Ongoing Efforts

Chapel versions 1.21 / 1.22

April 9 / 16, 2020

chapel_info@cray.com

chapel-lang.org

@ChapelLanguage

CRAY

a Hewlett Packard Enterprise company

CRAY CHAPEL

Outline

- [Constrained Generics]()
- ['ofi' Comm Layer]()
- [Shasta Chapel Module]()

Constrained Generics

Background: Generics in Chapel

- Today, the Chapel compiler follows the C++ strategy for generics
  - generic functions are instantiated and then type-checked
- However, this approach presents several problems:
  - generic code might not compile for all calls, with potentially confusing errors
  - long compile times
  - confusing point-of-instantiation rule
- Currently requires complex where-clauses for simple argument constraints
  - e.g., "the argument should be iterable and yield strings"
  - "the argument should have a 'this' method"

Goals of Constrained Generics

- Constrained generics enable more expressive programs
  - Allow users to indicate requirements of function interfaces more cleanly

```
proc foo(arg: ?T) where T implements Iterable(int) { ... }
```

- Can help improve point-of-instantiation
- Might help with compile-time issues
  - indicate requirements in generic function prototypes
  - type-check generic functions before instantiation
  - instantiate generic functions later in compilation

Unconstrained Generic

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

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

// instantiates with 't' replaced by 'int'

// functions from point-of-instantiation are in scope

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

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

```
double(1);
```

Unconstrained Generics and Visibility

```
module Lib {
```

point-of-instantiation rule

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

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

CHIP 2 Constrained Generic

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

// can resolve 'double' without instantiating

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

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

// can use internal table to easily instantiate later

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

Constrained Generics and Visibility

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

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

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

High Level Design Questions

- Some questions are less contentious, or easier to change with feedback
  - What is the terminology, syntax, and style to use?
  - Do we support both methods and function calls?
  - Can the compiler infer implementation of an interface?
- Others have less certainty, but aren't critical decisions
  - Can one assert an interface is met separately from an 'implements' block?
  - How can functions with the same name be handled?
- A major critical decision remains
  - How will the language handle generic functions without constraints?

Functions Without Constraints

Unconstrained Generics

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

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

- Potential strategies for resolving constrained and unconstrained generics:
  1. Resolve them both late — after instantiation
  2. Resolve them both early — before instantiation
  3. Use hybrid strategies
  4. Disallow unconstrained generics
     - Not really under consideration as it would prevent script-like code

Resolving Them Both Late

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

Pros:

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

Cons:

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

Resolve Both Early

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

Resolve Both Early: An Example

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

- Strategy: generate anonymous interface for 'arg'

```
interface double_arg_0 {
  proc self.show(): Writable;
  proc add(a: self, b: self): self;
}
```

- Could it furthermore type-check while building up this list of constraints?

```
proc int.show(): string {
  return "int " + this:string;
}
proc add(a: int, b: int): int {
  return a + b;
}
// Could compiler infer that the constraint is met?
double(1);
```

Resolve Both Early: Challenges

- Supporting common patterns in generic code will require additional effort
  - compile-time conditionals, e.g. 'if t == string then ...'
    - would require compile-time evaluation while resolving the interface
    - might lead to groups of interfaces, or conditionals *in* interfaces
  - inferred variable and return types

```
// potentially three anonymous interfaces
var x = arg.foo(); var y = x.bar(); return y.baz();
```

  - generic varargs, param loops

Resolve Both Early: Graph of Constraints

- One possibility: construct a graph combining constraints and type inference
  - Significantly more complicated than other strategies, likely prone to bugs
  - Unknown impact on compile-time to create and evaluate such graphs
- Error messages for such graphs may not be an improvement over today
  - Error messages for complex graphs might need to show a 'callstack'
  - Could the spec and compiler clearly communicate...

    ... how a graph/interface was built?

    ... how a graph/interface was evaluated?

Resolve Both Early: Summary

Cray a Hewlett Packard Enterprise company

Pros:

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

Cons:

- Supporting type inference will require significant implementation effort
- Unknown impact on compilation speed for unconstrained generics
- Changes visibility rules for unconstrained generics in existing code
- Error messages for anonymous interfaces may be no better than today

Hybrid Strategies

Resolve constrained generics before instantiation, unconstrained generics after

Pros:

- Limits changes for existing generic code
- Compilation speed improvements with early type checking
  - Could require use of constraints in standard/internal libraries

Cons:

- Mixing constrained and unconstrained generics will require special attention
- Does not improve compilation speed for unconstrained generics
- Different visibility rules between the two might be confusing

Unconstrained Generics: Summary

- "Both Late": least amount of effort for additional type checking

- "Both Early" and "Hybrid" offer potential compile-time improvements
  - Unknown or non-existent compile-time improvements for unconstrained
- "Both Early" unifies visibility, "Hybrid" requires more user attention

- "Both Early" has impact on existing programs with unconstrained
  - changes visibility rules
  - might not help compile-times or error messages
  - "Hybrid" has minimal impact by comparison

Unconstrained Generics: Next Steps

- Intend to pursue the hybrid strategy
  - More confident in implementation (timescale, maintenance, stability)
  - Allows for compilation time improvements
  - Minimal impact on existing programs
- Address open questions around hybrid strategy
  - Difference in visibility rules between constrained and unconstrained
  - How constrained and unconstrained would interact between/inside functions
- Path to "both early" option still remains open going forward

Hybrid Strategy: Open Questions

- How to address the difference in visibility rules?
  - Better compiler error messages?
  - Alter unconstrained rules for common cases, if the impact is acceptable?
- Explore how to handle interaction between constrained and unconstrained
  - Require a function to have all arguments constrained or all unconstrained?
  - Calling unconstrained functions inside constrained functions?
    - Could disallow such calls as an easy-to-remember rule
  - Or fall back on late resolution when strategies mix?
- Should users have a way to require usage of constrained generics in a module?

Hybrid Strategy: Open Questions (cont.)

- Interaction with class management?
  - Could choose to treat arguments as 'borrowed' in absence of annotation

```
// currently has generic management
proc foo(arg : MyClass)
// could treat as 'borrowed' to allow both classes and records
proc bar(arg : Writable)
```

Other Design Questions

High Level Design Questions

- Some questions are less contentious, or easier to change with feedback
  - What is the terminology, syntax, and style to use?
  - Do we support both methods and function calls?
  - Can the compiler infer implementation of an interface?
- Others have less certainty, but aren't critical decisions
  - Can one assert an interface is met separately from an 'implements' block?
  - How can functions with the same name be handled?
- A major critical decision remains
  - How will the language handle generic functions without constraints?

Asserting an interface is met

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

```
int implements Doubleable;
```

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

```
proc myAdd(a: int, b: int): int { ... }
int implements Doubleable {
  proc add(a: int, b: int): int { return myAdd(a, b); }
}
```

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

CHIP 2 Constrained Generic: Separate Clause

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

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

Handling Duplicate Function Names

- Sometimes generic code might use two interfaces with the same function names

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

Handling Duplicate Function Names

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

```
where t implements Addable with accum as accumA, Sumable
```

  - Could allow a cast syntax, e.g., '(arg: Addable).accum()'
  - Or could require a wrapper method:

```
proc doubleableShow(arg: Doubleable) { arg.show(); }
doubleableShow(arg);
```

- Universal method syntax helps to enable this in Rust
  - 'arg.show()' can be written as 'show(arg)' - so e.g., 'Doubleable.show(arg)'

Terminology, Syntax, and Style Choices

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

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

Method and Function Signatures in Interfaces

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

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

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

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

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

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

  - Use universal methods like Rust does:

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

‘ofi’ Comm Layer

ofi Comm : Background, This Effort

CRAY
a Hewlett Packard Enterprise company

Background: Ongoing development of a libfabric-based comm layer

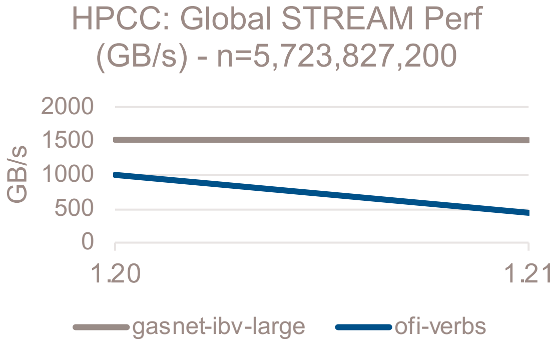
- In 1.20, passed most testing on all our targets with appropriate provider(s)
- Performance on par with gasnet but with slow cases, hangs, and variability

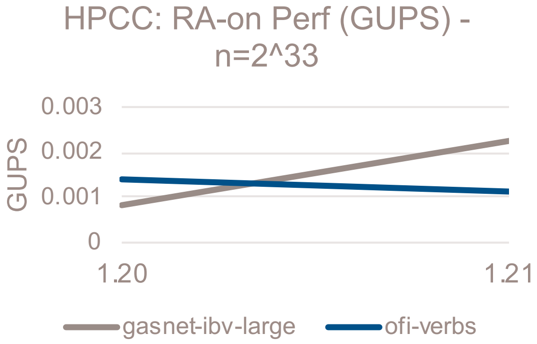
This Effort: Ongoing improvements

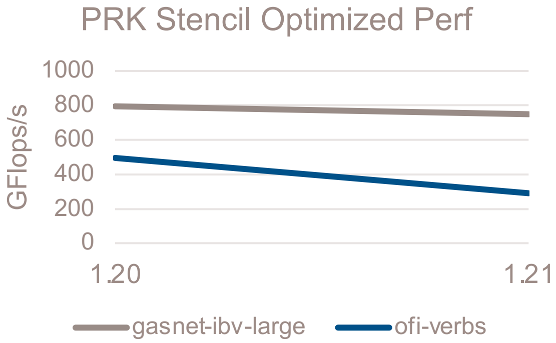
- Began nightly functional and performance testing
- Improved most slow cases
- Added unordered GETs, PUTs, AMOs
- Diagnosed all hangs; fixed all but one and have a plan for that one
  - One transaction progress fix caused widespread performance regression

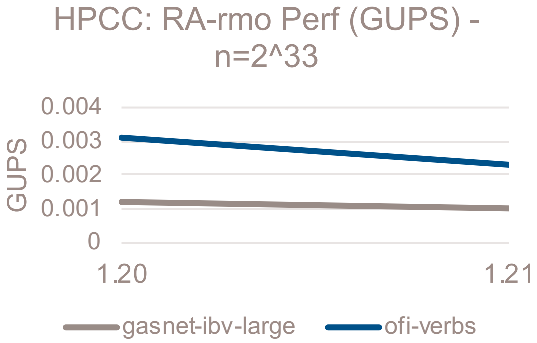
ofi Comm: Benchmark Performance

- 16-node Cray CS, InfiniBand network



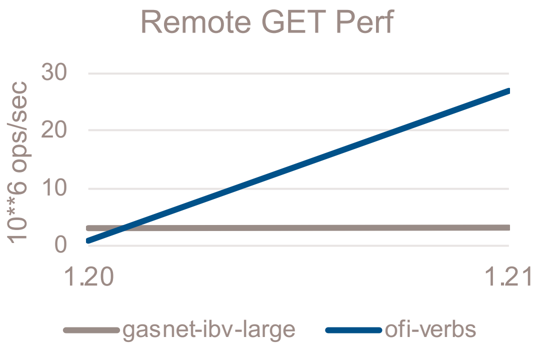


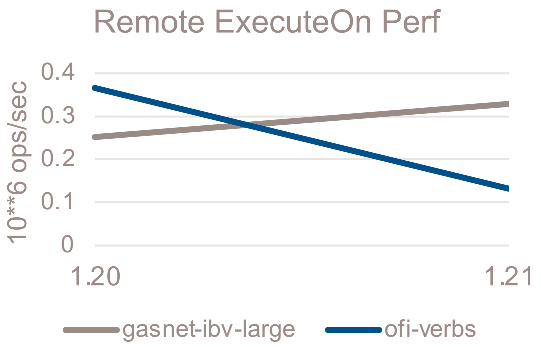


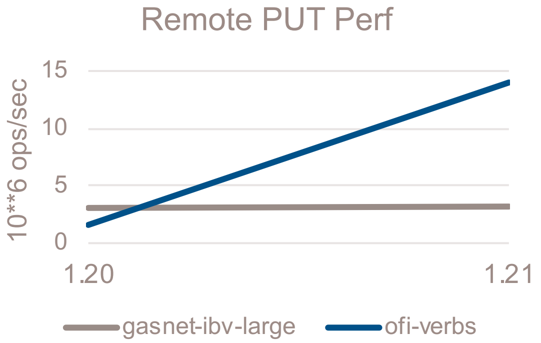


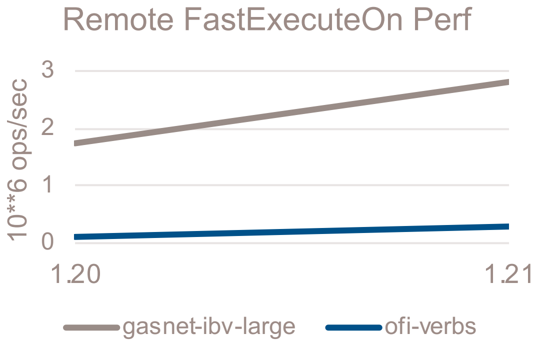
ofi Comm: Microbenchmark Performance

- 16-node Cray CS, InfiniBand network









ofi Comm: Next Steps

- Continue improving provider and platform portability

- Continue improving performance
  - Recover performance loss due to transaction-progress fix
  - Adjust memory consistency model (MCM) handling in compiler, modules

Shasta Chapel Module

Shasta Chapel Module

Background:

- Early-access Shasta customers had a pre-built Chapel from the start

This Effort:

- Normalized Shasta builds: removed special cases, cleaned up *ad hoc* logic

Status:

- Same 2 configs as in 1.20: comm=none and ofi, everything else default

Next Steps:

- Expand configurations to full supported set

FORWARD LOOKING STATEMENTS

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

THANK YOU

QUESTIONS?

chapel_info@cray.com
@ChapelLanguage
chapel-lang.org

CRAY
a Hewlett Packard Enterprise company

cray.com
@cray_inc
linkedin.com/company/cray-inc-/