calls may be included
  mayThrowErrorsAlso();
}
try! mayThrowErrors();       // halts on error
```

- **If an error is handled with no matching 'catch' clause:**
  - 'try' propagates the error
    - To an outer 'try', or out of the procedure (which must be marked 'throws')
  - 'try!' halts instead of propagating
  - Single statement form relies on this behavior

try/catch

- **'catch' clause list matches against an 'Error' at run-time**
  - If a type filter matches the error, that block will be executed
  - Lack of a type filter means that all errors match

```
try {
  trickyOperation(badArg);
} catch err: IllegalArgumentError {  // IllegalArgumentError, subtypes
  writeln("illegal argument!");
} catch err: MyError {               // MyError, subtypes
  throw err;
} catch {                            // catch-all
  writeln("unknown error!");
}
```

Default and Strict mode

- **Two modes to support the tradeoff between…**
  …ensuring propagation of errors is clear (strict)
  …drafting code quickly (default)
- **Strict mode enforces visible control flow**
  - All calls to throwing procedures must be enclosed within 'try' / 'try!'
  - Otherwise, an error will be raised at compile-time
- **Default mode supports rapid prototyping**
  - Throwing calls need not be enclosed in 'try' / 'try!'
  - If the enclosing procedure is marked 'throws', propagate errors
  - Otherwise, halt on errors
- **Strict mode enabled with a compiler flag, --strict-errors**
  - Otherwise, compiler uses default mode
  - Expect to support more fine-grained approaches in the future
    - e.g., specify strictness per-module (or even per-function?)

Next steps

- **Seek feedback on design and implementation**
- **Address the known limitations**
  - Especially with regards to parallelism and multilocale
- **New features**
  - Create a standard set of 'Error' classes
  - Enable throwing errors from iterators
  - Implement a 'defer' construct for state cleanup
  - Design and implement a fine-grained strict mode
- **Integrate error handling into the standard library**
- **Handle runtime errors by throwing Chapel errors**

Thank you! Questions?

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

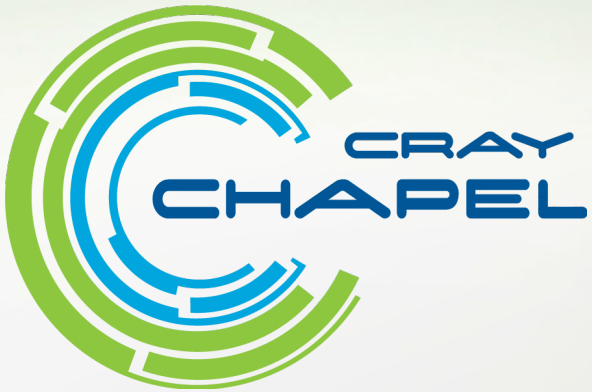
All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.



CRAY

THE SUPERCOMPUTER COMPANY