Hewlett Packard Enterprise

CHAPEL 1.24 RELEASE NOTES: ONGOING EFFORTS

Chapel Team

March 18, 2021

OUTLINE

- Interfaces
- Supporting Resized Arrays of Non-Nilable Classes
- Standard Library Stabilization
- Foreach Loops: Indicating Vectorizability
- Targeting GPUs from Chapel
- Revamping the Compiler


INTERFACES

INTERFACES
Background: Traditional Generics

- Generics in Chapel have followed the C++ template strategy for generics so far
  - a generic function is type-checked / resolved only after being instantiated for a call
  - we call them “traditional generics”

- Downsides of traditional generics include:
  - a generic function might appear correct because it works for some arguments, yet fail to type-check for others
  - a typechecking error appears within the generic function even if it is the caller’s fault

```
proc sort(A: []) { … if A[i] < A[j] then … }   // if '<' is unresolved, whose fault is it?
```

  - a complex point-of-instantiation rule is necessary (e.g., see 1.23 release notes)
  - a verbose where-clause is often needed to express a constraint
    - e.g., “the argument should be iterable and yield strings”

- See 1.21 release notes for a detailed overview

INTERFACES

Background: Interfaces

- “Interfaces” for Chapel were proposed in CHIP 2, similar to “concepts” in C++20

- Expected benefits for Chapel include:
  - cleaner specification of requirements for a generic function
  - guaranteed “once and for all” type-check of interface-constrained functions
  - simpler resolution rules without involving the point of instantiation
  - potentially improved compilation time if generic instantiation is delayed and/or reduced

INTERFACES
This Effort: Overview

- An initial implementation of interfaces for Chapel
  - chose an implementation strategy
  - documented the design questions that came up and chose initial answers to some of them
  - chose terminology
  - introduced the keywords ‘interface’ and ‘implements’

INTERFACES

This Effort: Basic Features

interface declaration

interface formal(s)

```
interface Comparable(T) {
  proc lt(x: T, y: T): bool;   // a less-than comparison
}
```

required function

interface-constrained function

```
proc minFn(in x: ?T, in y: T): T
  where T implements Comparable
{
  if lt(x, y)
   then return x;
   else return y;
}
```

interface constraint

call to required function 'lt'

implements statement

```
int implements Comparable;
```

implementation of required function

```
proc lt(x: int, y: int): bool
  return x < y;
```

call to constrained function 'minFn'

```
writeln(minFn(3, 4));
```

- 'minFn' is a generic function; it can be called only when the argument type implements 'Comparable'
- it is legal to invoke required function 'lt' within 'minFn'
- 'int' implements 'Comparable' because an implementation of the required function 'lt' is provided for 'int'

INTERFACES

This Effort: Syntactic Conveniences

'Self' is an implicit interface formal

```
interface LocaleArray {
  proc getNth(arr: Self, n: int): locale;
}
```

When an interface declaration does not list formals, a single formal 'Self' is introduced implicitly.

equivalent to '(arr: ?T) where T implements LocaleArray'

```
proc front(arr: LocaleArray): locale
  return getNth(arr, 0);

proc getNth(collection, n)   // an implementation of getNth()
  return collection[n];      // for any indexable type
```

'arr: LocaleArray' means the type of 'arr' must implement the 'LocaleArray' interface

use this syntax with arbitrary type expressions

```
implements LocaleArray(Locales.type);
writeln(front(Locales));
```

The syntax 'T implements I' is available only when 'T' is an identifier.

INTERFACES
This Effort: Associated Types

```
interface ArrayLike {
  type eltType;
  proc getNth(arr: Self, n: int): eltType;
}

proc front(arr: ArrayLike): arr.eltType
  return getNth(arr, 0);

proc getNth(collection, n)
  return collection[n];

implements ArrayLike(Locales.type);
writeln(front(Locales));
```

associated type(s)

associated type of 'arr'

the associated type is 'locale'

Each type implementing 'ArrayLike' must define its associated type by providing a type field or a type method 'eltType'.

Constrained functions can refer to an associated type using the dot notation.

Here, the associated type is obtained as if by querying 'Locales.eltType'.

INTERFACES

This Effort: Inference of ‘implements’ statements

```
interface ArrayLike {
  type eltType;
  proc getNth(arr: Self, n: int): eltType;
}

proc front(arr: ArrayLike): arr.eltType
  return getNth(arr, 0);

proc getNth(collection, n)
  return collection[n];

// implements ArrayLike(Locales.type);
writeln(front(Locales));
```

inferred automatically for the call ‘front(Locales)’

inference is disabled for empty interfaces

When calling a constrained function like ‘front’, must ensure each of its constraints is satisfied with an ‘implements’ statement.

If no applicable ‘implements’ statement is visible in the source code, attempt to infer it automatically.

This “structural” semantics is chosen to ease transition from traditional to interface generics: reduce the need to introduce ‘implements’ statements for callers of generic functions.

“Nominative” semantics (no inference) is provided for empty interfaces.

INTERFACES

This Effort: Resolution Semantics

- Resolution of generics uses a hybrid approach
  - traditional generics are handled as before—no change

- For an interface-constrained function:
  - resolve it “early”, before any instantiation
    - the early-resolved function remains generic
  - instantiate it for each invocation—like traditional generics, except:
    - no further resolution / type-checking of the instantiated body
    - within the body, calls to required functions are directed to their implementations
  - handle it like an instantiated traditional generic from there on, for example:
    - when translating abstract to concrete argument intents
    - during lifetime checking, optimizations, …

INTERFACES

This Effort: Early Resolution Example

```
interface Comparable(T) { proc lt(x: T, y: T): bool; }
```

an interface-constrained function

```
proc minFn(in x: ?T, in y: T): T
  where T implements Comparable
{
  if lt(x, y)
    then return x;
    else return y;
}
```

early-resolved into

early-resolved generic (pseudocode)

```
proc minFn(in x: T, in y: T): T
  where
    type T
      implements Comparable
{
  if lt(x, y)
    then return x;
    else return y;
```

a unique type

this call is resolved to the required fn

INTERFACES

This Effort: Instantiation Example

```
interface Comparable(T) { proc lt(x: T, y: T): bool; }
```

early-resolved generic (pseudocode)

```
proc minFn(in x: T, in y: T): T
  where
    type T
      implements Comparable
{
  if lt👆(x, y)
  then return x;
  else return y;
}
```

instantiated into

instantiation for T=int and the applicable implements statement

```
proc minFn(in x: int, in y: int): int
{
  if lt(x, y)
  then return x;
  else return y;
}
```

this call is redirected to the implementation of 'lt'

```
int implements Comparable; // lt = 👇
```

```
proc lt(x: int, y: int): bool
  return x < y;
```

```
minFn(3, 4);
```

this call to 'minFn' resolves to this instantiation using this implements statement

INTERFACES
Status

- Simple interface-based code works
  - a set of basic features is implemented
  - 86 passing tests in our test suite

- Introduced the keywords ‘interface’ and ‘implements’

- Semantics, implementation, terminology are still in flux

- Gathered a set of basic design questions
  - listed on issue #8629, also discussed on the following slides
  - the current implementation reflects certain design choices, open to revision

INTERFACES

Next Steps

- Solidify the implementation
  - implement missing features
  - fix bugs
- Resolve design questions
  - solidify terminology
- Apply interfaces to our modules
  - use this to guide design choices, prioritize implementation work
- Leverage interfaces to speed up compilation, reduce the size of generated code
- Provide explanations upon compiler errors like “interface constraint(s) were not satisfied”

INTERFACES

Next Steps: Open Design Questions

- Key design questions follow, more on issue #8629
- Should we allow interface inference? (issue #16952)
  - i.e., an interface constraint is satisfied when all its required functions are available
  - a.k.a. “structural” vs. “nominative” interfaces
  - “structural” interfaces would ease transition from traditional generics
- How to resolve name conflicts? (issue #10802)
  - e.g., a called function is defined by multiple interfaces
  - e.g., an interface constraint is satisfied by ‘implements’ statements from multiple modules
  - how to ensure an interface implementation is used consistently?
  - introduce something like Rust’s Orphan Rules?
- Should early resolution include lifetime checking and nil checking?

INTERFACES

Next Steps: Open Design Questions (continued)

- How do interface generics interact with traditional generics? (issue #16985)
  - what is the transition path from traditional to interface generics?
  - can traditional generics be mixed with interface generics?
  - can required functions be interface-constrained?

- How to handle types and params within interface-constrained functions?
  - cannot resolve type/param-dependent code “once and for all”
  - introduce a compile-time way to check if a given type implements an interface?

- How to reason about generic ‘implements’ statements? (issue #16820)
  - how to detect that a required function is implemented for some instantiations and not others?
  - how to resolve the body of a generic implementation?
  - should ‘implements’ statements with class types be generic over memory management? nilability? (#17262)


SUPPORTING RESIZED ARRAYS OF
NON-NILABLE CLASSES

SUPPORTING RESIZED ARRAYS OF NON-NILABLE CLASSES


INTRODUCTION

RESIZED ARRAYS OF NON-NILABLE

Refresher: Non-nilable Class Types

- Non-nilable class types were added in Chapel 1.20
  - A nilable class is differentiated from a non-nilable class by a postfix '?' on the type

```
class C { var x = 0; }
var x1 = new shared C();     // Create a non-nilable class variable of type 'shared C'
var x2 = new shared C?();    // Create a nilable class variable of type 'shared C?'
var x3 = x2!;                // Create a non-nilable class variable by unwrapping 'x2'
```

- Use of non-nilable class types offers some big advantages over nilable class types
  - The compiler can remove implicit 'nil' checks for non-nilable types
    - Because it knows a non-nilable type can never be 'nil'
  - It makes nilablility checks explicit instead of implicit
    - By unwrapping with the '!' operator

RESIZED ARRAYS OF NON-NILABLE

Problem: Cannot Resize an Array of Non-nilable Classes

- Arrays of non-nilable classes cannot have their domains resized

```
class C { var x = 0; }
var D = {0..0};
var A: [D] shared C = [new shared C()];

// Halt: can't resize domains whose arrays' elements don't have default values
D = {0..1};
```

- The resize of the domain 'D' will halt at runtime
  - The array 'A' will try to default-initialize newly added slots
  - Yet, a non-nilable class type has no default value (since it cannot be 'nil')

RESIZED ARRAYS OF NON-NILABLE

Partial Solution: Resize Method on Arrays

- When first tackling this problem, we considered adding a resize method to arrays
  - However, this only works when the array is 1:1 with its domain

```
var D = {0..0};
var A1: [D] shared C = [new shared C()];

// A helper method to resize an array that works when it is 1:1 with its domain
A1.resize({0..1}, new shared C());

var A2 = A1;

// Error: Cannot resize an array that is not 1:1 with its domain
A1.resize({0..2}, new shared C());
```

RESIZED ARRAYS OF NON-NILABLE

Partial Solution: Resize Method on Domains

- Next, we explored adding a resize function to domains that takes a factory record
  - However, this only works when the initialization logic is straightforward

```
var D = {0..0};
var A1: [D] shared C = [new shared C()];
var A2: [D] shared C = A1;

// Resizer will initialize the new slot of 'a1'
// with 'new shared C()'.
var myResizer = new resizer();
D.resize({0..1}, myResizer);
```

```
record resizer {
  // Resizer defines a `this` method which takes the
  // new domain, the current index, and the
  // element type of the array.
  proc this(dom, idx, type eltType) {
    return new eltType(idx);
  }
}
```

RESIZED ARRAYS OF NON-NILABLE

Partial Solutions are Insufficient

- What we learned: resize functions on domains and arrays aren't flexible enough for all cases
  - For arrays, when the array is not the only array declared over a domain
  - For domains, when the initialization logic is complex:

```
var D = {0..1};
var A1: [D] shared C = [new shared C()];
var A2: [D] shared C = A1;

// What if I want to fill the new slots of 'A1' and 'A2' in different ways?
// No way to know which array I'm resizing.
// The 'this' method can't have access to the array itself,
// because some slots aren't initialized.
record resizer {
  proc this(dom, idx, type eltType) {
    return new eltType(idx);
  }
}
```

RESIZED ARRAYS OF NON-NILABLE

Exploring More General Solutions

- The following slides explore some more general solutions to the ‘resize’ problem

- Each successive solution adds more implementation complexity
  - Imagine it as a point on a sliding scale

- Complexity and benefit do not necessarily grow in a linear fashion
  - A particular solution may be a huge jump in complexity…
  - …but only provide a minor gain in benefit!

- Solutions need not be mutually exclusive
  - Not every approach follows naturally from the previous one
  - We can revisit individual features later

RESIZED ARRAYS OF NON-NILABLE

Evaluation Criteria

- Ideally, the approach we settle on should satisfy the following constraints:
  - Avoid consuming any extra memory
    - Rely on nilable/non-nilable classes having the same memory footprint
  - Be easy to identify and search for within a codebase
    - To quickly identify weak points in the code

RESIZED ARRAYS OF NON-NILABLE

Brief Description of the Next Sections

- The following sections present solutions at different levels of abstraction

1. Solve the ‘resize problem’ with low-level functions
2. Improve approach 1 by adding some abstractions and runtime checks
3. Improve approach 2 by adding a new language feature
4. Improve approach 2 by adding more abstractions and potentially a new language feature
5. Explore an alternative new language feature that is higher-level

POTENTIAL SOLUTIONS: LOW-LEVEL FUNCTIONS

RESIZED ARRAYS OF NON-NILABLE

Baseline Solution Using Low-level Functions

- A baseline solution uses a few different low-level functions to resize arrays
  - This code snippet proposes a complete solution to the problem

```
use Memory.Initialization;

class C { var x = 0; }

// The following steps will resize 'D' and initialize the new element of 'A'
var D = {0..0};
var A: [D] shared C = [new shared C()];
A.suspendElementManagement();
D = {0..1};
moveInitialize(A[1], new shared C());
A.resumeElementManagement();
```

RESIZED ARRAYS OF NON-NILABLE

Methods to Suspend and Resume Element Management

- A baseline solution uses a few different low-level functions to resize arrays
  - This code snippet proposes a complete solution to the problem

```
use Memory.Initialization;

class C { var x = 0; }

// The following steps will resize 'D' and initialize the new element of 'A'
var D = {0..0};
var A: [D] shared C = [new shared C()];
A.suspendElementManagement();
D = {0..1};
moveInitialize(A[1], new shared C());
A.resumeElementManagement();
```

The first two pieces of this solution are the calls 'suspendElementManagement()' and 'resumeElementManagement()'.

RESIZED ARRAYS OF NON-NILABLE

Explaining ‘suspendElementManagement()’ and ‘resumeElementManagement()’

- When the method ‘suspendElementManagement()’ is called, an array will not:
  - Deinitialize elements when it goes out of scope or is shrunk
  - Initialize new elements when grown

- Calling ‘resumeElementManagement()’ undoes the above changes

```
var D = {0..0};
var A: [D] shared C = [new shared C()];

// Suspend element management so that new elements of ‘A’ are not initialized
A.suspendElementManagement();
D = {0..1};
moveInitialize(A[0], new shared C());
// Resume element management so that ‘A’ will deallocate its elements
A.resumeElementManagement();
```

RESIZED ARRAYS OF NON-NILABLE

Initializing New Elements Using a Move

- A baseline solution uses a few different low-level functions to resize arrays
  - This code snippet proposes a complete solution to the problem

```
use Memory.Initialization;

class C { var x = 0; }

// The following steps will resize 'D' and initialize the new element of 'A'
var D = {0..0};
var A: [D] shared C = [new shared C()];
A.suspendElementManagement();
D = {0..1};
moveInitialize(A[1], new shared C());
A.resumeElementManagement();
```

The last piece of this solution is the call to 'moveInitialize()'.

RESIZED ARRAYS OF NON-NILABLE

Explaining the ‘moveInitialize()’ Function

- The ‘moveInitialize()’ function is one of a few functions that perform a ‘low-level move’
  - The ‘Memory.Initialization’ module contains ‘moveInitialize()’ and friends
    - It is new as of Chapel 1.24.0

- The ‘moveInitialize()’ function takes two arguments, a left-hand-side and a right-hand-side
  - The LHS is overwritten by the RHS without being deinitialized first
    - This makes ‘moveInitialize()’ suitable for first-time initialization

```
D = {0..1};                                  // Resize ‘D’ to have 2 elements
moveInitialize(A[0], new shared C());        // Initialize ‘A[0]’ with ‘new shared C()’
A.resumeElementManagement();                 // Remember to resume element management
```

- View the docs for the ‘Memory.Initialization’ module

RESIZED ARRAYS OF NON-NILABLE

Review: Baseline Solution Using Low-level Functions

- A baseline solution uses a few different low-level functions to resize arrays
  - This code snippet proposes a complete solution to the problem

```
use Memory.Initialization;

class C { var x = 0; }

// The following steps will resize 'D' and initialize the new element of 'A'
var D = {0..0};
var A: [D] shared C = [new shared C()];
A.suspendElementManagement();
D = {0..1};
moveInitialize(A[1], new shared C());
A.resumeElementManagement();
```

RESIZED ARRAYS OF NON-NILABLE

Evaluating the Current Solution Space

- One action item is required to implement this solution in full
  - Add array methods to suspend/resume element management

- There are some problems with this solution that could be improved upon
  - For N non-nilable arrays over a domain, we must suspend/resume element management N times
  - There's no way to know if we initialized every element of an array correctly
  - It can be hard to identify where unsafe actions take place
  - Every call to 'suspendElementManagement()', must be matched by 'resumeElementManagement()'
  - The code is low-level, and errors lead to crashes/leaks

POTENTIAL SOLUTIONS: ADDING CHECKS AND ABSTRACTIONS

RESIZED ARRAYS OF NON-NILABLE

Goal: Improve the Baseline Solution

- This section tries to improve the ergonomics of the baseline solution
  - Yet, without adding any new language features

- The section highlights a problem, then offers one or more solutions
  - There may be multiple ways to address a particular problem

```
// Here's the code that is needed for the baseline solution
var D = {0..0};
var A: [D] shared C = [new shared C()];
A.suspendElementManagement();
D = {0..1};
moveInitialize(A[1], new shared C());
A.resumeElementManagement();
```

RESIZED ARRAYS OF NON-NILABLE

Problem: Too Much Boilerplate Code

- When resizing more than one array at a time, the baseline solution has too much boilerplate
  - For N non-nilable arrays over a domain, we have suspend/resume element management N times

```
var D = {0..0};
var A1, A2: [D] shared C = [new shared C()];   // Declare two arrays over 'D'

A1.suspendElementManagement();   // Each array must suspend element management
A2.resumeElementManagement();

D = {0..1};
moveInitialize(A1[1], new shared C());
moveInitialize(A2[1], new shared C());

A1.suspendElementManagement();   // Each array must resume element management
A2.resumeElementManagement();
```

RESIZED ARRAYS OF NON-NILABLE

Solution 1: Iterate Over a Domain's Arrays

- We could solve this problem by adding an iterator to loop over all a domain's arrays
  - Loop over arrays and make them suspend/resume element management

```
var D = {0..0};
var A1, A2: [D] shared C = [new shared C()];   // Declare two arrays over 'D'

// Loop over all the arrays of 'D' and make them suspend element management
for arr in D.arrays() do arr.suspendElementManagement();

D = {0..1};
moveInitialize(A1[1], new shared C());
moveInitialize(A2[1], new shared C());

// Loop over all the arrays of 'D' and make them resume element management
for arr in D.arrays() do arr.resumeElementManagement();
```

RESIZED ARRAYS OF NON-NILABLE

Solution 2: A Function on Domains

- We could solve this problem by adding a new method to the domain type
  - It would cause all arrays to suspend/resume element management at once

```
var D = {0..0};
var A1, A2: [D] shared C = [new shared C()];   // Declare two arrays over 'D'


D.allArraysSuspendElementManagement();

D = {0..1};
moveInitialize(A1[1], new shared C());
moveInitialize(A2[1], new shared C());

D.allArraysResumeElementManagement();
```

RESIZED ARRAYS OF NON-NILABLE

Problem: No Way to Know if We Resized Correctly

- When resizing an array with the baseline approach, we could skip an element and never know it
  - Best outcome would be a crash right away, but you might not get that

```
var D = {0..0};
var A: [D] shared C = [new shared C()];


A.suspendElementManagement();

D = {0..2};   // Resize to 3 elements total
moveInitialize(A[1], new shared C());   // Initialize A[1]


A.resumeElementManagement();

writeln(A[2]);   // We forgot to initialize A[2], so now we get undefined behavior—the program may crash…or it may not!
```

RESIZED ARRAYS OF NON-NILABLE

Solution: Add an Optional Check

- We can attach some sort of check to ‘resumeElementManagement()’
  - It would iterate through the elements of a non-nilable array and check to see if any are nil

```
var D = {0..0};
var A: [D] shared C = [new shared C()];


A.suspendElementManagement();

D = {0..2};   // Resize to 3 elements total
moveInitialize(A[1], new shared C());   // Initialize A[1]


A.resumeElementManagement();   // Halt: One or more non-nilable arrays over ‘D’ contains uninitialized elements
```

RESIZED ARRAYS OF NON-NILABLE

Solution: Add an Optional Check

- We can attach some sort of check to ‘resumeElementManagement()’
  - It would iterate through the elements of a non-nilable array and check to see if any are ‘nil’
  - The check could be optional, via an argument

```
A.resumeElementManagement(checks=true);
```

- Such a check would add a small linear cost to all resized non-nilable arrays
  - To use ‘nil’ as a sentinel value, new elements must be zeroed
    - This zeroing may interfere with first-touch
- Adding the check to ‘resumeElementManagement()’ could work as well
  - But in some cases, suspend/resume calls occur far apart

RESIZED ARRAYS OF NON-NILABLE

Evaluating the Current Solution Space

- Despite some initial abstractions, three problems remain:
  - It is still difficult to search for and identify unsafe code in large programs
    - Even when abstracted into functions, the function to search for varies based on codebase
  - Every call to ‘suspendElementManagement()’, must be matched by ‘resumeElementManagement()’
    - Or else a resized array will never deinitialize its elements
  - Incorrect code will lead to leaks and crashes
    - Checks cannot catch misuse of uninitialized array slots

POTENTIAL SOLUTIONS: UNSAFE BLOCKS

RESIZED ARRAYS OF NON-NILABLE

Goal: Make Unsafe Code Easier to Find

- A problem with the ‘baseline’ approach is that it relies heavily on convention
  - Nothing keeps a user from making unsafe function calls anywhere in the code
    - Many different functions and call sites to keep track of and search for

- Introduce new syntax that encourages users to avoid spreading out unsafe code
  - The Rust language has a similar feature, called an ‘unsafe block’

  - In Rust an unsafe block gives users ‘unsafe superpowers’:
    - Dereference a raw pointer (kind of like c_ptr)
    - Call an unsafe function or method
    - Access mutable static memory
    - Access union fields

  - Unsafe blocks are a static (lexically-scoped) construct

RESIZED ARRAYS OF NON-NILABLE

Explain the ‘unsafe’ Block Modifier

- We could introduce the keyword ‘unsafe’ as a modifier, attachable to any block
  - Certain functions and methods could only be called within an ‘unsafe’ block
    - Exactly like a regular block in every other respect

```
var D = {0..0};
var A: [D] shared C = [new shared C()];

// Our previous code pattern, but now certain calls can only be made inside the scope of a “unsafe” block
unsafe {
  A.suspendElementManagement();
  D = {0..1};
  moveInitialize(A[1], new shared C());
  A.resumeElementManagement();
}
```

RESIZED ARRAYS OF NON-NILABLE

Explain the ‘unsafe’ Function Modifier

- We could introduce the keyword ‘unsafe’ as a modifier attachable to any function
  - Marking a function as ‘unsafe’ makes it callable only within an unsafe block

- Mark all the functions used in our baseline solution as ‘unsafe’
  - They can only be called from within an unsafe block

```
unsafe proc array.enterNoinitState();

unsafe proc array.leaveNoinitState();

unsafe proc moveInitialize(ref lhs: ?t, in rhs: t);
```

RESIZED ARRAYS OF NON-NILABLE

Benefits of Unsafe Blocks

- Judicious use of unsafe blocks will make dangerous low-level code easier to find
  - Now a user can just search for ‘unsafe’ instead of having to know every function

- Unsafe blocks and functions provide value beyond this specific use-case
  - Users can create their own unsafe functions
    - E.g., when wrapping a low-level C API

RESIZED ARRAYS OF NON-NILABLE

Downsides of Unsafe Blocks

- Unsafe blocks still rely on convention to be effective
  - Nothing prevents a user from wrapping every function in an ‘unsafe’ block

```
module Foo {
  // E.g., wrapping ‘main’ in an unsafe block…
  proc main() {
    unsafe { … }
  }
}
```

  - To avoid this, we can encourage best practices like Rust
    - The Rust programmer’s handbook advises users to:
      - Keep unsafe blocks short and sweet
      - Wrap unsafe blocks in a safe API

RESIZED ARRAYS OF NON-NILABLE

Questions About Unsafe Blocks

- Unclear how tightly we would bolt ‘unsafe’ into the language beyond newly added functions
  - Do we retrofit ‘c_ptr' derefences and existing low-level APIs into ‘unsafe’ functions?
    - This might introduce a lot of new warning/error messages
  - Do we mark certain collection access patterns as ‘unsafe’?
    - Recall that indexing into a collection when ‘parSafe=true’ was deprecated in Chapel 1.23

```
use Map;
var m: map(int, int, parSafe=true);
m[0] = 0;   // Error: ‘map.this()’ can only be called in an unsafe block
```

RESIZED ARRAYS OF NON-NILABLE

Evaluating the Current Solution Space

- Despite further abstractions, two problems remain:
  - Every call to ‘suspendElementManagement()’, must be matched by ‘resumeElementManagement()’
    - Or else a resized array will never deinitialize its elements
  - Incorrect code will still lead to leaks and crashes
    - Unsafe blocks only make such errors easier to find

POTENTIAL SOLUTIONS: AUTOMATICALLY RESUME ELEMENT MANAGEMENT

RESIZED ARRAYS OF NON-NILABLE

Problem: Forgetting to Call ‘resumeElementManagement()’

- A problem with the ‘baseline’ approach is that users must call ‘resumeElementManagement()’
  - Forgetting to do so means the resized array will no longer deinitialize elements

```
var D = {0..0};
var A: [D] shared C = [new shared C()];


A.suspendElementManagement();
D = {0..1};
moveInitialize(A[1], new shared C());
// Forgetting to call ‘resumeElementManagement()’ means ‘A’ will not
// deinitialize its three elements when it goes out of scope
```

RESIZED ARRAYS OF NON-NILABLE

Solution 1: Use a ‘defer’ Block

- Use a ‘defer’ block to ensure ‘suspendElementManagement()’ is called on all paths out of a block

```
var D = {0..0};
var A: [D] shared C = [new shared C()];

{
  A.suspendElementManagement();
  // By using a ‘defer’ block, we couple the calls to suspend/resume element management
  // more tightly together, and ensure we never forget to call ‘resumeElementManagement()’
  defer { A.resumeElementManagement(); }
  D = {0..1};
  moveInitialize(A[1], new shared C());
}
```

RESIZED ARRAYS OF NON-NILABLE

Solution 2: Using a Scope-Based Token

- Rely on the deinitializer of a scope-based token to ensure ‘resumeElementManagement()’ is called
  - The token will call ‘resumeElementManagement()’ for arrays declared over a domain when it goes out of scope

```
var D = {0..0};
var A: [D] shared C = [new shared C()];

{
  // The ‘token’ will suspend/resume element management for all arrays over ‘D’
  const token: unsafeResizeToken = D.unsafeResizeAllArrays({0..1});
  moveInitialize(A[1], new shared C());
}
```

RESIZED ARRAYS OF NON-NILABLE

Solution 2: Using a Scope-Based Token

- Below is an example of what the 'unsafeResizeToken' record might look like

```
record unsafeResizeToken {
  ref domainToResize;   // Imagine that we can store record fields by 'ref'

  // All arrays over 'domainToResize' call 'suspendElementManagement()' in the token initializer
  proc init(ref domainToResize) {
    this.domainToResize = domainToResize;
    this.complete();
    for A in domainToResize.arrays() do A.suspendElementManagement();
  }

  // All arrays over 'domainToResize' call 'resumeElementManagement()' in the token deinitializer
  proc deinit() {
    for A in domainToResize.arrays() do A.resumeElementManagement();
  }
}
```

RESIZED ARRAYS OF NON-NILABLE

Solution 3: Adding a Context Manager to the Language

- The token-based approach has some problems that make it less than ideal
  - It will not work if the user does not capture the token in a variable

```
// Failing to capture the token here means that it will be destroyed immediately
D.unsafeResizeAllArrays({0..1});
moveInitialize(A[1], new shared C());
```

  - It will not work if the token and array being resized are declared in the same scope
    - The token must be shorter-lived than the array

```
var A: [D] shared C = [new shared C()];
const token = unsafeResizeAllArrays({0..1});
moveInitialize(A[1], new shared C());
// The array 'A' will never resume element management, because 'token' is at the same scope
```

RESIZED ARRAYS OF NON-NILABLE

Solution 3: Adding a Context Manager to the Language

- To fix these problems, we could add support for a *context manager* to the language
  - Like those offered by the Python language
    - Syntax looks like: 'manage' <expression> { … }

```
// Here the 'token' still exists, but is used by the context manager instead
// of waiting for it to go out of scope.
manage D.unsafeResizeAllArrays({0..1}) {
  moveInitialize(A[1], new shared C());
}
```

- The context manager accepts a value of any type that defines the appropriate methods
  - The methods are named 'enterThis()' and 'leaveThis()'

- The context manager calls 'enterThis()' at scope start and 'leaveThis() at scope end
  - The token record used in solution #2 should define these methods

RESIZED ARRAYS OF NON-NILABLE

Solution 3: Adding a Context Manager to the Language

- How the token record might be implemented in a context manager-based approach

```
record unsafeResizeToken {
  ref domainToResize;   // Imagine that we can store record fields by 'ref'

  proc init(ref domainToResize) { this.domainToResize = domainToResize; }

  // The 'enterThis()' method makes all arrays over 'D' call 'suspendElementManagement()'
  proc enterThis() {
    for A in domainToResize.arrays() do A.suspendElementManagement();
  }

  // The 'leaveThis()' method makes all arrays over 'D' call 'resumeElementManagement()'
  proc leaveThis() {
    for A in domainToResize.arrays() do A.resumeElementManagement();
  }
}
```

RESIZED ARRAYS OF NON-NILABLE

Evaluating the Current Solution Space

- Proposed three different ways to automate calling ‘resumeElementManagement()’
  - Use ‘defer’ blocks
  - Use a scope-based token
  - Add context managers to the language

- Despite additional abstractions, still left with a problem that can’t be ignored:
  - Incorrect code will still lead to leaks and crashes
    - Unsafe blocks only make such errors easier to find

POTENTIAL SOLUTIONS: RESIZE BLOCKS

RESIZED ARRAYS OF NON-NILABLE

Introducing the Resize Block

- Having to write unsafe low-level code is arguably disappointing
  - It would be nice if there was a higher-level way of doing things

- Add a new language feature to increase abstraction
  - We'll call this feature the 'resize' block

```
var D = {0..0};
var A: [D] shared C = [new shared C()];'

// Using a resize block to resize 'D' and initialize the new elements of 'A'
resize {
  D = {0..1};
  A[1] move= new shared C();
}
```

RESIZED ARRAYS OF NON-NILABLE

Goals of the Resize Block

- The resize block provides at least as many features as the previous approaches
  - It automates and hides calls to ‘suspendElementManagement()’ and ‘resumeElementManagement()’
  - It performs consistency checks to make sure all array elements are initialized
  - It is easy to identify in code due to the keyword ‘resize’

- The resize block makes stronger safety guarantees than previous approaches
  - Newly added array elements cannot be used until they are initialized
  - Consistency checks are always performed instead of being optional

- It provides safety through a combination of compile-time and runtime checks
  - If any edge cases are found, support will be improved to cover them

RESIZED ARRAYS OF NON-NILABLE

Rules Within Resize Blocks

- The ‘resize’ block behaves like a regular block with some additional restrictions

  - Newly added elements of a resized array cannot be used until they are initialized

```
var D = {0..0};
var A: [D] shared C = [new shared C()];
resize {
  D = {0..1};
  writeln(A[1]);   // Error: Element of ‘A’ is not initialized yet
}
```

  - The operator ‘move=’ is the only function that can initialize newly added array elements

```
resize {
  D = {0..1};
  A[1] move= new shared C();
  writeln(A[1]);   // OK, A[1] has been initialized first
}
```

RESIZED ARRAYS OF NON-NILABLE

Rules Within Resize Blocks

- The ‘resize’ block functions like a regular block with some additional restrictions

  - All newly added elements of a resized array must be initialized before exiting the block

```
var D = {0..0};
var A: [D] shared C = [new shared C()];
resize {
  D = {0..2};
  A[1] move= new shared C();
}   // Halt: resized array ‘a’ contains uninitialized elements
```

RESIZED ARRAYS OF NON-NILABLE

Adding the ‘move=’ Operator

- Add ‘move=’ to the language as a variation of the simple assignment (‘=’) operator
  - It provides behavior like the ‘moveInitialize()’ function used in previous approaches

- Unlike ‘moveInitialize()’, safety checks will ensure uses of ‘move=’ are valid

```
var x = new shared C();
x move= new shared C();    // Error: illegal use of ‘move=‘ on initialized variable ‘x’
```

- The ‘move=’ operator behaves like the ‘moveInitialize()’ function
  - An error occurs if the right-hand-side of ‘move=’ is used again

```
var r1: someRecord;
var r2 move= r1;    // Error: cannot use ‘r1’ to move-initialize ‘r2’ because it’s still in use
writeln(r1);        // Note: ‘r1’ is next used here
```

RESIZED ARRAYS OF NON-NILABLE

Resize Block Rules Might Need to be Broadened

- The resize block introduces two new rules:
  - Elements of a resized array cannot be used until they are initialized
  - Only the 'move=' operator can initialize newly added array elements

- These rules may not be broad enough to eliminate all unsafe behavior
  - Here is a program that may require additional rules

```
var D = {0..0};
var A: [D] shared C = [new shared C()];

proc getNewValue() { return A[1]; }   // How do we analyze the 'A[1]' within 'getNewValue()'?

resize {
  D = {0..1};
  A[1] move= getNewValue();   // A[1] would still point to uninitialized memory after this statement
}
```

SUMMARY

RESIZED ARRAYS OF NON-NILABLE

Summary

- The following slides illustrate code using a combination of each approach
  - Five different levels (or mix and match):
    - Low-level functions
    - Also add abstractions and safety checks
    - Also add unsafe blocks
    - Also automate calls to ‘resumeElementManagement()’
    - Explore resize blocks

RESIZED ARRAYS OF NON-NILABLE

Example: Low-Level Functions

- Solution using low-level functions and methods

```
var D = {0..0};
var A1, A2: [D] shared C = [new shared C()];


  A1.suspendElementManagement();
  A2.suspendElementManagement();

  D = {0..1};

  moveInitialize(A1[1], new shared C());
  moveInitialize(A2[1], new shared C());

  A1.resumeElementManagement();
  A2.resumeElementManagement();
```

RESIZED ARRAYS OF NON-NILABLE

Example: Abstractions and Safety Checks

- Solution that adds abstractions and checks

```
var D = {0..0};
var A1, A2: [D] shared C = [new shared C()];



  D.suspendElementManagementAllArrays();

  D = {0..1};

  moveInitialize(A1[1], new shared C());
  moveInitialize(A2[1], new shared C());

  D.resumeElementManagementAllArrays(checks=true);
```

RESIZED ARRAYS OF NON-NILABLE

Example: Unsafe Blocks

- Solution that adds abstractions, checks, and unsafe blocks

```
var D = {0..0};
var A1, A2: [D] shared C = [new shared C()];

unsafe {
  D.suspendElementManagementAllArrays();


  D = {0..1};

  moveInitialize(A1[1], new shared C());
  moveInitialize(A2[1], new shared C());

  D.resumeElementManagementAllArrays(checks=true);
}
```

RESIZED ARRAYS OF NON-NILABLE

Example: Automate Calls to ‘resumeElementManagement()’ Using Defer

- Solution that adds abstractions, checks, unsafe blocks, and defer

```
var D = {0..0};
var A1, A2: [D] shared C = [new shared C()];

unsafe {
  D.suspendElementManagementAllArrays();
  defer { D.resumeElementManagementAllArrays(checks=true); }

  D = {0..1};

  moveInitialize(A1[1], new shared C());
  moveInitialize(A2[1], new shared C());
}
```

RESIZED ARRAYS OF NON-NILABLE

Example: Automate Calls to ‘resumeElementManagement()’ Using Tokens

- Solution that adds abstractions, checks, unsafe blocks, and defer

```
var D = {0..0};
var A1, A2: [D] shared C = [new shared C()];

unsafe {
  const token = D.unsafeResizeAllArrays({0..1}, checks=true);




  moveInitialize(A1[1], new shared C());
  moveInitialize(A2[1], new shared C());
}
```

RESIZED ARRAYS OF NON-NILABLE

Example: Automate Calls to ‘resumeElementManagement()’ Using Context Managers

- Solution that adds abstractions, checks, unsafe blocks, and context managers

```
var D = {0..0};
var A1, A2: [D] shared C = [new shared C()];

unsafe manage D.unsafeResize({0..1}, checks=true) {


  D = {0..1};


  moveInitialize(A1[1], new shared C());
  moveInitialize(A2[1], new shared C());
}
```

RESIZED ARRAYS OF NON-NILABLE

Example: Resize Blocks

- Solution that uses resize blocks

```
var D = {0..0};
var A1, A2: [D] shared C = [new shared C()];

resize {


  D = {0..1};


  A1[1] move= new shared C();
  A2[1] move= new shared C();
}
```

RESIZED ARRAYS OF NON-NILABLE

Next Steps

- Determine what degree of support we should provide for resizing arrays of non-nilable
- Implement that decision and gain experience with it
- Provide guidance for users who wish to write their own collections of non-nilable values


STANDARD LIBRARY
STABILIZATION

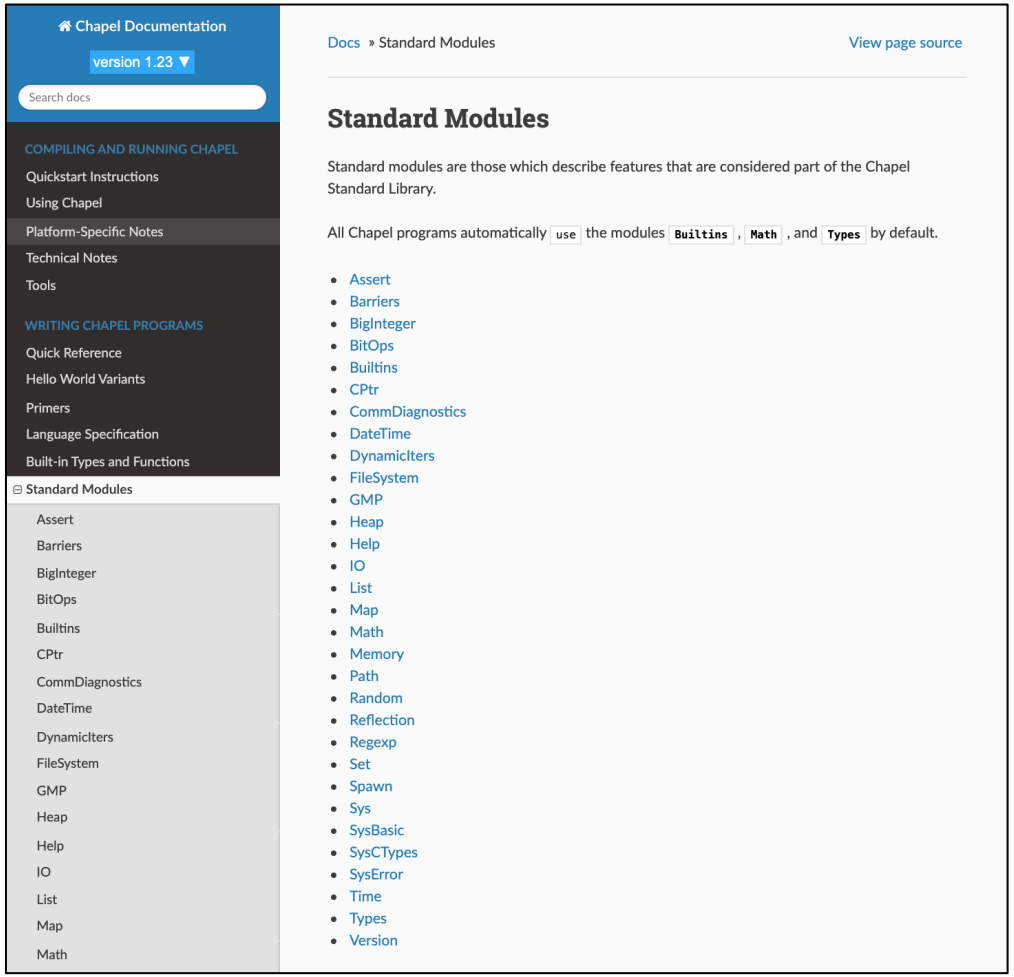
STANDARD LIBRARY STABILIZATION

Background

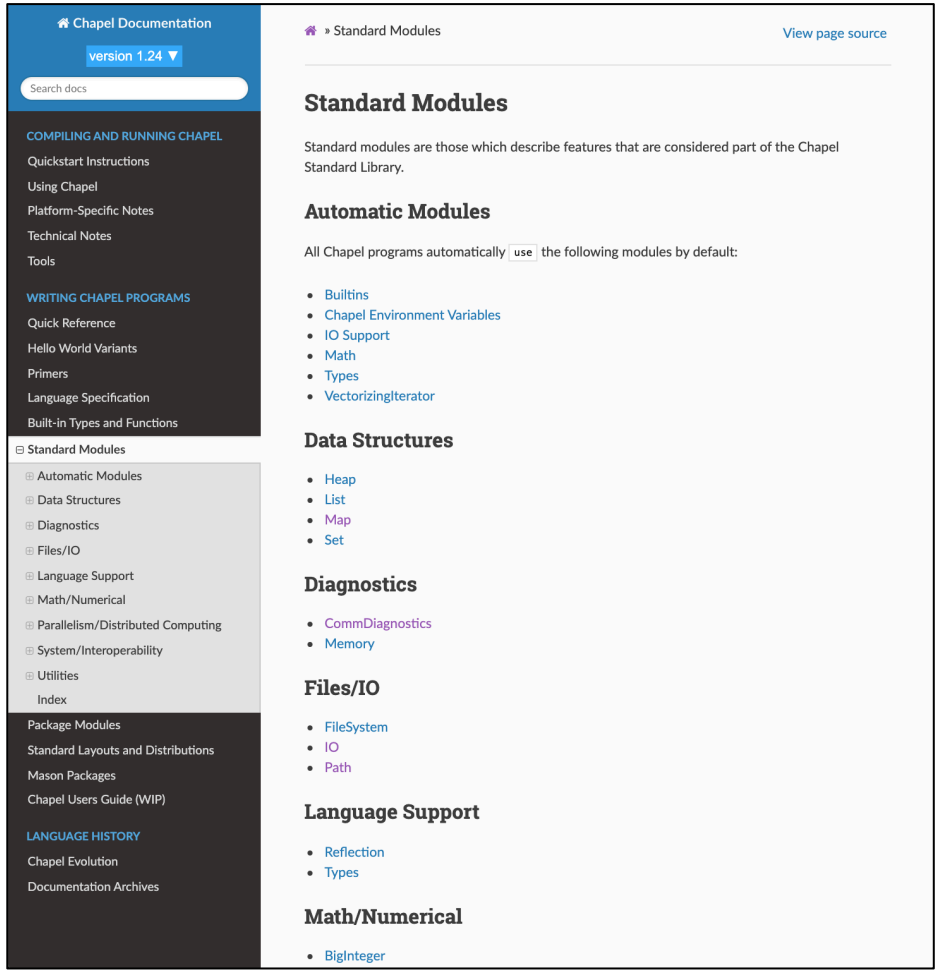
- In recent discussions, we've agreed that Chapel 2.0 should include stabilizing the standard libraries:
  - Builtins, Chapel Environment Variables
  - Heap, List, Map, Set
  - CommDiagnostics, Memory
  - FileSystem, Automatic IO, IO, Path
  - Reflection, Types
  - BigInteger, BitOps, GMP, Math, Random
  - Barriers, DynamicIters, VectorizingIterator
  - CPtr, Spawn, Sys, SysBasic, SysCTypes, SysError
  - DateTime, Help, Regexp, Time, Version
  - HaltWrappers
- plus library-like features on types defined by the language, such as methods and functions on standard types:
  - String, Bytes
  - Ranges, Domains, Arrays
  - Shared, Owned

CATEGORIZED STANDARD LIBRARY DOCUMENTATION

Background: modules were listed alphabetically



This Effort: grouped modules into categories



STANDARD LIBRARY STABILIZATION

This Effort

- Started this effort via a bi-weekly review of a module, taking turns leading a discussion
- For each module, consider things like:
  - name of module itself
  - names of public types, enums, global variables, constants, …
  - names of public procedures, arguments
  - behaviors / definitions of all public symbols
- Have tried to avoid getting overly distracted with:
  - issues of documentation unrelated to behavior / breaking changes
  - future extensions to the module when they're non-breaking
- Resolve what we're able to, file issues for things that require more attention

STANDARD LIBRARY STABILIZATION

Status

- The bold entries below have now received a round of review:
  - **Builtins**, Chapel Environment Variables
  - Heap, List, Map, Set
  - **CommDiagnostics**, Memory
  - FileSystem, Automatic IO, IO, **Path**
  - Reflection, Types
  - BigInteger, BitOps, GMP, Math, Random
  - Barriers, DynamicIters, VectorizingIterator
  - CPtr, Spawn, Sys, SysBasic, SysCTypes, SysError
  - DateTime, Help, **Regexp**, **Time**, Version
  - HaltWrappers

  - String, Bytes
  - **Ranges**, Domains, Arrays
  - **Shared**, Owned

STANDARD LIBRARY STABILIZATION

- Time
- Builtins
- CommDiagnostics
- Shared
- Range
- Regexp
- Path

TIME MODULE

Background:

- The Time module contains a variety of functions and types related to the concept of time

Actions Taken:

- Agreed to migrate features to DateTime: 'Day', 'getCurrentTime', 'getCurrentDate', 'getCurrentDayOfWeek'
  - These features categorically belong in the DateTime module, but existed in Time because it predated DateTime
- Agreed that 'sleep' should live in a different module
  - 'sleep' seems closer to tasking than to 'Time'
- Agreed to rename the 'Timer' record to 'stopwatch' and update its interface (issue #16393)
  - 'Timer' behaves more like a stopwatch with start/stop behavior than a timer with set/start behavior
- After these changes, the Time module will only contain the stopwatch type
  - Agreed to merge it into a different module or rename the Time module

Open Discussions:

- Which module should contain 'sleep'?
- Where should stopwatch be defined? Possibly in DateTime? (possibly renaming it to 'Time'?) (issue #16394)
- Should all measurement-related modules (Time, CommDiagnostics, Memory) be grouped together?

BUILTINS MODULE

Background:

- The Builtins module was added in 1.23
  - Created to close a memory leak related to module initialization order
  - Functions in it were previously in ChapelBase and Assert

- Builtins is automatically included and contains these functions:
  - 'assert', 'compilerError', 'compilerWarning', 'compilerAssert', 'exit'

- In 1.23, the documentation had these top-level sections:
  - Built-in Types and Functions
  - Standard Modules

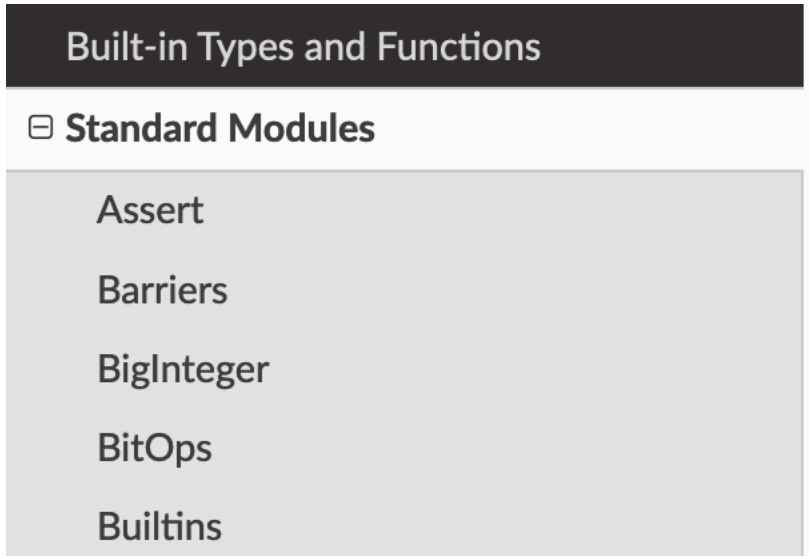
- But Builtins was listed under "Standard Modules" rather than "Built-in Types and Functions"

BUILTINS MODULE

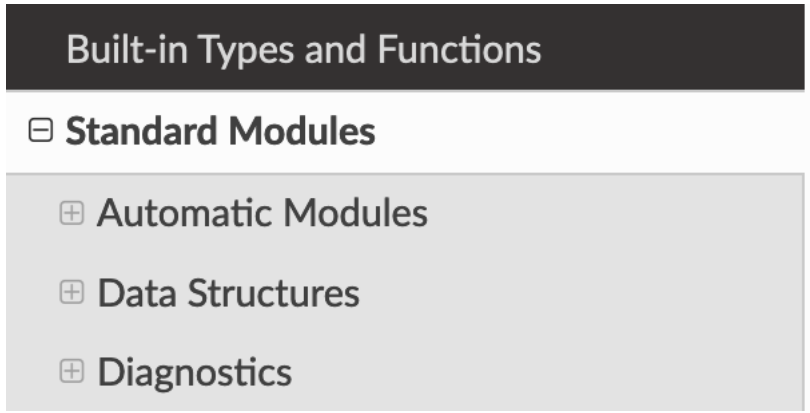
Actions Taken:

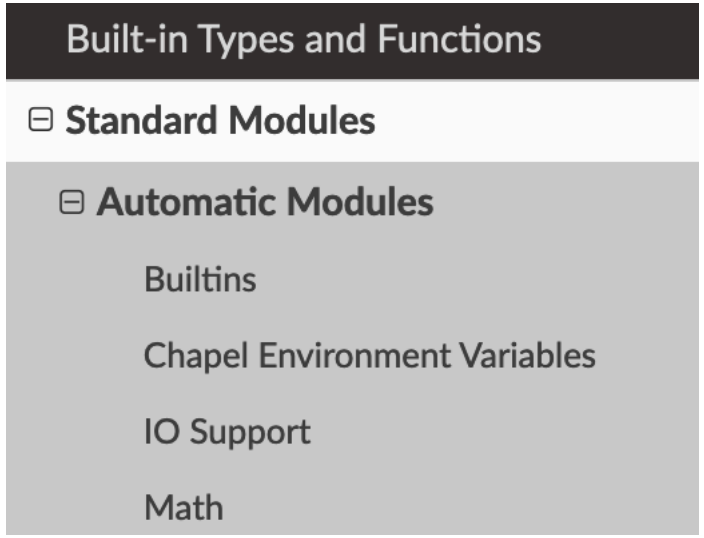
- Began changing the “Built-in Types and Functions” section to include only types
- Moved several modules from “Built-in Types and Functions” to “Standard Modules”
  – Chapel Environment Variables, IO Support, Vectorizing Iterator, and Misc. Functions
- These automatically-included standard modules are now listed in an “Automatic Modules” subsection

1.23 documentation



1.24 documentation





BUILTINS MODULE

Open Discussions:

- Should ‘compilerError’ be included by default? Should they be renamed? (issue #16807)

- Should it be possible to disable ‘assert’ checking? (issues #16810 and #12466)
  - Probably need an always-on variant and one that is off with ‘--fast’ but can be toggled with other flags

- The name of the Builtins module is arguably too broad. What organization and naming would be better?
  - Should ‘halt’, ‘exit’, and ‘assert’ be in a module named by topic? (issue #16808)
  - Should ‘compilerError’, ‘compilerWarning’, ‘compilerAssert’ be in a module named by topic? (issue #16809)

- Should we move the module docs for internal types to the spec? (issue #16814)
  - There is currently a lot of duplication between module documentation and spec sections for built-in types
  - Could then remove the “Built-in Types and Functions” section to reduce confusion with automatic standard modules

COMM DIAGNOSTICS MODULE

Background:

- The CommDiagnostics module gives users simple tools for reasoning about communication in their code
  - similar to taking timings by bracketing code with start/stop calls
  - supports two capabilities:
    - counting communication events (e.g., number of puts, gets, active messages, …)
    - logging communication events to the console

Open Discussions:

- Should we consider this module part of the Chapel 2.0 stabilization effort?
  - pro: it's a very useful module
  - con: it's somewhat low-level and subject to change as new types of communication events are introduced
- Concepts for revamping the design:
  - unify the two modes into one set of calls (issue #16955)
  - take a more object-oriented approach (issue #16956)
  - improve output in terms of flexibility, formats, event descriptions, etc. (issues #16957, #16958)

SHARED MODULE

Background:

- The Shared module contains the interface for ‘shared’ class instances
  - manages deletion of a class instance, allowing multiple owners

Actions Taken:

- Agreed to deprecate ‘retain’ and ‘clear’ functions because
  - these functions are not frequently used by users
  - the ‘retain’ function is the same as ‘create’
  - the ‘clear’ function can be replaced with ‘create(nil)’
- Documentation
  - Agreed to remove ‘init’ and ‘deinit’ functions from documentation since they are not user-facing
  - Agreed to remove the swap operator (‘<=>’) from the shared module API page to a more general type page

Open Discussions:

- Should we deprecate assignment (‘proc =’) or ‘create’ as they have the same functionality? (issue #17461)

RANGE MODULE

Background:

- 'range' is a well-established built-in datatype, implemented mostly in the internal module ChapelRange
- reviewed functions are mostly properties like 'isBounded', transformations like 'exterior', common ops like '=='

Actions Taken:

- upheld the style convention for a property method to be paren-less unless named 'is.....' or 'has.....' (#17128)
- decided to replace the ranges online docs page with a primer on ranges, to avoid duplicating the spec page

Open Discussions:

- should 'r+i' (addition of a range and an int), also 'i+r' and 'r-i', be specialized to return a new range? (#17101)
- should 'range.high' and 'range.low' return aligned bounds? (issue #17130)
- should 'range.size' return an 'int' regardless of the range's 'idxType' ? (issue #7530)
- should 'BoundedRangeType' be shortened to 'enum rangeBounds {low, high, both, none}' ? (issue #17126)
- better names for 'indexOrder', 'indexToOrder' (issue #14884) and 'stridable', 'aligned' (issue #17131)
- organization and/or renaming of range transformations (issue #17127) and set-like operations (issue #17125)
- other topics discussed in issues #17122, #17123, #17124, #17129, #17132

REGEXP MODULE

Background:

- The Regexp module provides regular expressions based on Google's RE2 library

Actions Taken:

- 'Regexp.compile()' option 'dotnl' has been deprecated and replaced with 'dotAll'
  - Makes '.' match any character, even newline
  - 'dotnl' was based on RE2's 'dotnl' option, while 'dotAll' is based on Python's 'DOTALL' option
- 'regexp.ok' and 'regexp.error()' have been deprecated
  - 'Regexp.compile()' already raises Chapel errors, so these were unnecessary
- In 1.25, the names 'Regexp', 'regexp' and 'BadRegexpError' will be deprecated
  - in favor of the shorter 'Regex', 'regex' and 'BadRegexError', respectively
- In 1.25, the name 'reMatch' will be deprecated in favor of 'regexMatch'

Open Discussions:

- Should the environment variable 'CHPL_REGEX', 'CHPL_RE2', or both be used? (issue #17306)
- Should Regexp be a standard module or a package module? (issue #17220)
- Should tertiary methods on 'string' and 'bytes' be defined in this module? (issue #17226)
- Which approach should be used to compile regular expressions in Chapel? (issue #17187)

PATH MODULE

Background:

- The Path module contains mostly string-based operations on paths
- Other relevant modules:
  - FileSystem: does operations on files themselves
  - IO: does operations on the contents of files

Actions Taken:

- Argument names were made more consistent
  - Some arguments were named ‘name’, but most were named ‘path’
  - In 1.24, they are all named ‘path’

Open Discussions:

- Should the Path module contain methods on the ‘file’ type?
  - Currently there are some tertiary methods on ‘file’ in the Path module
  - Should we move those to the IO module where ‘file’ is defined?
  - Should we also limit them to very basic operations, such as getting the basename and dirname from a ‘file’?
  - See issue #16748 for more on this discussion

STANDARD LIBRARY STABILIZATION

Next Steps

- Continue this effort, potentially increasing our rate
- Make sure we resolve open issues for modules that have been reviewed
  - re-reviewing if changes were substantial or the first pass felt incomplete

FOREACH LOOPS: INDICATING VECTORIZABILITY

FOREACH LOOPS
Background

- Chapel is designed so that ‘forall’ loops can be vectorized

- Additionally, there is a way to create a vectorized loop that does not create tasks:

```
for i in vectorizeOnly(1..n) { ... }
```

  - Useful for cases when the overhead of creating tasks is too high
  - ‘vectorizeOnly’ asserts to the compiler that the loop is vectorizable—the compiler does not check

- Vectorization hints are not currently generated by the C back-end
  - Instead, the C compiler will vectorize what it can without changing behavior vs. sequential code

- However, such hints can provide nice speedups when they are used appropriately
  - Prototype integration with the Region Vectorizer demonstrated nice speedups are possible

FOREACH LOOPS
Background: A Problem

- Prototype efforts revealed a problem with RandomStream followers
  - Before 1.23, all yielding follower loops were considered vectorizable
  - But RandomStream's follower loops mutate the RNG state per iteration as follows:

```
var state = initialRngState();
for i in 0..#nIters {
  state = updateRngState(state);
  yield produceValueFromState(state);
}
```

  - As a result, the RandomStream follower loops aren't actually vectorizable as written

- In 1.23, used an undocumented pragma to mark these RNG followers as non-vectorizable

FOREACH LOOPS

This Effort

- How can we solve the language design problem with such followers?
  - Parallel zippering with RandomStream is an important motivating example for Chapel's design
  - So, the pragma-based solution is not good enough

- Considered language design alternatives
- Chose the direction of adding a new 'foreach' keyword
  - concept: parallel, yet not as parallel as 'forall'—instead, use only the current task
    - 'for' < 'foreach' < 'forall' < 'coforall'

FOREACH LOOPS

This Effort: Alternatives Considered

- Could change vectorization strategy to come from leader iterators, rather than followers
  - But this would complicate simple examples and make the memory access order less obvious
  - In a vectorized setting, follower iterators would operate with a stride of the vector width

- Also considered using ‘vectorizeOnly’ to mark follower loops but that also has problems:
  - ‘vectorizeOnly’ asserts the loop is always vectorizable
  - Need something that says only that a specific follower loop is vectorizable
    - (and doesn’t say anything about code it is zippered with or code in the loop body)

- Settled on the strategy of adding the new ‘foreach’ keyword to solve this problem

FOREACH LOOPS

This Effort: Defining ‘foreach’

- ‘foreach’ indicates that a specific loop is vectorizable
  - In a follower iterator loop, applies to the follower loop body itself
    - not to the invoking ‘forall’ loop’s body or to zippered followers
  - In a zippered setting…

    ```
    foreach (a,b,c) in zip(...)
    ```

    … ‘foreach’ indicates the loop body is vectorizable but doesn’t assert anything about the invoked iterators

- ‘foreach’ will replace ‘vectorizeOnly’
- ‘foreach’ will support loop intents similar to ‘forall’
  - RandomStream iterators can use ‘in’/‘var’ intents to create vector-lane-local RNG state and support vectorization
- Follower iterators using ‘for’ will no longer be considered vectorizable
- Follower iterators will generally need to use ‘foreach’ to indicate vectorization is possible

- The keyword ‘foreach’ is reserved in 1.24 but completing the implementation is an ongoing effort

FOREACH LOOPS

Status and Next Steps

Status:

- Language design discussion around ‘foreach’ is complete
- ‘foreach’ keyword is reserved in 1.24
- Modules in the release still use a pragma for follower loops to indicate vectorizability

Next Steps:

- Implement full support for the ‘foreach’ keyword
- Address problems with cases like Diamond Tiling iterators (e.g., with ‘forwave’)
  - Such parallel “wavefront” iterators are not technically correct today because:
    - ‘forall’ indicates order independence
    - yet time-stepping has a required order, so these iterators are not completely order-independent
- Work towards enabling Region Vectorizer or something like it to better vectorize by default


TARGETING GPUS FROM CHAPEL

TARGETING GPUS FROM CHAPEL

Background

- GPU support is a natural fit for Chapel
  - Chapel has a goal of supporting any parallel algorithm on any parallel architecture
  - GPU support is frequently requested by Chapel users (and potential users)
  - The GPU execution model is a natural extension of Chapel's support for data parallelism

- GPU programming is tricky and unique
  - By offering a unified programming model for GPUs, Chapel's appeal would only increase

TARGETING GPUS FROM CHAPEL

Eventual Goal

- Easily execute parallel code on the GPU using conventional Chapel features
  - For example:
    - Stream:

```
on getGPUSublocale(){
  forall (a, b, c) in zip(A, B, C) {
    a = b + alpha*c;
  }
}
```

    - Reduction:

```
on getGPUSublocale(){
  var sum = 0;
  forall a in A with (+ reduce sum) {
    sum += a;
  }
}
```

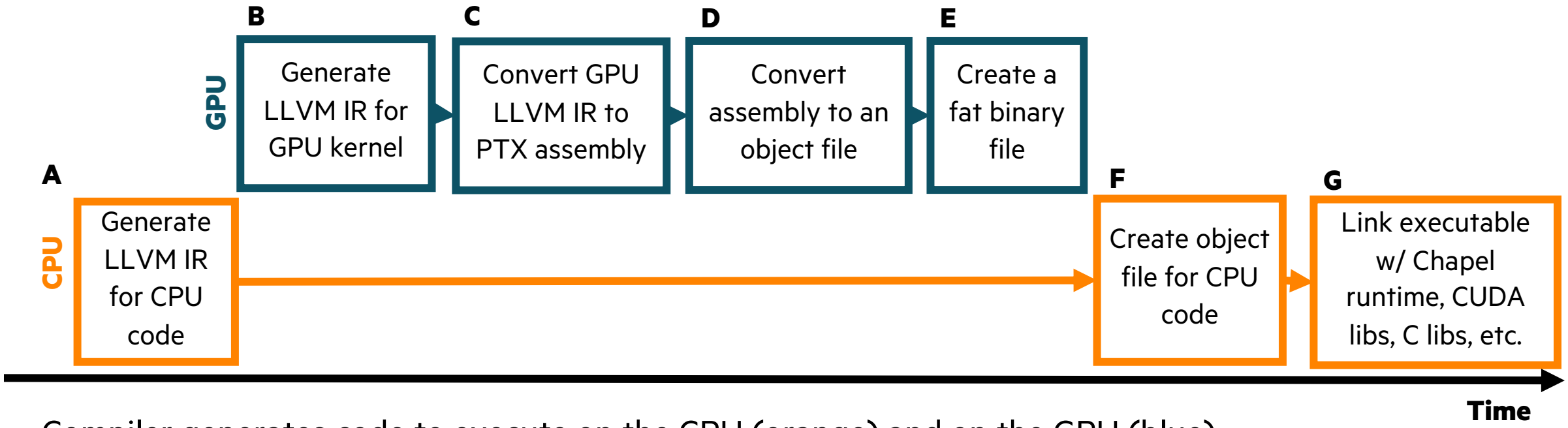
TARGETING GPUS FROM CHAPEL

This Effort

- Get Chapel code running on the GPU by whatever means necessary
  - Prototyped GPU code generation for a trivial kernel
  - Investigated data movement between CPU and GPU
  - Demonstrated launching kernels from Chapel

TARGETING GPUS FROM CHAPEL

Prototyping GPU Code Generation



- Compiler generates code to execute on the CPU (orange) and on the GPU (blue)
- GPU code generation strategy outlined by the blue boxes is intended to match Clang's CUDA support

TARGETING GPUS FROM CHAPEL

Prototyping GPU Code Generation (Example)

Chapel

```
pragma "codegen for GPU"
export proc add_nums(a: c_ptr(real(64))){
  a[0] = a[0]+5;
}

var output: real(64);
output = cudaFuncs();
```

Manual CUDA Driver API Calls

```
extern {
  #define FATBIN_FILE "chpl__gpu.fatbin"
  double cudaFuncs(){
    fatbinBuffer = <read FATBIN_FILE into buffer>
    cuModuleLoadData(&cudaModule, fatbinBuffer);
    cuModuleGetFunction(&function, cudaModule,
                        "add_nums");

    CUdeviceptr devBufferA;
    cuMemAlloc(&devBuffeA, sizeof(double));
    a = <user input>;
    cuMemcpyHtoD(devBufferA, &a, sizeof(double))

    void *KernelParams[] = { &devBufferA };
    cuLaunchKernel(function, <grid and block size>,
                   … , KernelParams, …));
    cuMemcpyDtoH(&a, devBufferA, sizeof(double));
    return a;
  }
}
```

Read fat binary and create a CUDA function

Move data to GPU

Launch CUDA function

Move data to CPU

TARGETING GPUS FROM CHAPEL

Data Movement From CPU to GPU

- The previous slide included many manual CUDA API calls

- The CUDA calls for data movement are out of place in Chapel
  - Chapel already supports implicit communication between locales as part of the PGAS model
  - Should ideally do the same between CPU and GPU [sub]locales as well

- Removed explicit function calls for data movement between GPU and CPU
  - First draft: Adjusted the Chapel memory allocator to:
    - register a memory range on the CPU for use with the GPU when allocating memory
    - then, manage the registered memory range the same way it normally manages memory
  - What does this look like for the user?

TARGETING GPUS FROM CHAPEL

Data Movement From CPU to GPU (Example)

Chapel

```
pragma "codegen for GPU"
export proc add_nums(a: c_ptr(real(64))){
  a[0] = a[0]+5;
}

var funcPtr = createFunction();
var a = [<user input>];
launchKernel(funcPtr, c_ptrTo(a));
writeln(a);
```

Manual CUDA Driver API Calls

```
extern {
  #define FATBIN_FILE "chpl__gpu.fatbin"
  double createFunction(){
    fatbinBuffer = <read FATBIN_FILE into buffer>
    cuModuleLoadData(&cudaModule, fatbinBuffer);
    cuModuleGetFunction(&function, cudaModule,
                        "add_nums");}

    CUdeviceptr devBufferA;
    cuMemAlloc(&devBuffeA, sizeof(double));
    a = <user input>;
    cuMemcpyHtoD(devBufferA, &a, sizeof(double))
  void launchKernel(void *function, double* a){
    void *KernelParams[] = { &a };
    cuLaunchKernel(function, <grid and block size>,
                   … , KernelParams, …));}
    cuMemcpyDtoH(&a, devBufferA, sizeof(double));
    return a;
}
```

Read fat binary and create a CUDA function

Move data to GPU

Launch CUDA function

Move data to CPU

TARGETING GPUS FROM CHAPEL

Data Movement From CPU to GPU (Example)

Chapel

```
pragma "codegen for GPU"
export proc add_nums(a: c_ptr(real(64))){
  a[0] = a[0]+5;
}

var funcPtr = createFunction();
var a = [<user input>];
launchKernel(funcPtr, c_ptrTo(a));
writeln(a);
```

Manual CUDA Driver API Calls

```
extern {
  #define FATBIN_FILE "chpl__gpu.fatbin"
  double createFunction(){
    fatbinBuffer = <read FATBIN_FILE into buffer>
    cuModuleLoadData(&cudaModule, fatbinBuffer);
    cuModuleGetFunction(&function, cudaModule,
                        "add_nums");}

  void launchKernel(void *function, double* a){
    void *KernelParams[] = { &a };
    cuLaunchKernel(function, <grid and block size>,
                   … , KernelParams, …));}

}
```

Read fat binary and create a CUDA function

Launch CUDA function

TARGETING GPUS FROM CHAPEL

Launching Kernels From Chapel

- First draft:
  - Add a primitive to the Chapel compiler
  - Arrange for the primitive to emit `cuLaunchKernel` during code generation
  - This approach was implemented and allows kernels to be launched from Chapel

```
__primitive("gpu kernel launch", funcPtr,
            <grid and block size>,…,
            c_ptrTo(a), …);
```

- Background on Chapel primitives:
  - uses function call syntax, but doesn't adhere to normal function call behavior
  - instead, special operations that the compiler understands and knows how to directly translate
  - generally not intended for end-user use, so typically used internally or as a stopgap (as in this case)

TARGETING GPUS FROM CHAPEL

Launching Kernels From Chapel (Example)

Chapel

```
pragma "codegen for GPU"
export proc add_nums(a: c_ptr(real(64))){
  a[0] = a[0]+5;
}

var funcPtr = createFunction();
var a = [<user input>];
launchKernel(funcPtr, c_ptrTo(a));


writeln(a);
```

Manual CUDA Driver API Calls

```
extern {
  #define FATBIN_FILE "chpl__gpu.fatbin"
  double createFunction(){
    fatbinBuffer = <read FATBIN_FILE into buffer>
    cuModuleLoadData(&cudaModule, fatbinBuffer);
    cuModuleGetFunction(&function, cudaModule,
                        "add_nums");}

    void launchKernel(void *function, double* a){
    void *KernelParams[] = { &a };
    cuLaunchKernel(function, <grid and block size>
                   … , KernelParams, …));}

}
```

Read fat binary and create a CUDA function

Launch CUDA function

TARGETING GPUS FROM CHAPEL

Launching Kernels From Chapel (Example)

Chapel

```
pragma "codegen for GPU"
export proc add_nums(a: c_ptr(real(64))){
  a[0] = a[0]+5;
}

var funcPtr = createFunction();
var a = [<user input>];
__primitive("gpu kernel launch", funcPtr,
            <grid and block size>,…,
            c_ptrTo(a), …);
writeln(a);
```

Manual CUDA Driver API Calls

```
extern {
  #define FATBIN_FILE "chpl__gpu.fatbin"
  double createFunction(){
    fatbinBuffer = <read FATBIN_FILE into buffer>
    cuModuleLoadData(&cudaModule, fatbinBuffer);
    cuModuleGetFunction(&function, cudaModule,
                        "add_nums");}

}
```

Read fat binary and create a CUDA function

TARGETING GPUS FROM CHAPEL

Status

- Within Chapel 1.24:
  - Prototyped native GPU code generation

- Not yet released:
  - Removed explicit calls for data movement between GPU and CPU in an extern block
  - Launched kernels in Chapel rather than in an extern block

TARGETING GPUS FROM CHAPEL

Status: Example Code with Chapel 1.24

Chapel

```
pragma "codegen for GPU"
export proc add_nums(a: c_ptr(real(64))){
  a[0] = a[0]+5;
}

var output: real(64);
output = cudaFuncs();
```

Manual CUDA Driver API Calls

```
extern {
  #define FATBIN_FILE "chpl__gpu.fatbin"
  double cudaFuncs(){
    fatbinBuffer = <read FATBIN_FILE into buffer>
    cuModuleLoadData(&cudaModule, fatbinBuffer);
    cuModuleGetFunction(&function, cudaModule,
                        "add_nums");

    CUdeviceptr devBufferA;
    cuMemAlloc(&devBuffeA, sizeof(double));
    a = <user input>;
    cuMemcpyHtoD(devBufferA, &a, sizeof(double))

    void *KernelParams[] = { &devBufferA };
    cuLaunchKernel(function, <grid and block size>,
                   … , KernelParams, …));
    cuMemcpyDtoH(&a, devBufferA, sizeof(double));
    return a;
  }
}
```

Read fat binary and create a CUDA function

Move data to GPU

Launch CUDA function

Move data to CPU

TARGETING GPUS FROM CHAPEL

Status: Example Code with Development Branches

Chapel

```
pragma "codegen for GPU"
export proc add_nums(a: c_ptr(real(64))){
  a[0] = a[0]+5;
}

var funcPtr = createFunction();
var a = [<user input>];
__primitive("gpu kernel launch", funcPtr,
            <grid and block size>,…,
            c_ptrTo(a), …);
writeln(a);
```

Manual CUDA Driver API Calls

```
extern {
  #define FATBIN_FILE "chpl__gpu.fatbin"
  double createFunction(){
    fatbinBuffer = <read FATBIN_FILE into buffer>
    cuModuleLoadData(&cudaModule, fatbinBuffer);
    cuModuleGetFunction(&function, cudaModule,
                        "add_nums");}

}
```

Read fat binary and create a CUDA function

TARGETING GPUS FROM CHAPEL

Next Steps

Near-term (details in the following slides):

- Remove reading the fat binary from an extern block
- Indicate to the compiler when to generate GPU code without a pragma
- Handle ‘forall’ loops and have the compiler and/or user determine grid and block size
- Try writing more complicated kernels for the GPU and address problems that arise

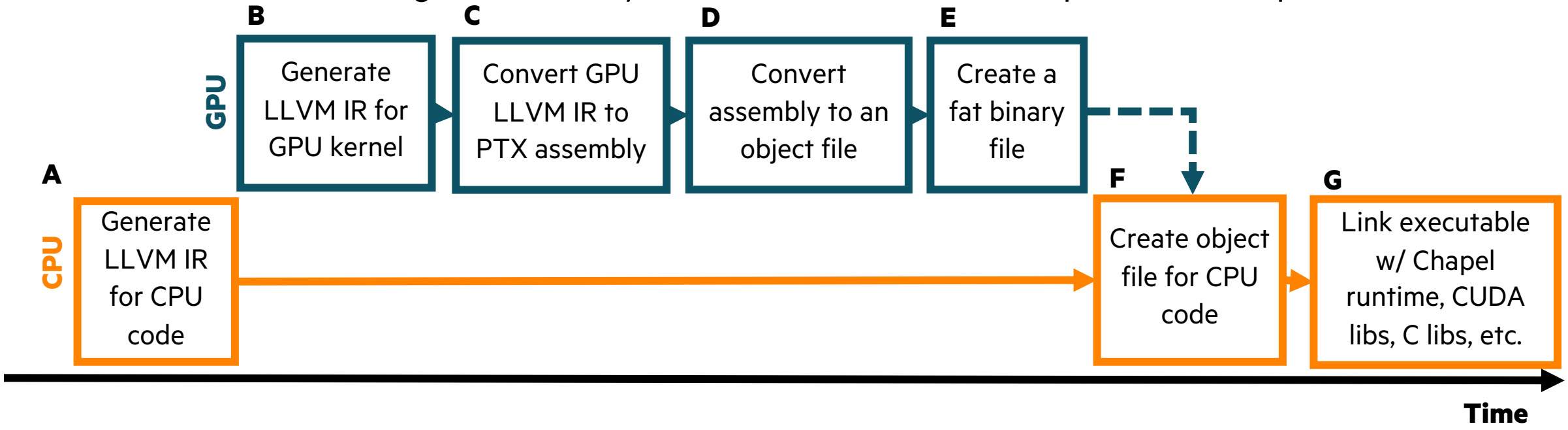
Longer-term:

- …

TARGETING GPUS FROM CHAPEL

Next Steps: Reading the Fat Binary

- We want to remove reading the fat binary from the extern block in the previous examples



- Following Clang CUDA, the fat binary file (E) should be embedded in the CPU object file (F)
- The compilation flow shown above supports this embedding

TARGETING GPUS FROM CHAPEL

Next Steps: Reading the Fat Binary (Example)

Chapel

```
pragma "codegen for GPU"
export proc add_nums(a: c_ptr(real(64))){
  a[0] = a[0]+5;
}

var a = [<user input>];
__primitive("gpu kernel launch", add_nums,
            <grid and block size>,…,
            c_ptrTo(a), …);
```

Manual CUDA Driver API Calls

```
extern {
  #define FATBIN_FILE "tmp/chpl__gpu.fatbin"
  void* createFunction(){
    fatbinBuffer = <read FATBIN_FILE into buffer>
    cuModuleLoadData(&cudaModule, fatbinBuffer);
    cuModuleGetFunction(&function, cudaModule,
                        "add_nums");}
}
```

Read fat binary and create a CUDA function

TARGETING GPUS FROM CHAPEL

Next Steps: Reading the Fat Binary (Example)

Chapel

```
pragma "codegen for GPU"
export proc add_nums(a: c_ptr(real(64))){
  a[0] = a[0]+5;
}

var a = [<user input>];
__primitive("gpu kernel launch", add_nums,
            <grid and block size>,…,
            c_ptrTo(a), …);
```

Manual CUDA Driver API Calls

```
extern {
  #define FATBIN_FILE "tmp/chpl__gpu.fatbin"
  void* createFunction(){
    fatbinBuffer = <read FATBIN_FILE into buffer>
    cuModuleLoadData(&cudaModule, fatbinBuffer);
    cuModuleGetFunction(&function, cudaModule,
                        "add_nums");}
}
```

Read fat binary and create a CUDA function

Chapel will:

- Arrange to include the fat binary in the executable
- Call `cuModuleLoadData` once during program initialization
- Call `cuModuleGetFunction` for each kernel

TARGETING GPUS FROM CHAPEL

Next Steps: When to Generate GPU Code

- Currently, we use a pragma to indicate which code should generate a GPU kernel:

```
pragma "codegen for GPU"
export proc add_nums(a: c_ptr(real(64))){
  a[0] = a[0]+5;
}
```

- In the future, use ‘on’ statements and ‘forall’ loops to indicate what should be run on the GPU

TARGETING GPUS FROM CHAPEL

Next Steps: When to Generate GPU Code (Example)

Chapel – Current

```
pragma "codegen for GPU"
export proc add_nums(a: c_ptr(real(64))){
  a[0] = a[0]+5;
}

var a = [<user input>];
_primitive("gpu kernel launch", add_nums,
           <grid and block size>,…,
           c_ptrTo(a), …);
writeln(a);
```

Chapel – Next Steps

```
var a = [<user input>];
on getGPUSublocale(){
  a = a + 5;
}
writeln(a);
```

- What to run on the GPU is specified with an ‘on’ statement instead of a pragma
- Primitive is no longer needed to launch a GPU kernel
- Data no longer has to be explicitly passed as a C pointer

TARGETING GPUS FROM CHAPEL

Next Steps: Forall Loops and Determining Grid and Block Size

- How do we handle ‘forall’ loops?

```
var a = [<user input>];
on getGPUSublocale(){
  forall x in a {
    x = x + 5;
  }
}
writeln(a);
```

- Launching a GPU kernel requires information such as grid and block size

- In the future, use information in the ‘forall’ loop to determine grid and block size

TARGETING GPUS FROM CHAPEL

Next Steps: Forall Loops and Determining Grid and Block Size (Example)

Chapel – Next Steps

```
var a = [<user input>];
on getGPUSublocale(){
  forall x in a
    x = x + 5;
}
writeln(a);
```

Chapel – Compiler Translation

```
var a = [<user input>];
on getGPUSublocale(){
  const (gridX, gridY) = Forall.getGridSize();
  const (blockX, blockY) = Forall.getBlockSize();
  __primitive("gpu kernel launch", forall_kernel,
              gridX, gridY, blockX, blockY,
              c_ptrTo(a));
}

extern {
  __global__ void forall_kernel(float *a) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    a[index] = a[index] + 5;
  }
}
```

TARGETING GPUS FROM CHAPEL

Next Steps

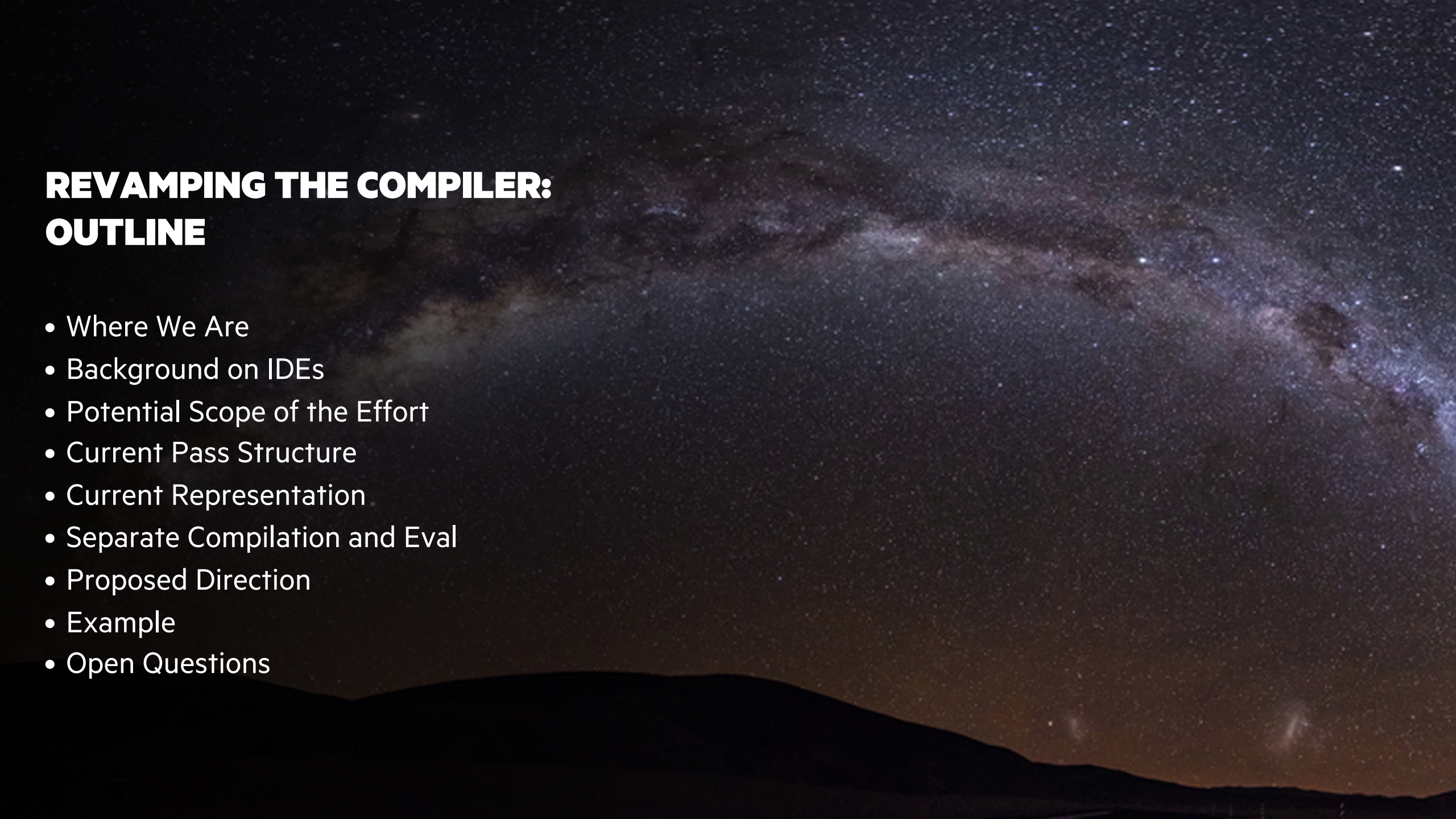
Near-term:

- Remove reading the fat binary from an extern block
- Indicate to the compiler when to generate GPU code without a pragma
- Handle ‘forall’ loops and have the compiler and/or user determine grid and block size
- Try writing more complicated kernels for the GPU and address problems that arise

Longer-term:

- Support for explicit memory management
- Target multiple GPUs per locale
- Cover a variety of GPU types and vendors
- Explore RDMA integration with GPU memory


REVAMPING THE COMPILER



REVAMPING THE COMPILER: OUTLINE

- Where We Are
- Background on IDEs
- Potential Scope of the Effort
- Current Pass Structure
- Current Representation
- Separate Compilation and Eval
- Proposed Direction
- Example
- Open Questions


WHERE WE ARE

PROBLEMS WITH THE CURRENT COMPILER

- **Speed**
  - The current compiler is generally slow, and extremely so for large programs (~7s to 15 minutes)
- **Errors**
  - For incorrect programs, the compiler frequently displays only some of the errors at a time
  - Compilation errors can be hard for users to understand and resolve
- **Structure and Program Representation**
  - The compiler is structured only for whole-program analysis, preventing separate/incremental compilation
  - Unclear how to integrate an interpreter, provide IDE support, or 'eval' Chapel snippets
  - Compilation passes are highly coupled and rely on pattern matching more than they ought to
  - The compiler normalizes and adds temporaries too early and in too many cases
  - AST isn't clear, embeds assumptions that change during compilation, and is difficult to optimize
- **Development**
  - The modularity of the compiler implementation needs improvement
  - There is a steep learning curve to become familiar with the compiler implementation

WHAT IS WORKING WELL

- The current compiler code base has enabled the design and evolution of Chapel to date
- The compiler includes key optimizations that support program performance
- The compiler is relatively fast to build and very portable
- Chapel's internal and library modules are extensive and largely independent of the compiler
- The runtime libraries are well-architected and not in need of major changes

BACKGROUND: INTEGRATED DEVELOPMENT ENVIRONMENTS

INTEGRATED DEVELOPMENT ENVIRONMENTS

- IDE support beyond syntax highlighting is frequently asked about

- Good IDE integration requires the ability of the compiler to
  - behave more like a server
  - respond quickly to limited queries

- The next slides illustrate two ways to integrate with IDEs:
  - Language Server Protocol
  - Jupyter Protocol

ABOUT LANGUAGE SERVER PROTOCOL

- Language Server Protocol was developed for Visual Studio Code but works with other IDEs

- A language server can handle these events:
  - create file, replace this text with some new text
  - semantic highlighting (e.g., formals are one color, types another)
  - code completion (e.g., typing ‘myBlockArray.’ can pop up a list of methods)
  - hover—mouse over a symbol to learn more about its type
  - signature help—what is the signature of the function called here?
  - other stuff:
    - go to declaration, definition, type definition, implementation
    - find references
    - find symbols matching query
    - highlight uses/reads/writes of symbol
    - format code, rename symbol

ABOUT JUPYTER PROTOCOL

- Jupyter Protocol supports Jupyter Notebooks

- A Jupyter Kernel can handle these messages:
  - execute
  - introspection
  - completion
  - history
  - code completeness
    - decide if input is “complete” (e.g., are we in the middle of a function declaration?)

- Jupyter supports many non-textual kinds of output (e.g., HTML, SVG)

POTENTIAL SCOPE OF THE COMPILER EFFORT

POTENTIAL SCOPE: COMPILER ARCHITECTURE

- Address these problems with the current architecture:
  - normalization is done before function resolution, requiring frequent workarounds
  - name resolution is implemented twice (once for functions, once for everything else)
  - complex dependencies between passes
  - AST classes are overloaded and could more simply represent Chapel concepts
  - whole-program architecture presents challenges for separate compilation, IDEs, and other tooling

Improving the architecture will enable:

- Separate compilation
- Autocompletion for IDEs
- REPL, 'eval', and compiling some Chapel code in order to load it into a running program
- Separately testing each compiler pass
- Easier development of debuggers, linters, and other tools

POTENTIAL SCOPE: SEPARATE COMPILATION

- Support separate compilation
  - via object files or an equivalent
  - and/or by redefining what “compile” and “link” means to support generics and other Chapel features

- Support incremental recompilation
  - caching portions of a program that have already been compiled to make similar compilation requests faster

Adding these features should:

- make typical uses of the Chapel compiler much faster
- allow users to better understand and work with compile & link steps during compilation
- better enable large projects to use Chapel

POTENTIAL SCOPE: DYNAMIC AND INTERACTIVE COMPILATION

- Support dynamically incorporating new code into a running Chapel program
  - including, potentially, into each node's executable for a multi-locale run
  - in order to support:
    - the ability to 'eval' Chapel code from a running Chapel program
    - a debugger's expression evaluation engine
- Include an interpreted/interactive Chapel mode (REPL)
- Enable autocompletion and other interactive IDE features
- Improve error reporting to show more errors at a time with better context

These features can:

- improve the learning experience for new Chapel programmers
  - with labeling features in IDEs and better debuggers
- enable better understanding of the performance of Chapel programs
- enable interactive data exploration


CURRENT PASS STRUCTURE

SUMMARY OF CURRENT COMPILER PASSES

pass	time for Hello World	time for Arkouda	approx lines of .cpp
parse+	0.5s	0.8s	10,000 lines
scopeResolve	0.4s	0.7s	4,500 lines
normalize+	0.9s	2.2s	9,000 lines
resolve	2.0s	165s	35,000 lines
post-resolve	0.3s	16s	16,000 lines
lowerIterators	0.1s	7.1s	6,000 lines
parallel	0.1s	17s	2,000 lines
optimization	0.5s	46s	7,000 lines
insertWideRefs+	0.1s	15s	6,000 lines
codegen	0.5s	48s	20,000 lines
makeBinary	1.1s	422s	-
TOTAL OF ABOVE	6.0s	724s	114,000 lines
TOTAL	6.4s	743s	170,000 lines in ‘compiler/*’

EXPLAINING THE EARLY PASSES

parse+:

- creates AST and lowers some simple patterns, including ‘begin on’ and ‘var a, b : int’

scopeResolve:

- for everything other than function calls, establishes what each identifier refers to
- computes class hierarchies

normalize+:

- adds call expression temporaries for nested calls like ‘g()’ in ‘f(g())’
- lowers many generic argument types into ‘where’ clauses
- creates default implementations of many functions (e.g., ‘=’ and ‘==’ for records)
- pulls task bodies out into task functions

resolve:

- establishes the types of everything
- establishes which routine is called for each function call
- lowers many constructs

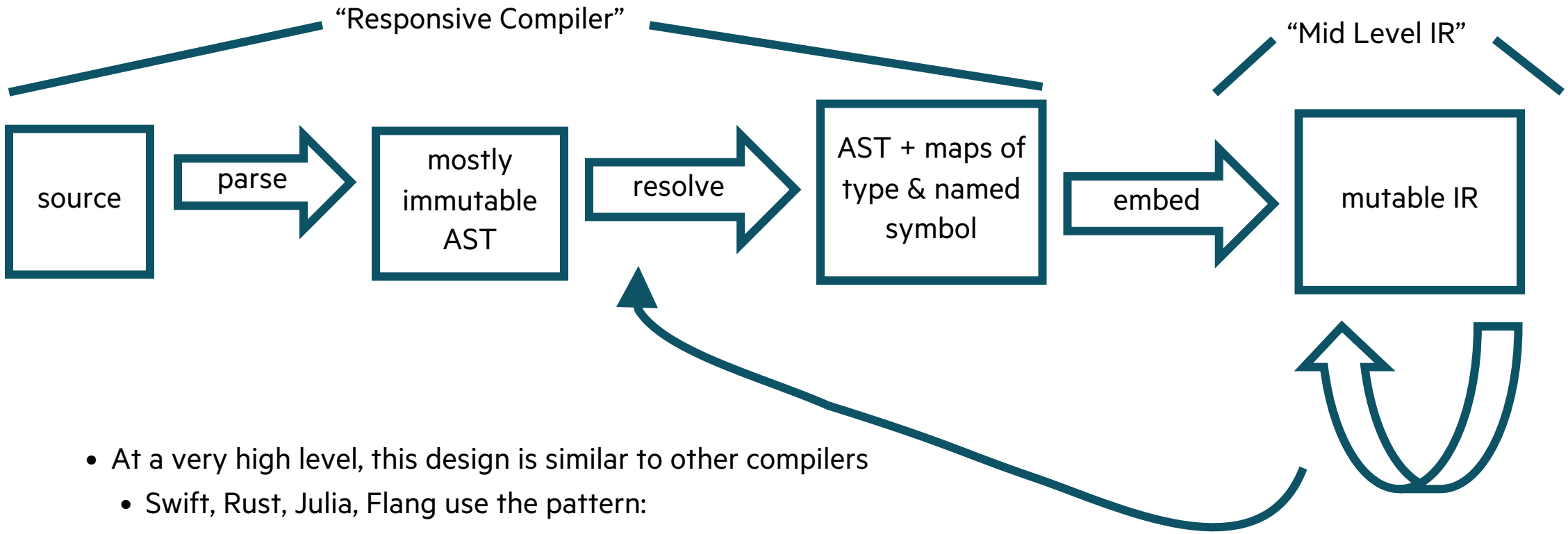
SEPARATE COMPILATION

SEPARATE COMPILATION

- We would like to support separate compilation
  - Challenging because there are generic functions and no equivalent to header files
  - Compiled libraries will store AST or source code for generic functions in case new instantiations are needed
- In a separate compilation scenario, both “compile” and “link” steps need a more flexible pass structure
- For “compile”, need to be able to compile a library without also compiling its dependencies
- For “link”, do not want to go through entire compilation process
- Rather, “link” should be limited to:
  - instantiating generics as necessary
  - connecting calls to concrete functions to their concrete implementations
- Neither of these are possible with the current rigid whole-program pass structure
- Compiler architecture improvements can make both problems easier to solve

PROPOSED DIRECTION

IMPROVED COMPILER ARCHITECTURE



- At a very high level, this design is similar to other compilers
  - Swift, Rust, Julia, Flang use the pattern:
    - parse ⟶ AST ⟶ mid-level IR (and progressively lower)
- Earliest part of compilation is most relevant for IDE integration
  - so is designed to be very incremental

CHANGES TO GET TO IMPROVED ARCHITECTURE

- Create a new AST more faithful to source code for early passes
- Develop a new pass architecture with less rigid ordering
  - make passes typically run per-function rather than whole-program and otherwise be idempotent
- Convert the new AST into the old AST to enable incremental development
- Gradually port later passes over to a new IR more suited for optimization

RESPONSIVE COMPILERS

- Matsakis' talk, *Responsive Compilers*[1], presents a vision for good IDE support:
  - highly incremental and demand-driven—just process enough to answer a query
    - e.g., how to complete newBlock<tab>
    - fast response times are key for a satisfying experience
  - even with error sentinel AST nodes, error-handling can be tricky
    - need to show errors again when an incremental result is reused

- The strategy relies on:
  - structuring compilation in terms of many fine-grained queries
    - e.g., what is the type of this variable?
  - framework uses these queries to manage dependencies among results
  - each query saves its result and is re-run when necessary
  - query results are represented separately from the input—which tends to mean a lot of maps
  - AST elements are given IDs to support these maps

1 – *Responsive Compilers*, Nicholas Matsakis, PLISS 2019

RESPONSIVE COMPILER STRUCTURES

- Assign scoped IDs to all Symbols and Expressions
  - use '#<number>' to handle the possibility of multiple symbols defined with the same name
  - use '.' for nested scopes
  - use '@' for the i'th expression inside of a symbol in a postorder traversal
  - E.g.

```
module M {          // M has ID M#0
  proc func() {     // func has ID M#0.func#0
    var x: int;     // symbol 'x' has ID M#0.func#0.x#0
    f(x);           // use of 'x' here has ID M#0.func#0@4
  }
}
```

- Each AST node includes an ID for itself and can contain other AST within it

- AST nodes use IDs to refer to other nodes outside of it (e.g., parent, prev, next)
  - prevents accessing outside AST without going through incremental framework

RESPONSIVE COMPILER QUERIES

- parse(filePath) ⟶ AST for file (which also establishes IDs)
- locate(AST) ⟶ (line number, column number)
- ast(ID) ⟶ AST for that identifier
- getDefinedIn(Expr, name) ⟶ Symbols defined in ‘Expr’ named ‘name’
- getVisible(Expr, name) ⟶ Symbols visible from ‘Expr’ named ‘name’
- types(Expr) ⟶ map from Symbols to Types for Symbols defined in ‘Expr’
- resolve(Expr) ⟶ map from Identifiers to Symbols they refer to

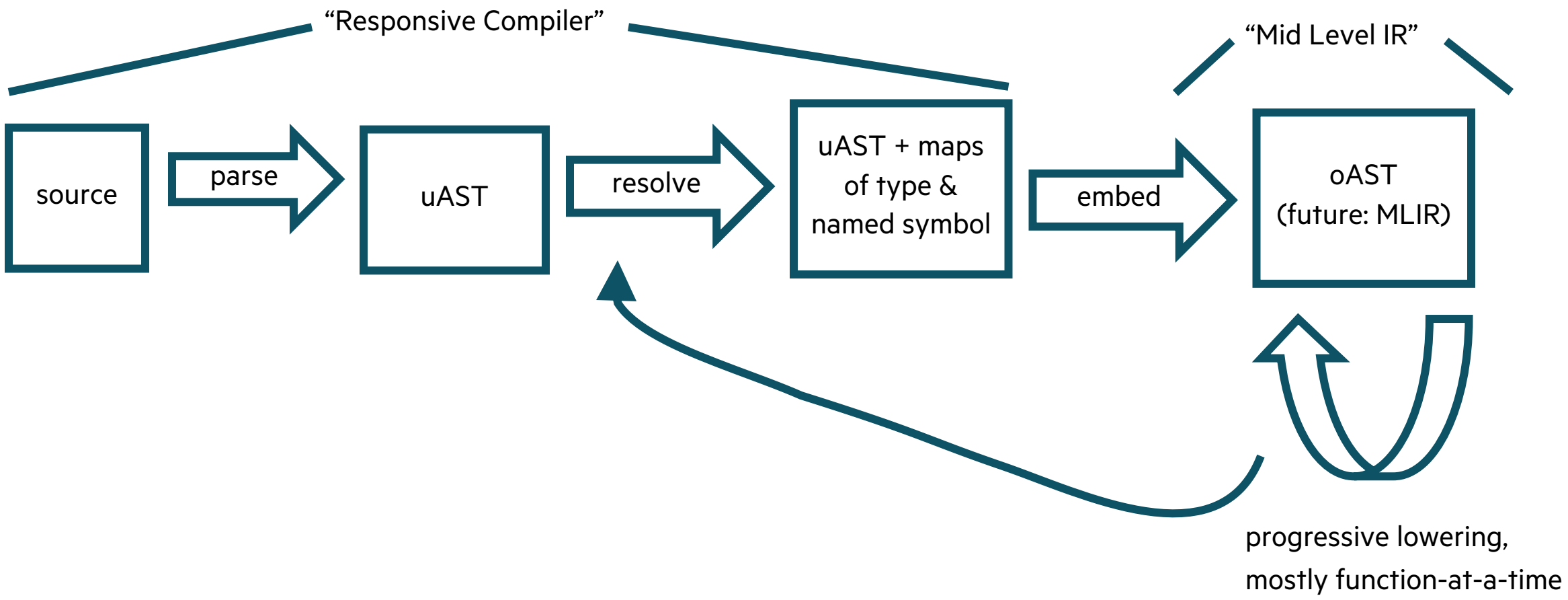
NEW AST AND OLD AST

- Let's call the current AST, Old AST (oAST)
- We'll introduce a new untyped AST, called uAST which:
  - is relatively faithful to grammar (simply represents things like 'forall'; represents comments and spacing)
  - refers to things by name rather than as a particular symbol
  - has no temporaries and no flow-sensitive analysis
  - is mostly immutable after parsing—but can be added to (say, when parsing a new file later)
  - includes error sentinels (e.g., a function body might be Error if we could not figure out more)
  - uses private fields and more unified interfaces
  - is designed with compiler performance in mind
- New uAST will eventually be converted into a mid-level IR
- Mid-level IR will:
  - refer back to the uAST
  - support progressive lowering transformations (in-place)
  - start out as oAST
  - eventually be represented as MLIR/LLVM

CHANGES TO PASS STRUCTURE

- Just after parsing, compiler will process uAST to resolve names to symbols
  - in a highly incremental manner (to support IDEs as well as incremental compilation)
- This phase will include:
  - choosing ‘ref’ or ‘const ref’ variants for return-intent overloading
  - instantiating unconstrained generics including vararg functions
  - establishing concrete intents
  - establishing types of everything
  - establishing when variables are initialized and when values are copied (for split initialization and copy elision)
- Focus here is on establishing properties and not on lowering
- The mid-level IR will be progressively lowered, preferring a function-at-a-time approach

FUTURE COMPILER ARCHITECTURE



GETTING THERE FROM HERE

- The next 3 slides show different tracks of work to achieve these goals
- Steps should occur in order within each track
- Each track can proceed independently, as needed

CHANGING REPRESENTATION IN PHASES

Phase 1:

- Define new uAST classes
- Adjust the parser to create uAST classes
- Add code to translate the uAST classes to oAST

Phase 2:

- Develop framework for responsive compilation (re-using saved query results when possible)
- Gradually, add new code to:
  - establish what names refer to (variables, function calls)
  - establish types of symbols and values of params
  - instantiate unconstrained generics
- Add code to translate uAST + maps for types, etc. into oAST

Phase 3:

- Define MLIR dialect to embed resolved uAST
- Gradually port passes to MLIR, translating the final MLIR form into oAST for remaining passes

CHANGING PASS STRUCTURE IN PHASES

Phase A:

- Develop pass manager framework inspired by MLIR/LLVM concepts
  - Concept of “isolated from above” is important—pass can’t modify things outside of the unit it accepts
  - Function-at-a-time pass
  - Modifies-everything pass
- In the future, it might be the MLIR pass manager - until then, need a look-alike to apply to oAST
- It should be easy to figure out which passes have been applied to a particular FnSymbol

Phase B:

- Update existing passes to use the pass manager - focusing especially making function-at-a-time passes

ENABLING SEPARATE COMPILATION

Phase Alpha:

- make every pass function-at-a-time or idempotent
  - Done separately from adding a PassManager—e.g., each pass loops over FnSymbols
  - Adjust oAST FnSymbol to track which passes have been applied to it
- success criterion: the top-level compiler code can run each pass twice without messing anything up

Phase Beta:

- create “library” file format for separate compilation
  - table of contents listing symbols available
  - source code / uAST (for generics)
  - LLVM IR / C / .o data for compiled concrete functions
- adjust the compiler to emit “library” files

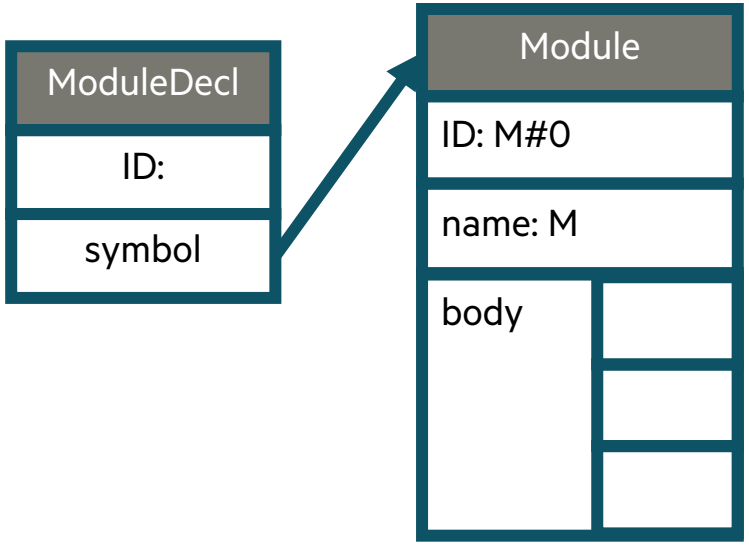
Phase Gamma:

- adjust the compiler to accept a “library” file as input instead of / in addition to .chpl files
- in this mode the compiler will take the role of the “linker”


EXAMPLES

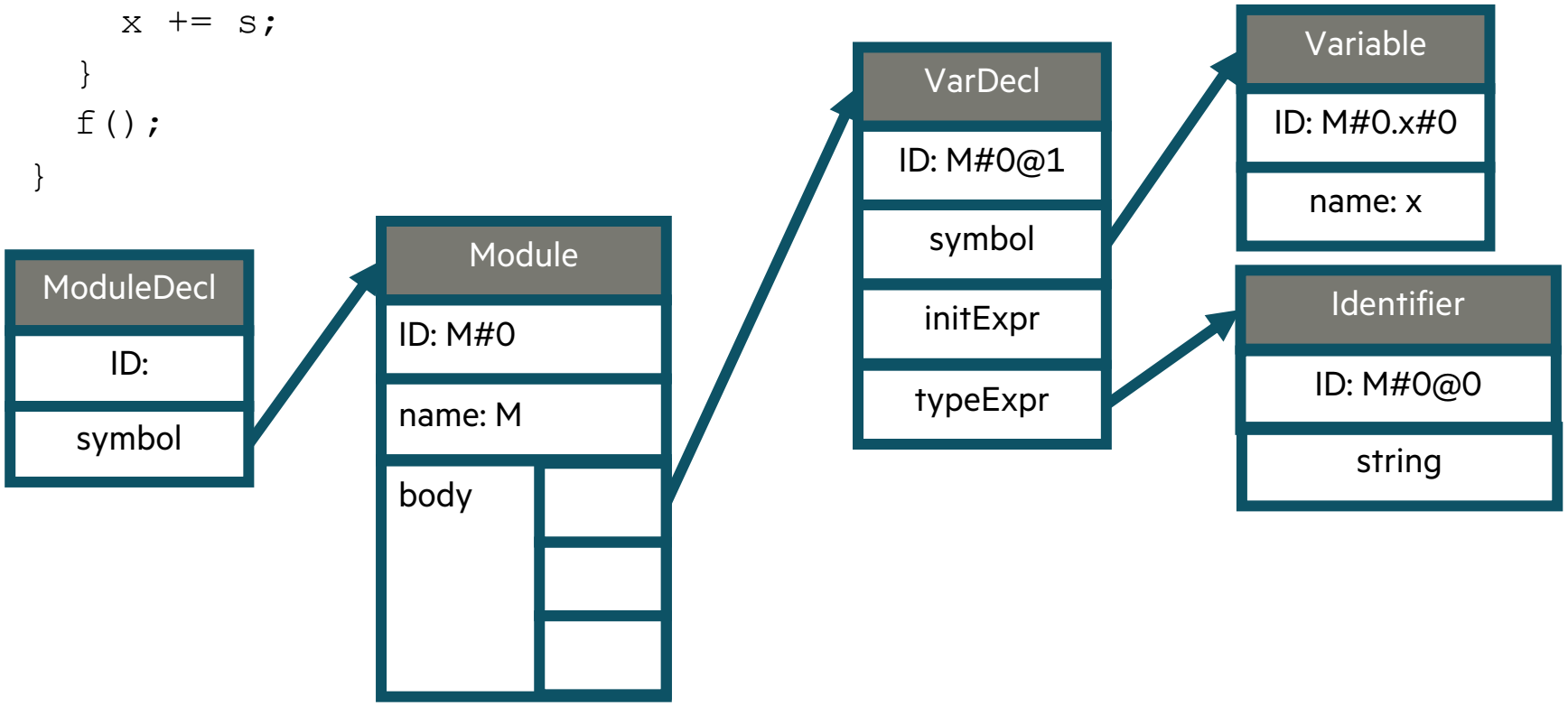
EXAMPLES: AST

```
module M {
  var x: string;
  proc f() {
    var s = "hello";
    x += s;
  }
  f();
}
```



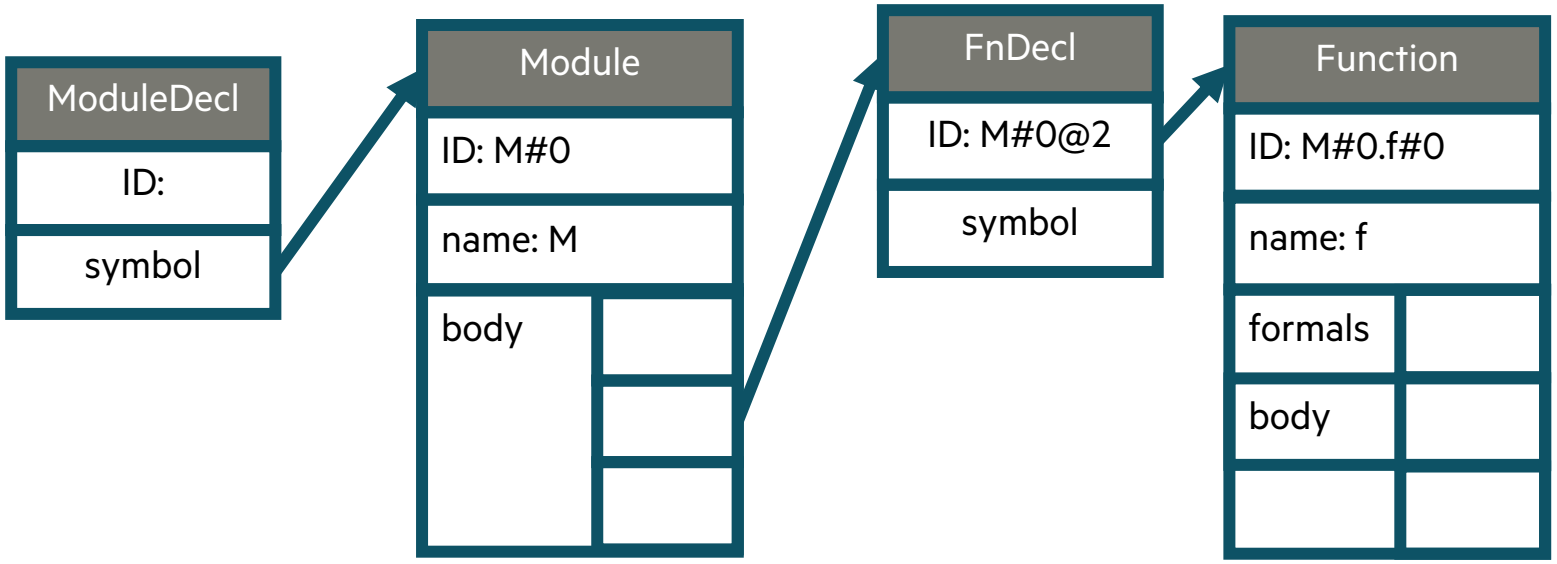
EXAMPLES: AST

```
module M {
  var x: string;
  proc f() {
    var s = "hello";
    x += s;
  }
  f();
}
```



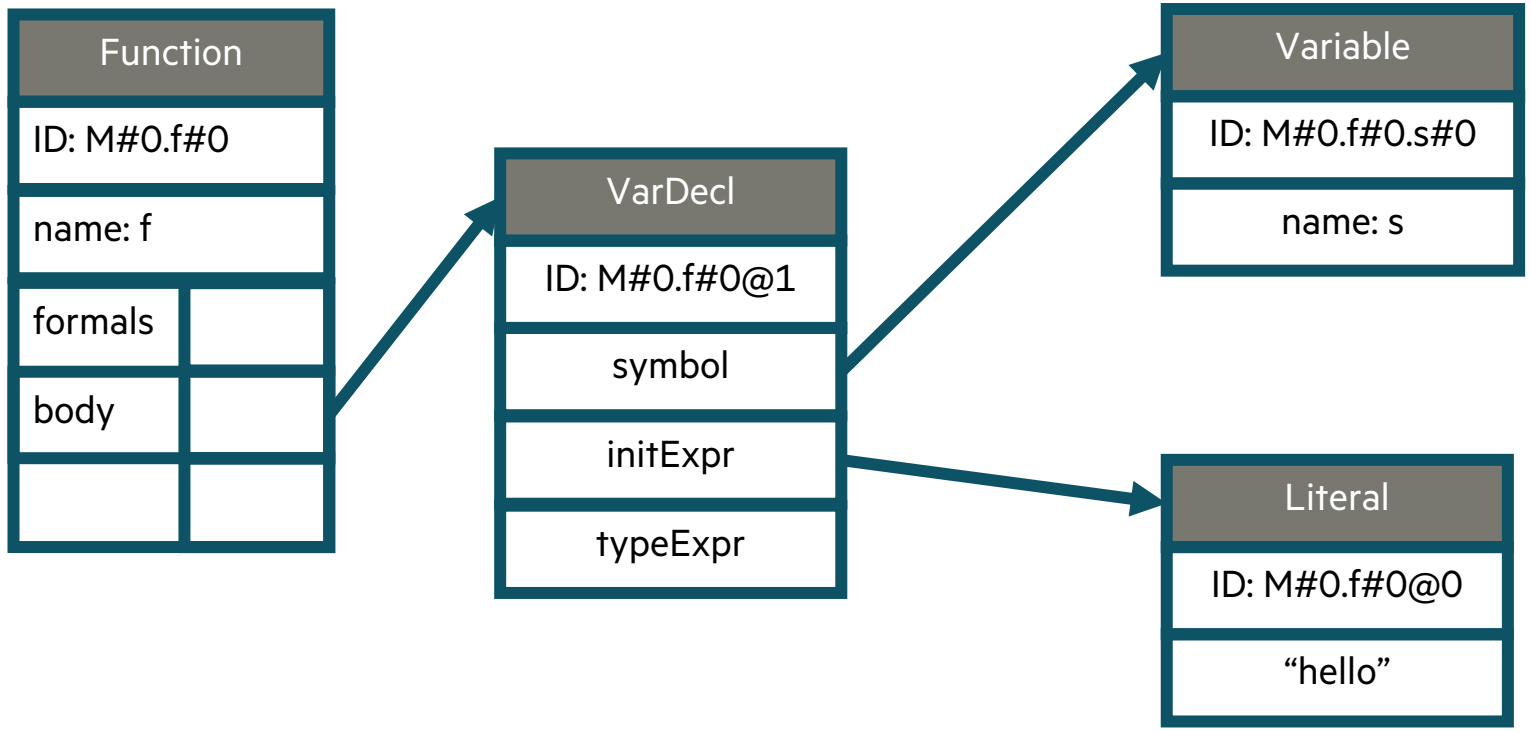
EXAMPLES: AST

```
module M {
  var x: string;
  proc f() {
    var s = "hello";
    x += s;
  }
  f();
}
```



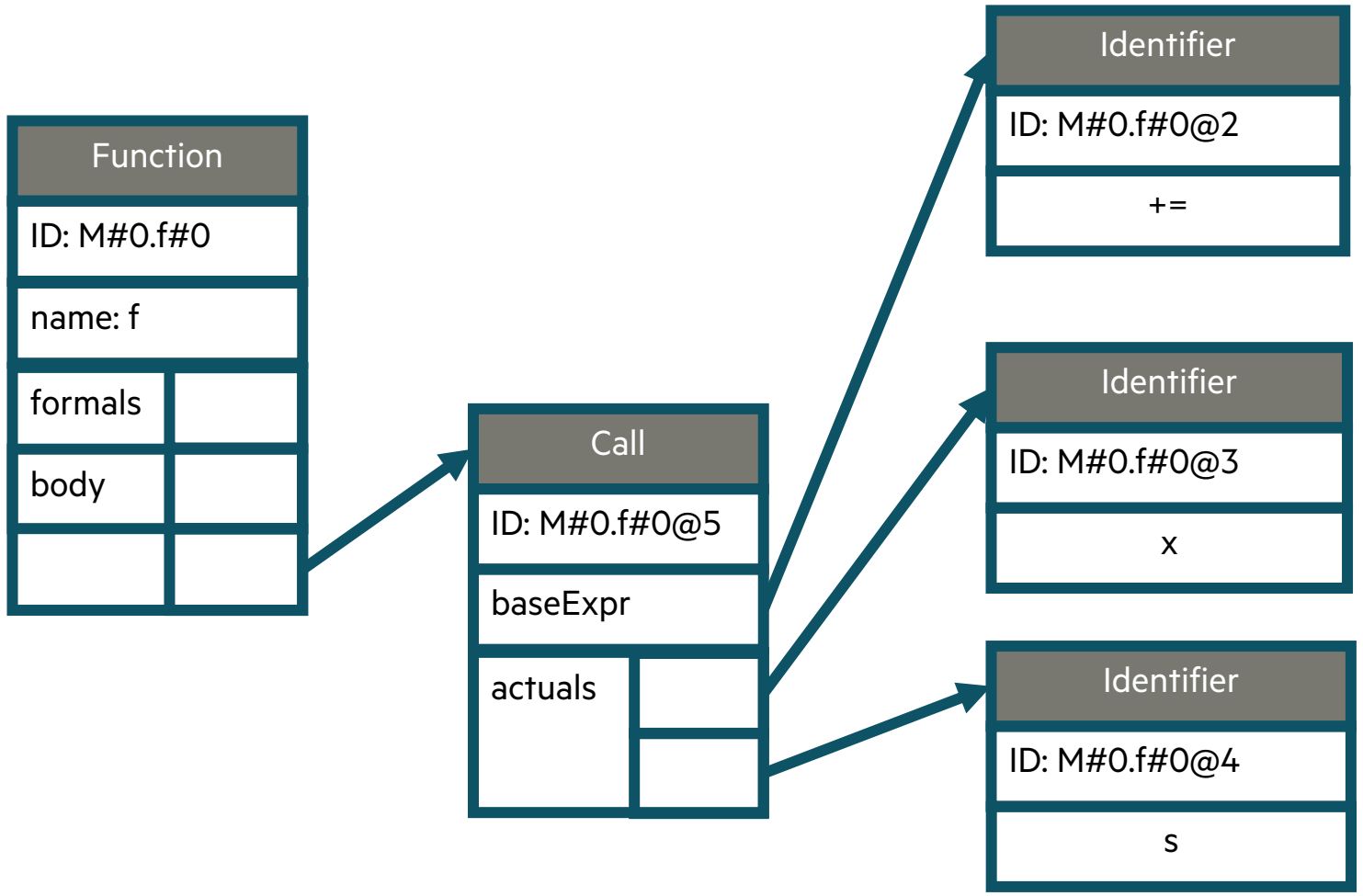
EXAMPLES: AST

```
module M {
  var x: string;
  proc f() {
    var s = "hello";
    x += s;
  }
  f();
}
```



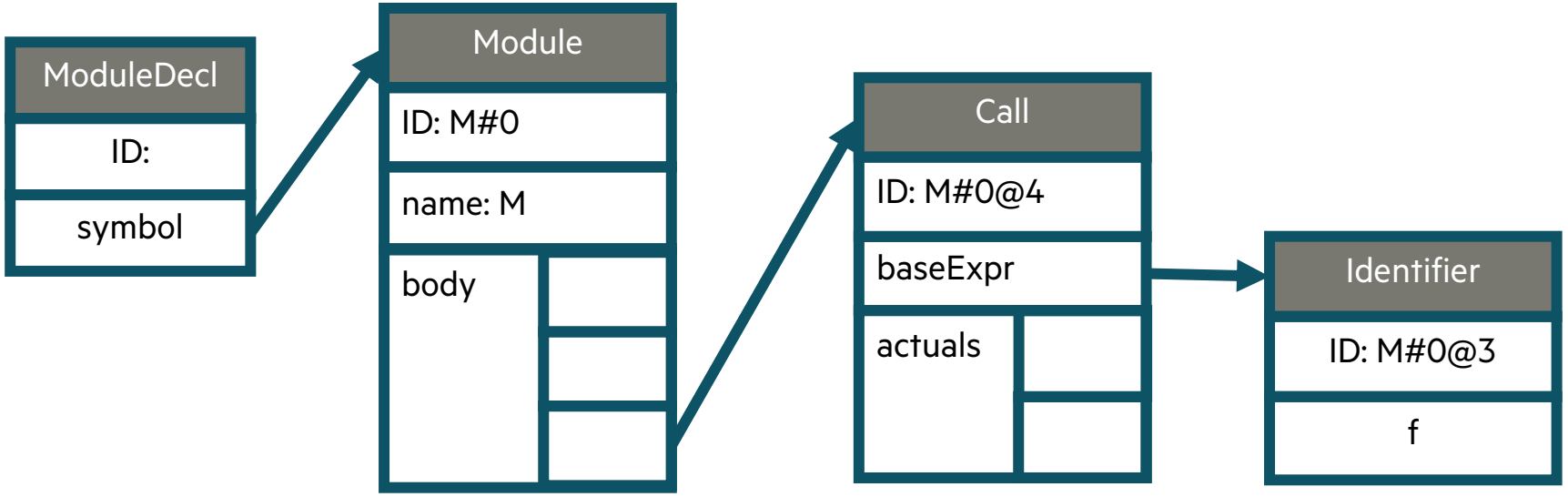
EXAMPLES: AST

```
module M {
  var x: string;
  proc f() {
    var s = "hello";
    x += s;
  }
  f();
}
```



EXAMPLES: AST

```
module M {
  var x: string;
  proc f() {
    var s = "hello";
    x += s;
  }
  f();
}
```



EXAMPLES: ESTABLISHING A TYPE

```
module M {
  var x: string;
  proc f() {
    var s = "hello";
    x += s;
  }
  f();
}
```

- How to establish the type of ‘x’?

```
type(x)
```

- This might be implemented with ‘resolve(M)’ which traverses M’s body
  - establishing the type of ‘x’ by figuring out what ‘string’ refers to
  - that in turn would use

```
getVisible(M.body, "string")
```

  - to find visible things called ‘string’
  - and then it would choose the best candidate ‘string’ or issue an ambiguity error

EXAMPLES: FORALL OPTIMIZATION

- Consider a simple forall statement

```
forall elt in A {
  elt = 1;
}
```

- There are several optimizations relevant for forall statements
  - Automatic Aggregation
  - Automatic Local Access
  - Unordered Optimization
  - Fast Follower Optimization
- These are currently spread across several passes

EXAMPLES: FORALL OPTIMIZATION CURRENTLY

- just before normalize
  - Opts: add a block containing the optimized form that can be selected by later passes or at runtime
- normalize adds call temporaries
- during resolution
  - add shadow variables to forall
  - Opts: determine if optimization is possible based upon types
- callDestructors
  - check lifetimes and check for nil dereferences
  - Opts: check that scopes of arrays are compatible with optimization
- lowerIterators
  - lower ‘forall’ statements into other constructs
  - Opts: remove aggregation from recursive iterators
- loopInvariantCodeMotion
  - Opts: transform assignments into unordered/aggregated operations

Opts in **5** passes!

EXAMPLES: FORALL OPTIMIZATION IN THE FUTURE

- Entire optimization happens on the mid-level IR
- Flow might look like this:
  - Opts: add a block containing the optimized form that can be selected by later passes or at runtime
    - but do so with the benefit of type information
    - identify recursive iterators here as well since calls are resolved
  - lowering 'forall' to include shadow variables
  - lowering for copy/move/deinit
  - check lifetimes and nil dereferences
    - Opts: check that scopes of arrays are compatible with optimization
  - lowering 'forall' statement into other constructs
  - (Opts: possible to do further cleanup, checking, or transformation here if it helps)

Opts in **2** passes!

- Note: If each optimization adds its own optimized version, code size can grow exponentially
  - Fix: adjust optimizations to share optimized and fallback versions when possible

STATUS AND NEXT STEPS

REVAMPING THE COMPILER: STATUS AND NEXT STEPS

Status:

- Have received positive feedback on this approach from the core team, but are interested in additional input
- Working on hiring to help staff this effort and/or backfill for current developers

Next Steps:

- Get started!


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