Language Improvements

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.

Outline

- [Lexical Scoping Changes]
- [New Interpretation of Formal Array Arguments]
- [Type Methods and Iterators]
- [Public/Private Module-level Symbols]
- [Memory Consistency Model]
- [Other Language Changes]

Lexical Scoping Changes

Lexical Scoping: Background

- **Variables used to be kept alive past their lexical scopes:**

```
{
  var A: [1..n] real;
  var count$: sync int;
  var x: real;
  begin with(ref x) {  …A…   …count$…   …x…   }
  // ^^^ this task and its references to A, count$, and x could outlive their scope
}  // So, traditionally, Chapel has kept these variables alive past their logical scope
```

- **Consequences of this approach:**
  - moves logical stack variables to the heap (like `x` and `count$` above)
    - incurs overhead for each access in original scope
  - incurs reference counting overhead
    - (or memory leaks in cases where we hadn't yet added reference counting)
  - complicates the implementation
  - not particularly valued or leveraged by users
  - arguably surprising ("x still exists even though it left scope?")

Lexical Scoping: This Effort

- **Simplify language definition and implementation**
  - live references to variables leaving scope are now user errors:

```
var flag$: sync bool;     // flag$ starts empty
{
  var x: real;
  begin with(ref x) {     // create task referring to x
    flag$;                // block task until flag$ is full
    …x…                   // user error: access to x occurs after it leaves scope
  }                       // end of task
}                   // x's scope closes
flag$ = true;       // fill flag$ only after x's scope closes
```

Lexical Scoping: Impact

Impact:

- Can now store logical stack variables on the stack
- Can improve and simplify the implementation
  - close some longstanding memory leaks
  - remove unnecessary reference counting
- Now freeing sync/single variables for the first time
  - This has been a longstanding source of memory leaks in the compiler
    - Previously, took very conservative “some task may be referring to it” approach
    - And never bothered to reference count…

Lexical Scoping: Status and Next Steps

Status:

- Other opportunities for leveraging new semantics still exist
- Program may crash if tasks refer to variables after they leave scope

Next Steps:

- Take further advantage of new semantics
  - stop heap-allocating variables in unnecessary cases
  - stop (or reduce) reference counting of arrays, domains, domain maps
  - close other leaks related to old semantics
- Implement runtime safety checks to guard against user errors
  - disable these checks for performance runs

New Interpretation of Formal Array Arguments

Array Formals: Background

- **Domains in formal array types used to perform reindexing**
  - rationale: permits functions to use most natural indexing scheme

```
var A: [1..n, 1..n] real;                   // declare n x n array

factor(A[lo..#b, lo..#b]);                  // pass b x b slice of A into routine

proc factor(X: [1..b, 1..b] real) {         // reindex actual as {1..b, 1..b}
  …X[1,b]…                                  // X[1,b] refers to A[lo, lo+b-1]
}
```

  - inspired by Fortran

Array Formals: Background

- **In practice, this reindexing feature was rarely leveraged**
  - Most cases used formals to assert/document an actual arg's domain:

```
var A: [1..n] real;
compute(A);
proc compute(X: [1..n] real) … // indicate expectation that actual is {1..n}
```

  - For such "identical domain" cases, reindexing adds extra overhead
    - our implementation of reindexing may be unnecessarily heavyweight
    - yet, there will be cases in which it will never be free
      - (e.g., reindexing may generate a distinct domain/array representation)
  - In practice, performance-sensitive users fell back to workarounds:

```
proc compute(X: [] real);            // give up on describing the domain
proc compute(X: [/*1..n*/] real);    // document domain with a comment
```

Array Formals: This Effort

- **Given overheads and programmers' intuition…**

  …interpret domains in formal arrays as constraints, not reindexing

  - specifically, "actual's index set must match formal's"

```
const DL = {1..b, 1..b},                    // local {1..b, 1..b} domain
      DD = DL dmapped Cyclic(…);            // distributed {1..b, 1..b} domain

var AL: [DL] real,                          // local {1..b, 1..b} array
    AD: [DD] real,                          // distributed {1..b, 1..b} array
    AZ: [0..#b, 0..#b] real,                // local {0..b-1, 0..b-1} array
    A: [1..n, 1..n] real;                   // local {1..n, 1..n} array

proc factor(X: [1..b, 1..b] real) …         // expect {1..b, 1..b} array
factor(AL);                                 // OK: index sets equal
factor(AD);                                 // OK: index sets equal
factor(AZ);                                 // error: different index sets
factor(A[1..b, 1..b]);                      // OK: after slice, index sets equal
```

Array Formals: This Effort

- **Given overheads and programmer intuition…**
  …interpret domains in formal arrays as constraints, not reindexing
  - if formal has non-default domain map, actual's domain map must be ==

```
const DL = {1..b, 1..b},                  // local {1..b, 1..b} domain
      DD = DL dmapped Cyclic(…);          // distributed {1..b, 1..b} domain

var AL: [DL] real,                        // local {1..b, 1..b} array
    AD: [DD] real,                        // distributed {1..b, 1..b} array
    AZ: [0..#b, 0..#b] real,              // local {0..b-1, 0..b-1} array
    A: [1..n, 1..n] real;                 // local {1..n, 1..n} array

proc factor(X: [DD] real) … // expect {1..b, 1..b} array distributed like DD
factor(AL);                               // error: different domain maps
factor(AD);                               // OK: index sets, domain maps ==
factor(AZ);                               // error: different index sets
factor(A[1..b, 1..b]);                    // error: after slice, dom. maps differ
```

Array Formals: This Effort

- **Given overheads and programmer intuition…**

  …interpret domains in formal arrays as constraints, not reindexing

  - note that the actual and formal need not share a single domain/domain map

```
const DL = {1..b, 1..b},                    // local {1..b, 1..b} domain
      DD = DL dmapped Cyclic(1);            // distributed {1..b, 1..b} domain

var AL: [DL] real,                          // local {1..b, 1..b} array
    AD: [{1..b, 1..b} dmapped Cyclic(1)] real,  // dist. {1..b, 1..b} arr.
    AZ: [0..#b, 0..#b] real,                // local {0..b-1, 0..b-1} array
    A: [1..n, 1..n] real;                   // local {1..n, 1..n} array

proc factor(X: [DD] real) … // expect {1..b, 1..b} array distributed like DD
factor(AL);                                 // error: different domain maps
factor(AD);                                 // OK: index sets, domain maps ==
factor(AZ);                                 // error: different index sets
factor(A[1..b, 1..b]);                      // error: after slice, dom. maps differ
```

Array Formals: This Effort

- **Given overheads and programmer intuition…**

…when reindexing is desired, user can insert at call-site manually:

```
var AZ: [0..#b, 0..#b] real;

proc factorL(X: [1..b, 1..b] real) …    // expect {1..b, 1..b} array
factorL(AZ.reindex(1..b, 1..b});        // OK: after reindex, ind. sets equal
```

…ultimately support a sugar for "reindex to formal's domain" case (?)

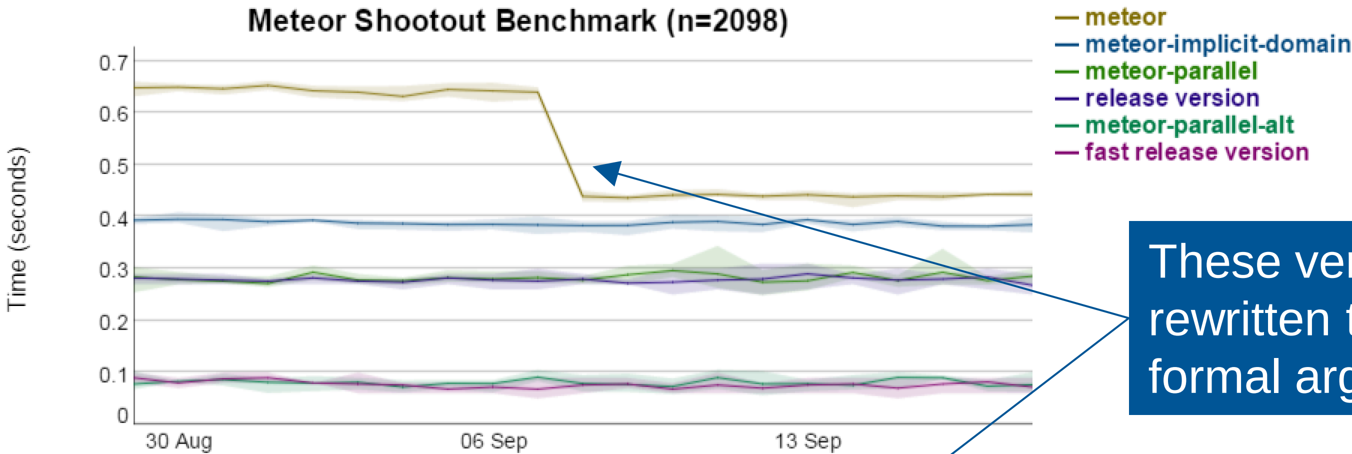
- Imagine something like:

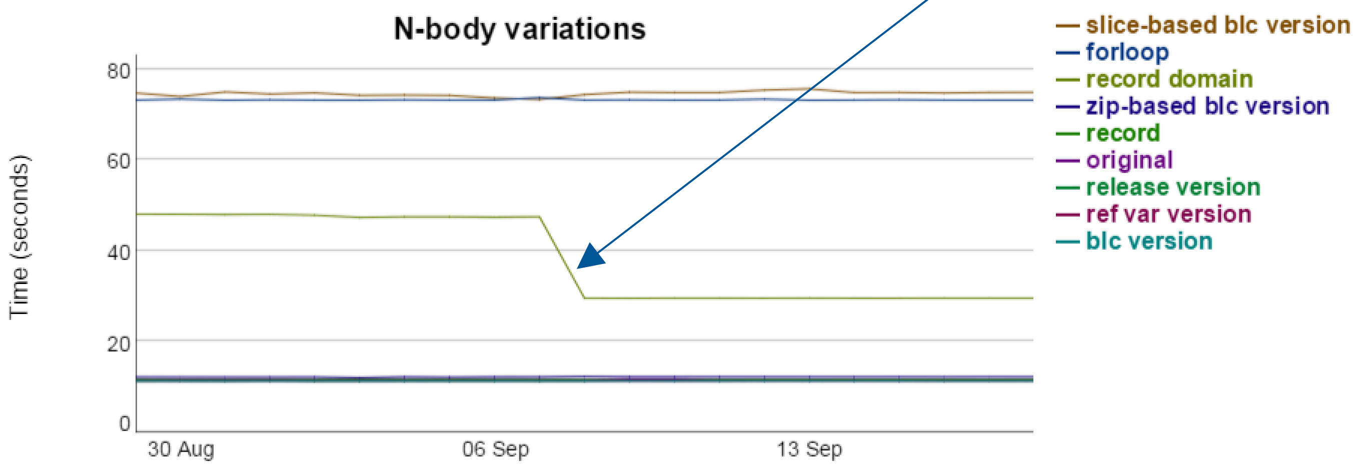
```
factorL(AZ.reindex(auto));
factorL(=>AZ);
```

(not that we're particularly excited about either of these…)

Array Formals: Impact

- Programs that used array formals to assert size improved:





These versions were never rewritten to avoid domains in formal arguments

Array Formals: Impact

Programs relying on reindexing needed to be rewritten:

for example, in examples/benchmarks/hpcc/fft.chpl, the following:

```
proc butterfly(wk1, wk2, wk3, X:[0..3]) {
  var x0 = X[0] + X[1],
      x1 = X[0] - X[1],
  …
}
```

became:

```
proc butterfly(wk1, wk2, wk3, X: [?D]) {
  const i0 = D.low,
        i1 = i0 + D.stride,
        …
  var x0 = X[i0] + X[i1],
      x1 = X[i0] - X[i1],
  …
}
```

Array Formals: Impact

Programs relying on reindexing needed to be rewritten:

for example, the following (naïve) dgemm-like routine:

```
proc dgemm(p: int, q: int, r: int,
           A: [1..p, 1..q] ?t,
           B: [1..q, 1..r] t,
           C: [1..p, q..r] t) {
  for i in 1..p do
    for j in 1..r do
      for k in 1..q do
        C[i,j] -= A[i,k] * B[k,j];
```

became:

```
proc dgemm(A: [?AD] ?t, B: [?BD] t, C: [?CD] t) {
  for (ai,ci) in zip(AD.dim(1), CD.dim(1)) do
    for (bj,cj) in zip(BD.dim(2), CD.dim(2)) do
      for (ak,bk) in zip(AD.dim(2), BD.dim(1)) do
        C[ci,cj] -= A[ai,ak] * B[bk,bj];
```

Array Formals: Status and Next Steps

Status:

- array formals are now, arguably, less surprising in Chapel
  - though some power/convenience has also been removed
  - though overheads have been removed for common “assert size” cases

Next Steps:

- design and implement a sugar for “reindex to match formal’s domain”

Type Methods and Iterators

Type Methods: Motivation

- **Chapel has only supported methods on variables/values**
  - for example:

```
class C {
  proc foo() { … }
}
proc int.foo() { … }
var myC = new C();
myC.foo(); 3.foo(); // OK to call methods on values/variables
C.foo(); int.foo(); // errors: methods can only be called on values, not types
```

- **Users have long requested "static methods"**
  - goal: to call methods on types rather than just variables/values:

```
C.bar();
int.bar();
```

Type Methods: This Effort

- **This release adds support for defining methods on types:**
  - for example:

```
class C {
  proc type bar() { … }
}
proc type int.bar() { … }

C.bar();          // OK to call type methods on matching types
int.bar();        // OK to call type methods on matching types
real.bar();       // error: no method named bar() defined on type 'real"
```

- **Syntactic approach:**
  - like a 'type' constraint/intent on an argument…

```
proc baz(type t) { … }    // baz()'s argument must be a type
```

    …yet applied to the implicit 'this' argument
  - mirrors pre-existing 'param' and 'ref' intents for 'this' arguments

Type Methods: This Effort

- **Note that, unlike C++ static methods, can't call on values:**

```
class C {
  proc type bar() { … }
}
proc type int.bar() { … }
var myC = new C();

myC.bar();      // error: can't call type methods on values/variables
3.bar();        // error: can't call type methods on values/variables
```

rationale:

- Consistent with 'type' intents in traditional Chapel argument contexts
- Removes convenience, not power (one can always call: `x.type.bar()`)
- Avoids questions about whether dynamic dispatch applies

Type Methods: Status and Next Steps

Status: now implemented in compiler and available for use

- one bug filed against type methods on generic types

Next Steps:

- fix bug for type methods on generic types
- start using in our own standard module development as appropriate

Public/Private Module-level Symbols

Public/Private: Background

- **All module level symbols were public**
  - Internal modules used various approaches for “private” symbols
    - prefix identifiers with ‘`_`’ or ‘`chpl_`’
    - put symbols in submodules and use explicit module naming to access:

      `…PrivateSubModule.privateRoutine()…`
  - Neither of these are ideal solutions for namespace control

- **Privacy controls are supported by most languages, e.g:**
  - C++’s private, public, protected fields
  - Rust symbols are private by default, can specify as public
  - Python symbols can’t be explicitly named w/o import

- **Chapel has always planned to provide better controls**
  - Yet not considered a priority compared to parallelism, performance, …
  - Until now – has become a FAQ as more libraries are written

Public/Private: This Effort

- **public/private are now keywords**
  - Public remains the default

```
private var foo = …;
public proc bar() { … }
       proc baz() { … }      // public, since not decorated
```

- **Can be used in declarations of:**
  - Modules
  - Vars, consts, and params
    - including configs
  - Procedures and iterators

Public/Private: Next Steps

- **Greater control over module 'use' statements**
  - '**`only`**' keyword to specify symbols to import

    ```
    use M1 only foo, bar;
    ```

  - '**`except`**' keyword to exclude symbols when importing

    ```
    use M1 except foo, bar;
    ```

  - Add the ability to rename symbols when importing

    ```
    use M1 only foo as M1foo;    // final syntax TBD
    ```

- **Improve module 'use' transitivity**

  ```
  module M {  var x: int;  }
  module M2 { use M; }
  …use M2;…   // is 'x' visible here?
  ```

  - Currently, the answer is always "yes"
  - Ultimately, would like to support both transitive and non-transitive uses
    - Challenges may exist, related to where generics are instantiated
  - Plan to support this via public and private 'use' statements

Public/Private: Next Steps

- **Extend public/private to type definitions**

```
private type foo = …;
private enum bar = { … };
private class C { … }
private record R { … }
```

- **Support public/private members of classes/records**
  - Determine inheritance story
    - Do we want/need an equivalent of ‘protected’?
    - Can a subclass of a ‘private’ class be ‘public’?
    - etc.

Memory Consistency Model

Memory Consistency Model: Background

- **Motivation:**
  Are you reasoning about code?
  You need a memory consistency model (MCM).
  - MCM: rules to follow in order to get consistent, predictable results.
- **Chapel has traditionally had an informal rule:**
  Use sync/single vars to order memory operations.
  - Didn't cover atomic vars, program order with parallel constructs, etc.
- **An informal model is not enough:**
  Are you reasoning rigorously about code?
  You need a *formal* memory consistency model.
  - Safer and more dependable than experience-based, ad-hoc coding

Memory Consistency Model: This Effort

- **Formalize and codify the Chapel MCM**
  - Work undertaken by MCM subteam

- **Gathered requirements, studied other memory models**

- **Produced new spec chapter: Memory Consistency Model**

Memory Consistency Model: Highlights

- **Based on:**
  sequential consistency (SC) for data-race-free programs

  … all Chapel tasks agree on the interleaving of memory operations and this interleaving results in an order consistent with the order of operations in the program source code.

  Chapel Language Spec, v0.98

- **Derived from models for similar languages**
  - Reminiscent of C11, C++11, Java, UPC, Fortran 2008, …

- **Describes effects of dynamic task creation, termination**

- **Plus: adds limited non-SC ordering**

Memory Consistency Model: SC Ordering

- **Operations on atomic/sync/single vars are ordered w.r.t. each other**
  - All observers see the same order
  - Consistent with program order
- **Operations on regular vars are <u>not</u> ordered w.r.t. each other**
  - Except: within a task, same-address ops are seen in program order
  - Ops on regular vars are ordered w.r.t. ops on atomic/sync/single vars
- **Task create/sync implies program order, thus mem order**
- **Data locality does not matter**

Memory Consistency Model: Non-SC Ordering

- **For improved performance in specific circumstances**

- **Relaxed atomics**
  - Not ordered with respect to each other
  - Ordered only with respect to non-relaxed atomics and syncs/singles
  - Results guaranteed (it's still atomic) and will be seen "eventually"
  - Example usage: parallel sum-reduction result variable

- **Unordered operations on regular vars**
  - Not ordered with respect to each other
    - ➢ Within a task, same-address operations may **not** be seen in program order
  - Ordered with respect to atomic operations
    - ➢ Not guaranteed to be seen until forced explicitly by atomic operation
  - Example usage: `A[idx[i]] = …`, where `idx[]` is a permutation
  - Neither syntax nor implementation done yet

Memory Consistency Model: Whither Now?

Impact:

- Less confusion about when memory operations are observable
- Less ad-hoc code to synchronize or communicate between tasks

Status:

- Conceptual model is complete and formalizes:
  - ✓ our intended (but informal) memory model
  - ✓ our implementation as it has existed for some time
- MCM chapter in spec is all-new and matches conceptual model

Next Steps:

- Improve spec for `memory_order` type constants and semantics
- Define syntax and semantics for unordered memory operations
- Research collaboration: Tatsuya Abe (Riken) plans to add Chapel to his MCM description language

Other Language Changes

Other Language Changes and Improvements

- **Renamed 'blank intent' to 'default intent' for clarity**
- **Changed 'use' to 'require' for external dependences**

```
use "foo.h", "-lfoo";  ⇒  require "foo.h", "-lfoo";
```

- **Added support for 'continue' statements in 'param' loops**
- **'select' now only evaluates its expression once**
- **Added support for == and != on domain maps**
  - Intuitively: "Do these map domains to the system in the same way?"
- **Added support for hexadecimal floating point literals**
  - (co-developed by Damian McGuckin, Pacific Eng. Systems Int'l)
- **Added formfeed ('\f') to the set of whitespace characters**
  - (co-developed by Damian McGuckin, Pacific Eng. Systems Int'l)

Interoperability Improvements

- **Added support for renaming external records in Chapel**
  - C struct declarations using typedefs…

    ```
    typedef struct stat { … } stat;
    ```

    …can use distinct names within Chapel if desired:

    ```
    extern "stat" record c_stat { … }
    ```

  - This capability can also support structs that aren't typdef'd in C:

    ```
    typedef struct stat { … };
    ```

    …as follows:

    ```
    extern "struct stat" record c_stat { … }
    ```

- **Added a 'chplmalloc' library permitting external code to call the Chapel allocator**
- **Stopped converting c_strings to strings when passed to generic arguments in external procedures**

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/