Advanced Chapel Programming

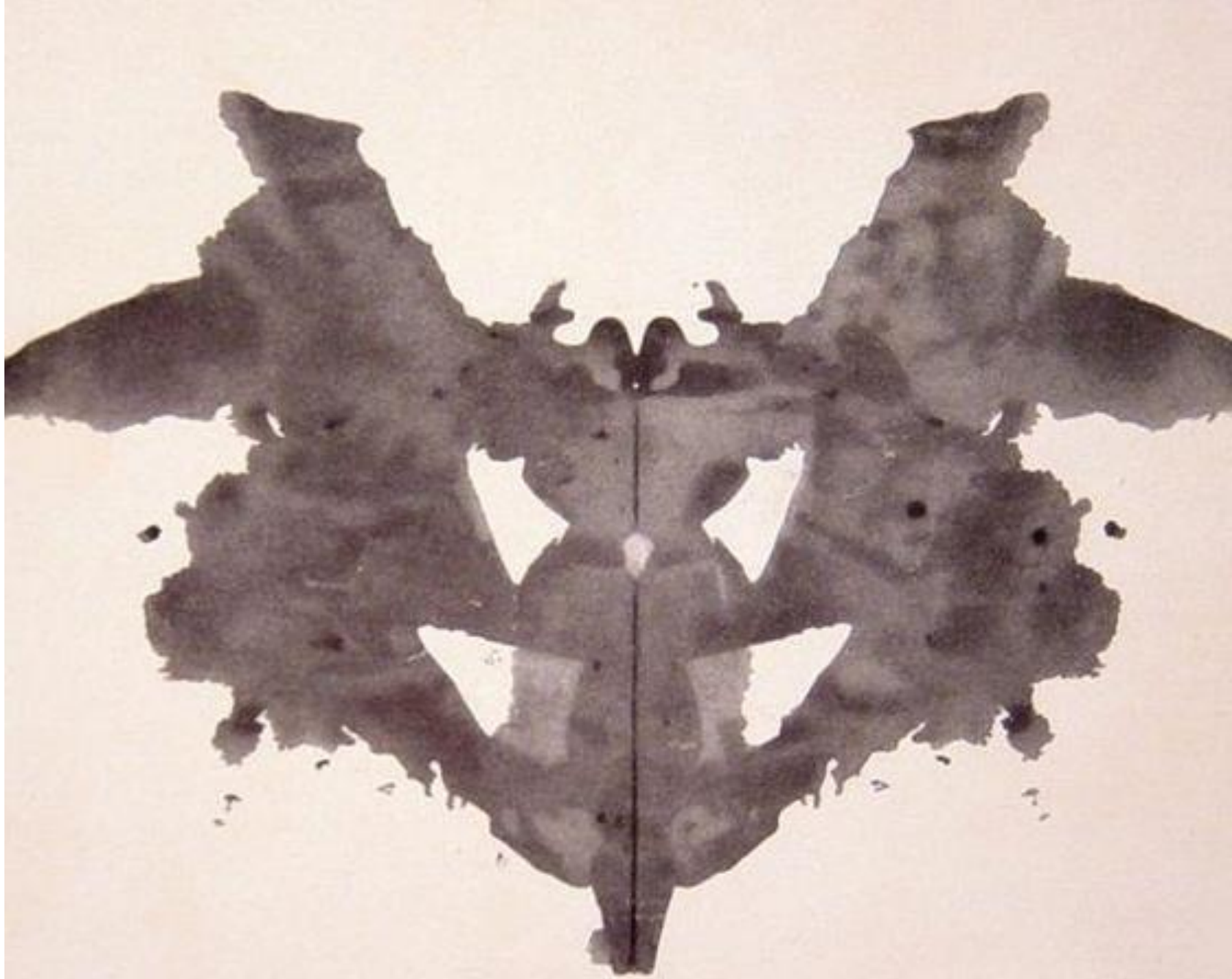
Daniel Fedorin, HPE

What does ‘Advanced Chapel’ Mean to You?’

— “Advanced Chapel” can mean a lot of things

— I picked features that I thought will be particularly relevant for the tutorials today

— Much “advanced” material cannot be covered in this time



Not Covered Here

— Performance work
- Engin Kayraklioğlu will give a performance debugging tutorial later today! (9:40 PT)
- Michael Ferguson has written an extensive optimization technical note

— Advanced parallel programming
- Shreyas covered parallel loops in his tutorial yesterday in much depth
- Advent of code 2022 days 11 and 12 cover low-level task spawning and synchronization
- With these building blocks, the algorithms matter more than the language

— Interfacing with C/C++/Fortran/Python
- Scott Bachman's NetCDF blog posts cover using a C library from Chapel
- Brandon Neth's & Michelle Strout's blog posts, in collaboration with [C]Worthy, cover interop with Fortran
- The 'Python' module bundled as of release 2.4 can be used to create and run Python interpreters

Not Covered Here

Parallel Loops Demo

Optimization technical note

AoC day 11 (syncs, coforall)

AoC day 12 (atomics, coforall)

C Interop (w/NetCDF)

FORTRAN Interop (w/MARBL)

Python module documentation

Motivation & Goals

— Many of Chapel's lower-level features require a deeper understanding of the specifics of the language
— This is true both for how the feature is implemented, and for how the user ought to engage with it
— This can muddy the waters while trying to make the jump

— Today, I want to give a primer on advanced features that come up in lower-level uses of Chapel
  • A lot of this turns out to be Chapel's type system

Compile-Time Generics

Many of Chapel's features are written using generic code

— Distributions, Iterators, Serializers

At the surface-level, Chapel's generics were designed to be familiar to folks coming from Python.

— Type annotations are not required

— Programs can be gradually enhanced with types after prototyping

— Functions can inspect arguments' types to adjust their behavior

Python achieves this with dynamic typing

— Conditions on values' types execute at runtime

— Type annotations don't do anything…

— …So, they can be added later.

But Chapel is statically typed!

Compile-Time Generics

Consider this function:

```
proc foo(x, y) do return x + y;
```

'foo' does not have enough static type information to be compiled as-is.

— This information is provided from calls to the function.

— For each set of argument types, we produce a copy of the function that has the corresponding types statically

— In simple cases, think of this as substitution.

— These copies are called *instantiations*.

Consider calls 'foo(1.0, 2)' and 'foo(1i, 1.0)'. This creates two instantiations:

```
proc foo(x: real, y: int) do return x + y;
proc foo(x: imag, y: real) do return x + y;
```

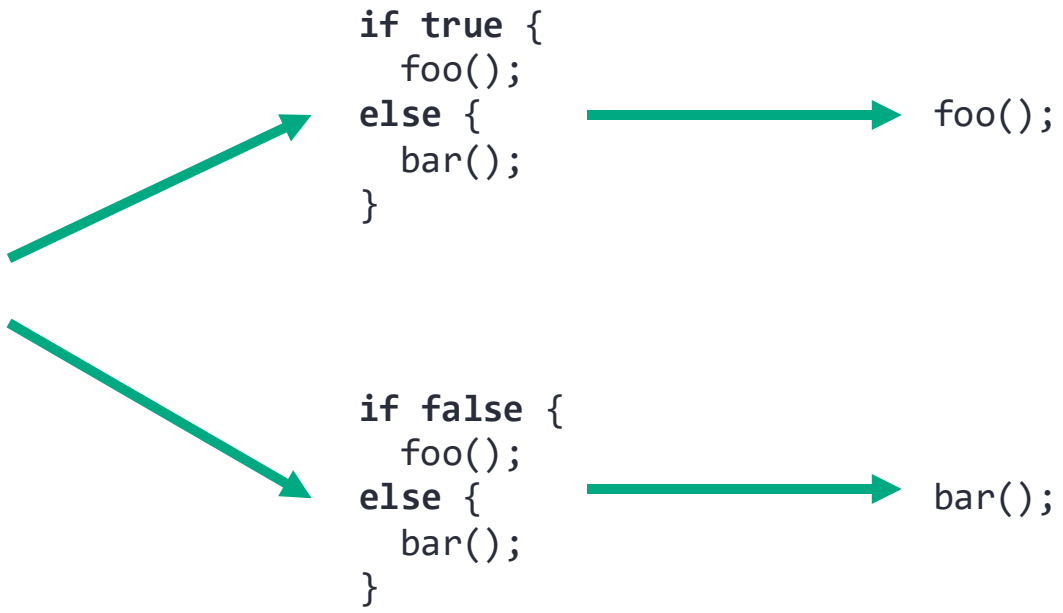
Each instantiation is resolved separately.

Compile-Time Generics

Conditionals on types are also handled at compile-time

— This is because conditionals with compile-time-known conditions are are eliminated

```
if x.type == string {
  foo();
else {
  bar();
}
```



ℹ Recall each instantiation gets one copy of this conditional

Resolution Order

Type resolution proceeds in a fixed order

— Roughly: left-to-right, top-to-bottom

— This enables early-returning patterns and prevents dead code (which may not compile) from being resolved

```
proc myHash(x) {
  if x.type == int {
    return x;
  }
  if !canResolveMethod(x, “hash”) then
    return 0; // no hash method, return default value

  return x.hash();
}
```

Resolution Order

Type resolution proceeds in a fixed order

— Roughly: left-to-right, top-to-bottom

— This enables early-returning patterns and prevents dead code (which may not compile) from being resolved

```
proc myHash(x: someType) {
  if someType == int {
    return x;
  }
  if !canResolveMethod(x, “hash”) then
    return 0; // no hash method, return default value

  return x.hash();
}
```

Resolution Order

Type resolution proceeds in a fixed order

— Roughly: left-to-right, top-to-bottom

— This enables early-returning patterns and prevents dead code (which may not compile) from being resolved

```
proc myHash(x: someType) {
  if someType == int {
    return x;
  }
  if !canResolveMethod(x, “hash”) then
    return 0; // no hash method, return default value

  return x.hash();
}
```

Resolution Order

Type resolution proceeds in a fixed order

— Roughly: left-to-right, top-to-bottom

— This enables early-returning patterns and prevents dead code (which may not compile) from being resolved

```
proc myHash(x: someType) {
  if false {
    return x;
  }
  if !canResolveMethod(x, “hash”) then
    return 0; // no hash method, return default value

  return x.hash();
}
```

Resolution Order

Type resolution proceeds in a fixed order

— Roughly: left-to-right, top-to-bottom

— This enables early-returning patterns and prevents dead code (which may not compile) from being resolved

```
proc myHash(x: someType) {


  if !canResolveMethod(x, “hash”) then
    return 0; // no hash method, return default value

  return x.hash();
}
```

Resolution Order

Type resolution proceeds in a fixed order

— Roughly: left-to-right, top-to-bottom

— This enables early-returning patterns and prevents dead code (which may not compile) from being resolved

```
proc myHash(x: someType) {


  if !canResolveMethod(x, “hash”) then
    return 0; // no hash method, return default value

  return x.hash();
}
```

Resolution Order

Type resolution proceeds in a fixed order

— Roughly: left-to-right, top-to-bottom

— This enables early-returning patterns and prevents dead code (which may not compile) from being resolved

```
proc myHash(x: someType) {


  if true then
    return 0; // no hash method, return default value

  return x.hash();
}
```

Resolution Order

Type resolution proceeds in a fixed order

— Roughly: left-to-right, top-to-bottom

— This enables early-returning patterns and prevents dead code (which may not compile) from being resolved

```
proc myHash(x: someType) {


  return 0; // no hash method, return default value

  return x.hash()
}
```

Resolution Order

Type resolution proceeds in a fixed order

— Roughly: left-to-right, top-to-bottom

— This enables early-returning patterns and prevents dead code (which may not compile) from being resolved

```
proc myHash(x: someType) {

  return 0; // no hash method, return default value

}
```

Example Application: ForwardModeAD

Each instantiation of a generic function is resolved separately

— So, we can use overloading to change the calls in the function body

Luca Ferranti's ForwardModeAD does this for forward-mode automatic differentiation

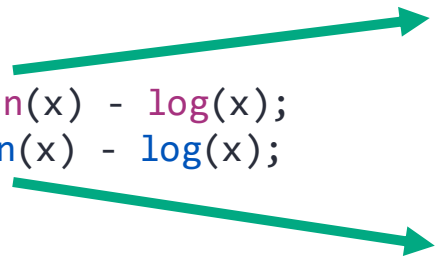
— This leans on the concept of "dual numbers", which carry their value and an approximation of a derivative

— These have the form 'a + bε', where can be thought of as a "small constant" ($\varepsilon^2 = 0$)

— By overloading '+', 'sin', etc. for these numbers, we can define operations that also compute derivatives

```
// generic
proc f(x) do return exp(-x) * sin(x) - log(x);

// instances (created by compiler)
proc f(x: real) do return exp(-x) * sin(x) - log(x);
proc f(x: dual) do return exp(-x) * sin(x) - log(x);
```



```
// Standard
proc exp(x: real) {}
proc sin(x: real) {}
proc log(x: real) {}

// ForwardModeAD
proc exp(x: dual) {}
proc sin(x: dual) {}
proc log(x: dual) {}
```

“Dependent” Typing

Generic Chapel procedures support a degree of dependency

— For example, the following procedure accepts three arguments of the same type:

```
proc fooMatching(x, y: x.type, z: y.type) do return x + y + z;
```

— In general, type expressions in formals can reference preceding formals

— To support this, resolution of formals notionally proceeds left-to-right

```
fooMatching(1, 2, 3);
proc fooMatching(x, y: x.type, z: y.type) do return x + y + z;
proc fooMatching(x: int, y: int, z: y.type) do return x + y + z;
proc fooMatching(x: int, y: int, z: int) do return x + y + z;
proc fooMatching(x: int, y: int, z: int) do return x + y + z;
```

“Dependent” Typing

In general, type expressions in formals can reference preceding formals

— This doesn’t have to be limited to exact matches

```
proc fooMatching(x, y: (x.type, x.type, ?)) do return x + y[0] + y[1];

fooMatching(1, (2, 3, “Hello”));

proc fooMatching(x, y: (x.type, x.type, ?)) do return x + y[0] + y[1];
proc fooMatching(x: int, y: (int, int, ?)) do return x + y[0] + y[1];
proc fooMatching(x: int, y: (int, int, string)) do return x + y[0] + y[1];
```

Type Queries

Instead of '.type', you can use *type queries* to extract type information from inferred formal types

— In this example, we accept an 'int' and then as many 8-bit values as we need to construct it

```
proc intAndBytes(x: int(?w), ref bs: int(8)...(w/8)) {}

var b0, b1: int(8);
intAndBytes(0x2025 : int(16), b0, b1);

proc intAndBytes(x: int(?w), ref bs: int(8)...(w/8)) {}
proc intAndBytes(x: int(16), ref bs: int(8)...(16/8)) {}
proc intAndBytes(x: int(16), ref bs_0: int(8), ref bs_1: int(8)) {}
```

Type Queries

Instead of '.type', you can use *type queries* to extract type information from inferred formal types

— In this example, we accept an array and a value of the same type as the array's elements

```
proc arrayAndElement(A: [] ?t, element: t) {}
```

Applications: Standard Library

Lots of standard library procedures use type queries and dependent formals. Some eclectic examples:

— Sets only allow union, intersection, etc. on same-element-type sets

```
operator set.|(const ref a: set(?t, ?), const ref b: set(t, ?))
operator set.&(const ref a: set(?t, ?), const ref b: set(t, ?))
```

— The 'Random' module uses array element type queries to constrain 'fillRandom' and others

```
proc fillRandom(ref arr: [] ?t, min: t, max: t, seed: int)
```

— Comparators for sorting use the same-type constraint we've already seen

```
proc Self.compare(x, y: x.type)
```

Applications: Serializers

The user-facing serializer API uses dependent types and type queries:

— In various helpers like 'startRecord', the type of 'this' is used to parameterize the 'fileWriter'

```
proc Serializer.startRecord(writer: fileWriter(false, this.type), name: string, size: int) throws;
```

— In other parts of the API, generics are written using type queries

```
proc T.serialize(writer: fileWriter(?), ref serializer: ?st) throws
```

Serializers Demo
Covers reading and writing custom data structures

`where` clauses

Instead of conditionals in a function body, types can be restricted as part of the function's signature

```
proc foo(x, y)
  where isNumericType(x.type) && isNumericType(y.type)
  do return x + y;
```

Unlike what we've seen, this happens *before* a function's body is resolved

— 'where' clauses are checked when picking applicable overloads

— The following program is unambiguous

```
proc bar(x) where x.type == string {} // (1)
proc bar(x) where x.type != string {} // (2)

bar("hello") // picks (1)
bar(1234)    // picks (2)
```

where clauses

Functions with where clauses are also considered “more specific” and are preferred when possible

— The following program is also unambiguous

```
proc bar(x) where x.type == string {} // (1)
proc bar(x)                        {} // (2)

bar(“hello”) // picks (1)
bar(1234)    // picks (2)
```

`param` values

So far, we've seen that generic functions:

— Have the types of their arguments determined at compile-time

— Are instantiated (creating copies) for each possible combination of argument types

- Each instantiation can be thought of as getting subjected to substitution of generic types with the determined types

— Have their body resolved step-by-step, in a way that is sensitive to early returns and compile-time conditions*

So far, the substitutions and copies have only occurred for different *types*

With 'param's, we can generalize this to values

* also true for regular functions, but typically less important

Compile-Time Generics with `params`

Consider the following function:

```
proc fooParam(param x, param y) param do return x + y;
```

Consider calls ‘fooParam(1.0, 2)’ and ‘fooParam(3, 2)’. This creates two instantiations:

```
proc fooParam(param x: real [=1.0], param y: int [=2]) param do return 1.0 + 2;
proc fooParam(param x: int [=3], param y: int [=2]) param do return 3 + 2;
```

Note that the values of ‘x’ and ‘y’ were replaced with their constants, like we saw with ‘.type’

Calls to functions with the ‘param’ return intent are replaced with their return value

— The return value must be a compile-time-known constant

```
return 1.0 + 2      /* ===> */ return 3.0;
fooParam(1.0, 2)    /* ===> */ 3.0
```

Compile-Time Generics with `params`

The following types can become ‘params’:

— **`int`**

— **`uint`**

— **`real`**

— **`string`**

— any **`enum`**

`params` You Might Have Seen

— Lists, Sets, and other data structures are split into two families: ones that are safe to use in parallel, and ones not

```
var l: list(int, parSafe=true); // parSafe is a param argument to the type constructor, and a field
```

— Some compiler settings are actually ‘param’ global variables used by the standard modules

```
$ chpl -sdebugDataPar=true myprog.chpl # debugDataPar is a global constant for rectangular data
```

— Whether a range has an upper bound, a lower bound, or both is a ‘param enum’ on the range

```
(1..).bounds // = boundKind.low
```

— Conditions are “compile-time removed” and affect resolution order precisely when their conditions are ‘param’

```
if returnsParamTrue() then foo();  /* ===> */ foo()
```

Application: Writing Parallel Iterators

Custom parallel iterators require generics, ‘param’ enums, and ‘where’ clauses.

— From the **Parallel Iterators** primer, an example of a standalone parallel iterator:

```
iter count(param tag: iterKind, n: int, low: int = 1) where tag == iterKind.standalone
```

Parallel Iterators Demo
Covers developing custom parallel iterators

Summary

Many Chapel features use generic functions and types as part of their implementation, such as:

— Iterators
— Serializers
— Many standard data structures

Understanding this features helps write more powerful code and better understand the standard library.

Chapel's generics:

— Are statically-typed, and stamped out by the compiler depending on the argument types
— Support compile-time simplification of 'type' and 'param' expressions to eliminate unused code
— Allow formals' types and values to depend on the types and values of preceding formals
— Can use 'where' clauses to provide more specific / constrained implementations of functions