Hewlett Packard Enterprise

CHAPEL 1.23 RELEASE NOTES: LIBRARY IMPROVEMENTS

Chapel Team

October 15, 2020

OUTLINE

- 'Version' Module
- Collections of Classes Improvements
- Parallel-Safe API for List & Map
- New Collection Types
- Comm Diagnostics Tables
- Standard Library Namespaces
- Namespace Inspection


‘VERSION’ MODULE

‘VERSION’ MODULE
Background and This Effort

Background:

- It can be useful for Chapel code to statically reason about the version of ‘chpl’ being used to compile it
  - to ensure that certain features are available
  - to make code portable across multiple versions of the compiler

```
param lowBound = if (chplVersion < createVersion(1,22)) then 1 else 0;
```

- Previously, there hadn’t been an official / easy way to do this
- Similarly, one might want to associate version numbers with library modules

This Effort:

- Added a new standard ‘Version’ module for this purpose, supporting:
  - **sourceVersion:** a type to represent static ‘major.minor.update’ version numbers (plus an optional ‘commit’ string)
  - **comparison operators (<, >, <=, …):** to support comparisons between version values at compile time
  - **chplVersion:** the version of ‘chpl’ being used to compile the code
  - **createVersion():** a factory function for creating new version numbers

‘VERSION’ MODULE

Impact and Next Steps

Status:

- ‘Version’ module is available in Chapel 1.23
  - See ‘Version’ documentation: https://chapel-lang.org/docs/modules/standard/Version.html

Impact:

- Chapel code can now contain and reason about version numbers
- Chapel programs can now be written to be sensitive to compiler and library versions

Next Steps:

- Get user experience with ‘Version’ module features and behavior
- Consider introducing version numbers into package modules


COLLECTIONS OF CLASSES
IMPROVEMENTS

COLLECTIONS OF CLASSES IMPROVEMENTS

Background

Background:

- Most collections did not support all class types (e.g., management types, nilable vs. non-nilable) in 1.22

	list	map	set	fixed array	resized array	assoc array	sparse	tuple
owned t	✅	🔷	🔷	✅	🔷	🔷	🔷	✅
shared t	✅	✅	❌	✅	🔷	🔷	🔷	✅
borrowed t	❌	❌	✅	✅	🔷	🔷	🔷	✅
unmanaged t	✅	✅	❌	✅	🔷	🔷	🔷	✅
(shared t, shared t)	❌	❌	❌	❌	🔷	🔷	🔷	✅
owned t?	✅	✅	🔷	✅	✅	✅	✅	✅
shared t?	✅	✅	❌	✅	✅	✅	✅	✅
borrowed t?	❌	❌	❌	✅	✅	✅	✅	✅
unmanaged t?	✅	✅	❌	✅	✅	✅	✅	✅
(shared t?, shared t?)	✅	✅	❌	✅	✅	✅	✅	✅
record	✅	✅	✅	✅	✅	✅	✅	✅

Key

✅ Working ❌ Not yet working 🔷 Not expected to work

COLLECTIONS OF CLASSES IMPROVEMENTS

This Effort

This Effort: Increased support for a wider range of class types in collections

- Below are the class types supported in this release

	list	map	set	fixed array	resized array	assoc array	sparse	tuple
owned t	✅	✅	✅	✅	🔷	🔷	🔷	✅
shared t	✅	✅	✅	✅	🔷	🔷	🔷	✅
borrowed t	✅	✅	✅	✅	🔷	🔷	🔷	✅
unmanaged t	✅	✅	✅	✅	🔷	🔷	🔷	✅
(shared t, shared t)	✅	✅	✅	❌	🔷	🔷	🔷	✅
owned t?	✅	✅	✅	✅	✅	✅	✅	✅
shared t?	✅	✅	✅	✅	✅	✅	✅	✅
borrowed t?	✅	✅	✅	✅	✅	✅	✅	✅
unmanaged t?	✅	✅	✅	✅	✅	✅	✅	✅
(shared t?, shared t?)	✅	✅	✅	✅	✅	✅	✅	✅
record	✅	✅	✅	✅	✅	✅	✅	✅

COLLECTIONS OF CLASSES IMPROVEMENTS

This Effort

This Effort: Increased support for a wider range of class types in collections

- Below are the class types supported in this release

	list	map	set	fixed array	resized array	assoc array	sparse	tuple
owned t	✅	✅	✅	✅	🔷	🔷	🔷	✅
shared t	✅	✅	✅	✅	🔷	🔷	🔷	✅
borrowed t	✅	✅	✅	✅	🔷	🔷	🔷	✅
unmanaged t	✅	✅	✅	✅	🔷	🔷	🔷	✅
(shared t, shared t)	✅	✅	✅	❌	🔷	🔷	🔷	✅
owned t?	✅	✅	✅	✅	✅	✅	✅	✅
shared t?	✅	✅	✅	✅	✅	✅	✅	✅
borrowed t?	✅	✅	✅	✅	✅	✅	✅	✅
unmanaged t?	✅	✅						✅
(shared t?, shared t?)	✅	✅						✅
record	✅	✅						✅

- Bug related to default initialization of tuple array elements containing non-nilable classes

COLLECTIONS OF CLASSES IMPROVEMENTS

Status and Next Steps

Status:

- Nearly all class types are now supported for each collection type

Next Steps:

- Fix support for fixed-size arrays of tuples containing non-nilable classes


PARALLEL-SAFE API
FOR LIST AND MAP

PARALLEL-SAFE API FOR LIST AND MAP

Background

- The indexing methods for ‘list’ and ‘map’ each return references to elements
- It is trivial for users to invalidate references to elements
  - Particularly dangerous in parallel codes

```
use List;
var lst = new list(int, parSafe=true);
coforall i in 0..1 with (ref lst) {
  lst.append(i);
  ref v = lst[i];
  if v == 0 then lst.pop(i);
  else writeln(v);

}
```

The call to ‘pop()’ could end up moving the elements of ‘lst’ around in memory.

This could invalidate the other task’s reference stored in ‘v’.

PARALLEL-SAFE API FOR LIST AND MAP

Potential Solutions

- Prevent reference invalidation when using lists and maps
- Some potential solutions:
  - Never move elements
    - **Pro**: References can still be used
    - **Con**: Greatly limits API, not possible for every operation
  - Smart references
    - **Pro**: Prevents reference invalidation until all references are out of scope
    - **Con**: Too easy to cause deadlock, would require compiler support
  - Atomic blocks
    - **Pro**: Elegant, language-level solution
    - **Con**: Too risky to add an unproven language feature
  - Disable the 'this()' method when 'parSafe=true'
    - **Pro**: Prevents reference invalidation
    - **Con**: Indexing is more elegant than named methods

PARALLEL-SAFE API FOR LIST AND MAP

Potential Solutions

- Prevent reference invalidation when using lists and maps
- Some potential solutions:
  - Never move elements
    - **Pro**: References can still be used
    - **Con**: Greatly limits API, not possible for every operation
  - Smart references
    - **Pro**: Prevents reference invalidation until all references are out of scope
    - **Con**: Too easy to cause deadlock, would require compiler support
  - Atomic blocks
    - **Pro**: Elegant, language-level solution
    - **Con**: Too risky to add an unproven language feature
  - Disable the 'this()' method when 'parSafe=true'
    - **Pro**: Prevents reference invalidation
    - **Con**: Indexing is more elegant than named methods

This is the approach we chose

- Other approaches were either too involved, or did not totally prevent reference invalidation

PARALLEL-SAFE API FOR LIST AND MAP

This Effort

- Deprecate the ‘this()’ method for lists and maps initialized with ‘parSafe=true’

```
var m = map(int, int, parSafe=true);
m[0] = 0;   // warning: indexing a map initialized with ‘parSafe=true’ has been deprecated
```

- Add methods to the ‘list’ type to work around the deprecation of ‘this()’

```
use List;

class C { var x = 0; }
var lst: list(shared C, parSafe=true);
lst.append(new shared C());

var b = lst.getBorrowed(0);     // Use ‘getBorrowed()’ to get a borrow of a class element
var v = lst.getValue(0);        // Use ‘getValue()’ to get a copy of an element
lst.set(0, new shared C(16));   // Use ‘set()’ to set the value of an element
```

PARALLEL-SAFE API FOR LIST AND MAP

The ‘update()’ Method

- **Problem**: Without references, users must copy an element when they want to read it
  - Reads of large elements using ‘getValue()’ become prohibitively expensive
    - It also becomes impossible to update an element without making a copy

- **Solution**: Add a new ‘update()’ method to ‘list’ and ‘map’
  - Enables users to reference an element in a task-safe manner
    - Accepts an index to update and a generic updater object

```
// The signature of ‘update()’ for map
proc update(const ref k: keyType, updater) throws …
```

  - See documentation for list and map

  - The old ‘update()’ method for map has been renamed to ‘extend()’
    - Existing calls to ’update()’ will produce a deprecation warning

PARALLEL-SAFE API FOR LIST AND MAP

The ‘update()’ Method

- Users can pass a class, record, or first-class function to the ‘update()’ method
  - If a class or record is used, it must define a ‘this()’ method that returns a value

```
// Define an updater object with a ‘this()’ method that updates a
// map value and returns ‘none’
record myUpdater {
  var newValue = 0;

  // The ‘this()’ method accepts a key and value from a map
  proc this (const ref k, ref v) {
    // Update a map value with ‘newValue’
    v = newValue; return none;
  }
}
```

```
// Making use of the ‘update()’ method on a map
use Map;

var m: map(int, int, parSafe=true);
m.add (0, 0);

// Initialize ‘myUpdater’ as our updater object
var idx = 0;
var updater = new myUpdater(16);

// Update m[0] with ‘updater’
m.update(idx, updater);

// Prints {0: 16}
writeln(m);
```

PARALLEL-SAFE API FOR LIST AND MAP

Status

- The ‘this()’ method has been deprecated for lists and maps initialized with ‘parSafe=true’
  - Users will see deprecation warnings starting with this release
    - These warnings may become errors in future releases

- Users migrating to this release may need to adjust code to silence deprecation warnings
  - Calls to ‘this()’ for parallel-safe lists and maps will need to be replaced

```
// 1.22
var m = new map(int, int, parSafe=true);
// Implicitly adds (0, 0) to m and assigns the value 16
m[0] = 16;

// Use indexing to read and write elements
if m.contains(0) && (m[0] % 2) == 0 {
  m[0] *= 2;
}
```

```
// 1.23
var m = new map(int, int, parSafe=true);
// Potentially add the key 0, then assign the value 16
m.addOrSet(0, 16);

// Replace reads with ‘getValue()’ and writes with ‘set()’
if m.contains(0) && (m.getValue(0) % 2) == 0 {
  m.set(0, m.getValue(0) * 2);
}
```

PARALLEL-SAFE API FOR LIST AND MAP

Impact

- Users may make use of new methods to write code that is not susceptible to reference invalidation

- Preventing references from being returned gives us more flexibility in implementation choice
  - More complicated lock-free data structures may be used
    - This lock-free map is one example: Issue #14409

PARALLEL-SAFE API FOR LIST AND MAP

Next Steps

- Consider adding new types designed to be parallel-safe and removing the ‘parSafe’ parameter
  - The semantics of list and map now vary greatly depending on the value of their ‘parSafe’ parameter
    - Collections added this release may also require ‘update()’ methods and accessor methods
    - This is complex to document and hard for users to keep track of
  - Adding new collections designed to be parallel-safe would promote separation of concerns
    - These new types would be designed to prevent reference invalidation from the outset
  - We could deprecate the ‘parSafe’ parameter on list and map
    - This would simplify the implementation requirements for these collections

PARALLEL-SAFE API FOR LIST AND MAP

Next Steps

- Consider improving the capabilities of first-class functions and formalizing their design
  - The ‘update()’ method has been designed to accept first-class functions
    - Updater records with a generic ‘this()’ method are currently preferred

- Explore unguarded collections wrapped in locks as an alternative strategy for parallel-safety
  - Collections wrapped in locks could make use of the ‘this()’ indexing method safely
    - This strategy could be pursued independently of our ‘parSafe=true’ collection story


NEW COLLECTION TYPES

NEW COLLECTION TYPES

Background and This Effort

Background:

- Chapel 1.22 had several collections:
  - list
  - map
  - set

This Effort:

- Add new collection modules

- Implemented as a Google Summer of Code project
  - Student: Yujia Qiao
  - Mentors: Krishna Kumar Dey (Chapel GSoC 2019 Alum), Paul Cassella, Engin Kayraklioglu

NEW COLLECTION TYPES

‘Heap’ Standard Module

- ‘heap’ can be used to store data in a way that enables fast sorted retrieval and consumption

```
use Heap;
var h = new heap(int);   // creates a max-heap
for i in someRandomIntStream() do
  h.push(i);
for i in h.consume() do
  writeln(i);   // print items in sorted order
```

- Different comparators can be used to define ordering

```
var h = new heap(int, comparator=myComparator);
```

- Like other collections, parallel-safety can be enabled

```
var h = new heap(int, parSafe=true);
```

- See ‘Heap’ documentation: https://chapel-lang.org/docs/modules/standard/Heap.html

NEW COLLECTION TYPES

‘OrderedSet’ Package Module

- ‘orderedSet’ represents a set that maintains its items in a sorted order

```
use OrderedSet;
var s = new orderedSet(int);
for i in someRandomIntStream() do
  s.add(i);
for item in s do
  writeln(s);     // unique elements will be printed in order
```

- Different comparators can be used to define ordering

```
var s = new orderedSet(int, comparator=myComparator);
```

- Similar to other collections, parallel-safety can be enabled

```
var s = new orderedSet(int, parSafe=true);
```

- See ‘OrderedSet’ documentation: https://chapel-lang.org/docs/modules/packages/OrderedSet.html

NEW COLLECTION TYPES

Next Steps

- Collections stabilization
  - Adjust parallel-safe interface (see “Ongoing Efforts” slides)
  - Review standard collections for interface consistency, naming
- Design questions
  - Should we parametrize different implementations, or are they different collections?
    - See: https://github.com/chapel-lang/chapel/issues/15913
- Merge open pull requests for additional collections
  - ‘OrderedMap’ module
    - See: https://github.com/chapel-lang/chapel/pull/16271
  - ‘UnrolledLinkedList’ module
    - See: https://github.com/chapel-lang/chapel/pull/16244
- Promote ‘vector’ from a test-only type to standard modules:
  - See: https://github.com/chapel-lang/chapel/pull/16048


COMM DIAGNOSTICS TABLES

COMM DIAGNOSTICS TABLES

Background and This Effort

Background:

- 'CommDiagnostics' is a module for counting communication events
- Traditionally, users have printed out the array of records that is returned:

```
writeln(getCommDiagnostics());
(execute_on_nb = 2997) (put = 999, execute_on_fast = 999) (put = 999, execute_on_fast =
999) (put = 999, execute_on_fast = 999)
```

This Effort:

- Improve readability by supporting a new 'printCommDiagnosticsTable' routine:

```
printCommDiagnosticsTable();
| locale | put | execute_on_fast | execute_on_nb |
| -----: | --: | --------------: | ------------: |
|      0 |   0 |               0 |          2997 |
|      1 | 999 |             999 |             0 |
|      2 | 999 |             999 |             0 |
|      3 | 999 |             999 |             0 |
```

- An optional argument says to print "empty" columns too (those that are all-zero, like the 'get' column here)

COMM DIAGNOSTICS TABLES

Impact and Next Steps

Impact:

- Makes it much easier to see communication patterns using ‘CommDiagnostics’
- Output format is compatible with markdown (e.g., for use on GitHub issues and PRs)

locale	put	execute_on_fast	execute_on_nb
0	0	0	2997
1	999	999	0
2	999	999	0
3	999	999	0

Next Steps:

- Review the ‘CommDiagnostics’ module as part of the library stabilization effort
  - e.g., routine names seem unnecessarily verbose

STANDARD LIBRARY NAMESPACES

STANDARD LIBRARY NAMESPACES

Background and This Effort

Background:

- Chapel programs have been able to access certain standard module symbols without a 'use'/'import' statement
  - In some cases, this is by design—e.g., 'writeln("Hello, world!");'
  - Others have been unintentional

```
const myPtr: c_ptr(c_int);    // this has compiled, but ought to require 'use CPtr, SysCTypes;'
```

    - root cause: presence of 'public use' statements within internal modules
- Recent releases have improved this situation
  - However, a few cases remained due to internal module entanglement

This Effort:

- Eliminated remaining cases of internal modules unintentionally leaking standard module symbols
  - Made 'use' / 'import' private by default within internal modules, as in user code
  - Rewrote internal modules to avoid 'public use' of standard modules
- Related to this effort, also moved two modules to more appropriate locations
  - 'CPtr' (was 'internal', now 'standard') and 'LinkedLists' (was 'standard' now 'packages')

STANDARD LIBRARY NAMESPACES

Impact and Next Steps

Impact:

- Code must now explicitly ‘use’ / ‘import’ standard modules
  - key cases that are no longer auto-available: ‘Sys’, ‘SysBasic’, ‘SysCTypes’, ‘CPtr’, ‘DSIUtil’

Next Steps:

- review public symbols defined by internal modules
  - make them ‘private’ when possible
  - prefix them with ‘chpl_’ otherwise
- review standard library interfaces as part of the “Chapel 2.0” effort
- introduce a way for Chapel code to opt out of auto-available standard modules (e.g., ‘Math’)


NAMESPACE INSPECTION

NAMESPACE INSPECTION

Background and This Effort

Background: Desired a way to know what symbols are visible in a given scope

This Effort: Add a primitive to print the visible symbols from any point in the code

```
use Sort;

__primitive("get visible symbols",
            ignoreBuiltinModules=true);
```

```
getVisible.chpl:3: Printing symbols visible from here:
  $CHPL_HOME/modules/packages/Sort.chpl:265: defaultComparator
  …
  $CHPL_HOME/modules/packages/Sort.chpl:472: sort
  $CHPL_HOME/modules/packages/Sort.chpl:504: isSorted
  $CHPL_HOME/modules/packages/Sort.chpl:541: sorted
  $CHPL_HOME/modules/packages/Sort.chpl:3060: DefaultComparator
  $CHPL_HOME/modules/packages/Sort.chpl:3211: ReverseComparator
```

NAMESPACE INSPECTION

Status and Next Steps

Status:

- The new primitive can dump a list of symbols visible from any point in the code
- Named arguments are used to filter the list

```
ignoreInternalModules // default = true
ignoreBuiltinModules  // default = false
```

- Currently more of a feature for Chapel developers than users

Next Steps:

- Extend the implementation to make it more of a user feature
  - Make it into a function instead of a primitive
  - Add the function to the Reflection module
  - Return an array of symbol names instead of printing them at compile time
- Optimize the implementation (currently *O(#GlobalSymbols)*)
- Consider how it should work w.r.t. overloaded functions (within a module / across modules)

OTHER LIBRARY IMPROVEMENTS

OTHER LIBRARY IMPROVEMENTS

For a more complete list of library changes and improvements in the 1.23 release, refer to the following sections in the CHANGES.md file:

- ‘Standard Library Modules’
- ‘Package Modules’
- ‘Bug Fixes’
- ‘Deprecated / Removed Library Features’


THANK YOU
https://chapel-lang.org
@ChapelLanguage