CHAPEL 1.27.0/1.28.0 RELEASE NOTES:
LANGUAGE IMPROVEMENTS

Chapel Team
June 30, 2022 / September 15, 2022
OUTLINE

- @Unstable Attribute
- Language Stabilization
- Documenting Built-in Types in the Spec
- Task Intents on ‘this’
- Ignoring Unreachable Code
- Shadowing Fields
- Parentheses-less Methods
- Range Improvements
- Implicit Numeric Conversions
- Overload Resolution
- Module Scoping
@UNSTABLE ATTRIBUTE
@UNSTABLE ATTRIBUTE

Background

- Features are **unstable** if it is known that further discussion and adjustments to them are needed

```
proc type datetime.fromTimestamp(tick: real, tz: shared TZInfo?)

The datetime that is tick seconds from the epoch
```

⚠️ Warning

tzinfo is unstable; its type may change in the future

- Users of unstable features will be notified if they compile their code with the flag `--warn-unstable`
Background

• The 2.0 stabilization effort has classified many features and symbols as unstable
  • E.g., timezone support, unions, GPU support

• Future library developers may want to label symbols as unstable

• Old way to mark a feature as unstable was not user-facing
  • It was also only applicable to modules and functions

    proc foo() {
        if chpl_warnUnstable {
            // Symbols that start with 'chpl_' are generally considered an implementation detail
            compilerWarning("foo is unstable and may change in future releases");
        }
        ...
    }

• Had to label unstable symbols by hand in the documentation
@UNSTABLE ATTRIBUTE

This Effort

• Added ‘@unstable’, an attribute that can be applied to any symbol

  ```
  @unstable module ArgumentParser { ... }
  @unstable class Foo { ... }
  ```

• An optional message can be provided, overriding the default message of "<name> is unstable"

  ```
  // Prints "warning: x is unstable" when 'x' is accessed
  @unstable var x: string;
  ```

  ```
  // Prints "warning: y is unstable and may change type in the future" when 'y' is accessed
  @unstable "y is unstable and may change type in the future" var y: int;
  ```

• This attribute will generate warnings when code is compiled with ‘--warn-unstable’
  • It will also insert warnings into documentation generated using ‘chpldoc’
    - If a symbol's documentation already includes the word "unstable", this will be skipped
@UNSTABLE ATTRIBUTE
Impact and Next Steps

Impact:

- Users can now mark any symbol as unstable
- All compile-time uses of ‘chpl_warnUnstable’ in Chapel’s modules have been replaced with this attribute
  - one dynamic case remains for arrays with a negative stride; ‘@unstable’ can’t really help with execution-time cases

Next Steps:

- Suppress warnings triggered in Chapel's provided libraries [#20541]
  - User can't do anything about them, shouldn't have to see them when using ‘--warn-unstable'
  - Might re-enable with a developer-oriented flag

- Potentially skip generating documentation warnings in other cases? [#20676]

/* This symbol is not considered stable */
@unstable var g: bool;
Next Steps (continued):

- Trigger unstable warnings when an unstable config is set via the command line [#20680]
  
  ````
  $ cat foo.chpl
  @unstable config var x: bool = false;
  $ chpl --warn-unstable foo.chpl
  $ ./foo --x=true  # This program should trigger a warning
  ```

- Promote ‘deprecated’ keyword to a user-facing feature by making it an attribute

- Add full attribute support
  – See slides 66-70 of the 1.25 release deck on ongoing efforts and discussion on GitHub [#14141]
LANGUAGE CHANGES SUPPORTING STABILIZATION
LANGUAGE CHANGES SUPPORTING STABILIZATION

Background and This Effort

- We have been working towards stabilizing Chapel language and standard library features
  - So that users can rely on them not changing in future releases
- This section discusses such changes for language features whose implementations are module-based
  - Primarily focuses on code in the internal modules, which aren’t user-facing
LANGUAGE CHANGES SUPPORTING STABILIZATION

- Class Management Updates
- Locale Updates
- Array Updates
- Deprecating the ‘<~>’ Operator
CLASS MANAGEMENT UPDATES
CLASS MANAGEMENT UPDATES

Background and This Effort

Background:

• Class management styles enable different strategies for de-initializing classes
• Some operations between managed classes can be confusing and less than helpful
  – Implicit casting could result in unwanted behavior in some cases
  – The behavior of ‘new borrowed’ was unclear

This Effort:

• Confine the set of allowed operations to those that have a clear application
• Deprecate the more confusing and unhelpful operations
  – More explicit applications of the deprecated behavior are still allowed
CLASS MANAGEMENT UPDATES

Status

- Assignment and initialization of ‘shared’ from ‘owned’ is deprecated
  
  ```
  myShared = myOwned;  // emits a deprecation warning
  ```
  
  ```
  myShared = myOwned: shared;  // OK
  ```

- ‘new borrowed’ is now deprecated
  
  ```
  var x = new borrowed C();  // emits a deprecation warning
  ```
  
  ```
  var tx = new owned C();
  var x = tx.borrow();  // OK
  ```

- Initializing or assigning a borrowed from ‘new owned’ / ‘new shared’ / ‘new unmanaged’ now warns
  
  ```
  myBorrowed = new shared C();  // emits a compiler warning
  ```
  
  ```
  var ty = new shared C();
  myBorrowed = ty.borrow();  // OK
  ```

- Nilable variants (e.g., ’C?’) have similar deprecations and warnings
LOCALE UPDATES
**Background:**

- The ‘locale’ type represents a unit of target architecture (e.g., compute node, GPU, ...)
- Locales are used to set computation and memory-allocation policies via on-statements and distributions
- While stabilizing the ‘locale’ interface, we considered its interaction with our newly added GPU support

**This Effort:**

- Deprecated ‘locale.callStackSize’
  - Rationale: rarely used, not applicable to new hardware types, precludes stack-size heterogeneity across individual tasks
- Marked ‘locale.numPUs’ as unstable
  - Rationale: in the context of new hardware (e.g., GPUs), it's not clear what should be considered a Processing Unit
- Improved the module documentation:
  - annotated method return types
  - improved accuracy of various method descriptions
ARRAY UPDATES
Arrays

Background:
- Arrays are supported by an internal module whose library-like features are being reviewed

This Effort:
- Marked `equals()` as being unstable
  - We have an `==` operator and are considering having a standalone `equals()`
- Made array methods `.count()`, `.find()`, `.reverse()`, `.sorted()` unstable
  - We previously thought about removing these but had second thoughts upon seeing the impact
- Removed deprecated support for `.front()` / `.back()`
- Removed deprecated `arrayIndicesAlwaysLocal` config param

Next Steps: Resolve the remaining open questions
- Should `idxType` return a tuple for multi-dimensional arrays? [#19141]
  - You can currently do this with standalone `index()` pseudo-type
- Where to put `.count()`, `.find()`, `.reverse()`, `.sorted()` [#18089]
  - Keep on array, deprecate, or move to other modules (e.g., move `find` to `Search`?)
DEPRECATING THE ‘<~>’ OPERATOR
**DEPRECATED THE ‘<~>’ OPERATOR**

**Background and This Effort**

**Background:** The ‘<~>’ binary operator supported reading and writing IO with one set of code

- Allowed for C++-style chained IO
  
  ```cpp
  readerOrWriter <~> nameStr <~> favInt <~> langStr; // can read/write “Taylor 42 python”
  ```

- Had the goal of reducing redundancy in writing read/write routines for user types

- Has not stood the test of time
  - Reading complex types often requires more care than simply reading a field at a time
  - Frustratingly, cannot use string literals with reading channels, which expect a mutable value
    ```cpp
    myReader <~> "name: " <~> nameStr; // OK when writing; when reading, get: ‘error passing "name: " to ref-intent’
    ```

**This Effort:** Deprecate the ‘<~>’ operator for channels and in the language

- Generally, replaced uses of ‘<~>’ with ‘read’ or ‘write’ method calls on the channel

- Generally, used the version of ‘read’ that accepts a type, since it throws on EOF
  - In the value version, would need to check the Boolean return value to detect EOF
DEPRECATED THE ‘<~>’ OPERATOR
Impact, Status, and Next Steps

**Status:** ‘<~>’ usage results in a deprecation warning in 1.28

**Next Steps:** Remove support entirely for 1.29
  - Continue deprecating IO features that combine reading and writing into a single function/method
DOCUMENTING BUILT-IN TYPES IN THE SPEC
MIGRATE DOCS FOR BUILT-IN TYPES
Background and This Effort

Background:
• The Built-In Types docs (generated via ‘chpldoc’) had entries describing features in the Language Specification – Spec chapters are written and maintained manually, and could get out of sync with module-based implementations

This Effort:
• Migrate more of the duplicate entries generated by ‘chpldoc’ into the Language Specification
MIGRATE DOCS FOR BUILT-IN TYPES

Status, Impact, and Next Steps

Status:
• New Language Specification pages have been created for 'Strings' and 'Bytes'
• Built-In docs for 'Tuples' and 'Ranges' have been merged into respective Language Specification pages

Impact:
• Prevents Language Specification docs from getting out of sync with type implementations

Next Steps:
• Migrate the remaining Built-In Types pages [#18027]
  – Remove the "Built-in Types and Functions" section itself
TASK INTENTS ON ‘THIS’
TASK INTENTS ON ‘THIS’
Background

• Explicit task intents on ‘this’ were not supported prior to 1.28

• When there is no explicit intent on ‘this’:
  • ‘this’ is passed by default task intent into a ‘forall’, yet it can be accessed directly in a ‘coforall’
  • when ‘this’ is a record: each field gets its own shadow variable passed by default intent into a ‘forall’ or ‘coforall’

```plaintext
record R {
    var x: real;
    var A: [D] int;
}
proc R.update() {
    forall idx in D
        // ‘with (ref this)’ and other explicit intents were not available in 1.27
        { this = new R(); // disallowed: in a forall loop ‘this’ is a shadow variable with default intent ‘const ref’
            this.x = idx; // disallowed: ‘this.x’ is a shadow variable with default intent ‘const in’
            this.A[idx] = 1; // okay: ‘this.A’ is a shadow variable with default intent ‘ref’
        }
}
```
**TASK INTENTS ON ‘THIS’**

This Effort

- Enabled explicit task intents on ‘this’
  - When an explicit intent on ‘this’ is present: fields of a record ‘this’ do **not** get their own shadow variables
  - When there is no explicit intent on ‘this’: no changes in behavior

```plaintext
record R {
    var x: real;
    var A: [D] int;
}
proc R.update() {
    forall idx in D
        with (in this) // available in 1.28: causes each task to get its own copy of ‘this’ to modify
    {
        this = new R(); // now allowed: ‘this’ is a shadow variable with explicit intent ‘ref’
        this.x = idx; // now allowed: modifying a field of a mutable shadow variable ‘this’
        this.A[idx] = 1; // okay: modifying a field of a mutable shadow variable ‘this’
    }
}
```
### Task Intents on ‘This’

#### Next Steps

- Allow customizing the default intent for a record at its declaration [#19211](#19211)
- Enable explicit intents for fields of ‘this’
- Switch ‘coforall’ loops to pass ‘this’ by default intent—to match ‘forall’

```plaintext
record R {
  var x: real;
  var A: [D] int;
}

proc R.update() {
  coforall idx in D
    // using the default intent for ‘this’
  {
    this = new R(); // currently allowed but should be disallowed because default task intent for record is ‘const ref’
    this.x = idx;
    this.A[idx] = 1;
  }
}
```
IGNORING UNREACHABLE CODE
Background and This Effort

**Background:** the following code produced a surprising error:

```pascal
proc numIndices(r: range(?)) {
    if r.isBounded() then
        return …;
    compilerError("r must be bounded");
}
writeln(numIndices(1..3));
```

This Effort: made the compiler ignore unreachable code, i.e.:

- after ‘return’, ‘throw’, ‘break’, ‘continue’, and calls to ‘halt()’ or ‘exit()’
- after ‘if’ statements containing the above
  – ignoring param-folded branches, if applicable
- unreachable code is ignored through the end of the current block
- exception: compiler does not ignore code after ‘throw’ or ‘halt()’ if it contains a ‘return’
  – this allows ‘return’ statement(s) to define implicitly-inferred return type
- compiler also does not ignore nested type and function declarations

should be unreachable when ‘r’ is bounded
yet the compiler generated "r must be bounded"
IGNORING UNREACHABLE CODE

Impact and Next Steps

**Impact:** behavior matches programmer’s intuition about unreachable code

```plaintext
proc createSameManagement(x) {
    if isOwnedClass(x) {
        return new owned MyClass2();
    }
    return new shared MyClass2();
}

var y = createSameManagement(new owned MyClass1());
```

**Next Steps:**

- Remove the exception for code containing ‘return’ after ‘throw’ or ‘halt()’
- Gather user feedback [#20673]
- Decide whether unreachable code should result in compilation errors or warnings
Background: Historically, Chapel has allowed a subclass field to shadow a superclass field, e.g.:

```chapel
class Base {
    var field: int;
}
class MyClass: Base {
    var field: real;
}
var c = new MyClass();
writeln(c.field); // printed 0.0
```

- Problematic because the two fields above have the same name and usage but different behavior

This Effort: Stopped allowing subclass fields to shadow superclass fields
- The compiler now emits a compilation error in the above case

Impact: A confusing pattern is no longer allowed
PARENTHESES-LESS METHODS
PAREN-LESS METHODS

Background

- Chapel allows methods to be declared without parentheses
  - These are called *parentheses-less methods* or *paren-less methods*
- Paren-less methods support changing the fields of a ‘class’ or ‘record’ to code without interface changes
  - Field access and paren-less method calls use the same syntax

- For example:

  ```plaintext
  record myRecord {
    proc parenless { return 15; }
  }
  var r: myRecord;
  writeln(r.parenless);  // outputs 15
  ```
**PAREN-LESS METHODS**

This Effort

- Noticed problems when a paren-less method on a subclass had the same name as a superclass field
- Improved ‘override’ checking for paren-less methods
  - ‘override’ now required when a paren-less method has the same name as a superclass field or paren-less method
- Improved convenience of paren-less methods that return a ‘type’ or ‘param’
  - These can now be invoked even on nilable classes without requiring ‘!’
  - This convenience was already available for ‘type’ or ‘param’ fields

```grit
class C {
    param field = 1;
    proc parenless param { return 2; }
}
```

```grit
var c: borrowed C? = nil;
writeln(c.field);        // has been OK: ‘c’ being ‘nil’ is irrelevant since ‘field’ is known at compile-time
writeln(c.parenless);   // now OK for similar reasons
```
PAREN-LESS METHODS

Impact

• Addressed some strange and surprising bugs observed when developing unrelated features
• Methods without parentheses are now more capable of replacing fields
RANGE IMPROVEMENTS
RANGE IMPROVEMENTS

Background and This Effort

**Background:**
- Ranges are a core type in Chapel
  - Represent regular integer sequences, as used in many for-loops
    ```chapel
    1..n  // represents the integers 1, 2, 3, ..., n
    ```
  - Used to define rectangular domains and arrays
- Range features are being reviewed for Chapel 2.0 stabilization, taking user feedback/experiences into account

**This Effort:**
- Addressed a few sticking points:
  - Flexibility of the ‘by’ and ‘#’ operators w.r.t. integer types
  - Queries of low/high bounds
  - Interpretation of unbounded ranges
- Also studied the impacts of other potential changes (not covered here)
RANGE IMPROVEMENTS

- ‘by’/‘#’ Flexibility
- Low/High Bound Queries
- Unbounded Loops
RANGES: ‘BY’ / ‘#’ FLEXIBILITY
Background:

- Every range has an ‘idxType’ that represents the type of the values it represents
  
  ```java
  const r = 1..10;
  ...r.idxType...  // evaluates to ‘int’
  ```

- The ‘by’ and ‘#’ operators can be applied to ranges to create new sub-ranges
  
  ```java
  ...r by 2...
  ...r#3...
  ```

- Traditionally, the bit-widths of the integer arguments for ‘by’ and ‘#’ have matched the range’s ‘idxType’
  - Historical rationale: similar to other operators, like ‘+’, whose formal arguments have matching bit-widths
  - Worked fine for the common case of ‘int’ ranges
  - Turned out to be frustrating for users with narrower ‘idxType’s (e.g., ‘range(int(8))’)

  ```java
  var r = 1..10:int(8);
  config const count = 2;
  ...r#count...
  ```

  - Note that such frustrations did not apply to ‘param’ values, due to support for downcasting

  ```java
  ...r#2...
  ```

  // OK, since the compiler will downcast ‘2’ to int(8)
RANGES: ‘BY’ / ‘#’ FLEXIBILITY
This Effort and Impact

This Effort:
- In Chapel 1.28, changed ‘by’ and ‘#’ operator definitions to accept any ‘integral’ value
- Effectively:
  \[
  \text{operator by}(r: \text{range}(?), \text{val}: r.\text{idxType}) \quad \Rightarrow \quad \text{operator by}(r: \text{range}(?), \text{val}: \text{integral})
  \]
  \[
  \text{operator } #(r: \text{range}(?), \text{val}: r.\text{idxType}) \quad \Rightarrow \quad \text{operator } #(r: \text{range}(?), \text{val}: \text{integral})
  \]
- New rationale:
  - Unlike ‘+’, the ‘by’ and ‘#’ operators are asymmetrical due to taking a range and an integer; more like methods on a range
  - Actual integer values passed to these operators matter far more than the bit-widths of their representation
- New out-of-bounds conditions result in halts
  - e.g., stride values that cannot be represented in the bit-width of the range's ‘idxType’

Impact:
- Ranges with small integer ‘idxType’s are now easier to use

```chapel
var r = 1..10:int(8);
config const count = 2;
...r#count...  // now OK!
```
RANGES: ‘BY’ / ‘#’ FLEXIBILITY

Next Steps

Next Steps:

• These operators currently accept ‘bool’ values—should they? [#20114]
  – Yes: Other operators, like ‘+’, accept bool, treating them as 0/1
  – No: Supports cases that arguably aren’t particularly compelling or motivating
    – Counting by ‘bool’ gives a sub-range of 0 or 1 elements
    – Illegal to stride by 0, and striding by 1 is uninteresting

• ‘integral’ arguments don’t currently accept ‘bool’ values—should they? [#20125]
  – Yes: ‘bool’ can be viewed as a 1-bit integer
  – No: Could enable usage patterns that a routine’s author did not intend or anticipate
RANGES:
LOW/HIGH BOUND QUERIES
RANGES: LOW/HIGH BOUND QUERIES

Background:

- Ranges (and domains) with non-unit strides potentially support two interpretations of their bounds; for example:
  
  ```chapel
  const r = 1..10 by 2
  // has a “pure” high bound of 10
  // has a “practical” or “aligned” high bound of 9
  ```

- For this reason, Chapel has supported two sets of queries:
  
  ```chapel
  ...r.high... // evaluated to 10, the “pure” high bound
  ...r.alignedHigh... // evaluated to 9, the “practical” or “aligned” high bound
  ```

- However, this has resulted in some brittle generic code when written with only unit-stride cases in mind:
  
  ```chapel
  proc last(A: [?D] ?t) {
    // return the last element in an array
    return A[D.high];
  }
  var A: [1..10 by 2] real;
  writeln(last(A)); // results in an out-of-bounds due to trying to access ‘A[10]’ here, which doesn’t exist
  ```

- Our experience is that users typically care about aligned bounds, yet most naturally reach for ‘.low’/.high’
**RANGES: LOW/HIGH BOUND QUERIES**

This Effort and Impact

**This Effort:**

- Updated these range and domain queries to better reflect typical usage
- In Chapel 1.27.0:
  - Introduced new `.lowBound`/`.highBound` queries to return the “pure” bounds of a range or domain
  - Added a warning for applications of `.low`/`.high` to strided cases to make users aware they’d start returning aligned values
  - Added a ‘config param’ for opting into the new behavior now
  - Updated our module code to use `.lowBound`/`.highBound` or `.alignedLow`/`.alignedHigh` in strided contexts
- In Chapel 1.28.0:
  - Changed `.low`/`.high` to return the aligned bounds
  - Deprecated the ‘config param’

**Impact:**

- Implements what we believe to be more intuitive interface for Chapel’s ranges
- Found and fixed previously unknown instances of module and test code that had been using the wrong bounds
RANGES: LOW/HIGH BOUND QUERIES

Status and Next Steps

Status:

• Ranges and rectangular domains now support three pairs of queries for bounds:
  
  ```
  ...r.lowBound... / ...r.highBound... // return pure bounds
  ...r.low...       / ...r.high...    // return aligned bounds
  ...r.alignedLow... / ...r.alignedHigh... // return aligned bounds
  ```

Next Steps:

• Decide whether to retain or deprecate `.alignedLow`/`.alignedHigh` [#20606]
  
  – Retaining them would permit programmers to make their queries very explicit / self-descriptive if desired
  – But, it could also cause confusion to readers of code
    
    – Potential user: “How is `.alignedHigh` different from `.high`?” [reads documentation] “Wait, it’s not?”
    
    – If we retained it, would we encourage a coding style that avoided `.low`/`.high`?
RANGES: UNBOUNDED LOOPS
RANGES: UNBOUNDED LOOPS

Background

Background:

- Chapel ranges can omit one or both bounds:
  
  - `1..`  // conceptually represents all positive integers
  - `..-1`  // conceptually represents all negative integers
  - `..`  // conceptually represents all integers

- Questions have come up about the interpretation of unbounded `bool` or `enum` ranges
  - For integer ranges, the lack of a bound has been considered a conceptual “infinity”
  - Yet, these types are inherently finite, so...

```plaintext
enum color {red, green, blue};
color.red..  // should this be equivalent to 'color.red..color.blue'?
false..      // should this be equivalent to 'false..true'?
```

- Without breaks/returns, for-loops over unbounded ranges iterated forever or generated out-of-bounds errors

```plaintext
for i in (1:uint(8)..) do  // spun forever, wrapping around after exceeding 'max(uint(8))'
  writeln(i);
for c in (color.red..) do  // printed 'red', 'green', 'blue' then got 'error: halt reached - enumerated type out of bounds...' 
  writeln(c);
```
**RANGES: UNBOUNDED LOOPS**

This Effort and Next Steps

---

**This Effort:**

- Improved behavior for serial loops over unbounded ranges
  - Loops over unbounded bool / enum ranges now stop once the values have been exhausted
    ```
    for c in (color.red..) do
        writeln(c);
    ```
    
  - Loops over unbounded integer ranges now halt if they are about to wrap around
    ```
    for i in (250:uint(8)..) do
        writeln(i);
    ```

**Next Steps:**

- Extend new interpretation of unbounded ‘bool’ / ‘enum’ ranges to other operations and methods as well
  ```
  ...false..).high... // should arguably evaluate to ‘true’
  ```
- Update specification of unbounded ranges to define behavior as being context-specific rather than ‘infinite’ (?)
- Potentially, improve loops over unbounded integer ranges to include the most extreme value
  - e.g., would be preferable if the ‘uint(8)’ example above yielded ‘255’ before halting
- Consider adding support for parallel loops over unbounded ranges
RANGES: WRAP-UP
RANGE IMPROVEMENTS
Status and Next Steps

Status:
• Improved some use cases that have emerged in user interactions and 2.0 stabilization

Next Steps:
• Address next steps from previous sections
• Resolve other range stabilization topics:
  – How generic of a type should ‘range’ be?
  – Expression of a range type’s boundedness and stridability
  – Role of alignment
  – Definition of range slicing for corner cases
IMPLICIT NUMERIC CONVERSIONS
IMPLICIT NUMERIC CONVERSIONS

Background

- The compiler supports implicit conversions (sometimes called coercions) between some numeric types
- For example, the program below uses an implicit conversion:

```plaintext
proc f(arg: int) { }
var myInt8: int(8);
f(myInt8); // 'myInt8' will implicitly convert from 'int(8)' to 'int' for this call
```

- There are additional implicit conversions from ‘param’ or literal values, for example:

```plaintext
proc g(arg: int(8)) { }
g(1); // '1' has type 'int' but can implicitly convert to 'int(8)' because it fits and is a compiler-known value (a 'param')
```

- These additional implicit conversions are allowed when the ‘param’ value fits into the destination type
- They are called ‘param’ narrowing conversions
IMPLICIT NUMERIC CONVERSIONS

Background

- The following simplified table shows the relevant implicit numeric conversions allowed in 1.26
  - the table does not include the ‘param’ narrowing conversions discussed on the previous slide

When is implicit conversion allowed from a value of one type to another type?

<table>
<thead>
<tr>
<th>From</th>
<th>To uint(t)</th>
<th>To int(t)</th>
<th>To real(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint(s)</td>
<td>s ≤ t</td>
<td>s &lt; t</td>
<td>s = t = 64 or s &lt; t</td>
</tr>
<tr>
<td>int(s)</td>
<td>s ≤ t</td>
<td></td>
<td>s = t = 64 or s &lt; t</td>
</tr>
<tr>
<td>real(s)</td>
<td></td>
<td></td>
<td>s ≤ t</td>
</tr>
</tbody>
</table>

- These rules were inspired by C#, yet Chapel did not always match C# behavior
- Note that the above rules do not allow:
  - ‘int(64)’ to implicitly convert to ‘real(32)’
  - ‘int(32)’ to implicitly convert to ‘real(32)’
  - ‘int(t)’ to convert to ‘uint(s)’ for any t or s
IMPLICIT NUMERIC CONVERSIONS

Problems with ‘+’

• Chapel users have complained about surprising behavior when working with ‘real(32)’:

  ```chapel
  var myInt32: int(32), myInt64: int(64), myReal32: real(32);
  myInt32 + myReal32 // results in a 'real(64)’—makes it hard to work with 32-bit values
  myInt64 + myReal32 // results in a 'real(64)’—makes it hard to work with 32-bit values since 'int(64)’ is the default integer
  ```

  This behavior is unlike other languages and surprising because it implicitly changes the ‘real’ width:
  • it messes up any error analysis that compares precision with 32-bit floating point to 64-bit floating point
  • could lead to a surprising lack of performance since 32-bit floating point is faster than 64-bit

• A similar surprising situation exists when combining ‘int’ and ‘uint’:

  ```chapel
  var myInt32: int(32), myUint32: uint(32), myInt64: int(64), myUint64: uint(64);
  myInt32 + myUint32 // results in an 'int(64)'
  myInt64 + myUint64 // error: illegal use of '+' on operands of type uint(64) and signed integer
  ```

• This behavior is also problematic:
  • Change in integer width from 32-bit to 64-bit can be surprising
  • The special error overload of ‘operator +’ interferes with patterns like ‘myUint8 + myUint64’
**IMPLICIT NUMERIC CONVERSIONS**

Problems with ‘+’: Explanation

- Built-in operators like ‘operator +’ generally consist of overloads for each numeric type, e.g.

  ```
  operator +(a: int(32), b: int(32)): int(32) { ... }
  operator +(a: int(64), b: int(64)): int(64) { ... }
  operator +(a: uint(32), b: uint(32)): uint(32) { ... }
  operator +(a: uint(64), b: uint(64)): uint(64) { ... }
  operator +(a: real(32), b: real(32)): real(32) { ... }
  operator +(a: real(64), b: real(64)): real(64) { ... }
  ```

- myInt32 + myReal32 // results in a ‘real(64)’
- myInt64 + myReal32 // results in a ‘real(64)’
- myInt32 + myUint32 // results in an ‘int(64)’

‘real(64)’ is the only type that both ‘int(32)’ and ‘real(32)’ could implicitly convert into.

Surprising behavior: it changes floating point width when it is not expected to.

‘int(64)’ is the only type that both ‘int(32)’ and ‘uint(32)’ could implicitly convert into.

Surprising behavior: it changes integer width when it is not expected to.
IMPLICIT NUMERIC CONVERSIONS
Problems with User-Defined Functions

• A surprising situation existed when creating ‘int’/’uint’/’real’ overloads:

  proc plus(a: int(32), b: int(32)): int(32) { ... }
  proc plus(a: int(64), b: int(64)): int(64) { ... }
  proc plus(a: uint(32), b: uint(32)): uint(32) { ... }
  proc plus(a: uint(64), b: uint(64)): uint(64) { ... }
  proc plus(a: real(32), b: real(32)): real(32) { ... }
  proc plus(a: real(64), b: real(64)): real(64) { ... }

  var myInt32: int(32);
  var myInt64: int(64);
  var myUint32: uint(32);
  var myUint64: uint(64);

  plus(myInt32, myUint32) // results in an ‘int(64)’
  plus(myInt64, myUint64) // results in a ‘real(64)’

• The built-in ‘min’/’max’ implementation had a similar problem and behaved in a surprising way
This Effort

- Sought to remove these surprising cases
- To do so, enabled more implicit numeric conversions to make the rules more consistent

When is implicit conversion allowed from a value of one type to another type?

<table>
<thead>
<tr>
<th>From</th>
<th>To uint(t)</th>
<th>To int(t)</th>
<th>To real(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint(s)</td>
<td>s ≤ t</td>
<td>s &lt; t</td>
<td>all s, t</td>
</tr>
<tr>
<td>int(s)</td>
<td>s ≤ t</td>
<td>s ≤ t</td>
<td>all s, t</td>
</tr>
<tr>
<td>real(s)</td>
<td></td>
<td></td>
<td>s ≤ t</td>
</tr>
</tbody>
</table>

• Some highlights of new implicit conversions:

  - int → uint
  - int → real(32)
  - int(32) → uint(32)
  - int(32) → real(32)

• Achieving these changes required reworking overload resolution rules (see next section)
**IMPLICIT NUMERIC CONVERSIONS**

**Impact (1/2)**

- Resolved the surprising behavior shown in the previous examples for `+`:

```
<table>
<thead>
<tr>
<th>x type</th>
<th>y type</th>
<th>x + y type in 1.28</th>
<th>x + y type in 1.27</th>
</tr>
</thead>
<tbody>
<tr>
<td>int(32)</td>
<td>real(32)</td>
<td>real(32)</td>
<td>real(64)</td>
</tr>
<tr>
<td>int(64)</td>
<td>real(32)</td>
<td>real(32)</td>
<td>real(64)</td>
</tr>
<tr>
<td>int(32)</td>
<td>uint(32)</td>
<td>uint(32)</td>
<td>int(64)</td>
</tr>
<tr>
<td>int(64)</td>
<td>uint(64)</td>
<td>uint(64)</td>
<td>error</td>
</tr>
<tr>
<td>uint(8)</td>
<td>uint(64)</td>
<td>uint(64)</td>
<td>error</td>
</tr>
</tbody>
</table>
```

- Also resolved the surprising behavior shown in the previous examples for a user-defined `plus`:

```go
var myInt32: int(32), myInt64: int(64), myUint32: uint(32), myUint64: uint(64);
plus(myInt32, myUint32) // now results in a 'uint(32)' rather than an 'int(64)'
plus(myInt64, myUint64) // now results in a 'uint(64)' rather than a 'real(64)'
```
IMPLICIT NUMERIC CONVERSIONS

Impact (2/2)

- Now easier to use narrow-width types, like ‘real(32)’ or ‘uint(32)’
- Chapel implicit conversions are now closer to C/C++
  - However, ‘int’ to ‘uint’ implicit conversions could seem to change the sign of a value in some cases
    - Ameliorated by an experimental warning for implicit conversions from a signed integral type to unsigned: ‘--warn-int-uint’
IMPLICIT NUMERIC CONVERSIONS

Next Steps

• Improve checking for when ‘int’ to ‘uint’ implicit conversion could seem to change the sign of a value
  • Should it be possible to enable runtime checking for it? [#20543]
  • Should it be possible to request a warning about this? [#20687]
  • Should it be possible to request a warning for all numeric implicit conversions? [#20687]

• ‘--warn-int-uint’ is available in 1.28 as an experimental warning for signed to unsigned conversions
  • This warning needs more design review before becoming user facing
OVERLOAD RESOLUTION
OVERLOAD RESOLUTION

Background

- **Overload resolution** chooses the *most specific candidate(s)* from the set of candidate overloads
  - This process is also called *disambiguation*

- For example, in the code below, there are two overloads of ‘f’

```java
proc f(arg: real) { }
proc f(arg: int) { }
```

- When resolving a call such as:
  ```java
  f(1)
  ```
  - The compiler first determines that the set of candidates are the two ‘proc f’ overloads above
  - Then, the compiler chooses the most specific candidate from those overloads, in this case:
    ```java
    proc f(arg: int) { }
    ```
OVERLOAD RESOLUTION
Problems

• The overload resolution rules had several problems in 1.27:

  • They were difficult to describe and difficult to implement
    – language specification used about 3 pages of dense text to describe them

  • They had an unusual design, as compared to other languages
    – the more specific relation included 4 levels of preference: strong/weak/weaker/weakest

  • Observed cases where the more specific relation was non-transitive
    – this was a serious problem since other algorithmic elements consider it a partial order

• Changing the implicit numeric conversions has required modifying these rules
  • adding implicit conversions adds candidates for overload resolution
  • changes to implicit numeric conversions described in the previous section are a recent example
OVERLOAD RESOLUTION
This Effort

- Adjusted the overload resolution rules to address the problems and to make them:
  - More understandable
  - More typical (as compared to other languages)
  - More sound — the *more specific* relation needs to be transitive
  - Less brittle in the face of changes to the implicit numeric conversions

- Added checking that the *more specific* relation is transitive with ‘--verify’

- Developed and then abandoned an initial effort to improve these rules
  - It used a heuristic to avoid ‘plus(myInt64, myUint64)’ resulting in a ’real(64)’
  - This heuristic seemed a bit too unprincipled
  - Allowing ‘int’ to ‘uint’ implicit conversions removed the need for the heuristic
OVERLOAD RESOLUTION
This Effort: Outline of Rules

• Discard each candidate...
  ...that is less visible than (or shadowed by) another candidate.
  ...that uses promotion if there is a candidate that does not use promotion
  ...with a less-specific argument mapping than another candidate
    – See the language specification for more details on more specific argument mappings
  ...with more formals requiring implicit conversion than another candidate
    – For this step, implicit conversions between ‘real(w)’, ‘imag(w)’, and ‘complex(2*w)’ are not considered.
  ...with more formals requiring negative-param-to-uint than another candidate
    – negative-param-to-uint is when a negative ‘param’ is converted to a ‘uint’ type of any width
  ...with more formals requiring param-narrowing than another candidate
    – param-narrowing is when a ‘param’ is converted to a narrower type only because it fits
    – For example, the param ’1’ of type ‘int’ can be passed to an ‘int(8)’ formal
OVERLOAD RESOLUTION

Impact

- Overload resolution rules are now more reasonable
  - Their description fits in ~2 pages rather than ~3
  - The *more specific* relation is more typical and transitive
  - Interactions among different rules are easier to reason about

- However, several programs have changed behavior
  - The following slides will show examples
  - It is possible to encounter these in applications, but we have observed that to be rare in practice
OVERLOAD RESOLUTION
Impact: Param Expression Behavior

• Some expressions consisting of mixed-type literal or ‘param’ values now have different behavior. E.g.:
  
  \[
  \text{1:} \text{int}(8) + 2 \quad \text{ // now results in an ‘int(64)’ rather than an ‘int(8)’}
  \]

• This change is caused by the rules change to consider ‘param’ in a separate step

• One positive aspect is that now the behavior matches the same expression with ‘var’s:

  \[
  \text{var one = 1;}
  \text{var two = 2;}
  \text{one:} \text{int}(8) + \text{two} \quad \text{ // the result type of ‘int(64)’ is the only one that makes sense here}
  \]

• Here are a few similar examples that behave differently

  \[
  \text{((-2):} \text{int}(32))^{53} \quad \text{ // now results in ‘int(64)’ rather than ‘int(32)’}
  \text{1:} \text{int}(8) \ldots 100: \text{uint}(8) \quad \text{ // now results in a range with index type ‘uint(8)’ rather than ‘int(16)’}
  \]
OVERLOAD RESOLUTION
Impact: Mixed int/uint overloads

• The following program now shows a change in behavior

```r
proc fn(a: int(8)) { ... }
proc fn(a: uint(64)) { ... }
fn(42:int(64)); // now calls the ‘uint(64)’ version rather than the ‘int(8)’ version
```

• This change is caused by the rules change to more strongly avoid ‘param’ narrowing conversion
OVERLOAD RESOLUTION
Impact: Function visibility and shadowing

• The following program now shows a change in behavior

```plaintext
proc f(arg: int) { ... }
proc main() {
  proc f(arg) { ... }
  f(1); // now calls the inner generic 'f' rather than the outer concrete 'f'
}
```

• This change is caused by the rules change to consider visibility first

• This change makes it less likely that changing a library will interfere with an application, e.g.,
  • 'f' might have been defined in an application that 'use's a library
  • then, the library adds an 'f' that is a better match
  • with this visibility adjustment, the 'f' from the application will be preferred, preserving the old behavior
  • otherwise, would expect an overload set error from the compiler
OVERLOAD RESOLUTION

Next Steps

• Address open design questions:
  • How should range literals defined with mixed-type ‘param’ bounds behave? [#20545]
  • Should overloads with implicit conversion be preferred over instantiating? [#20539]
  • When should shadowing occur for operator functions and methods? [#20540]
  • Could/should disambiguation work on uninstantiated functions? [#20649]

• The next slides describe a few of these in more detail
Open Issue: Range Literals

- Range literals using a non-default type use that type as the index type:
  
  ```
  1: int(8)..10  // index type is ‘int(8)’ before and after this effort
  ```

- This behavior no longer matches the behavior of ‘+’

  ```
  1: int(8) + 10  // now results in an ‘int(64)’ instead of an ‘int(8)’
  ```

- Should range literals be updated to match the behavior of ‘+’ in terms of inferring the index type?
Open Issue: Implicit Conversion vs Instantiation

• The overload resolution rules sometimes prefer an implicit conversion over instantiation

• For example, in this program, the second overload is called:

```plaintext
proc g(arg)       { writeln("in generic g"); }  
proc g(arg: real) { writeln("in real g"); } 
g(1); // currently, calls 'proc g(arg: real)'
```

• But, specifying the generic type ‘integral’ in the first overload causes it to be preferred

• This behavior differs from both C++ and C# behavior

• We are considering changing it to match C++ / C# in this regard

• Have not yet evaluated this change in terms of its impact on tests
MODULE SCOPING
**SCOPING**

*Background*

- *Shadowing* is when a more closely scoped variable or routine hides other variables or routines

- Here is an example with variables:

  ```
  var x: string;
  {
    var x: int; // this ‘x’ is said to ‘shadow’ the outer ‘x’
    ...x... // here, ‘x’ refers to the ‘var x: int’ from the preceding line
  }
  ```

- This is an example with functions:

  ```
  proc f() { ... }
  {
    proc f() { ... } // this ‘f’ is said to shadow the outer ‘f’
    f(); // here, ‘f( )’ calls the inner ‘proc f’ from the preceding line
  }
  ```
A *shadow scope* is an intermediate scope considered just outside of the one containing the declarations.

‘use’ statements work with two shadow scopes:

- one for the names of ‘use’d modules
- one for the contents of ‘use’d modules

For example:

```plaintext
module M {
    var x = "M.x"
}
module Program {
    use M;
    var x = "Program.x"
    ...x...
}
```

// compiler's view within module Program
{
    // outer shadow scope for ‘use’d module names
    symbol M -> module M
    {
        // inner shadow scope for contents of 'use'
        symbol x -> M.x
        {
            // scope containing regular module-level declarations
            symbol x -> Program.x
        }
    }
}

In other words, the shadow scope is what causes the mention of ‘x’ above to refer to ‘Program.x’

- rather than being an ambiguity due to ‘M.x’ also being visible
**SCOPING**

The First Problem

- Shadowing behaved differently for functions and variables
- For example:

```plaintext
module M {
    var X = "M.X";
    proc f() { writeln("M.f"); }
}

module N {
    var X = "N.X";
    proc f() { writeln("N.f"); }
    proc main() {
        use M;
        writeln(X);  // referred to M.X
        f();        // resulted in an ambiguity error
    }
}
```
The Second Problem

- The structure of ‘public use’ statements impacted shadowing
- In the following example, ‘Dependency.X’ was preferred, because the path to it is shorter
  - problematic because the internal structure of ‘Library’ and its helper modules is an implementation detail
  - better to have it result in ambiguity

```plaintext
module Library {  
    public use LibraryImpl, Dependency;
}

module LibraryImpl {  
    public use LibraryDetail;
}

module LibraryDetail {  
    var X = "LibraryDetail.X";
}

module Dependency {  
    var X = "Dependency.X";
}

module Application {  
    proc main() {  
        use Library;  
        writeln(X); // used to refer to Dependency.X  
        // Library → LibraryImpl → LibraryDetail.X == 3 hops  
        // Library → Dependency.X == 2 hops, so it won
    }
}
```
SCOPING
This Effort

- Identified problems with the language design of shadowing
  - many of these issues were identified when implementing scope resolution in ‘dyno’

- Simplified the shadowing rules and made them more consistent between variables and functions
  - removed the need to consider the ‘use’ / ‘import’ structure for dependencies
  - now a module only provides a single, flat, view of public symbols — a bill of sale

- In particular, made the following language design choices:
  - removed shadow scopes for ‘import’ and ‘public use’
  - kept shadow scopes for ‘use’ and ‘private use’
  - adjusted ‘public use’ to no longer bring in the module name
  - adjusted overload resolution to no longer consider methods as subject to shadowing

- We will discuss each of these in the following slides
SCOPING

Removed shadow scopes for ‘import’

• 'import' / 'private import' / ‘public import’ no longer introduce a shadow scope
• ‘import’ed symbols now behave more similarly to locally-defined ones
• Supports the bill of sale view

```plaintext
module M {
    import N.x;

    var x = "M.x";

    proc main() {
        writeln(x); // now a multiply-defined symbol error instead of referring to M.x
    }
}

module N {
    var x = "N.x";
}
```
SCOPING

Removed shadow scopes for ‘public use’

• ‘public use’ no longer introduces a shadow scope
• Supports the bill of sale view

```plaintext
module M {
  public use N;

  var x = "M.x";

  proc main() {
    writeln(x); // now a multiply-defined symbol error instead of referring to M.x
  }
}

module N {
  var x = "N.x";
}
```
SCOPING
Kept shadow scopes for ‘use’ and ‘private use’ (1/3)

• ‘private use’ and its shorter synonym ‘use’ still use two shadow scopes
  • it has been this way since 1.19, at least, but not fully documented

• The next two slides show the rationale with two examples
SCOPING
Kept shadow scopes for ‘use’ and ‘private use’ (2/3)

- The shadow scope allows a library to add symbols and be less likely to break code that ‘use’s it
  - one key case is the automatic modules in the standard library
- Below, ‘X’ is used in ‘Program’, and that still works when ‘Library’ gains a symbol with the same name

```plaintext
module Library {  
  // version 1
  ...
}

module Library {  
  // version 1.1 — additive change should not be breaking
  var X = "Library.X";
  ...
}

module Program {  
  use Library;
  var X = "Program.X";
  ...X... // refers to Program.X
}

module Program {  
  use Library;
  var X = "Program.X";
  ...X... // still refers to Program.X
}
```
• The second shadow scope for the module name supports modules with the same name as a class

```rust
module MyDist {
  var x: int;
  class MyDist {
  }
}
module Program {
  use MyDist;
  // refers to the class rather than the module
  new MyDist()...
}
```

• This pattern has worked historically, with some caveats — including inhibiting qualified access

```rust
module Program2 {
  use MyDist;
  // error: ‘MyDist’ here refers to the class, not the module
  MyDist.x...;
}
```

• Users coming from other programming languages expect to be able to use this pattern
Adjusted ‘public use’ to no longer bring in the module name

- Since ‘public use’ exists to enable a kind of bulk re-export, it does not bring in the module name
- That way, internal details of helper modules can be hidden as an implementation detail
- You can opt in to bringing in the module name by adding an ‘as’ clause:
  ```
  public use M as M;  // ‘as’ form opts in to bringing in the name ‘M’
  ```
- Or by separately importing the module name:
  ```
  public import M;  // bring in the module name ‘M’ only
  public use M;      // bring in everything else from the module
  ```
- This design allows the example from the previous slide to work even with ‘public use’:
  ```
  module MyDist {
    class MyDist { }
  }
  module Program {
    public use MyDist;
    ...MyDist...  // refers to the class rather than the module
  }
  ```
SCOPING
Adjusted overload resolution to no longer consider methods as subject to shadowing

• Methods are no longer subject to shadowing
• For example, the program below now results in an ambiguity error
  • If we revisit this decision, changing from ambiguity to something else would be a non-breaking change

```plaintext
module Library {
    record rec {
        proc method() { writeln("Library's rec.method()"); }
    }
}

module Program {
    use Library;  // note: import Library; import Library.rec; has equivalent behavior
    proc rec.method() { writeln("Program's rec.method()" ); }
    proc main() {
        var r = new rec();
        r.method();  // now ambiguity instead running Program's rec.method()
    }
}
```
SCOPING

Impact

• Shadowing is now more consistent between variables and functions
• Scoping behavior is simplified and more predictable
• ‘use’ and ‘import’ statements are better at hiding implementation details
SCOPING

Next Steps

- Resolve open design questions about automatically included modules:
  - should it be possible to explicitly ‘use’/‘import’ the automatic modules? [#19313]
    - this is a feature that could probably be added in a non-breaking way
  - should it be possible to define a module that shadows an automatically included symbol? [#19312]
    - it was possible before 1.27 but it is not currently possible
    - perhaps it would only be possible when explicitly ‘use’ing/‘import’ing the automatic modules

- Resolve an open design question about ‘use someEnum’:
  - should ‘use someEnum’ create a shadow scope? [#19367]
    - It does today, and it would be a breaking change if we changed it

- Implement some warnings for potentially confusing cases
  - need warnings for differences between public and private use [#19780]
OTHER LANGUAGE IMPROVEMENTS

For a more complete list of language changes and improvements in the 1.27.0 and 1.28.0 releases, refer to the following sections in the CHANGELOG.md file:

- ‘Semantic Changes / Changes to the Chapel Language’
- ‘Deprecated / Unstable / Removed Language Features’
- ‘New Features’
- ‘Feature Improvements’
- ‘Error Messages / Semantic Checks’
- ‘Documentation’
THANK YOU

https://chapel-lang.org
@ChapelLanguage