Hewlett Packard Enterprise

CHAPEL 1.29.0/1.30.0 RELEASE NOTES: LANGUAGE IMPROVEMENTS

Chapel Team

December 15, 2022 / March 23, 2023

LANGUAGE CHANGES IN CHAPEL 1.29 AND 1.30

Background and This Effort

Background: Chapel 2.0

- Goal is to provide a version of the language that is stable
  - Features that are documented as being unstable may change in future minor releases
  - New non-breaking changes can still be made
  - Major changes to features declared stable will trigger a new major version of the language

This Effort:

- Implemented new features requested by users or aiding with stabilization
- Address other issues in need of attention
  - User questions have led to some clarifications/simplifications
  - Dyno/compiler rework of type/call resolution has uncovered some rough edges

STATUS OF LANGUAGE STABILIZATION

Stabilized in 1.29 or 1.30, and Next Steps

Stabilized in 1.29 or 1.30:

- Added initial support for throwing initializers, sufficient for supporting standard module use cases
- Stabilized the '.find( )’ method on arrays
- Improved range slicing behaviors
- Stabilized zipped serial loops over unbounded ranges
- Made overload resolution for generic vs. typed arguments consistent
- Added support for single statement routines and removed the exception for ‘return’ statements
- Removed support for unary negation on 'uint(w)'
- Deprecated 'bool(w)'

Next Steps:

- Generics: handling of generic records/classes and partial instantiation
- Approach for special method naming
- Consider removing support for default ‘ref-maybe-const’ intents
- Make sure tuple semantics are appropriate w.r.t. ‘ref’ vs. ‘const’ behavior

OUTLINE

- Attributes
- Throwing Initializers
- Changes to Yielding Tuples
- ‘.transmute( )’ Method
- Array and Range Features
- Class Management Updates
- Untyped vs. Generic Formals
- Single-Statement Routines
- Unary Negation of ‘uint’s
- Deprecation of ‘bool(w)’

ATTRIBUTES

ATTRIBUTES
Background

- For some time, Chapel users and developers have been interested in support for *attributes*
  - Purpose: a means of communicating information to the compiler, or other tools, without language changes

```
// sample attributes:
@attribute1
proc bar() { … }

@attribute2(arg1="value", arg2=1, arg3=1.0, arg4=true, arg5=1..10)
proc foo() { … }
```

- In the meantime, Chapel has been making use of pragmas, and occasionally keywords, for such purposes
  - These approaches were not as flexible or attractive
  - Pragmas were never intended to be a user-facing feature

ATTRIBUTES
This Effort

- Implemented a generalized attribute feature
  - Developed syntax to support attributes in more places than pragmas had been (e.g., loops)
  - Added support for multiple (optionally named) arguments
  - Defined the notion of *tool namespaces*
    - e.g., '@chpldoc.nodoc' is an attribute specific to the 'chpldoc' tool

- Implemented some initial attributes: '@unstable', '@deprecated', and '@chpldoc.nodoc':

```
@deprecated(since="1.30", notes="foo is deprecated", suggestion="use newFoo instead")
proc foo() { … }

@unstable(category="experimental", issue="1234", reason="testing a new feature")
proc bar() { … }

@chpldoc.nodoc
proc baz() { … }
```

- Removed the developer-oriented 'deprecated' keyword

ATTRIBUTES
Status and Next Steps

Status:

- Added attribute support in 1.30.0
- The tool names ‘chpl’ and ‘chpldoc’ are reserved for use by the Chapel team
- Flags can be used to control how the compiler reacts to tool names
  - Ignore all tool names by passing ‘--no-warn-unknown-attribute-toolname’ to ‘chpl’
  - Ignore a specific tool name by passing ‘--using-attribute-toolname=<toolname>’ to ‘chpl’

Next Steps:

- Implement additional attributes according to our needs and user requests, for example:
  - Control memory alignment, e.g., ‘@chpl.align(n)’
  - Indicate a loop should always be unrolled, e.g., ‘@chpl.unroll(n)’
- Continue to refine our philosophy about what should be supported as an attribute vs. a language feature
- Remove the “no doc” pragma

THROWING INITIALIZERS

THROWING INITIALIZERS

Background and This Effort

Background: Initializers could not be declared with 'throws'

- Only supported 'try!' without catch blocks

```
proc init(…) {                               // Couldn't declare with 'throws'
  this.x = try! someThrowingFunc();          // Will halt if an error is thrown
}
```

This Effort: Added initial support for throwing initializers

- Throwing calls can now be made after all fields are initialized

```
class Foo {
  proc init(…) throws {
   …
   this.complete();          // Guarantees all fields are initialized
   someThrowingProc();       // Any thrown error will be propagated out of 'init'
  }
}
```

THROWING INITIALIZERS

Impact and Next Steps

Impact:

- Throwing initializers are used by types in the ‘BigInteger’, ‘IO’, and ‘Regex’ modules
  - ‘Regex’ can now be stabilized for 2.0

Next Steps:

- Expand support for other throwing patterns:
  - Support 'throw' statements in initializer bodies
  - Support 'try!'/'try' with 'catch' blocks
- Explore supporting throwing code before field initialization is complete

BEHAVIOR UPDATES WHEN YIELDING TUPLES

YIELDING TUPLES
Background and This Effort

Background: tuples are intended to behave like a collection of individual variables

- Specifically, w.r.t. carrying a value vs. a reference
  - e.g., ‘f((myInt, myArray))’ passes ‘myInt’ by ‘const in’ and ‘myArray’ by ‘ref’ if ‘f’s formal has default intent

- Default yield intent was “by value” for almost all types
  - except it was “by reference” for tuple components that are arrays, records, or similar
    - due to an oversight in specification and implementation
  - yielding by value was chosen to match returning by value for default return intent

This Effort:

- Reconciled the behavior of yielding tuples with yielding individual values
  - “by value” default yield intent now includes tuple components of all types

YIELDING TUPLES
Impact

- Record-like types are now yielded by value by default, whether in a tuple or standalone
  - e.g., consider the following statements in a procedure or iterator with the default return/yield intent:

```
return myRecord;          // returns 'myRecord' by value, as before
return (myRecord, 0);     // ditto
yield myRecord;           // yields 'myRecord' by value, as before
yield (myRecord, 0);      // now yields 'myRecord' by value, too
```

- Some adjustments were required to accommodate this change:
  - StencilDist's 'boundaries' iterator is now annotated with a pragma to retain the "yield by reference" behavior
    - 'boundaries' yields (element, index) pairs and allows updating 'element' in the loop body
  - DistributedFFT code now needs to distinguish between owning and borrowing 'fftw_plan' pointers

```
forall (plan, myzRange) in yPlan.batch() {
  ...   // within loop body, 'plan' is now a copy of a Chapel record wrapping a long-lived 'fftw_plan'
}       // when a loop iteration finishes, 'plan' is now deinitialized, however the wrapped 'fftw_plan' should not be destroyed
```

- Yielding behavior for the default intent is now explicitly defined in the language specification

YIELDING TUPLES
Next Steps

- Finalize the default yielding behavior: should it be by value or by default argument intent? [#21888]
  - yield by value:
    - analogous to returning: passing something back to the outside of the function
    - suits iterators that create new records for the purpose of yielding them
    - the current default
  - yield by default intent
    - arrays, string, records, record-like types will be yielded by reference
    - analogous to argument passing: treats a loop iteration like a (shorter lived) function call
    - more suitable for iterators that yield records external to the iterator
    - currently, no user-facing way to achieve this when yielding records within tuples

- Provide a means for users to specify value/reference behavior of each component explicitly

```
iter map.items() { ...                        // e.g., we want an iterator over '(key, value)' pairs in a map
  yield (const entry.key, ref entry.value);   // to yield keys by 'const' intent (value or ref) and values by 'ref'
... }
```

'.TRANSMUTE' METHOD

‘.TRANSMUTE’ METHOD

Background and This Effort

Background:

- Chapel’s type conversions typically attempt to preserve logical values when possible

  …`1: real`… *// results in 1.0*

  …`2.3: int`… *// results in 2, a necessary loss of precision due to the types involved*

- Sometimes, it is useful to convert between types in a way that preserves *bits* rather than logical values
  - e.g., ‘9218868437227405312’ == ‘0x7ff0000000000000’ == ‘inf’ when bits are interpreted as a floating-point value
  - yet ‘9218868437227405312: real’ == ‘9.21887e+18’

This Effort:

- Added a new ‘.transmute( )’ method that can convert between types of matching width, preserving bit patterns

  …`9218868437227405312.transmute(real)`… *// results in a ’real’ with the value ‘inf’*

- Currently, only supports conversions between ‘real(64)’ and ‘uint(64)’ as well as ‘real(32)’ and ‘uint(32)’
  - Supports both compile-time (‘param’) and execution-time transmutations

‘.TRANSMUTE’ METHOD

Impact, Status, and Next Steps

Impact:

- Addresses a longstanding user request

Status:

- Implemented in 1.30.0
- Currently considered unstable because design did not receive much attention prior to the release

Next Steps:

- Finalize interface design and stabilize
- Consider adding support for other types of matching width
  - e.g., transmute from an ‘imag’ to a ‘uint’, ‘int’, or ‘real’?
- Consider extending to richer types:
  - e.g., transmute a 1024-element array of ‘real(32)’ into a 512-element array of ‘uint(64)’?
  - e.g., transmute a 4-tuple of uint(8) into an ‘int(32)’?

ARRAY AND RANGE IMPROVEMENTS

- ‘.fullIdxType’ Query
- ‘.find( )’ Method on Arrays
- Array Literal Type Inference
- Range Slicing Improvements
- Unbounded ranges:
  - Serial Zipped Loops
  - with ‘enum’/’bool’ Indices

‘.FULLIDXTYPE’ QUERY

ARRAYS: '.FULLIDXTYPE' QUERY

Background:

- Chapel arrays have long supported an '.idxType' query for the per-dimension index type
  - matches the 'idxType' argument used when declaring range and domain types

```
var A: [1..100] real;              …A.idxType…    // evaluates to 'int' since A's only dimension is indexed by 'int's
var B: [1..100, 1..100] real;      …B.idxType…    // evaluates to 'int' since each dimension is indexed by 'int's
var C = ["hi" => 1, "bye" => 2];   …C.idxType…    // evaluates to 'string' since strings are used to index 'C'
```

- Have also desired some way of referring to the complete index type used by multidimensional arrays in practice
  - can think of this query as indicating "what type would a loop over this array's domain yield?"

```
var A: [1..100] real;              …A.???…        // would evaluate to 'int'
var B: [1..100, 1..100] real;      …B.???…        // would evaluate to '2*int'
var C = ["hi" => 1, "bye" => 2];   …C.???…        // would evaluate to 'string'
```

This Effort:

- Decided to name this query '.fullIdxType' and implemented it for Chapel 1.30
- Used it in the new 1-argument 'array.find( )' routine (see next section)

Next Steps:

- Explore whether Chapel could/should support implicit conversions between scalars of type 't' and '1*t' tuples

‘.FIND’ METHOD ON ARRAYS

ARRAYS: '.FIND' METHOD
Background and This Effort

Background:

- Chapel arrays have supported a '.find( )' method for quite some time
- However, its return type has not matched that of '.find( )' on 'bytes' or 'string' values

```
bytes.find(…): int;                                 // returns '–1' if the pattern was not found
string.find(…): byteIndex;                          // returns '–1' if the pattern was not found
[array].find(…): (bool, index(this.domain))         // returns whether or not the value was found + the index if it was
```

  – Traditional rationale for difference: No obvious sentinel index to return since arrays can have arbitrary indices
- In addition, its implementation has been serial
  – Not ideal for a parallel language, particularly when using it on distributed arrays

This Effort:

- Deprecated previous '.find( )' on arrays and introduced two new overloads (enabled with '-suseNewArrayFind'):
  – First overload is only supported on rectangular arrays

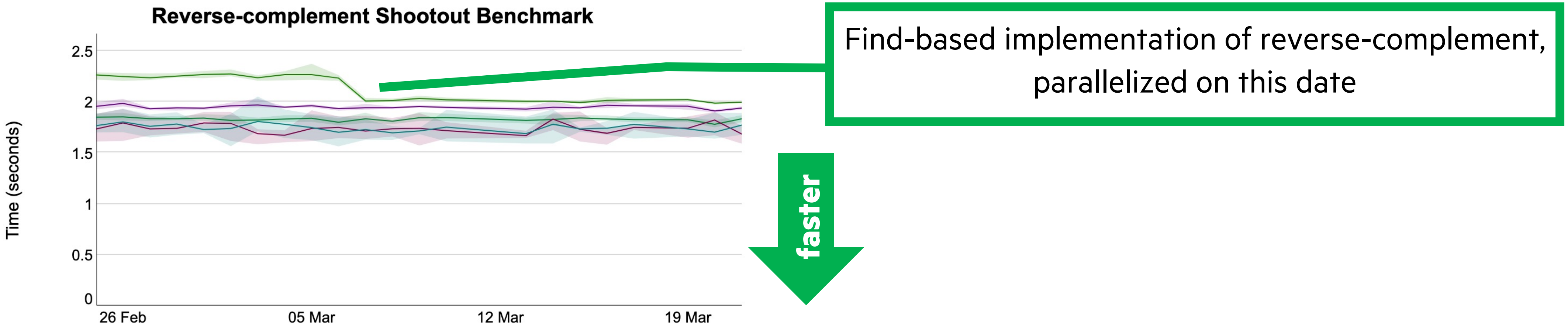
```
proc [array].find(val: eltType): fullIdxType;                  // returns 'domain.lowBound – 1' if 'val' is not found
proc [array].find(val: eltType, ref idx: fullIdxType): bool;   // returns 'true' & location in 'idx'; or 'false'
```

- Parallelized these new implementations

ARRAYS: '.FIND' METHOD

Impact

Impact: Parallelization helps at modest problem sizes (here, a local 64k-element array of 8-bit ints)



- Improvements for distributed arrays can be massive, due to properly aligning iterations with their array elements
  - E.g., communication counts for a 'find( )' on a 1,000,000-element array distributed across 4 locales:
  - Old serial version:

locale	gets
0	750,021
1	0
2	0
3	0

New parallel version:

locale	gets	active msgs	non-blocking active msgs
0	6	0	3
1	0	2	0
2	0	2	0
3	0	2	0

ARRAYS: '.FIND' METHOD

Status and Next Steps

Status:

- The interface and implementation of '.find( )' on arrays is now much improved

Next Steps:

- Optimize implementation for additional cases:
  - Make use of 'memchr( )' when searching for 8-bit values?
  - Squash parallelism for smaller arrays?
- Consider adding an 'indices' argument to restrict searches, as with 'string/bytes.find( )'?
  - Not as crucial for arrays since they support O(1) slicing, unlike 'string'/'bytes'
  - Yet, could be more efficient than slicing
- Make serial loops over distributed domains/arrays execute with proper affinity to indices/elements?

ARRAY LITERAL TYPE INFERENCE

ARRAY LITERAL TYPE INFERENCE

Background and This Effort

Background:

- Traditionally, Chapel has inferred an array literal's element type based on its first element:

```
[1.2, 3  ]    // inferred to be an array of 'real' due to '1.2'; since '3' can coerce to 'real', this is OK
[1,   2.3]    // inferred to be an array of 'int' due to '1'; since '2.3' can't coerce to 'int', this was an error
```

This Effort:

- Improved array literal inference to consider all elements
  - Implemented using return type inference for procedures, so has similar capabilities and limitations
  - Similarly improved 'LinearAlgebra' module's inference of 'Matrix' types based on input arrays
- Accelerated the compilation times of homogeneous array literals
  - Compilation times for 5060-element arrays:

expr types	previously	with PR
int-only	0:24	0:11
int/real	error	0:27

ARRAY LITERAL TYPE INFERENCE

Impact, Status, and Next Steps

Impact:

- Arrays with mixed, yet compatible, element types are now supported

```
[1.2, 3  ]    // still inferred to be an array of 'real'
[1,   2.3]    // now inferred to be an array of 'real'
```

- Improves productivity of users working with arrays and matrices

Status: Implemented in Chapel 1.29.0

Next Steps:

- Move inference logic from module code to compiler code to further accelerate compilation of array literals
- Add language support for multidimensional array literals

RANGE SLICING IMPROVEMENTS

RANGES: SLICING IMPROVEMENTS

Background and This Effort

Background: range slicing 'range1[range2]' is an intersection of index sequences: range1 ∩ range2

- Array and domain slicing perform per-dimension range slicing

This Effort: updated some slicing behavior to match intuition about how array slicing should behave

```
var A: [1..9] real;              // the following comments show (intuition) → (updated behavior)

for a in A[1..9 by -1] do        // reverse the traversal order → writes A[9], ..., A[1]
  writeln(a);

writeln(A[..6 by 2]);            // pick every 2nd index that is ≤6 → writes A[1..6 by 2] i.e., A[1], A[3], A[5]
```

- Updates for unaligned ranges

```
( 1..7 by -1 )[ ..4 by 2 ]       // intersect the bounds, apply the stride → 1..4 by -2
( 1..7 by -2 )[ ..4 by 2 ]       // "impose" 'align 1' on 2nd range to match 1st range → 1..4 by -2 align 1
( 1..7 by -3 )[ ..5 by 2 ]       // copy alignment from 1st range into 2nd range → 1..5 by -6 align 1
                                 // using 'align 4' would be just as valid → issue unstable warning
( ..5 by 2 )[ 1..7 by -3 ]       // we expect that users will not need to slice an unaligned range → disallow it for now
```

RANGES: SLICING IMPROVEMENTS

Impact, Status, and Next Steps

Impact:

- Range slicing behavior now follows our intuition

Status:

- Enabled new slicing behavior with negative strides when compiling with '-snewSliceRule'
  - by default, the previous behavior is preserved, with a deprecation warning
- Enabled new slicing behavior with unaligned ranges by default
  - this change affects only rare corner cases
- While there, added a warning when creating arrays and slices with negative strides
  - enabled by default

Next Steps:

- Enable new slicing behavior with negative strides by default
- Finalize behaviors for arrays and array slices of negative strides
  - Ensure correct implementation of array slices with negative strides

SERIAL ZIPPED LOOPS OVER UNBOUNDED RANGES

UNBOUNDED RANGES: ZIPPED SERIAL LOOPS

Background and This Effort

Background:

- A parallel zipped loop in Chapel is governed by its *leader* expression, which determines the policy for the loop

```
forall (a, b) in zip(A, B) …   // 'A' is the leader of this loop, so its parallel iterator determines how this loop will be run
```

- Unbounded ranges have special "follower" behavior when they are zipped with finite leaders

```
forall (i, j) in zip(lo..hi, 1..) …   // though '1..' is conceptually infinite, it will conform to the size of 'lo..hi'
```

- To date, unbounded ranges have not supported the leader role in *parallel* loops

```
forall (i, j) in zip(1.., lo..hi) …   // 'error: parallel iteration is not supported over unbounded ranges'
```

- However, they have been legal as leader expressions of *serial* zipped loops, and conformed to their follower(s)

```
for (i, j) in zip(1.., lo..hi) …   // ran for lo-hi+1 iterations, as though 'lo..hi' was the leader
```

  – This felt inconsistent, while also posing challenges for plans to support serial leader/follower iterators in the future

This Effort:

- Considered this a bug and decided to treat such loops as conceptually infinite, similar to 'for i in 1.. do …'

```
for (i, j) in zip(1.., lo..hi) …   // now results in a size mismatch if it doesn't 'break', 'return', or 'exit' before j == hi+1
```

- Added a compile-time warning for such cases to inform users of the change in behavior

UNBOUNDED RANGES: ZIPPED SERIAL LOOPS
Impact, Status, and Next Steps

Impact:

- Updated user codes in which serial loops were led by an unbounded range
  - Found more cases of this than we had anticipated
- Language now feels more consistent

Status: Implemented in Chapel 1.29.0

Next Steps:

- Develop plan for serial leader-follower iterators
- Permit users to write “unbounded, but willing to conform” iterators, similar to unbounded ranges
  - E.g., a serial iterator generating random numbers that conforms to its leader’s size/rank
- Improve general approach used for defining iterator families on a type (“leader-follower 2.0”)
- Add support for unbounded ranges to lead parallel loops?

UNBOUNDED RANGES OVER ENUM/BOOL

UNBOUNDED RANGES: ENUM / BOOL

Background, This Effort, and Status

Background:

- Chapel 1.27 improved support for looping over unbounded ranges with ‘enum’ and ‘bool’ indices

```
enum color { red, orange, yellow, green, blue, indigo, violet };
use color;
for c in (blue..) do …              // loops over ‘blue’, ‘indigo’, ‘violet’, then stops
```

- However, a few cases were still not implemented correctly:

```
for c in (blue.. by -1) do …        // should loop over ‘violet’, ‘indigo’, ‘blue’
                                    // instead, got ‘error: halt reached - iteration over range that has no first index’
```

This Effort:

- Added support for cases that were not working before:

```
for c in (blue.. by -1) do …        // now loops over ‘violet’, ‘indigo’, ‘blue’
```

Status: Unbounded ranges of ‘enum’ and ‘bool’ now support iteration more consistently

UNBOUNDED RANGES: ENUM / BOOL

Next Steps

Next Steps: Determine how other ops on unbounded ranges of ‘enum’ or ‘bool’ should behave [#20896]

- ’(blue..).last’:
  - ‘violet’ because that’s the last value iteration would reach?
  - Or undefined because it’s unbounded?
- ‘(blue..).high’:
  - ‘violet’: because that’s its high bound when iterating?
  - Or undefined because it’s unbounded?
- ‘(blue..) == (blue..violet)’
  - ‘true’ because they describe the same indices when iterating?
  - Or false, because they are not identical range values?

CLASS MANAGEMENT UPDATES

CLASS MANAGEMENT UPDATES
Background

- Chapel supports multiple ways to create and convert objects with different management strategies

```
var obj = owned.create(new unmanaged A());
var s: shared A?;
s.retain(obj.release());    // obj is now dead
obj = new A();
s = shared.create(obj);     // obj is now dead
```

- Managed objects' lifetimes can be manually controlled

```
obj.clear();                // obj is now dead
delete obj.release();       // same as 'obj.clear( )'
```

- This usage of methods vs. type methods…
  - is inconsistent
  - provides multiple ways to do the same thing

CLASS MANAGEMENT UPDATES

This Effort and Next Steps

This Effort:

- Added three additional experimental type methods intended to replace the previous API
  - ‘owned.adopt( )’, ‘owned.release( )’, and ‘shared.adopt( )’
  - One way to control object lifetime and convert management strategies

```
var obj = owned.adopt(new unmanaged A());    // instead of 'owned.create(…)'
var s = shared.adopt(owned.release(obj));    // instead of 's.retain(o.release( ))'
obj = new A();
s = shared.adopt(obj);          // instead of 'shared.create(o)'
delete owned.release(obj);      // instead of 'o.clear( )' or 'o.release( )'
```

Next Steps:

- Deprecate ‘create( )’, ‘retain( )’, ‘clear( )’, and ‘release( )’
- Allow assignment to ‘nil’ as a safer way to cut a lifetime short

```
obj = nil;    // obj is now dead
```

- Improve the interoperability between managed and unmanaged classes

UNTYPED FORMALS AND IMPLICIT CONVERSION

UNTYPED FORMALS
Background

- Implicit conversion and instantiation are two ways an actual might not precisely match a formal:

```
proc converts(arg: real) { … }
converts(1);          // implicitly converts the 'int' value 1 into the 'real' value 1.0 and calls 'converts(1.0)'

proc instantiates(arg) { … }
instantiates(1);    // instantiates the 'arg' formal with 'int' to generate 'proc instantiates(arg: int)' and calls that
```

UNTYPED FORMALS
Background

- Yet, what happens when the compiler needs to choose between these two for a single call?
  - Chapel has preferred to do implicit conversion rather than instantiate an untyped formal
    - For example, the call to 'g(1)' below would use implicit conversion to call the 'real' version:

```
proc g(arg) { … }              // #1
proc g(arg: real) { … }        // #2
g(1);    // called the 'real' version, #2
```

  - In contrast, when the formal had an explicit generic type, Chapel preferred to instantiate:

```
proc h(arg: integral) { … }    // #3
proc h(arg: real) { … }.       // #4
h(1);    // called the 'integral' version, #3
```

  - This differed from the C++ and C# behaviors in addition to being inconsistent between the 'g( )' and 'h( )' cases

UNTYPED FORMALS

This Effort and Impact

This Effort: Adjusted resolution rules to remove the special behavior for untyped formals

- Now the genericity of formals is only considered when the formals have the same type after instantiation
- Causes the example on the previous slide to behave more similarly to the 'integral' version:

```
proc g(arg) { … }           // #1
proc g(arg: real) { … }   // #2
g(1);    // calls the generic version, #1
```

Impact:

- Chapel behavior in this regard is now more similar to C++ and C#
- In rare cases, code that assumed the previous behavior needs to be adjusted. For example:

```
proc category(arg)                          { return "anything"; }
proc category(arg: real)                    { return "convertible to real"; }
// can be changed into:
proc category(arg)                          { return "anything"; }
proc category(arg)
  where isCoercible(arg.type, real) { return "convertible to real"; }
```

SINGLE-STATEMENT SUBROUTINES

SINGLE-STATEMENT SUBROUTINES
Background

Background:

- Since Chapel's inception, it has supported single-statement subroutines if the statement was a 'return'

```
proc computeAnswer()
  return 42;
```

- However, it has not supported other single-statement subroutines due to the potential for syntactic ambiguities

```
proc writeDebugMsg(msg)
  writeln("Debug: ", msg);    // syntax error: near 'writeln'
```

- Meanwhile, other syntactic constructs support single-statement forms via keywords like 'do' and 'then':

```
for i in 1..10 do
  writeln(i);
```

```
if verbose then
  writeln("Blah blah blah");
```

- These asymmetries felt unsettling going into Chapel 2.0
  - Should 'return' get special treatment?
  - Should we support other single-statement subroutines?

SINGLE-STATEMENT SUBROUTINES

This Effort and Status

This Effort: Decided to resolve these inconsistencies

- Deprecated the special-case for single-statement routines that are returns

```
proc computeAnswer()     // now results in: warning: Single-statement 'return' routines are deprecated;
  return 42;             //                 please insert 'do' before the 'return' or wrap the statement in curly brackets
```

- Added the ability to define single-statement subroutines using ‘do’:

```
proc writeDebugMsg(msg) do
  writeln("Debug: ", msg);
```

- Updated existing uses of the ‘return’ exception to use ‘do’ instead:

```
proc computeAnswer() do
  return 42;
```

Status: Implemented in 1.30.0

UNARY NEGATION OF UNSIGNED INTEGERS

UNARY NEGATION

Background:

- Historically, the result of unary negation on an unsigned integer depended on its width:

Unsigned Integer Type	Result of Unary Negation
unt(64), uint	compilation error
uint(32)	int(64)
uint(16)	int(32)
uint(8)	int(16)

- Potentially surprising to have arithmetic on 32-bit unsigned integers result in 64-bit signed integers

This Effort: Changed unary negation to result in a compilation error for any unsigned integer

Impact:

- Now easier to compute with a particular bit width of unsigned integers
- The error helps users catch unintentional mistakes in their code
- The error allows further adjustments as non-breaking changes

DEPRECATION OF ‘BOOL(W)’

DEPRECATION OF 'BOOL(W)'

Background:

- Chapel has supported fixed-width 'bool' values for years: 'bool(8)', 'bool(16)', 'bool(32)', 'bool(64)'
  - Rationale:
    - The width of Chapel's default 'bool' is implementation-defined
    - These variations gave programmers a means of specifying the bit-width of a specific bool's representation
- This approach has had some downsides:
  - One of the few sources of cycles in the graph of Chapel's implicit conversions
    - 'bool(8)' implicitly converts to 'bool(64)' which implicitly converts to 'bool(8)'
  - Has felt confusing to users, and often like overkill
    - "'bool' only requires one bit, so why do all these variations exist?"
- Meanwhile, have also wanted more control over the memory layout of other types
  - e.g., the ability to cache-align and/or pad an 'atomic int(32)' value

This Effort: Decided to deprecate 'bool(w)' and rely on forthcoming memory attributes to control layout

Status: Implemented in Chapel 1.30

Next Steps: Develop and implement attributes for memory alignment and/or padding

OTHER LANGUAGE IMPROVEMENTS

OTHER LANGUAGE IMPROVEMENTS

For a more complete list of language changes and improvements in the 1.29.0 and 1.30.0 releases, refer to the following sections in the CHANGES.md file:

- New [Language] Features
- Feature Improvements
- Semantic Changes/Changes to the Chapel Language
- Syntactic/Naming Changes
- Deprecated/Unstable/Removed Language Features
- Bug Fixes

THANK YOU

https://chapel-lang.org
@ChapelLanguage