Hewlett Packard
Enterprise

CHAPEL 1.23 RELEASE NOTES: LANGUAGE IMPROVEMENTS

Chapel Team

October 15, 2020

OUTLINE

- Point of Instantiation Improvements
- Method Resolution
- ‘use’ and ‘import’ Improvements
- Array Copy Initialization
- Array Noinit
- Other Language Stabilization Efforts

POINT OF INSTANTIATION (POI) IMPROVEMENTS

POINT OF INSTANTIATION

Background: POI Rule Prior to 1.23

- The POI rule applies when resolving a function call ‘fn(…)’ in a generic function (*GF*)
  - The POI is a call site of the *GF* for which it is instantiated w.r.t. its generic arguments
  - Visible functions for the call ‘fn(…)’ include those visible at the POI

  … and transitively at the POI of the generic function containing the call site of *GF*, if applicable

```
record MyR {...}
proc <(l:MyR, r:MyR) {...}
var A: [D] MyR;
use Sort;
sort(A);

module Sort {
  proc sort(Data: []) {
    quickSort(Data); }
  proc quickSort(Data: []) {
    ... if Data[i] < Data[j] then ...; }
}
```

(1) **proc <** is not visible in lexical scope
⇒ look it up at POI for ‘quickSort()’

POINT OF INSTANTIATION

Background: POI Rule Prior to 1.23

- The POI rule applies when resolving a function call ‘fn(…)’ in a generic function (*GF*)
  - The POI is a call site of the *GF* for which it is instantiated w.r.t. its generic arguments
  - Visible functions for the call ‘fn(…)’ include those visible at the POI

  … and transitively at the POI of the generic function containing the call site of *GF*, if applicable

```
record MyR {...}
proc <(l:MyR, r:MyR) {...}
var A: [D] MyR;
use Sort;
sort(A);

module Sort {
  proc sort(Data: []) {
    quickSort(Data); }
  proc quickSort(Data: []) {
    ... if Data[i] < Data[j] then ...; }
}
```

POI for ‘quickSort()’

POINT OF INSTANTIATION

Background: POI Rule Prior to 1.23

- The POI rule applies when resolving a function call ‘fn(…)’ in a generic function (*GF*)
  - The POI is a call site of the *GF* for which it is instantiated w.r.t. its generic arguments
  - Visible functions for the call ‘fn(…)’ include those visible at the POI

  … and transitively at the POI of the generic function containing the call site of *GF*, if applicable

```
record MyR {...}
proc <(l:MyR, r:MyR) {...}
var A: [D] MyR;
use Sort;
sort(A);

module Sort {
  proc sort(Data: []) {
    quickSort(Data); }
  proc quickSort(Data: []) {
    ... if Data[i] < Data[j] then ...; }
}
```

(2) **proc <** is not visible from ‘sort()’ either
⇒ look it up at POI for ‘sort()’

POINT OF INSTANTIATION

Background: POI Rule Prior to 1.23

- The POI rule applies when resolving a function call ‘fn(…)’ in a generic function (*GF*)
  - The POI is a call site of the *GF* for which it is instantiated w.r.t. its generic arguments
  - Visible functions for the call ‘fn(…)’ include those visible at the POI

  … and transitively at the POI of the generic function containing the call site of *GF*, if applicable

```
record MyR {...}
proc <(l:MyR, r:MyR) {...}
var A: [D] MyR;
use Sort;
sort(A);

module Sort {
  proc sort(Data: []) {
    quickSort(Data); }
  proc quickSort(Data: []) {
    ... if Data[i] < Data[j] then ...; }
}
```

POI for ‘sort()’

POINT OF INSTANTIATION

Background: POI Rule Prior to 1.23

- The POI rule applies when resolving a function call ‘fn(…)’ in a generic function (*GF*)
  - The POI is a call site of the *GF* for which it is instantiated w.r.t. its generic arguments
  - Visible functions for the call ‘fn(…)’ include those visible at the POI

  … and transitively at the POI of the generic function containing the call site of *GF*, if applicable

```
record MyR {...}
proc <(l:MyR, r:MyR) {...}
var A: [D] MyR;
use Sort;
sort(A);

module Sort {
  proc sort(Data: []) {
    quickSort(Data); }
  proc quickSort(Data: []) {
    ... if Data[i] < Data[j] then ...; }
}
```

(3) **proc <** on ‘MyR’ is now visible
⇒ use it for ‘Data[i] < Data[j]’

POINT OF INSTANTIATION

Background: POI Rule Prior to 1.23

- The Point of Instantiation (POI) rule also:
  - Chose a single POI arbitrarily among all call sites instantiating the *GF* with the same generic arguments
  - **Shared the instantiation** among all these call sites
- Sharing caused surprising, undesirable behavior

```
record MyR {...}
module User1 {
  proc <(l:MyR, r:MyR) {...}
  var A1: [D] MyR;
  sort(A1);
}
module User2 {
  proc <(l:MyR, r:MyR) {...}
  var A2: [D] MyR;
  sort(A2);
}
```

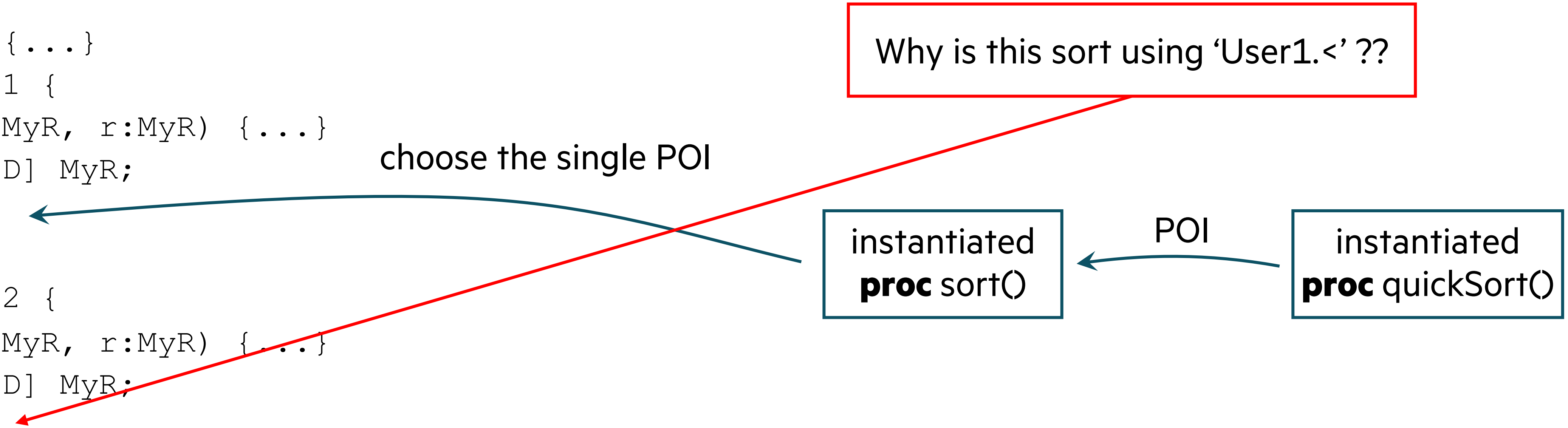
Why is this sort using ‘User1.<’ ??

POINT OF INSTANTIATION

Background: POI Rule Prior to 1.23

- The Point of Instantiation (POI) rule also:
  - Chose a single POI arbitrarily among all call sites instantiating the *GF* with the same generic arguments
  - **Shared the instantiation** among all these call sites
- Sharing caused surprising, undesirable behavior

```
record MyR {...}
module User1 {
  proc <(l:MyR, r:MyR) {...}
  var A1: [D] MyR;
  sort(A1);
}
module User2 {
  proc <(l:MyR, r:MyR) {...}
  var A2: [D] MyR;
  sort(A2);
}
```



Why is this sort using ‘User1.<’ ??

choose the single POI

instantiated **proc** sort()

POI

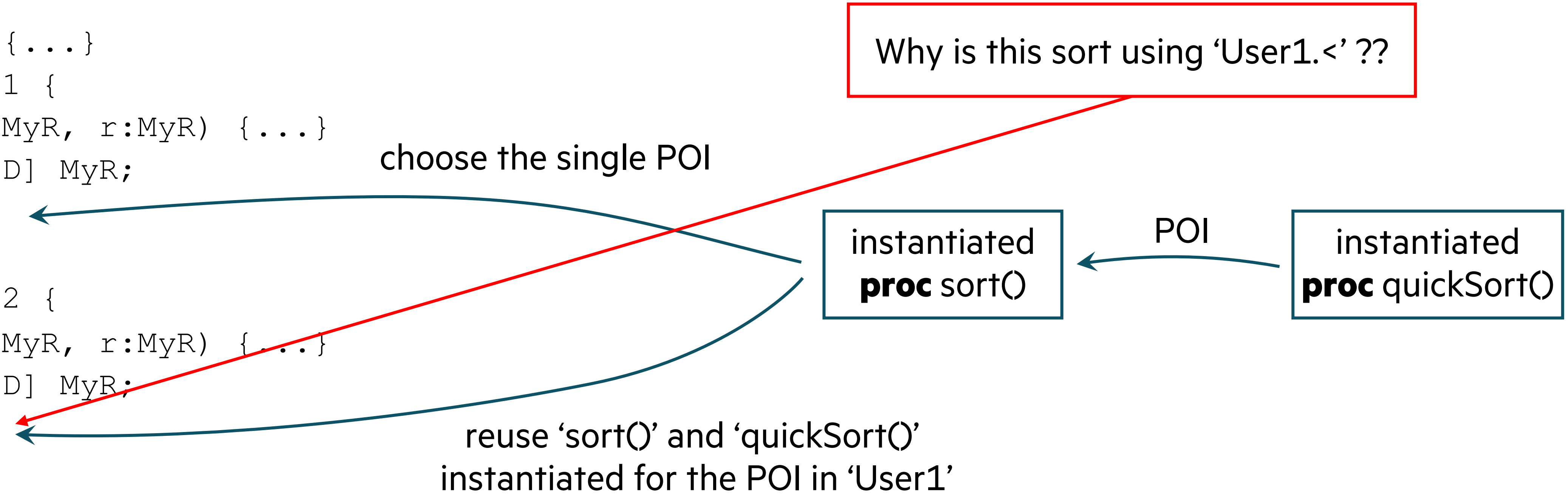
instantiated **proc** quickSort()

POINT OF INSTANTIATION

Background: POI Rule Prior to 1.23

- The Point of Instantiation (POI) rule also:
  - Chose a single POI arbitrarily among all call sites instantiating the *GF* with the same generic arguments
  - **Shared the instantiation** among all these call sites
- Sharing caused surprising, undesirable behavior

```
record MyR {...}
module User1 {
  proc <(l:MyR, r:MyR) {...}
  var A1: [D] MyR;
  sort(A1);
}
module User2 {
  proc <(l:MyR, r:MyR) {...}
  var A2: [D] MyR;
  sort(A2);
}
```



Why is this sort using 'User1.<' ??

choose the single POI

instantiated
proc sort()

POI

instantiated
proc quickSort()

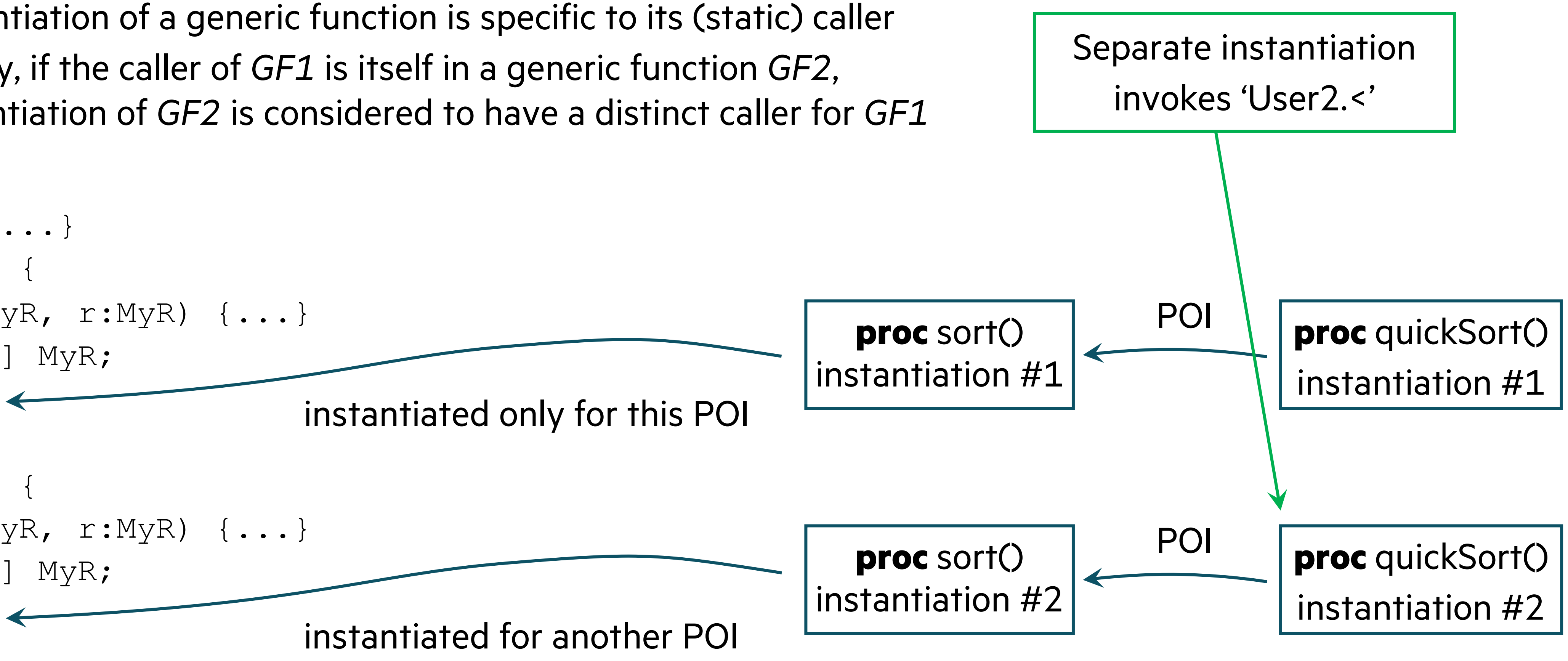
reuse 'sort()' and 'quickSort()'
instantiated for the POI in 'User1'

POINT OF INSTANTIATION

This Effort: Call-specific Instantiations

- The Point of Instantiation (POI) rule now **disallows reuse**:
  - Each instantiation of a generic function is specific to its (static) caller
  - Transitively, if the caller of *GF1* is itself in a generic function *GF2*, each instantiation of *GF2* is considered to have a distinct caller for *GF1*

```
record MyR {...}
module User1 {
  proc <(l:MyR, r:MyR) {...}
  var A1: [D] MyR;
  sort(A1);
}
module User2 {
  proc <(l:MyR, r:MyR) {...}
  var A2: [D] MyR;
  sort(A2);
}
```



Separate instantiation invokes ‘User2.<’

POI
POI
proc sort() instantiation #1
proc sort() instantiation #2
proc quickSort() instantiation #1
proc quickSort() instantiation #2

instantiated only for this POI

instantiated for another POI

POINT OF INSTANTIATION

This Effort: Clarified Language Design

- Clarified preference for callee context over caller context and improved implementation:
  - Search at POI(s) only if applicable candidates are **not** found at the lexical scope of the call
  - Once found, do not visit further POIs, if any

```
module M1 {
  use M2;
  proc bar() {}
  callFB2("M1");
}
```

```
module M2 {
  use M3;
  proc foo() {}
  proc bar() {}
  proc callFB2(arg) {
    callFB(arg);
  }
}
```

```
module M3 {
  proc foo() {}
  proc callFB(arg) {
    foo();
    bar();
  }
}
```

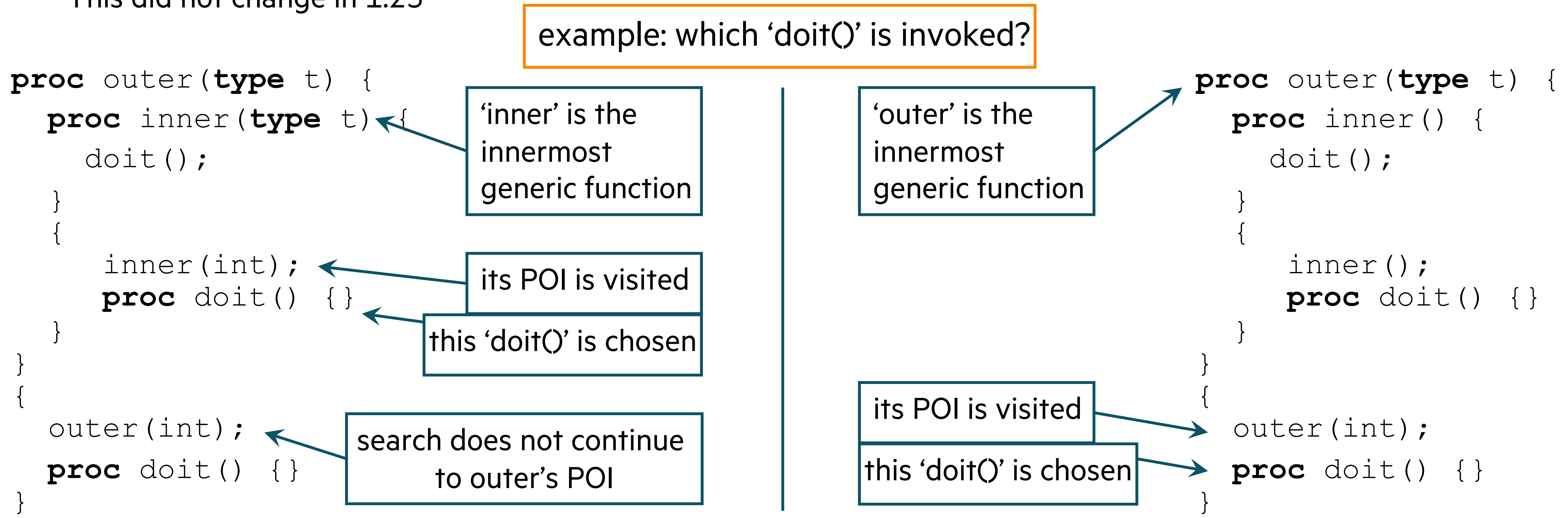
‘M3.foo’ is a candidate, so do not search at POI(s)

- Search at POI, which is in ‘M2.callFB2’
- There, ‘M2.bar’ is a candidate
- So, do not search at callFB2’s POI, which is in M1

POINT OF INSTANTIATION

This Effort: Clarified Language Design

- In the event a call is within a nested function:
  - Use the POI of the innermost generic function
  - If there is no enclosing generic function, then there is no POI and no search at POI
    - This did not change in 1.23



example: which ‘doit()’ is invoked?

```
proc outer(type t) {
  proc inner(type t) {
    doit();
  }
  {
     inner(int);
     proc doit() {}
  }
}
{
  outer(int);
  proc doit() {}
}
```

‘inner’ is the innermost generic function

its POI is visited

this ‘doit()’ is chosen

search does not continue to outer’s POI

```
proc outer(type t) {
  proc inner() {
    doit();
  }
  {
     inner();
     proc doit() {}
  }
}
{
  outer(int);
  proc doit() {}
}
```

‘outer’ is the innermost generic function

its POI is visited

this ‘doit()’ is chosen

POINT OF INSTANTIATION

Impact

- Improved handling of some scenarios with potential function hijacking
- Estimated <~5% compilation time increase
- Two benchmarks needed adjustments:
  - A local overload of ‘min()’ was used to change the behavior of the predefined min-reduction
    - In a variant of the ‘meteor’ CLBG benchmark
    - Arguably a form of function hijacking and an undesirable pattern
    - Adjustment: changed the input data to use the predefined min-reduction as-is
    - Alternatively, could apply a user-defined reduction

  - An overload of ‘+=’ on a user record type was used in a benchmark-specific AccumStencilDist distribution
    - In an elegant variant of CoMD
    - Adjustment: changed the AccumStencilDist distribution to invoke a distinctly-named function if provided
      - otherwise, it defaults to ‘+=’

  - Details and other adjustment choices in [issue 15948]

POINT OF INSTANTIATION

Impact, Discussion

- The CoMD case highlights an interesting scenario:

```
module Library {
  proc accumulate(ref lhs, rhs) { lhs += rhs; }
  proc updateFluff(A) {
    ... accumulate(A[i],A[j]) ...
  }
}
record MyR {...}
proc accumulate(ref lhs:MyR, rhs:MyR) {...}
var A: [D] MyR;
use Library;
updateFluff(A);
```

Intention: provide the default implementation

Intention: use caller’s implementation when available

The default implementation is always preferred by the new rules:

* It is visible from the lexical scope
* It is always applicable
* So, no search at POI(s)

- This scenario is currently addressed with a wrapper that checks for a user implementation
- Constrained generics will provide better support for this scenario

POINT OF INSTANTIATION

Status and Next Steps

Status:

- Implemented in 1.23
  - Exception: the previous implementation is used when resolving calls to ‘init()’ and ‘deinit()’
  - Compiler reuses instantiations when legal

Next Steps:

- Gain experience with the revised rule
- Define and implement the desired POI rules for special functions:
  - For ‘init’ and ‘deinit’, which currently use the old strategy
  - For ‘init=’, ‘_cast’, ‘+=’, and some others, which currently use the new strategy
- Implement constrained generics
  - A better way to write many of the codes that rely on the POI rule today

METHOD RESOLUTION

METHOD RESOLUTION

Background

- We’ve wrestled with how ‘use’ and ‘import’ impact the visibility of methods

- Typically have focused on whether the type’s scope was visible via ‘use’ or ‘import’ statements
  - Even going so far as to ignore whether the ‘use’/‘import’ was private or excluded the type

- But we encountered a case where it was reasonable to want to call methods when the type wasn’t visible
  - Here, the ‘use’ which enabled us to get an instance of R is not visible when you leave createR’s body

```
proc createR() {
  use One;
  var res = new R();
  return res;
}
var y = createR();
y.method1(); // Failed to resolve.  ‘R’ is defined in the module ‘One’, which is not visible to this scope
```

```
module One {
  record R {…}
  proc R.method1() {…}
}
```

METHOD RESOLUTION

This Effort: Type Definition Point

- Started resolving method calls by searching the type's definition point first
  - Now if you have an instance, you'll always be able to call methods that are defined in the type's scope

```
proc createR() {
  use One;
  var res = new R();
  return res;
}
var y = createR();
y.method1(); // Now works!
```

```
module One {
  record R {…}
  proc R.method1() {…}
}
```

  - Tertiary methods (secondary methods defined in other modules) still rely on 'use' and 'import' statements

- This new rule covered a lot of cases handled by previous rules
  - Though it doesn't cover all those old cases on its own

METHOD RESOLUTION

This Effort: Language Design Questions

- We considered how various adjustments to the rules would affect the language design
  - Should we be able to find methods when they're only available behind private uses or imports?
    - Or uses/imports that don't bring the type in explicitly?

```
module M {
  use One;
  proc R.extension() {…}
  proc otherFunc(): R {…}
}
use M only otherFunc;
var x = otherFunc();
x.extension();  // Should this continue to resolve?  Even though the 'use M' doesn't say anything about 'R'?
```

```
module One {
  record R {
    var x;
  }
}
```

  - Should listing the type in a use/import limitation clause impact all symbols relating to the type?
    - E.g. '+', 'initCopy', etc.

METHOD RESOLUTION

This Effort: Current Direction

- Decided:
  - To return to honoring the privacy and limitations of ‘use’ and ‘import’ statements when resolving methods
    - Behavior of ‘private use’ will be consistent and easily explainable
  - That listing a type in a limitation clause will impact visibility of its tertiary methods defined in that module
  - Still need to implement these changes
- Should we:
  - Search the type definition point for methods and operators?
    - Currently this is done for methods but not operators
  - Continue to **not** bring in related operators in the same scope when the type is brought in by a limitation clause?

METHOD RESOLUTION

Impact and Next Steps

Impact:

- An instance is now guaranteed to be able to call its methods that were defined at its declaration point
- The language will be more consistent and explainable with these rules
- The more consistent rules lend greater confidence to language stabilization

Next Steps:

- Finalize decisions on operators
- Adjust implementation for final decisions

‘USE’ AND ‘IMPORT’ IMPROVEMENTS

‘USE’ AND ‘IMPORT’ IMPROVEMENTS

Background

- The 1.21 and 1.22 releases included a number of namespace and module improvements, including:
  - The new ‘import’ statement
  - The introduction of “re-exporting” as a concept for ‘public import’ statements
  - The alteration of the default privacy of ‘use’ statements to be ‘private’ instead of ‘public’
  - The introduction of ‘this.’ and ‘super.’ prefixes for relatively referencing symbols
    - E.g. ‘import this.M’ when M is a submodule of the current module
  - The new requirement that a submodule must ‘use’ or ‘import’ its parent module before accessing its symbols

- These were all good changes, but work remained

‘USE’ AND ‘IMPORT’ IMPROVEMENTS

This Effort

- We focused on finishing these features for the 1.23 release, e.g.
  - Allowed ‘use’ to disable qualified access

```
use A as _;      // Renaming to ‘_’ means ‘A’ is not brought into scope
writeln(x);      // Thus, this is okay if ‘A’ defines ‘x’
writeln(A.x);    // But this is not, because ‘A’ is not brought into scope
```

  - Extended re-exporting to also apply to ‘public use’ statements

```
module Other {
  import Foo;
  // ‘Bar’ now can be treated like a submodule of ‘Foo’
  writeln(Foo.Bar.x);
}
```

```
module Bar {
  var x: int;
}
module Foo {
  public use Bar;
}
```

‘USE’ AND ‘IMPORT’ IMPROVEMENTS

This Effort

- Extended ‘import’ to support multiple expressions, e.g.

```
import A, B.x, C.{one, two, three};
```

- Fixed some bugs, including allowing ‘import super.foo’ when ‘foo’ is not a module symbol

```
module Outer {
  var foo = 7;
  module Inner {
    import super.foo; // Now works!
    writeln(foo);
  }
}
```

- Improved the ‘Modules’ chapter of the language spec w.r.t. ‘use’ and ‘import’
  - https://chapel-lang.org/docs/1.23/language/spec/modules.html

‘USE’ AND ‘IMPORT’ IMPROVEMENTS

Status and Next Steps

Status:

- ‘use’ and ‘import’ statements are now considered stable

Next Steps:

- Improve the situation when using a name that corresponds to a private function (see issue #14535)
  - Not valid to use a private function
  - But there may be a better match we could find
  - And the error message should be improved regardless

```
module A {
  use B;
  use C; // Today gives: “Error: ‘use’ of non-module/enum symbol C”
  // Should probably ignore the function because it is ‘private’
}
```

```
module B {
  var x: int;
  private proc C() {…}
}
```

ARRAY COPY INITIALIZATION

ARRAY COPY INITIALIZATION

Background

- In 1.22 and earlier, array variables declared with a type and an initializer used default-init-then-assign

```
var A: [1..n] int = 1..n;
// translates into default-init-then-assign:
var A: [1..n] int = 0;  // default initialize
A = 1..n;  // assign
```

- Led to different behavior with typed vs. untyped array variable declarations:

```
var B = A;  // copy-initializes elements
var C: [1..n] int = A;  // default-init-then-assign
var D = {1..n};
var E = createArrayWithDomain(D);   // no copy occurs
var F: [D] int = createArrayWithDomain(D);   // default-init-then-assign - lost opportunity for copy elision
```

  - The difference is observable for arrays containing record elements
    - Because instead of 'R.init=', each element is initialized with 'R.init' and then '='

- Split-init and copy-elision did not benefit arrays as much as they could

ARRAY COPY INITIALIZATION

Background: Runtime Types

- The main challenge with improving the situation is handling runtime types
- A typed array declaration generally specifies the runtime type of the array:

```
var A: [1..n] int = ...;
// This declaration indicates that A’s domain is {1..n}. This domain forms part of A’s runtime type.

var BDom = {0..n-1};
var B: [BDom] int = A;
// B must be initialized so that its domain is BDom
// B’s runtime type includes BDom
```

- The default-init-then-assign strategy handles these runtime types correctly

ARRAY COPY INITIALIZATION

This Effort

- Adjust implementation to remove default-init-then-assign for arrays
  - To bring array behavior closer to the expected behavior for other types
  - To enable language stabilization
- Copy array elements on array copy; move array elements on array move
  - While preserving array runtime types (note: move initialization supports copy elision)
- For example:

```
var B = A;  // allocates new array and copy-initializes elements
var C: [1..n] int = A;  // 'A' is dead, so copy is elided and elements are move initialized
var D = {1..n};
var E: [D] int = createArrayWithDomain(D);  // move-initializes array
var F: [D] int = createArrayWithDomain({0..n-1});  // move-initializes elements (see next slide)

proc createArrayWithDomain(D) {
  var ret: [D] int;
  return ret;
}
```

ARRAY COPY INITIALIZATION

This Effort: Allocations

- Allocations are now avoided in some cases
  - When move-initializing a new array with the same domain variable, re-use the existing element storage

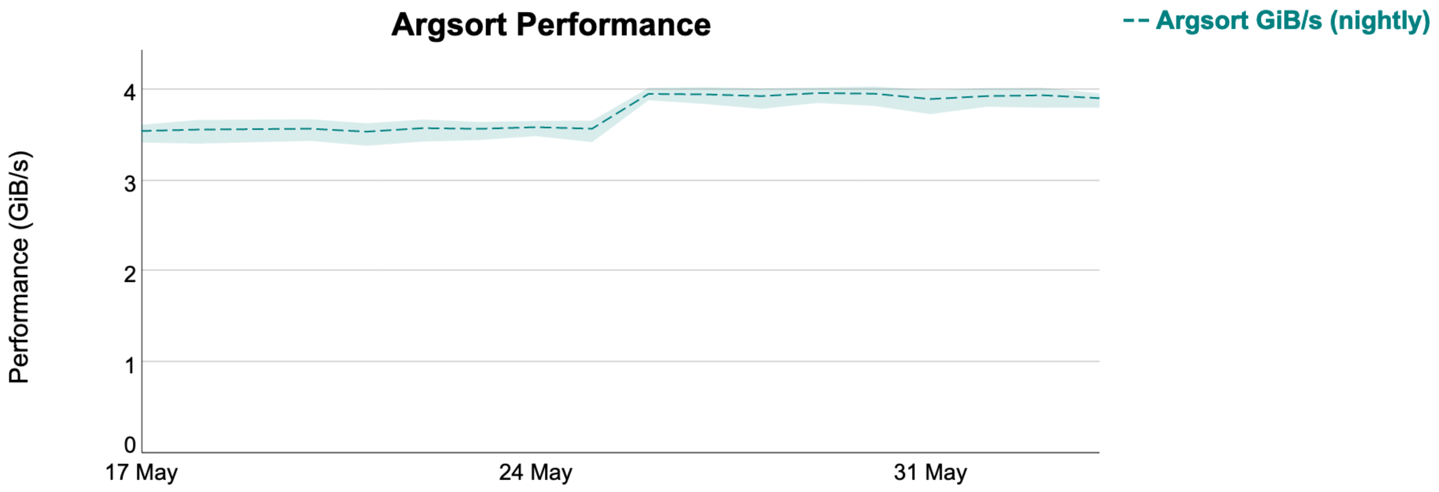
```
var D = {1..n};
var E:[D] int = createArrayWithDomain(D);  // same domain, so re-uses element storage and metadata
var F:[1..n] int = createArrayWithDomain({1..n}); //  different domain, so allocates new element storage
```

- Could reuse the element storage when move-initializing arrays with compatible type and shape
  - In initialization of ‘F’, there is room for ‘n’ elements in the source and destination buffers
  - Additionally, ‘F’ and the result of ‘createArrayWithDomain’ are the same array type
  - So, no need to allocate new storage for elements

ARRAY COPY INITIALIZATION

Impact

- Record behavior within arrays is closer to optimal
- Further optimization of array move initialization is possible without changing program behavior
- Provided a 10% improvement to Arkouda Argsort on 16-node XC



ARRAY COPY INITIALIZATION

Next Steps

- Avoid allocating new element storage for more cases of array move-initialization
- Adjust domains similarly to avoid copy-init-then-assign
- Remove other cases of default-init-then-assign for arrays, if any remain

- Avoid a coforall+on when implementation marks an array initialization as complete

- Create a user-facing way to move-initialize an array element
  - Necessary inside the domain map implementation
  - Also necessary for certain array 'noinit' use cases

- Address open language design questions around copy-init and move-init across locales

ARRAY COPY INITIALIZATION

Open Language Design Question: copy & locales

- Do records need a way to respond to copy initialize across locales?

```
var A: [1..n] R; // supposing R is a record type
on Locales[1] {
  var B = A;  // copy-initializes elements
}
... A ...;
```

- For array copy-initialization, bulk transfer of array elements is an important optimization
  - Otherwise we are doing 1 GET per array element

- Possible directions:
  - Don’t bulk-transfer in this case (possibly, rely on lower level caching such as ‘--cache-remote’)
  - Call ‘init=’ on the destination locale after bulk transfer (this is the current strategy)
  - Call ‘init=’ on the destination locale after bulk transfer and pass a ‘sourceLocale: locale’ argument
  - Call a new method on the record, e.g. ‘proc postcopy(from: locale)’
  - Allow record authors to opt-in to a serialize/deserialize mechanism when doing bulk transfer

ARRAY COPY INITIALIZATION

Open Language Design Question: move & locales

- Do records need a way to respond to **move** initialize across locales?

```
// supposing R is a record type
var MyBlockArray: [MyBlockDomain] R = returnCyclicArray();
```

- Bulk transfer of array elements is still important
  - And additionally comes up in the context of sorting

- Possible directions:
  - Don't bulk-transfer in this case (possibly, rely on lower-level caching such as '--cache-remote')
  - Don't notify the record at all
  - Try to notify the record by some combination of calling 'init=' and 'deinit'
  - Call a new method on the record e.g. 'proc postmove(from: locale)'
  - Allow record authors to opt into a serialize/deserialize mechanism
    - And have the serialize/deserialize distinguish between copy initialization and move initialization

ARRAY NOINIT

ARRAY NOINIT

Background and This Effort

Background: Some sort of ‘noinit’ has long been planned

- ‘noinit’ support was added in 1.9 but only for basic types
- It was removed in 1.19 to make progress on other issues
- The syntax was motivated by use cases with arrays

This Effort: Added support for ‘noinit’ specifically for arrays

```
var A: [1..n] int = noinit;
forall i in 1..n do A[i] = i;
```

- Using ‘noinit’ in this way allocates space for the elements but does not initialize the elements
- Elements can then be initialized with ‘=’
- Currently only works for arrays of trivially copyable types such as numeric types

ARRAY NOINIT

Next Steps

- Extend array ‘noinit’ to arbitrary element types
  - Probably requiring a call to a low-level ‘move’ function instead of ‘=’ to set the elements

- Choose a user-facing way to indicate when an array is completely initialized
  - To allow for registration with communication support
  - To allow for deinitializing records when the array is deinitialized

- Decide to what extent other collection types should support ‘noinit’
  - Perhaps other collection types should use initializer arguments for this rather than the keyword

OTHER LANGUAGE STABILIZATION IMPROVEMENTS

OTHER LANGUAGE STABILIZATION IMPROVEMENTS

Split Initialization Improvements

- Split initialization no longer considers nested function declarations
  - Consider this example:

```
1 { var x: int;
2   inner();
3   x = 1;
4   proc inner() { writeln(x); }  }
```

  - In 1.22, ‘x’ was default-initialized due to ‘inner()’ referring to it
  - Now, it results in a compilation error:

```
prog.chpl:2: error: 'x' is used before it is initialized
prog.chpl:1: note: 'x' declared here
prog.chpl:3: note: 'x' initialized here
```

  - Now more consistent with behavior for split-init of a module-scope variable

- Fixed several problems with split initialization involving ‘const’ variables, tuples, or ‘out’ intents

OTHER LANGUAGE STABILIZATION IMPROVEMENTS

Ordering Improvements

- Improved ordering of ‘inout’ copy-in and write-back operations to mesh well with ‘in’ and ‘out’
  - For example:

```
proc foo(inout a: R, inout b: R, in c: R, in d: R, out e: R, out f: R) { ... }

var a, b, c, d, e, f: R = ...
foo(a, b, c, d, e, f);
```

  - In 1.22:
    - copy-init from c, d, a, b
    - foo body
    - assign to b, a, e, f
  - In 1.23:
    - copy-init from a, b, c, d
    - foo body
    - assign to a, b, e, f

- Module-scope variables are now deinitialized in reverse initialization order

OTHER LANGUAGE STABILIZATION IMPROVEMENTS

‘in’ Argument Improvements

- Iterator expressions can now be passed to ‘in’ array arguments

```
proc foo(in X: [] int) { … }
foo([i in 1..10] i);
```

- Certain arguments no longer infer runtime types from their default values

```
var DD = {1..4};
var AA: [DD] int;
f(A=AA);

proc f(in D = {1..4}, in A = makeArray(D)) {
   // Is A.domain D or DD?
   // 1.22 - D
   // 1.23 - DD
}
proc makeArray(D : domain(1)) { var R : [D] int; return R; }
```

OTHER LANGUAGE STABILIZATION IMPROVEMENTS

Improved Checking

- Restricted types and intents for ‘extern’/‘export’ functions to working cases
  - ‘out’ and ‘inout’ are not currently allowed
  - records other than ‘extern’ records are not currently allowed

- Improved checking for invalid changes to an instantiated generic field

```
record R { param fixed; }
proc R.init=(rhs: R) {
  this.fixed = rhs.fixed;  // an error if ‘this.fixed’ is already established
}
var a = new R(1);
var b: R(2) = a;
```

  – compiled in 1.22

  – now results in:

```
error: Cannot replace an instantiated param field with another value
```

OTHER LANGUAGE IMPROVEMENTS

OTHER LANGUAGE IMPROVEMENTS

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

- ‘Semantic Changes / Changes to Chapel Language’
- ‘New Features’
- ‘Feature Improvements’ and
- ‘Deprecated / Unstable / Removed Language Features’

THANK YOU

https://chapel-lang.org
@ChapelLanguage