

A photograph of a person walking along a snowy mountain ridge. The sky is filled with dramatic, colorful clouds at sunset, transitioning from yellow to orange to red. The person is seen from behind, wearing a backpack and trekking poles, moving towards the horizon.

Ongoing Efforts

Chapel version 1.20
September 19, 2019

- ✉ chapel_info@cray.com
- 🌐 chapel-lang.org
- 🐦 @ChapelLanguage



Outline

- Epiring Values
- Constrained Generics



Epiring Values



Expiring Values

- Several open questions are connected to an idea of when a value is dead
 - when exactly `deinit()` is called for records
 - eliding copies from expiring values
 - ownership transfer from non-nil 'owned'
- We will consider these cases with some examples
- Then we will evaluate two candidate language rules for when a value is dead



Expiring Values: 'deinit' Example

- Currently, local record variables are always deinitialized at the end of the block
- However this can be surprising in some cases. For example:

```
proc main() {  
  
    var f = opentmp();  
  
    var A = [1,2,3,4];  
  
    var B: [A.domain] int;  
  
    f.writer().write(A);  
  
    f.reader().read(B); // file appears empty here... what happened?  
    // the temporary storing f.writer() is only deinitied here, flushing the buffered data  
}
```

- Should temporary record variables be deinitialized at the end of their statement?

Expiring Values: 'deinit' Behavior

- Deinitializing all temporaries at the end of the statement would prohibit this code:

```
proc main() {  
    ref slice = makeArray() [1..10];  
// during compilation, above statement becomes  
    temp t1:[1..100] int = makeArray();  
    temp t2 = createSlice(t2, 1..10);  
    ref slice = t2;  
// deinitializing t1 or t2 here would cause memory errors below  
    writeln(slice);  
}
```

```
proc makeArray() {  
    var A:[1..100] int;  
    return A;  
}
```

Expiring Values: Copy Elision Example

- The compiler already knows to elide copies for patterns like this:

```
record R { ... }

proc makeR() {
    return new R();
}

proc acceptIn(in arg: R) { ... }

acceptIn(makeR()); // no copy initialization here
```

- However this rule is connected to call-expression temporaries
 - The following code performs copy initialization even though it is not necessary

```
var x = makeR();

acceptIn(x);
```

Expiring Values: Ownership Transfer Example

- We have already seen this example:

```
proc f() {  
    var x: owned C = makeC();  
    var y = x; // ownership transfer from x to y  
    g(y);  
}
```

- Here, 'x' is not used after it is transferred out of
- Could this case be allowed when the compiler can prove such a property?

First Design for When a Value is Dead

- A local variable is dead:

- at the end of the block
 - if it is a user-level variable

```
var x: R;
```

- if it is a temporary in an initialization expression

```
var y = f(g()); // temps storing result of f, g are deinitied at the end of block
```

- Otherwise, it's dead at the end of the statement in which it is created

```
p(q()); // temp storing result of q is deinitialized after this statement
```

- Global variables continue to be destroyed at the end of the program's execution

First Design: 'deinit' Example

- This first design addresses the deinit example

```
proc main() {  
  
    var f = opentmp();  
  
    var A = [1,2,3,4];  
  
    var B: [A.domain] int;  
  
    f.writer().write(A); // temporary storing f.writer() is deinited here  
    f.reader().read(B); // reads the data written above  
  
}
```

- Unfortunately, it is not able to address the other motivating examples

First Design: Pros and Cons

Pros:

- addresses 'deinit' example
- similar to C++ and D

Cons:

- Does not improve ownership or copy elision examples
- Relies on distinguishing between compiler temporaries and user variables
- Users might need something like C++ std::move to optimize some cases



Suggested Design for When a Value is Dead

- A local variable is dead when an analysis shows it is no longer needed
- In particular:
 - at the end of a block, if an alias is potentially captured
 - otherwise, just after the last statement mentioning the variable
- The analysis considers mentions of 'x' to determine if it is potentially captured:
 - by developing a set of potential aliases to 'x'
 - then determining if any of them are potentially captured
- Inferred or explicit lifetime of called functions indicates alias capture potential

Suggested Design: Potential Aliases

- A call 'f(..., a, ...)' potentially creates an alias to 'a' if the lifetime constraint of 'f':
 - allows returning something with lifetime 'a'
- For example, these calls potentially create an alias:

```
identity(a);           proc identity(ref arg) ref { return arg; }  
a[1..10];              // array slicing  
a.borrow();            // borrowing
```

- These calls do not:

```
count(a);               proc count(arg): int { ... }  
globalTable.lookup(a);  // method inferred lifetime prohibits return of arg
```

Suggested Design: Potential Captures

- A user variable initialization potentially captures an alias to 'b' if:
 - it initializes a user variable that refers to / borrows from 'b' (e.g. 'ref x = b')
- For example, these variable initializations potentially capture an alias:

```
var x = b.borrow();  
  
ref c = b;  
  
const ref c = b;
```

- These do not:

```
var y: owned C = b;  
  
var z: R;  
  
var i: int = b;
```



Suggested Design: Potential Captures

- A call 'g(..., c, ...)' potentially captures an alias to 'c'
 - if the lifetime constraint of 'g' allows storing something with lifetime of 'c'
 - into another formal argument, or
 - into an outer / global variable
- For example, these calls potentially capture an alias:

```
set(localVariable, c);
```

```
saveInGlobal(c);
```

- These do not:

```
set(c, localVariable);
```

```
compare(localVariable, c);
```

```
proc set(ref lhs, const ref rhs)
```

```
    lifetime lhs >= rhs { ... }
```

```
proc saveInGlobal(c)
```

```
    lifetime c >= global { ... }
```

```
proc compare(a, b) { ... }
```

Suggested Design: 'deinit' Example

```
proc main() {  
    var f = opentmp();  
    var A = [1,2,3,4];  
    var B:[A.domain] int;  
    f.writer().write(A); // temporary storing f.writer() deinit'd here  
    f.reader().read(B); // reads the data written above  
}
```

- An alias to the temporary storing `f.writer()` is not potentially captured
 - because the '`channel.write`' call cannot save an alias to the channel
 - lifetime clause prevents saving it; return value not used



Suggested Design: Copy Elision Example

```
record R { ... }

proc makeR() { return new R(); }

proc acceptIn(in arg: R) { ... }

{
    var x = makeR();
    f();
    acceptIn(x);
    g();
}
```

- 'x' dead after the 'acceptIn' statement, so copy elision can occur in that call
 - copy elision would not be legal if 'x' could potentially be used in 'g'



Suggested Design: Ownership Transfer Example

```
proc f() {  
    var x: owned C = makeC();  
    var y = x; // ownership transfer from x to y  
    g(y);  
}
```

- ownership transfer from 'x' is legal since it is dead after the transfer
 - no potential alias capture, and 'var y = x' is the last statement mentioning 'x'

Suggested Design: Global Variables

- The design described so far only impacts local variables
- However, we'd like the 'f.writer()' example to behave the same in global scope
 - user-level global variables will always be destroyed at end of program
 - as any function call could refer to these
 - non-captured expression temporaries could be deallocated earlier



Suggested Design: RAII

- Suggested design interferes with some RAII patterns used in C++:

- an RAII lock:

```
var l = takenLock(lock);  
// automatically unlocked at end of block when l is deinitialized
```

- a global stack managing the current line number:

```
var tmp = pushLineNumber(line);  
// line number automatically popped when 'tmp' is deinitialized at end of block
```

- Here are two ways to resolve this problem:

- Create something like 'with' in Python (see e.g. [issue #12306](#))
 - Allow record declarations to request deinitialization only at end of block

```
attribute (no statement deinit) record RAIILock { ... }
```



Suggested Design: Pros and Cons

Pros:

- Addresses 'deinit', copy elision, and ownership transfer examples
- Does not rely on difference between temporaries and user variables
- Avoids the need for something like C++ std::move

Cons:

- More complex than Option 1
- Different from C++ and D
- Supporting RAII patterns in Chapel will require additional work



Expiring Values: Status and Next Steps

Status: Two definitions of expiring values have been proposed

Next Steps: Begin implementation of one of the proposals

- Options are discussed on issue [#13704](#)
- Copy elision and ownership transfer will require further compiler support



Constrained Generics



What are constrained generics?

- Today, the Chapel compiler follows the C++ strategy for generics
 - generic functions are instantiated and then type-checked
- However, this approach presents several problems:
 - generic code might not compile for all calls
 - long compile times
 - confusing point-of-instantiation rule
- Constrained generics are a strategy to:
 - indicate requirements in generic function prototypes
 - type-check generic functions before instantiation
 - instantiate generic functions later in compilation

Previous Work on Constrained Generics

- C++ needs constrained generics but it has taken ~20 years to add it
 - Effort started in 2000s [\[1\]](#)
 - 'concepts' will finally be included in C++20 [\[2\]](#)
- Constrained generics design for Chapel developed in 2012-2013
 - in partnership with Jeremy Siek and Chris Wailes
- Rust and Swift have constrained generics baked in
 - including allowing an implementation with a type known only at runtime

Compilation Speed and Generic Instantiations

- Since generic instantiation is a relatively slow part of compilation:
 - would like to get errors beforehand
 - would like to optimize the compiler to avoid duplicate instantiations
- Constrained generics help with the above
- Additionally, the feature enables a type to be known only at runtime
 - calls can bind dynamically to a table of function pointers
 - similar to virtual dispatch on classes
 - strategy can reduce compilation time but also reduces program performance



Example Generic



Unconstrained Generic

```
proc double(arg: ?t): t {  
    writeln("2x ", arg.show());  
    return add(arg, arg);  
}
```

```
proc int.show(): string {  
    return "int " + this:string;  
}
```

```
proc add(a: int, b: int): int {  
    return a + b;  
}
```

```
double(1);
```

Unconstrained Generic

```
proc double(arg: ?t): t {  
    writeln("2x ", arg.show());  
    return add(arg, arg);  
}
```

// instantiates with 't' replaced by 'int'
// functions from point-of-instantiation are in scope

```
proc double(arg: int): int {  
    writeln("2x ", arg.show());  
    return add(arg, arg);  
}
```

```
proc int.show(): string {  
    return "int " + this:string;  
}
```

```
proc add(a: int, b: int): int {  
    return a + b;  
}
```



```
double(1);
```

Unconstrained Generic

```
proc double(arg: ?t): t {  
    writeln("2x ", arg.show());  
    return add(arg, arg);  
}
```

// instantiates with 't' replaced by 'int'
// functions from point-of-instantiation are in scope

```
proc double(arg: int): int {  
    writeln("2x ", arg.show());  
    return add(arg, arg);  
}
```

```
proc int.show(): string {  
    return "int " + this:string;  
}
```

```
proc add(a: int, b: int): int {  
    return a + b;  
}
```

double(1);

Unconstrained Generic

What if 'add' is private?

```
// instantiates with 't' replaced by 'int'  
// functions from point-of-instantiation are in scope  
proc double(arg: int): int {  
    writeln("2x ", arg.show());  
    return add(arg, arg);  
}
```

```
proc int.show(): string {  
    return "int " + this:string;  
}
```

```
proc add(a: int, b: int): int {  
    return a + b;  
}
```

```
double(1);
```

Unconstrained Generic

```
proc double(arg: ?t): t {  
    writeln("2x ", arg.show());  
    return add(arg, arg);  
}
```

// instantiates with 't' replaced by 'int'
// functions from point-of-instantiation are in scope

```
proc double(arg: int): int {  
    writeln("2x ", arg.show());  
    return add(arg, arg);  
}
```

```
proc int.show(): string {  
    return "int " + this:string;  
}
```

private

```
proc add(a: int, b: int): int {  
    return a + b;  
}
```

double(1);

CHIP 2 Constrained Generic

```
proc double(arg: ?t): t
where t implements Doubleable {
    writeln("2x ", arg.show());
    return add(arg, arg);
}

interface Doubleable {
    proc self.show(): string;
    proc add(a: self, b: self): self;
}
```

```
int implements Doubleable {
    proc int.show(): string {
        return "int " + this:string;
    }
    proc add(a: int, b: int): int {
        return a + b;
    }
}
double(1);
```



CHIP 2 Constrained Generic

```
proc double(arg: ?t): t
where t implements Doubleable {
    writeln("2x ", arg.show());
    return add(arg, arg);
}
```

```
interface Doubleable {
    [0]: proc self.show(): string;
    [1]: proc add(a: self, b: self): self;
}
```

```
int implements Doubleable {
    proc int.show(): string { ←
        return "int " + this:string;
    }
    proc add(a: int, b: int): int { ←
        return a + b;
    }
}
double(1);
```

```
implementation Doubleable(int):
    [0]: show
    [1]: add
```

Rust-like Constrained Generic

```
proc double(arg: ?t): t
where T:Doubleable {
writeln("2x ", arg.show());
return add(arg, arg);
}
```

```
trait Doubleable {
proc self.show(): string;
proc add(a: self, b: self): self;
}
```

```
impl Doubleable for int {
proc int.show():string {
return "int " + this:string;
}
proc add(a: int, b: int): int {
return a + b;
}
}
double(1);
```

Swift-like Constrained Generic

```
proc double(arg: ?t): t
where T:Doubleable {
    writeln("2x ", arg.show());
    return arg.add(arg);
}

protocol Doubleable {
    // assumed to be methods
    proc show():string;
    proc add(other: self): self;
}
```

```
extension int: Doubleable {
    proc int.show(): string {
        return "int " + this:string;
    }
    proc int.add(other: int): int {
        return this + other;
    }
}
double(1);
```

Design Questions



High Level Design Questions

- What is the terminology, syntax, and style to use?
- Do we support both methods and function calls?
- Can one assert an interface is met separately from an 'implements' block?
- How can functions with the same name be handled?
- Can the compiler infer implementation of an interface?
- How will the language handle generic functions without constraints?

Terminology, Syntax, and Style Choices

- 'interface', 'trait', 'protocol', or 'concept' ?
- 'self' is the type being implemented by the constrained generic
 - Should it be explicitly named as an interface argument?
 - 'self' or 'Self' ? Should interface names be UpperCase or lowerCase?
- How exactly can one write a constraint on a generic argument?

```
proc f(arg: ?t) where t implements MyInterface // preferred
proc f(arg: ?t) where t: MyInterface
proc f(arg: implements MyInterface)
proc f(arg: impl MyInterface)
proc f(arg: MyInterface) // preferred
```



Method and Function Signatures in Interfaces

- CHIP 2 proposal uses the 'self' keyword to differentiate methods and functions:

```
proc add(a: self, b: self): self; // non-method  
proc self.show(): string; // method
```

- Alternatives:

- Identify methods with 'this' as the method receiver:

```
proc this.show(): string; // method
```

- Indicate methods with 'this' as the first argument:

```
proc show(this): string; // method
```

- Use universal methods like Rust does:

```
proc show(_:self): string; // show(1) works  
proc add(a: self, b: self): self; // a.add(b) works
```

Asserting an interface is met

- In Rust, 'implements' blocks always contain the implementation
- Compare with CHIP 2, which allows a statement like

```
int implements Doubleable;
```

- Even in Rust, the functions defined in an 'impl' are visible outside of the trait
 - so 'impl' block requirement is not due to visibility requirements
- Even if implementations must be in an 'implements' block, one could forward:

```
proc myAdd(a: int, b: int): int { ... }

int implements Doubleable {

    proc add(a: int, b: int): int { return myAdd(a, b); }

}
```

- Standalone 'implements' has low design impact, avoids duplicate methods

CHIP 2 Constrained Generic: Separate Clause

```
interface Doubleable {
    proc self.show(): string;
    proc add(a: self, b:self): self;
}

proc double(arg:?t):t
where t implements Doubleable {
    writeln("2x ", arg.show());
    return add(arg, arg);
}

proc int.show():string {
    return "int " + this:string;
}

proc add(a: int, b: int): int {
    return a + b;
}

int implements Doubleable;
double(1);
```



CHIP 2 Constrained Generic: Separate Clause

```
interface Doubleable {  
  
    proc self.show(): string;  
  
    proc add(a: self, b:self): self;  
}  
  
proc double(arg:?t):t  
  
where t implements Doubleable {  
  
    writeln("2x ", arg.show());  
  
    return add(arg, arg);  
}
```

```
proc int.show():string {  
  
    return "int " + this:string;  
}  
  
proc add(a: int, b: int): int {  
  
    return a + b;  
}  
  
int implements Doubleable;  
double(1);
```

Handling Duplicate Function Names

- Sometimes generic code might use interfaces using the same function names

```
interface Addable {  
  
    proc self.accum(other: self): self;  
  
}  
  
interface Accumable {  
  
    proc self.accum(other: self): self;  
  
}  
  
proc double(arg: ?t)  
  
where t implements Addable && t implements Accumable {  
  
    arg.accum(arg); //ambiguous: which accum?  
  
}
```

Handling Duplicate Function Names

- How can one resolve the ambiguity?
 - Could decorate functions with interface name, e.g. 'Doubleable.add(arg, 1)'
 - Could allow renaming in 'implements' statements, e.g.

```
where t implements Addable with accum as accumA, Sumable
```
 - Could allow a cast syntax, e.g. '(arg:Addable).accum()'
 - Or could require a wrapper method:

```
proc doubleableShow(arg: Doubleable) { arg.show(); }
doubleableShow(arg);
```
- Universal method syntax helps to enable this in Rust
 - 'arg.show()' can be written as 'show(arg)' - so e.g. 'Doubleable.show(arg)'

Inferring Implementation of an Interface

- Should the compiler be able to infer whether a type implements an interface?
 - e.g. if all of the methods/functions are available, it meets the interface
 - 'explicit' keyword would be available to opt out
- Swift and Rust do not allow this form of inference; but Go does
- Allowing the inference has some advantages:
 - more understandable to Python programmers
 - generics still behave similarly to duck typing
 - less code changes required to migrate generic code
 - reduces knowledge required to use generic libraries
 - e.g. 'implements' not required to specify a sort order



CHIP 2 Constrained Generic: Inferred

```
interface Doubleable {  
  
    proc self.show(): string;  
  
    proc add(a: self, b: self): self;  
}  
  
proc double(arg:?t):t  
  
where t implements Doubleable {  
  
    writeln("2x ", arg.show());  
  
    return add(arg, arg);  
}
```

```
proc int.show():string {  
  
    return "int " + this:string;  
}  
  
proc add(a: int, b: int): int {  
  
    return a + b;  
}  
  
// compiler infers  
// int implements Doubleable;  
// it would not do so if Doubleable it were declared  
// explicit interface Doubleable { ... }  
  
double(1);
```

Inferring Implementation of an Interface

- Inferring that a type implements an interface is not appropriate for all interfaces
 - cannot infer semantic properties like *multiplication on this type is commutative*
- CHIP 2 proposes allowing inference for interfaces not marked 'explicit'
 - aiming for best of both worlds
 - 'explicit' necessary when the interface communicates some property
 - beyond the existence of the functions or methods
- It is possible that the inference could add to compile times
 - In that event, compiler could warn about the compile time
 - and suggest adding 'implements' statements

Unconstrained Generics

- How will the language handle generic functions without constraints?
- Such generic code is very common in Chapel today

```
proc f(arg) { return arg + arg; }
```

- Potential strategies for resolving constrained and unconstrained generics:
 1. Resolve them both late — after instantiation
 2. Use separate strategies
 3. Resolve them both early — before instantiation



Resolving Them Both Late

- In this idea, a constraint on a generic would merely serve as extra type-checking
- It would not affect which functions are resolved

Pros:

- Easier to implement
- No language changes for unconstrained generics
- Constrained and unconstrained generics have similar visibility rules

Cons:

- does not help with complete type-checking of generic code
- does not replace the point-of-instantiation rule
- will not improve compile speed or reduce the amount of type-checking

Separate Strategies

- Idea: use different strategies for constrained and unconstrained generics
 - A constrained generic would resolve and type-check before instantiation
 - An unconstrained generic would resolve and type-check after instantiation
- Note: mixing the two will require special attention for these cases:
 - functions with some arguments constrained and some unconstrained
 - constrained generics calling unconstrained generics
 - unconstrained generics calling constrained generics

Separate Strategies

Pros:

- Limits changes for existing generic code

Cons:

- Mixing constrained and unconstrained generics will require special attention
- Misses opportunities to improve compilation speed for unconstrained generics
- Different visibility rules between the two might be confusing

Resolving Them Both Early

- Here, unconstrained and constrained generics would both resolve early
- Constrained generics would behave as described earlier
- Unconstrained generics would become more similar to constrained generics
 - compiler would infer both interface and 'implements' statements
 - then use constrained generic rules for function visibility

Revisiting the Earlier Unconstrained Example

```
proc double(arg:?t):t {  
    writeln("2x ", arg.show());  
    return add(arg, arg);  
}
```

- Could compiler infer from the above that the argument to 'double' needs to have a 'show' method and work with an 'add' function?
- Could it furthermore type-check while building up this list of constraints?

```
proc int.show(): string {  
    return "int " + this:string;  
}  
  
proc add(a: int, b: int): int {  
    return a + b;  
}  
  
double(1);
```

- Could it infer here that the constraint is met?

Resolving Them Both Early

- Supporting common patterns in generic code will require additional effort
 - compile-time conditionals, e.g. 'if $t == \text{string}$ then ...'
 - inferred variable and return types
- One possibility: construct a graph combining constraints and type inference



Resolving Them Both Early

Pros:

- Clear path for mixing constrained, unconstrained arguments/functions
- Potential to improve compilation speed for unconstrained generics

Cons:

- Unknown impact on compilation speed
- Supporting type inference will require additional implementation effort
- Changes visibility rules for unconstrained generics (see next slides)



Function Visibility in Generic Functions



Function Visibility in Generic Functions

- Function visibility in generic functions is tricky
 - Relies on the point-of-instantiation rule today
- Constrained generics would not use point-of-instantiation rule
 - It is replaced by 'implements' information
- Should unconstrained generics visibility rules match constrained generics?
 - Changes here can break existing code
 - but might not affect many programs in practice

Visibility Example

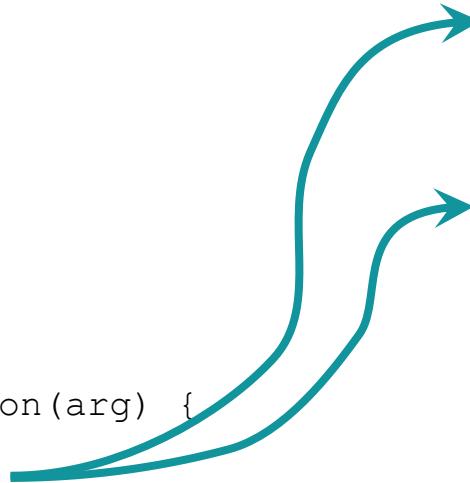
```
module Lib {  
  
    proc foo(arg:int(8)) { ←  
        writeln("Lib.foo(int(8))");  
    }  
  
    proc foo(arg:int) { ←  
        writeln("Lib.foo(int)");  
    }  
  
    proc genericFunction(arg) {  
        foo(arg); ←  
    }  
}
```

```
module Main {  
  
    use Lib;  
  
    proc main() {  
        var x = 1:int(8);  
        genericFunction(x);  
    }  
}
```

Visibility Example

```
module Lib {  
  
    proc genericFunction(arg) {  
        foo(arg);  
    }  
}
```

```
module Main {  
  
    use Lib;  
  
    proc foo(arg:int(8)) {  
        writeln("Lib.foo(int(8))");  
    }  
  
    proc foo(arg:int) {  
        writeln("Lib.foo(int)");  
    }  
  
    proc main() {  
        var x = 1:int(8);  
        genericFunction(x);  
    }  
}
```



Visibility Example

```
module Lib {
```

point-of-instantiation rule

```
proc genericFunction(arg) {  
    foo(arg);  
}
```

```
}
```

```
module Main {
```

```
use Lib;  
proc foo(arg:int(8)) {  
    writeln("Lib.foo(int(8))");  
}
```

```
proc foo(arg:int) {  
    writeln("Lib.foo(int)");  
}
```

```
proc main() {  
    var x = 1:int(8);  
    genericFunction(x);  
}
```

```
}
```

Visibility Example

```
module Lib {  
  
proc foo(arg:int) {  
    writeln("Lib.foo(int)");  
}  
  
proc genericFunction(arg) {  
    foo(arg); //overload sets error  
}  
}
```

```
module Main {  
  
use Lib;  
  
proc foo(arg:int(8)) {  
    writeln("Main.foo(int(8))");  
}  
  
proc main() {  
    var x = 1:int(8);  
    genericFunction(x);  
}  
}
```



Visibility Example

```
module Lib {  
  
    proc foo(arg:int(8)) {  
        writeln("Lib.foo(int(8))");  
    }  
  
    proc genericFunction(arg) {  
        foo(arg); //ok  
    }  
}
```

```
module Main {  
  
    use Lib;  
  
    proc foo(arg:int) {  
        writeln("Main.foo(int)");  
    }  
  
    proc main() {  
        var x = 1:int(8);  
        genericFunction(x);  
    }  
}
```



Visibility Example

```
module Lib {  
  
    proc foo(arg:int(8)) {  
        writeln("Lib.foo(int(8))");  
    }  
  
    proc genericFunction(arg) {  
  
        foo(arg); //ok  
    }  
}
```

```
module Main {  
  
    use Lib;  
  
    proc foo(arg:int) {  
        writeln("Main.foo(int)");  
    }  
  
    proc main() {  
  
        var x = 1:int(8);  
        genericFunction(x);  
        foo(x); //overload sets error  
    }  
}
```



Visibility Example

```
module Lib {  
  
    interface I {  
  
        proc foo(arg: self);  
    }  
  
    proc foo(arg:int(8)) {  
        writeln("Lib.foo(int(8))");  
    }  
  
    proc genericFunction(arg: ?t)  
    where t implements I {  
  
        foo(arg);  
    }  
}
```

```
module Main {  
  
    use Lib;  
  
    proc foo(arg:int) {  
        writeln("Main.foo(int)");  
    }  
  
    int implements I;  
    int(8) implements I; //overload sets error?  
  
    proc main() {  
  
        var x = 1:int(8);  
  
        genericFunction(x);  
    }  
}
```

Visibility Example

What if 'foo(int(8))' is private?

```
proc foo(arg:int(8)) {  
    writeln("Lib.foo(int(8))");  
}  
  
proc genericFunction(arg) {  
    foo(arg);  
}
```

```
module Main {  
  
    use Lib;  
  
    proc foo(arg:int) {  
        writeln("Main.foo(int)");  
    }  
  
    proc main() {  
        var x = 1:int(8);  
        genericFunction(x);  
    }  
}
```

Visibility Example

Historically, failed to compile.
Started working in 1.18.

```
private proc foo(arg:int(8)) {
    writeln("Lib.foo(int(8))");
}

proc genericFunction(arg) {
    foo(arg);
}
```

```
module Main {
    use Lib;
    proc foo(arg:int) {
        writeln("Main.foo(int)");
    }
}

proc main() {
    var x = 1:int(8);
    genericFunction(x);
}
```



Visibility Example

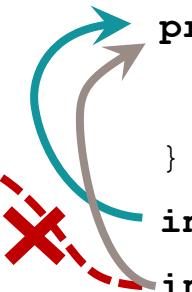
```
module Lib {  
  
    interface I {  
  
        proc foo(arg: self);  
  
    }  
  
    private proc foo(arg:int(8)) {  
        writeln("Lib.foo(int(8))");  
    }  
  
    proc genericFunction(arg: ?t)  
  
        where t implements I {  
  
            foo(arg);  
  
        }  
  
    }  
}
```

```
module Main {  
  
    use Lib;  
  
    proc foo(arg:int) {  
        writeln("Main.foo(int)");  
    }  
  
    int implements I;  
    int(8) implements I;  
  
    proc main() {  
  
        var x = 1:int(8);  
  
        genericFunction(x);  
  
    }  
}
```

Visibility Example

Fails with constrained generic visibility

```
private proc foo(arg:int(8)) {  
    writeln("Lib.foo(int(8))");  
}  
  
proc genericFunction(arg: ?t)  
where t implements I {  
    foo(arg);  
}
```



```
module Main {  
  
    use Lib;  
  
    proc foo(arg:int) {  
        writeln("Main.foo(int)");  
    }  
  
    int implements I;  
    int(8) implements I;  
  
    proc main() {  
        var x = 1:int(8);  
        genericFunction(x);  
    }  
}
```

Further Issue With Point of Instantiation

- The point of instantiation rule has a further wrinkle
- The compiler will *arbitrarily* select an instantiation with particular substitutions
- Usually OK for 'type' arguments
- Can be very confusing with 'param' arguments
- From the language specification:
 - When resolving function calls made within generic functions, there is an additional source of visible functions. Besides functions visible to the generic function's point of declaration, visible functions are also taken from one of the call sites at which the generic function is instantiated for each particular instantiation. The specific call site chosen is arbitrary and it is referred to as the *point of instantiation*.

Point of Instantiation and Multiple Call Sites

Consider an application using two libraries which use a generic library

```
module LibraryOne {  
    use GenericLibrary;  
  
    proc foo(i: int) { }  
    callFoo(1);  
}  
  
module Application {  
    use LibraryOne;  
    use LibraryTwo;  
}  
  
module LibraryTwo {  
    use GenericLibrary;  
  
    module GenericLibrary {  
        proc callFoo(param x) {  
            foo(x);  
        }  
    }  
}
```



Point of Instantiation and Multiple Call Sites

Consider an application using two libraries which use a generic library

```
module LibraryOne {  
    use GenericLibrary;  
  
    proc foo(i: int) { }  
  
    callFoo(1);  
}
```

```
module Application {  
    use LibraryOne;  
    use LibraryTwo;  
}
```

```
module LibraryTwo {  
    use GenericLibrary;  
  
    proc foo(i: int) { }  
  
    callFoo(1);  
}  
  
module GenericLibrary {  
    proc callFoo(param x) {  
        foo(x);  
    }  
}
```

A generic function is used
by two other libraries

Point of Instantiation and Multiple Call Sites

Consider an application using two libraries which use a generic library

```
module LibraryOne {  
    use GenericLibrary;  
    proc foo(i: int) {}  
    callFoo(1);  
}
```

```
module Application {  
    use LibraryOne;  
    use LibraryTwo;  
}
```

```
module LibraryTwo {  
    use GenericLibrary;  
    proc foo(i: int) {}  
    callFoo(1);  
}  
  
module GenericLibrary {  
    proc callFoo(param x) {  
        foo(x);  
    }  
}
```

'callFoo' implementation
expects 'foo' to be
provided by the caller

Point of Instantiation and Multiple Call Sites

```
module LibraryOne {  
    use GenericLibrary;  
  
    proc foo(i: int) { }  
  
    callFoo(1);  
}
```

```
module GenericLibrary {  
    proc callFoo(param x) {  
        foo(x);  
    }  
}
```

// instantiates as:

```
proc callFoo(param 1) {  
    foo(1);  
}  
}
```

```
module LibraryTwo {  
    use GenericLibrary;  
  
    proc foo(i: int) { }  
  
    callFoo(1);  
}
```

Point of Instantiation and Multiple Call Sites

```
module LibraryOne
  use GenericLibrary;
  proc foo(i: int) { }
  callFoo(1);
}
```

In LibraryTwo, callFoo(1)
invokes LibraryOne.foo() !

```
module GenericLibrary {
  proc callFoo(param x) {
    foo(x);
  }
  // instantiates as:
  proc callFoo(param 1) {
    foo(1);
  }
}
```

```
module LibraryTwo
  use GenericLibrary;
  proc foo(i: int) { }
  callFoo(1);
}
```

Point of Instantiation and Multiple Call Sites

Problem is solved by constrained generic visibility

```
module LibraryOne {  
    use GenericLibrary;  
  
    proc foo(i: int) { }  
  
    int implements Foo;  
  
    callFoo(1);  
}
```

```
module GenericLibrary {  
  
    proc callFoo(param x)  
  
    where implements Foo {  
  
        foo(x);  
    }  
}
```

```
module LibraryTwo {  
    use GenericLibrary;  
  
    proc foo(i: int) { }  
  
    int implements Foo;  
  
    callFoo(1);  
}
```

Next Steps for Constrained Generics



Further Features

- These slides focus on possible directions and language compatibility
- A constrained generics feature will likely also need:
 - Associated types
 - Interfaces with multiple arguments
 - Support for generic methods and functions
 - A form of interface type supporting run-time typing
 - Similar to virtual dispatch, but for interfaces
 - Expect this to be syntactically identified, e.g. 'any Doubleable'

Constrained Generics in Chapel: Next Steps

- Discuss and resolve open language design questions
- Decide whether or not to accept breaking language changes
 - especially around visibility in generic functions
- Begin implementation effort to aid in resolving design questions
 - especially regarding what is possible or practical

FORWARD LOOKING STATEMENTS

This presentation may contain forward-looking statements that involve risks, uncertainties and assumptions. If the risks or uncertainties ever materialize or the assumptions prove incorrect, the results of Hewlett Packard Enterprise Company and its consolidated subsidiaries ("Hewlett Packard Enterprise") may differ materially from those expressed or implied by such forward-looking statements and assumptions. All statements other than statements of historical fact are statements that could be deemed forward-looking statements, including but not limited to any statements regarding the expected benefits and costs of the transaction contemplated by this presentation; the expected timing of the completion of the transaction; the ability of HPE, its subsidiaries and Cray to complete the transaction considering the various conditions to the transaction, some of which are outside the parties' control, including those conditions related to regulatory approvals; projections of revenue, margins, expenses, net earnings, net earnings per share, cash flows, or other financial items; any statements concerning the expected development, performance, market share or competitive performance relating to products or services; any statements regarding current or future macroeconomic trends or events and the impact of those trends and events on Hewlett Packard Enterprise and its financial performance; any statements of expectation or belief; and any statements of assumptions underlying any of the foregoing. Risks, uncertainties and assumptions include the possibility that expected benefits of the transaction described in this presentation may not materialize as expected; that the transaction may not be timely completed, if at all; that, prior to the completion of the transaction, Cray's business may not perform as expected due to transaction-related uncertainty or other factors; that the parties are unable to successfully implement integration strategies; the need to address the many challenges facing Hewlett Packard Enterprise's businesses; the competitive pressures faced by Hewlett Packard Enterprise's businesses; risks associated with executing Hewlett Packard Enterprise's strategy; the impact of macroeconomic and geopolitical trends and events; the development and transition of new products and services and the enhancement of existing products and services to meet customer needs and respond to emerging technological trends; and other risks that are described in our Fiscal Year 2018 Annual Report on Form 10-K, and that are otherwise described or updated from time to time in Hewlett Packard Enterprise's other filings with the Securities and Exchange Commission, including but not limited to our subsequent Quarterly Reports on Form 10-Q. Hewlett Packard Enterprise assumes no obligation and does not intend to update these forward-looking statements.



THANK YOU

QUESTIONS?

-  chapel_info@cray.com
-  [@ChapelLanguage](https://twitter.com/ChapelLanguage)
-  chapel-lang.org



- cray.com 
- [@cray_inc](https://twitter.com/cray_inc) 
- linkedin.com/company/cray-inc-/ 