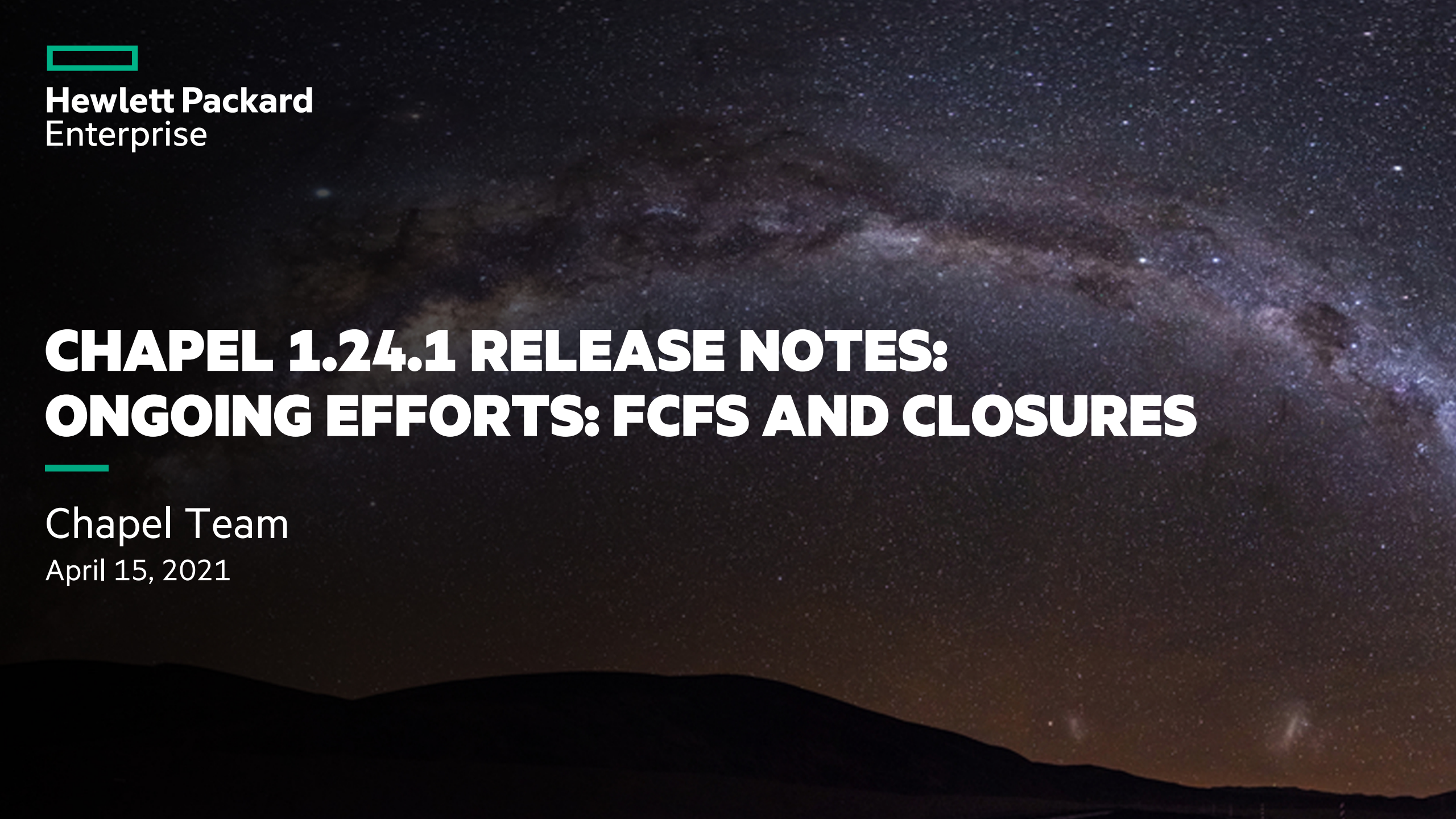

Hewlett Packard Enterprise

CHAPEL 1.24.1 RELEASE NOTES: ONGOING EFFORTS: FCFS AND CLOSURES

Chapel Team

April 15, 2021

FIRST-CLASS FUNCTIONS AND CLOSURES

Background

- Chapel's current support for first-class functions and closures was drafted by a 2010 summer intern
  - Features have received only slight amounts of attention since then
  - Works surprisingly well, given the level of effort
  - That said…
    - line between what works and what doesn't is often unclear
    - documentation is poor
    - syntactic choices aren't universally beloved

- Users ask about these features reasonably often
  - And arguably moreso in recent years

- This slide deck captures where we are today, and considers options for going forward

FIRST-CLASS FUNCTIONS AND CLOSURES

Background: Defining Terms

- First-class functions (FCFs) are functions that can be passed around like any other value
- A closure is a first-class function that captures one or more outer variables
- Capturing is the process of storing one or more outer variables to use later (by reference or value)
- An outer variable is any variable defined outside the scope of a function, that is not global

FIRST-CLASS FUNCTIONS AND CLOSURES

Background: Nested Functions and Capturing

- Chapel provides support for writing nested functions that can capture outer variables
  - Nested functions that capture are not considered closures
    - Unlike FCFs, they cannot escape the scope in which they were defined

```
proc f() {
  var x = 5;
  proc g() {
    writeln(x);    // Here 'x' refers to an outer variable defined in 'f'
                   // This reference is safe because 'g' cannot escape the scope of 'f'
  }
  g();    // Prints '5'
}
f();
```

FIRST-CLASS FUNCTIONS AND CLOSURES

Background: First-Class Functions

- Chapel provides support for treating functions as first-class values
  - Referred to as first-class functions (FCFs)

```
proc foo(x: int, y: int) { return x + y; }
// Create a variable 'fn' that refers to the function 'foo'
var fn = foo;
writeln(fn(1, 2));
```

- Chapel supports constructing FCFs from anonymous functions via the 'lambda' keyword

```
// Here 'fn' refers to an anonymous function
var fn = lambda(x: int, y: int) { return x + y; };
writeln(fn(1, 2));
```

- Function types can be constructed via use of the 'func' keyword

```
// The type of a function that takes two ints and returns an int
type t = func(int, int, int);
```

FIRST-CLASS FUNCTIONS AND CLOSURES

Background: Closures

- Chapel does not currently support the creation of closures
  - A closure is a FCF that captures one or more outer variables
    - The below program does not work today

```
proc main() {
  var x = 5;

  proc g1() { writeln(x); }

  var fn1 = g1;   // This way of creating a closure breaks today
  fn1();

  var fn2 = lambda() { writeln(x); };   // This way also breaks
  fn2();
}
```

FIRST-CLASS FUNCTIONS AND CLOSURES

Why Add Support for Closures?

- Closures make certain patterns easier to write, such as 'update()' calls for 'list' and 'map'
  - Recall that 'map.update()' provides a parallel-safe way to update an element in a map

```
record myUpdater {
  var x: int;
  proc this(const ref k, ref v) { v = x; }
}
var m: map(int, int);
m.add(0, 0);
var x = 5;   // Imagine 'x' is some expensive computation
m.update(0, new myUpdater(x));   // Update 'm[0]' to the value '5'
```

  - With support for closures, a lambda could be used instead of a record, reducing boilerplate

```
var x = 5;
m.update(0, lambda(k: int, ref v: int) {
              v = x;
            });
```

FIRST-CLASS FUNCTIONS AND CLOSURES

This Effort

- Explore improvements to nested functions, FCFs, and support for closures
  - Propose some possible answers to the following questions:
    - Should we change the syntax for constructing anonymous functions?
    - Should we change the syntax for expressing the type of a function?
    - How should nested functions capture outer variables?
    - How should closures capture outer variables?
    - What happens to the outer variables of a closure when it is returned?
    - Should closures with generic arguments be supported?
    - Should we allow the argument types of anonymous functions to be omitted?

FIRST-CLASS FUNCTIONS AND CLOSURES

Syntax for Anonymous FCFs

- Should we change the syntax for constructing anonymous functions?
  - Currently they are constructed using the 'lambda' keyword

```
var fn = lambda(x: int, y: int) {
           return x + y;
         };
```

- One option is to replace 'lambda' with a different existing keyword such as 'proc'

```
var fn = proc(x: int, y: int) {
           return x + y;
         };
```

- Another option is to replace 'lambda' with new syntax, for example:

```
var fn = |x: int, y: int| {
           return x + y;
         };
```

FIRST-CLASS FUNCTIONS AND CLOSURES

Syntax for the Type of a Function

- Should we change the syntax for constructing the type of a function?
  - Currently they are constructed using the ‘func’ keyword

```
// The type of a function that takes two ints and returns an int
type t = func(int, int, int);
```

- Function types could be expressed in a manner like function declaration

```
// The type of a function that takes two ints and returns an int
type t = proc(int, int): int;
```

- Or a more functional style of syntax could be introduced, for example:

```
// The type of a function that takes two ints and returns an int
type t = (int, int) -> int;
```

FIRST-CLASS FUNCTIONS AND CLOSURES

Capturing Non-Local Variables in Nested Functions

- Currently, nested functions capture outer variables by reference

```
proc foo() {
  var x = 5;
  proc bar() { writeln(x); }
  x = 6;
  bar();
}
foo();   // Prints '6'
```

- Non-local variables could be captured by default-intent instead
  - The above would print '5', because the default intent of 'int' is 'const in'

- Non-local variables could also be captured by value
  - Though this is probably not the behavior we want

FIRST-CLASS FUNCTIONS AND CLOSURES

Capturing Non-Local Variables in Closures

- How should a closure capture outer variables?
  - Recall that a closure is a FCF that refers to one or more outer variables

```
proc foo() {
  var a = 5;
  var fn = lambda() { writeln(a); }
  a += 1;
  fn();   // Should this print 5, or 6?
}
```

- Some possible options:
  - Non-local variables will always be captured by reference
  - Non-local variables will always be captured by value
  - Non-local variables will always be captured by default-intent
  - The user can use syntax to specify whether to capture by reference or by value

FIRST-CLASS FUNCTIONS AND CLOSURES

Option 1: Always Capture by Reference

- Non-local variables will always be captured by reference

```
proc foo() {
  var a = 5;
  // The lambda will capture 'a' by reference
  var fn = lambda() { writeln(a); }
  a += 1;
  fn();   // Prints '6'
}
```

- Care must be taken when a closure escapes the scope where it is created
  - Or else captured variables may refer to deallocated memory
  - We will discuss this more soon

FIRST-CLASS FUNCTIONS AND CLOSURES

Option 2: Always Capture by Value

- Non-local variables will always be captured by value

```
proc foo() {
  var a = 5;
  // The lambda will capture 'a' by value
  var fn = lambda() { writeln(a); }
  a += 1;
  fn();   // Prints '5'
}
```

- An expensive approach for large arrays / records

FIRST-CLASS FUNCTIONS AND CLOSURES

Option 3: Capture by Default-Intent

- Non-local variables will always be captured by default-intent

```
proc foo() {
  var a = 5;
  // Capturing a FCF by 'ref' could make it capture by reference
  var fn = lambda() { writeln(a); }
  a += 1;
  fn();   // Prints '5'
}
```

- The default-intent of 'int' is 'const in', so 'a' would be captured by value
  - Small types can be copied for convenience

FIRST-CLASS FUNCTIONS AND CLOSURES

Option 4: User Chooses Capture Mode

- The user can use syntax to specify the capture mode, for example:

```
proc foo() {
  var a = 5;
  // Creating a FCF with 'ref' could make it capture by reference
  var fn = ref lambda=() { writeln(a); }
  a += 1;
  fn();   // Prints '6'
}

proc foo() {
  var a = 5;
  // Creating a FCF with 'var' could make it capture by value
  var fn = var lambda() { writeln(a); }
  a += 1;
  fn();   // Prints '5'
}
```

FIRST-CLASS FUNCTIONS AND CLOSURES

What About C++ and Rust?

- In C++, the capture mode can be specified via syntax such as '[=]' (which copies)

```
int x = 5;
auto fn = [=] { std::cout << x << std::endl; };   // Move a copy of 'x' into 'func'
x = 6;
fn();   // Prints '5'
```

- In Rust, a captured variable may be borrowed, copied, or moved

```
let mut x = 5;
let func = move || { println!("{}", x); };   // Move a copy of 'x' into 'func'
x = 6;
func();   // Prints '5'
println!("{}", x); // Prints '6'
```

FIRST-CLASS FUNCTIONS AND CLOSURES

Returning Closures

- What should happen to the outer variables of a closure when it is returned?
  - If ‘y’ is still captured by reference after ‘foo()’ returns, it will refer to invalid memory locations

```
proc foo() {
  var y = 5;
  return lambda() { writeln(y); };
}
var fn = foo();   // What about the write of ‘y’ in ‘fn’?
```

- Some possible options:
  - Prohibit returning closures that capture outer variables by reference
  - On return, implicitly convert to a new closure that captures all outer variables by value
  - Migrate outer variables to the heap, with a reference count

FIRST-CLASS FUNCTIONS AND CLOSURES

Option 1: Prohibit Returning Closures that Capture by Reference

- One option is to prohibit returning closures that capture outer variables by reference

```
proc foo() {
  var y = 5;
  return lambda() { writeln(y); };   // Error: cannot return lambda that refers to outer variable 'y'
}
```

FIRST-CLASS FUNCTIONS AND CLOSURES

Option 2: Convert to a New Closure That Captures by Value

- Another option is to convert to a new closure that captures all outer variables by value

```
proc test() {
  var y = 5;

  proc foo() {
    // When the closure is returned, the value of 'y' is saved
    return lambda() { writeln(y); }
  }

  var fn = foo();
  y = 6;

  fn();   // Prints '5'
}
```

FIRST-CLASS FUNCTIONS AND CLOSURES

Option 3: Migrate Outer Variables to the Heap

- Another option is migrating all outer variables to the heap, with a reference count
  - This ensures that outer variables will outlive the scope that created them

```
proc foo() {
  var y = 5;
  // Create two closures that both print 'y'. The first closure increments 'y'
  var fn1 = lambda() { writeln(y); y += 1; };
  var fn2 = lambda() { writeln(y); };
  // Because the closures are returned, the compiler stores 'y' on the heap rather than the stack
  return (fn1, fn2);
}
{
  var (fn1, fn2) = foo();

  fn1();   // Prints '5'
  fn2();   // Prints '6'

}   // 'y' is freed here since 'fn1' and 'fn2' have left scope
```

FIRST-CLASS FUNCTIONS AND CLOSURES

Capture Rules May Apply to Other Constructs

- The capture rules that are chosen may apply to other constructs as well
  - Consider the following situation in which we define a function object

```
proc foo() {
  var y = 5;

   record r {
     proc this() { writeln(y); }   // Here 'y' may be captured, or this could be an error
   }

  return new r();
}

var r1 = foo();
r1();   // What this prints depends on the capture rules we choose, if we allow it at all
```

FIRST-CLASS FUNCTIONS AND CLOSURES

Supporting FCFs with Generic Arguments

- Currently, FCFs must specify an explicit type for each formal argument

```
proc foo(a) { writeln(a); }
var fn = foo;   // Error: cannot refer to generic function 'foo' by value
```

- Should we support FCFs with generic formal arguments?

```
proc foo(a) { writeln(a); }
var fn = foo;
fn(1);   // Prints '1'
```

FIRST-CLASS FUNCTIONS AND CLOSURES

Omitting Argument Types of Anonymous Functions

- Currently, anonymous 'lambda' functions must specify the type of every argument

```
var fn = lambda(x) { writeln(x); };   // Error: cannot capture generic lambda
```

- Should we allow the argument types of anonymous functions to be omitted?
  - This could be done even if we decide not to support creating generic FCFs

- Anonymous functions often appear inline
  - Which makes explicit argument types feel like boilerplate

- It would be nice if argument types could be omitted as a syntactic convenience

```
var fn = lambda(x) { writeln(x); };
fn(1);   // Prints '1'
```

FIRST-CLASS FUNCTIONS AND CLOSURES

Omitting Argument Types for Closures in Rust

- In Rust, the types of closure arguments may be omitted
  - If the closure is being returned, the returning function must declare an explicit return type
  - The type of the closure is made concrete the first time it is called
    - Cannot call the closure again using arguments of different types

```
let func = | x | { println!("{}", x); };
func(1);
func("1");    // Error: expected integer, found `&str`
```

- This design would differ from the current rules Chapel uses to instantiate generic functions
  - Today, a concrete function would be instantiated for each combination of argument types

- We could adjust the rules for anonymous functions to mimic Rust as a syntactic convenience
  - Would this cause too much confusion?

FIRST-CLASS FUNCTIONS AND CLOSURES

Status, Next Steps

Status: Discussion about capturing rules, FCFs, and closures is still at an early stage

Next Steps:

- Continue to study how other languages implement capturing rules and closures
- Identify which of the features discussed are most important to users
- Reach agreement within the team


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