Ongoing Efforts

Chapel Team, Cray Inc.

Chapel version 1.12

October 1st, 2015

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.

Context for this Slide Deck

- **Typically, our release notes focus on the release contents**
- **This release cycle saw a number of other important efforts**
  - Some of these were implementation efforts that weren't done in time
  - Others were more forward-looking strategic or design efforts
  - Both categories seemed worth reporting on in spite of being ongoing

Outline

- **Processing Twitter @mentions Graphs in Chapel**
  - Sidebar on I/O for Twitter Processing in Chapel
- **Error Handling Approach**
- **New String Implementation**
- **Fixing Record Semantics**
- **Constructor/Destructor Refinement**
- **Interactive Programming Environment (IPE) Update**

  a.k.a. Front-End / Internal Representation Refactoring (v2)

Processing Twitter @mentions Graphs in Chapel

Processing Tweets: Motivation

Motivating Question: Is Chapel useful for Data Analytics?

- What would it look like?
- What features are we missing?

Processing Tweets: Background

Twitter: an online social networking service that enables users to send and read short 140-character messages called "tweets" --Wikipedia

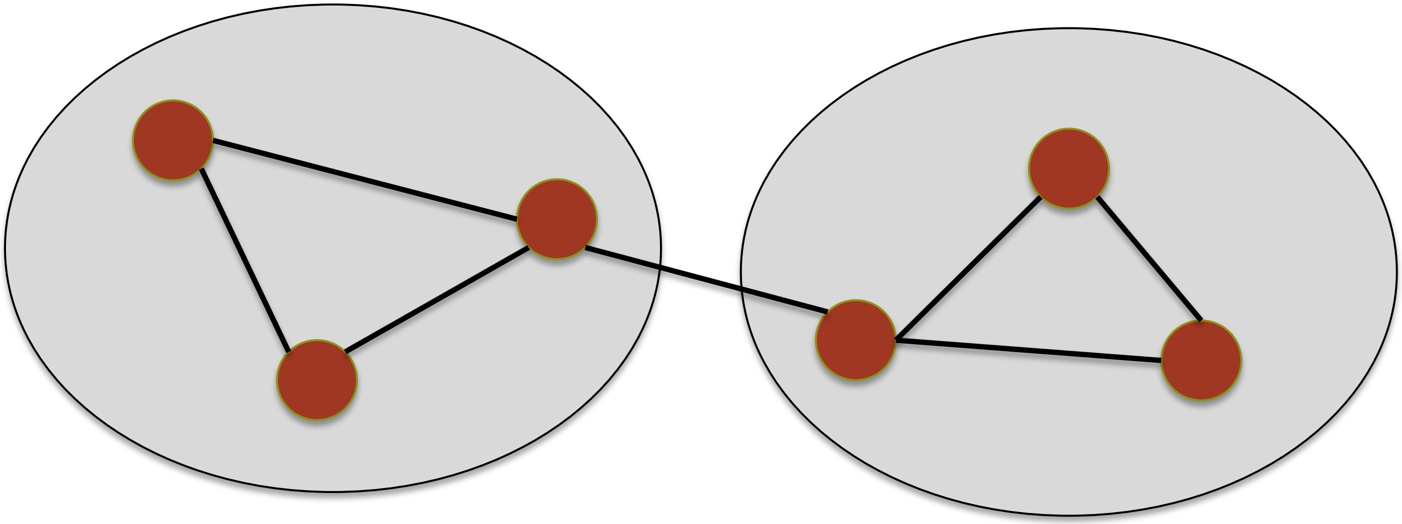
- tweets support referencing other users via @*username*

Benchmark: Label Propagation for Community Detection

- can be considered to capture a data analytics workflow
- see CUG'15 paper: *Implementing a social-network analytics pipeline using Spark on Urika XA*
- a few implementations of this benchmark exist
  - e.g., Spark

Processing Tweets: Computation Steps

- **Computation consists of these steps:**
  - Read in gzip files storing JSON-encoded tweets
  - Find pairs of Twitter users that @mention each other
  - Construct a graph from such users
  - Run a label propagation algorithm on that graph
  - Output the community structure resulting from label propagation



Community 1 Community 2

Processing Tweets: Label Propagation

Label Propagation Algorithm

(described in [*Near linear time algorithm to detect community structures in large-scale networks*])

1. Initialize the labels at all nodes in the network.
2. Set $i = 1$.
3. Arrange the nodes in the network in a random order and set it to X.
4. For each x in X, set node x's label to the one that occurs most frequently among neighbors, with ties broken uniformly randomly.
5. If every node has a label that the maximum number of neighbors have, stop the algorithm. Otherwise, set $i = i + 1$ and go to step 3.

Processing Tweets: Implementation Overview

- **< 400 lines of Chapel code**
  - plus a Graph module (< 300 lines, to become a standard module)
  - plus some improvements to existing Chapel modules
- **current version is single-locale**
  - ultimately, need to support multi-locale in order to run larger data sets
- **graph representation similar to other Chapel graph codes**
  - e.g., SSCA#2

Processing Tweets: I/O

- Reading the tweets to build the graph is ~1/2 of the code
- Command line input lists files and directories to process
- findfiles() iterator used to enumerate files in a directory
- Reads file using `gunzip` via the new Spawn module
- Uses new functionality for JSON I/O
  - concept: use types and I/O that ignore irrelevant fields
    - (details in a sidebar following this section)

Processing Tweets: Algorithm in Chapel

Algorithm closely matches the psuedocode:

```
var i = 0;
var go: atomic bool;
go.write(true);
while go.read(…) && i < maxiter {
  go.write(false);
  // for each x in the randomized order
  forall vid in reordered_vertices {
    // set the label to the most frequent among neigbors
    mylabel = labels[vid].read(memory_order_relaxed);
    maxlabel = mostCommonLabelInNeighbors(vid);
    if countNeighborsWith(vid, mylabel) <
       countNeighborsWith(vid, maxlabel) then
       go.write(true); // stop the algorithm if …
    labels[vid].write(maxlabel, memory_order_relaxed);
  }
  i += 1;
}
```

Processing Tweets: Caveats

The next few slides compare our Chapel version against a Spark version

Important Notes:

- Spark includes resiliency features while Chapel currently does not
- neither implementation is necessarily optimal

Processing Tweets: Productivity Comparison

Spark

- **RDDs are immutable**
  - create new RDD every iteration through algorithm
- **Algorithm written in terms of mapping a fn on data**
  - difficult to visit vertices in random order
  - movement of information is described as messages contributing to a new RDD
  - breaking ties randomly might require a custom operator

Chapel

- **Chapel arrays are mutable**
  - Algorithm can update labels in-place
- **Algorithm written in terms of parallel loops**
  - straightforward to visit vertices in random order
  - movement of information occurs through variable reads and writes
  - breaking ties randomly is an easy change

These differences reflect Spark's declarative nature vs. Chapel's imperative design.

Processing Tweets: Performance Comparison

- **We performed an initial performance comparison between our Chapel version and the Spark version**
  - preliminary results are promising
- **However, there are several caveats:**
  - the results are completely apples-to-oranges:
    - different architectures
    - different system scales
    - different data set sizes

    (reflects Chapel code being single-locale only, early stages of study)
  - a multi-locale Chapel version will likely perform very differently
    - multi-locale execution will be necessary for larger dataset scales
- **For these reasons, we've decided not to release results until we can perform a more rigorous study**
  - specifically, multi-locale Chapel, same data set, same architecture

Processing Tweets: Impact, Status, Next Steps

Impact:

- A positive early indication of Chapel's applicability to data analytics

Status:

- Have a prototype data analytics benchmark
  - reliant on pending modifications to Chapel library
- Productivity and performance are promising

Next Steps:

- Commit library modifications to master
- Create a multi-locale version
  - primary effort: multi-locale graph data structures / domain maps
- Compare performance with other implementations, scientifically
- Describe this study in a paper to disseminate the results, get feedback

Sidebar on I/O for Twitter Processing in Chapel

Example Tweet in JSON format

- **Tweets have 34 top-level fields**
  - including nested structures containing much more data

{ "coordinates": null, "created_at": "Fri Oct 16 16:00:00 +0000 2015", "favorited": false, "truncated": false, "id_str": "28031452151", "entities": { "urls": [{ "expanded_url": null, "url": "http://chapel.cray.com", "indices": [69, 100] }], "hashtags": [ ], "user_mentions": [{ "name": "Cray Inc.", "id_str": "23424245", "id": 23424245, "indices": [25, 30], "screen_name": "cray" }] }, "in_reply_to_user_id_str": null, "text": "Let's mention the user @cray -- here is an embedded url .... http://chapel.cray.com", "contributors": null, "id": 28039652140, "retweet_count": null, "in_reply_to_status_id_str": null, "geo": null, "retweeted": false, "in_reply_to_user_id": null, "user": { "profile_sidebar_border_color": "C0DEED", "name": "Cray Inc.", "profile_sidebar_fill_color": "DDEEF6", "profile_background_tile": false, "profile_image_url": "http://a3.twimg.com/profile_images/2342452/icon_normal.png", "location": "Seattle, WA", "created_at": "Fri Oct 10 23:10:00 +0000 2008", "id_str": "23502385", "follow_request_sent": false, "profile_link_color": "0084B4", "favourites_count": 1, "url": "http://cray.com", "contributors_enabled": false, "utc_offset": -25200, "id": 23548250, "profile_use_background_image": true, "listed_count": 23, "protected": false, "lang": "en", "profile_text_color": "333333", "followers_count": 1000, "time_zone": "Mountain Time (US & Canada)", "verified": false, "geo_enabled": true, "profile_background_color": "C0DEED", "notifications": false, "description": "Cray Inc", "friends_count": 71, "profile_background_image_url": "http://s.twimg.com/a/2349257201/images/themes/theme1/bg.png", "statuses_count": 302, "screen_name": "gnip", "following": false, "show_all_inline_media": false }, "in_reply_to_screen_name": null, "source": "web", "place": null, "in_reply_to_status_id": null }

Reading JSON Tweets

```
// define Chapel records whose fields reflect only
// the portions of the JSON data we care about

record TweetUser {
  var id: int;
}
record TweetEntities {
  var user_mentions: list(TweetUser);
}
record User {
  var id: int;
}
record Tweet {
  var id: int,
      user: User,
      entities: TweetEntities;
}
```

```
proc process_json(…) {
  var tweet: Tweet;

  while true {
    // "%~jt" format string:
    //     j: JSON format
    //     t: any record
    //     ~: skip other fields
    got = logfile.readf("%~jt",
                        tweet,
                        error=err);
    if got && !err then
      handle_tweet(tweet);
    if err == EFORMAT then ...;
    if err == EEOF then break;
}
```

Open Issue: How to Read Arrays from JSON

Current approach:

```
record TweetEntities {
  var user_mentions: list(TweetUser);
}
```

Desired approach:

```
record TweetEntities {
  var user_mentions: [1..0] TweetUser;
  // not possible to know array's length in advance; determined by file contents
  // would like a way to read this record that resizes the array appropriately…
}
```

Should Reading an Array Resize it?

- **Should we resize arrays on reads?**
  - File formats like JSON's variable-length arrays encode their sizes

Pros:
- makes reading arrays trivial for such file formats
- having the default record I/O function do this would simplify user burden

Cons:
- array resize on reads may be confusing / inconsistent
- traditionally, Chapel cannot resize arrays with shared domains
  - suggests only supporting this feature for arrays with unique domains
- not all file formats support self-descriptive array sizes

Challenges:
- how to support benefits without causing undue surprise?
- under what conditions should array reads resize vs. not?
- how to minimize user burden?

Error Handling Approach

Error Handling: Background

- **Chapel currently lacks a general strategy for errors**
- **Standard library uses two primary approaches at present:**
  - calls to `halt()`
  - optional output arguments (`out error: syserr`)
    - if argument provided, assumed user will handle; else call halt()
- **Each of these approaches has serious drawbacks:**
  - halting the program is not appropriate in library code
  - current output argument approach…
    …only returns error codes
    …doesn't permit users to easily add new error codes or state
- **A more general strategy is desired, supporting**
  - the ability to write bulletproof code
    - ideally, in a way that supports propagation of errors, like exceptions
  - the ability to get useful messages when errors are not handled

Error Handling: This Effort

- **Designing a new approach for error handling**
- **We considered:**
  - using generalized error objects instead of error codes
  - returning tuples encoding (result, error)
  - returning error objects via optional out arguments
  - exceptions along the lines of C++
  - an exception-like approach (inspired by Swift)
    - our current leading candidate

Error Handling: Swift Error Handling Model

- **Functions that can raise an error are declared with `throws`**

```
func canThrowErrors() throws { … }
func cannotThrowErrors() { … }
```

- **Calls that throw must be decorated with `try` or `try!`**
  - makes the control flow possibilities clear without inspecting the callee
  - `try` propagates the error to an enclosing `do/catch` block or out of a throwing function
  - `try!` halts if an error occurred

- **Programs can respond to errors with `do/catch` statements**

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

Error Handling: Expected Additions for Chapel

- **Throwing errors from iterators**
- **Ability to catch errors generated in runtime layers:**
  - Communication
  - Memory allocation
  - Task creation
  - Not mandatory to check for these
  - Try blocks are one option here

```
try {
  var A: [1..n] int;                  // array allocation could fail
  begin ...;                          // task launch could fail
  remoteB[i] = A[i] + A[i+1];         // communication could fail
}
```

- **Task joins propagate errors to parent tasks**
  - Occurs at end the of `sync`/`coforall`/`cobegin` blocks

Error Handling: Advantages of This Model

- **Represents a middle ground**
  - arguably acceptable to devotees of both exceptions and error codes
- **Easier to implement than stack-unwinding**
  - re-uses the existing return mechanisms
- **Fits well with existing task parallelism**

Error Handling: Next Steps

- **Create a detailed proposal**
  - Description of a Swift-like model in Chapel
  - Solicit feedback from the community
- **Investigate expected additions to the Swift model**
- **Start implementing**
  - Some parts will require larger changes
    - Handling errors from the runtime
    - Task joins propagating errors to the parent task

New String Implementation

Strings: Setting

Background:

- We have been working on re-implementing strings as a Chapel record
  - To remove special-case string-specific code from the compiler
  - To plug existing memory leaks for strings
  - To serve as a proxy for other interesting value types users might write
- Early drafts indicated problems with our implementation of records
  - Given how broadly strings are used, they serve as an acid test for records
  - Currently working on addressing these on the string-as-rec branch

This Effort:

- Define a record-based `string` type
- Convert string literals to type `string` (were previously `c_string`)
- Support a modern set of library routines
- Add support for unicode strings

Strings: Define a record-based `string` type

- **String record currently looks like:**

```
record string {
  var len: int = 0;
  var size: int = 0;
  var buff: c_ptr(uint(8))= nil;
  var owned: bool = true;
}
```

- **Implemented in the modules, used in the compiler**
  - Compiler hooks onto this type early in compilation
  - Alternative implementations could be swapped in easily
- **Lets us remove many special cases in the compiler**
  - `string` is handled (almost) like any other record

Strings: Convert string literals to type `string`

- **String literals have been implemented as type `c_string`**
  - Implicit coercions to **`string`** needed to be inserted in many cases
    - Caused a new **`string`** to be created over and over for the same literal

      `var x = "Hello, World"; // implicit coercion from c_string to string`
  - **`c_string`** provides **`param`** functionality
    - **`param`** records are a long ways off…

Strings: Convert string literals to type `string`

- **Added support for `param string`**
  - Specific to `string`, not generalized for all records
  - Supports the same operations as `c_string`
- **Made string literals type `string`**
  - `string` literals are constructed at the beginning of time
    - Locale private globals
  - Implicit coercions go away
  - `c_string` literals can be written using different syntax:

```
var x = c"Hello, World";
```

Strings: Add new library routines

```
this (substring)
startsWith
endsWith
find
count
rfind
replace
join
strip
partition
isUpper
isLower
isSpace
isAlpha
isDigit
isAlnum
isPrintable
isTitle
toLower
toUpper
toTitle
capitalize
+, *, ==, !=, <=, ...
```

- These all work for single-locale
- Multi-locale support forthcoming
  - Blocked by general record issues

Strings: Next steps

- **Get all of these changes onto master**
  - Requires the record fixes to be completed
- **Add a proper Unicode `string` type**
  - Should be used for any operations on text
- **Rename the current record to `bytes` (or …?)**
  - A sequence of `uint(8)`s with `string`-like operations
  - `bytes` is not intended for working with general text
  - Rather, for use in places like:
    - Networking
    - Filesystem operations

Fixing Record Semantics

Strings and Records: Background

The Problem: Record implementation found to have gaps

- Records of plain-old data work reasonably well…
- More interesting record types have significant problems
  - Constructor/destructor/assign not called in all cases that it should be
- Possible outcomes:
  - Uninitialized state
  - Memory leaks
  - Errors in multi-locale programs

The Approach: Extended memory management algorithms

- Flow-control algorithms to track ownership of objects
  - Ensure constructor/destructor/assignment invoked correctly
- Consistent handling for records and ref-counted types
  - e.g. domain maps, domains, arrays
- Had hoped to complete this work by 1.12 but failed to do so
- Work being pursued on string-as-rec branch

Strings and Records: This Effort and Status

Status:

- Regression count:
  - of 5500 existing tests, 22 failures for single-locale, 64 for multi-locale
- Developed 132 new stress tests for records
  - Single-locale: 9 fail on string-as-rec, while 30 fail on master
- Performance challenges:
  - 15% of performance tests more than 10% slower on string-as-rec
  - A few tests more than 100% slower
  - Believe these to be symptomatic of the current implementation
    - not of inherent overhead in doing Chapel's records correctly
- Significant problems for arrays
  - Additional calls to increment ref-counts for arrays $\Rightarrow$ a performance issue
  - Failure to decrement ref-counts $\Rightarrow$ a memory leak issue
  - Root cause of these problems likely to be linked to performance issues
  - Arrays have been special-cased in ways that wriggle around record issues
    - string-as-rec cleanup entangled with these special cases

Strings and Records: Next Steps

- **Tolerate inconsistencies between arrays and records**
  - but long-term goal is to unify the underlying implementation
- **Close the gap between string-as-rec and master**
  - string-as-rec has accumulated a lot of changes
    - some changes are independent of records/arrays
  - highlight material differences that contribute to…
    …improved behavior for records and strings
    …degraded behavior for ref-counted objects e.g. arrays
    …some differences may do both at the same time
  - isolate safe changes that enhance strings and records
    - Migrate them to master
- **Identify remaining errors on each branch**
  - string-as-rec and master have differing errors
  - focus on removing errors from master

Constructor/Destructor Refinement

Constructors/Destructors: Background

Background:

- Chapel's OOP features have traditionally been naïve in terms of…
  …constructors and destructors
  …initialization vs. assignment
  …user-defined default values, parallel initialization, …
- Need to get this right for correct resource management
  - memory, file descriptors, …

Constructors/Destructors: This Effort

Goals:

- improve design of constructors, destructors, assignment operators
- clarify when they are invoked

Approach:

- design features within a sub-team; then review with team & community
- using C++ and D as primary reference points

Topics:

- syntax and semantics of constructors for records, classes
- integration with `noinit` and default values
- postblit-based copy construction, inspired by D language
- serialize-/deserialize-based copying into tasks and/or `on`-statements
- semantics of returning records and arrays
- avoiding copies in argument passing and returning
- specifying when the compiler can make bit-wise copies of records

Constructors/Destructors: Work In Progress

Status:

- sub-team reaching consensus
- work-in-progress described by draft CHIPs:
  - CHIP 4: Constructor Syntax and Semantics
  - CHIP 5: Implement Object Copying Using a “Postblit” Method

Next Steps:

- finish proposed design
- get consensus on approach from broader team
- complete implementation

Interactive Programming Environment (IPE) Update

a.k.a. Front-End / Internal Representation Refactoring (v2)

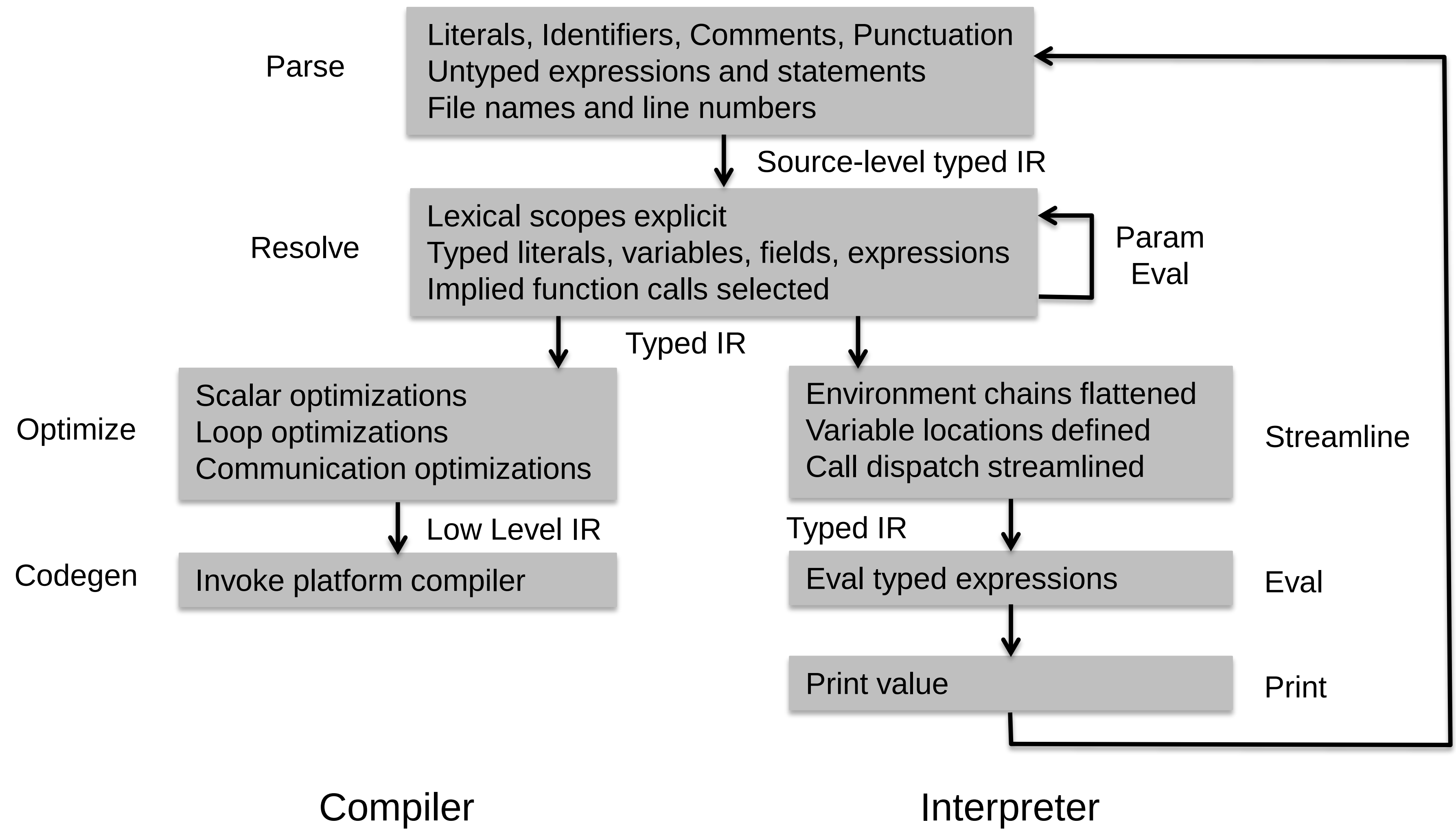
IPE/v2: Background

- **IPE integration hampered by current compiler arch. / IR**
  - Many passes implemented as program-wide transformations
    - Inconsistent with statement-by-statement interpretation
  - IR is lowered aggressively and early $\Rightarrow$ loss of high-level information
  - IR is not fully typed until deep in the pipeline
  - IR carries baggage from sprint of HPCS-era development
    - Nodes overloaded in meaning, esp. CallExpr, BlockStmt, FnSymbol
    - Casual mutation of public member variables
    - Fragile assumptions embedded into code
    - Difficult to reason about state of IR across passes
- **Meanwhile, other compiler efforts suffer from same**
  - Simplicity of adding new features e.g. “Concepts”
  - Ability to write new optimizations/transformations
    - e.g., array optimizations want high-level information lost in lowering
  - Desire for better error messages and diagnostics
  - Module documentation

IPE/v2: This Effort

- **Integrated support for both IPE and compiler**
  - Clean interface between front-end and back-ends
  - Unification of param resolution and interpreter
- **Deliver a high-level, fully-typed IR**
  - Represent Chapel's source-level semantics more naturally
  - Consistent semantic checking
  - Include clear references to source text
- **Overhaul type / function resolution**
  - Statement-oriented rather than whole-program
    - Respect relaxed ordering of declarations w.r.t. their uses
    - Respect potential for mutual cross-module dependencies
  - Be lazy where practical
    - Prefer to avoid expanding unreachable types/functions/generics

IPE/v2: A Shared Future for compiler and IPE



IPE/v2: Status

- **Initially an effort to refactor resolution**
  - Perform resolution immediately after parsing
    - Avoid relying on normalization
  - Extended early work for IPE
  - Limited progress
- **Expanded effort to include refactoring of parser**
  - Premature lowering of IR is an undesirable complication
  - Began to extend the set of IR nodes
  - Revised lexer to be “pure” and capture additional tokens
  - Modified scanner to begin to generate new IR nodes
- **Expanded investigation to contemplate a possible “v2”**
  - Should a new front-end be a step to a new back-end?
  - How to balance incremental integration vs. clean-sheet?

IPE/v2: Next Steps

- **Develop an end-to-end prototype for compilation**
  - Support same subset of Chapel as for IPE
  - No optimizations
  - Define extended integration path
- **Confirm ability to support IPE for same language subset**
- **Expand to support**
  - records and classes
  - iterators
  - generics
  - multi-tasking
  - multi-locale programs
- **Main Challenge: How much effort to place here how soon?**
  - For long-term health of Chapel, v2 seems essential
  - For short-term adoption, need to continue making obvious progress

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, URIKA, and YARCDATA. 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.

http://chapel.cray.com chapel_info@cray.com http://sourceforge.net/projects/chapel/