Towards Interfaces for Chapel

Chris Wailes
and
Jeremy Siek

INDIANA UNIVERSITY
BLOOMINGTON

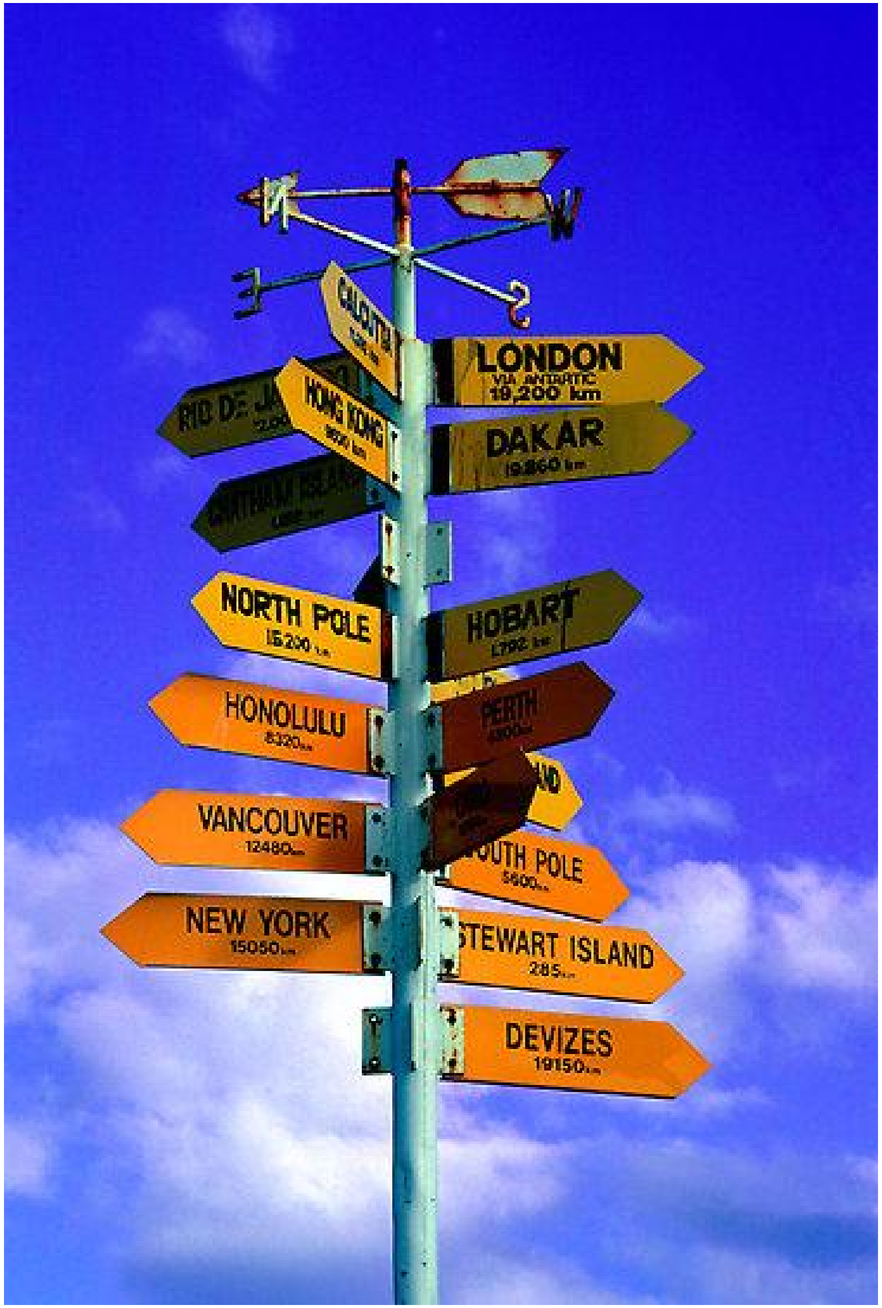
Why interfaces

How interfaces

Semantic changes

Compiler changes

Improving the compiler



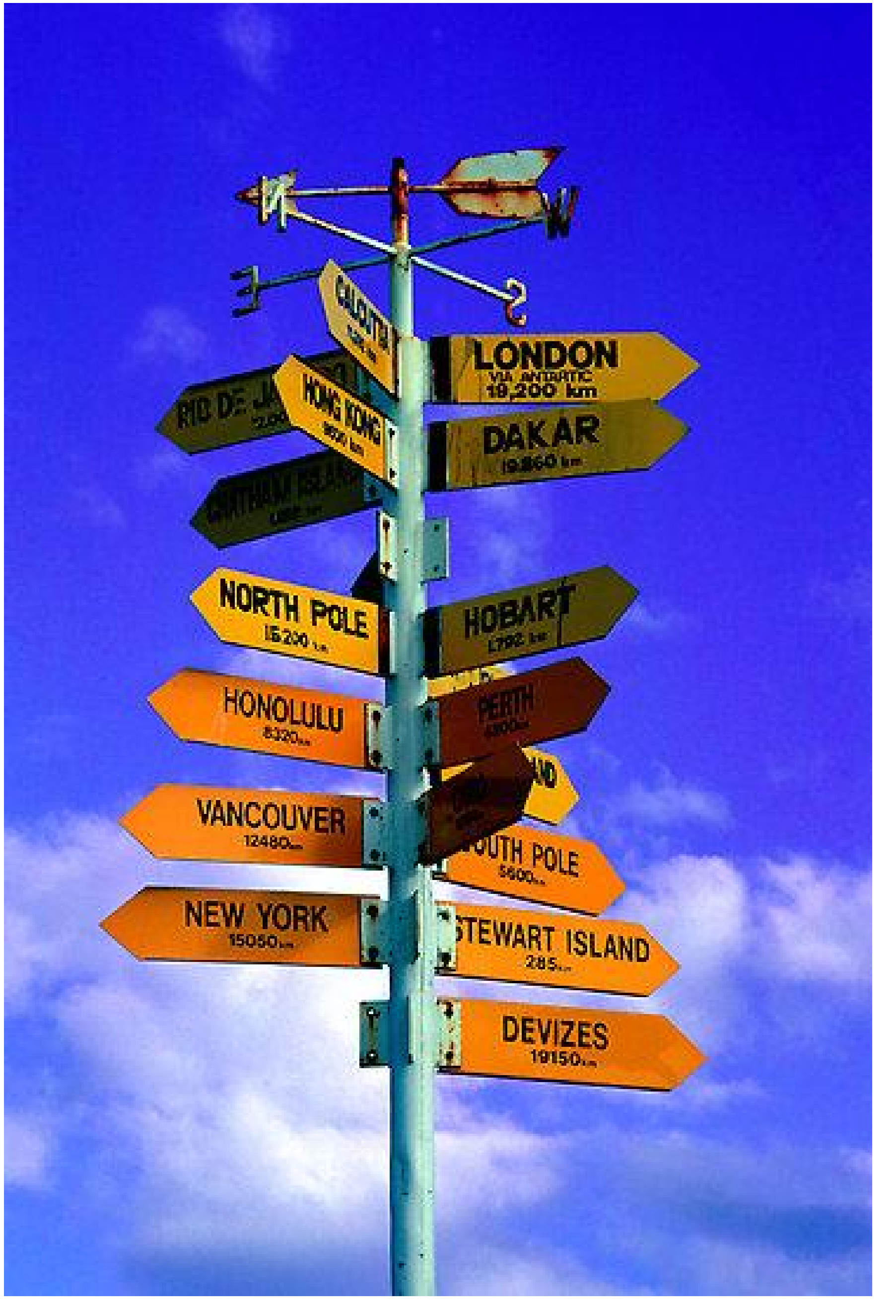
Why interfaces

How interfaces

Semantic changes

Compiler changes

Improving the compiler



Error messages are improved

```
use Sort;
class C { }
var A : [1..10] C;
QuickSort(A);
```

```
/modules/standard/Sort.chpl:58: In function 'InsertionSort':
/modules/standard/Sort.chpl:64: error: unresolved call '<(C, C)'
SortExample.chpl:4: Type 'C' does not implement the 'Comparable' interface
/modules/internal/ChapelBase.chpl:326: note: candidates are: <(a: string, b: string)
```

Constrained generics can be checked eagerly

```
proc InsertionSort(Data: [?Dom](?T))                                    {
     const lo = Dom.low;
     for i in Dom {
          const ithVal : T = Data(i);
          var inserted     = false;
          for j in lo..i-1 by -1 {
               if (ithVal < Data(j)) {        error: unresolved call '<(T, T)'
                    Data(j+1) = Data(j);
               } else {
                    Data(j+1) = ithVal;
                    inserted  = true;
                    break;
               }
          }
          if (!inserted) { Data(lo) = ithVal; }
     }
}
```

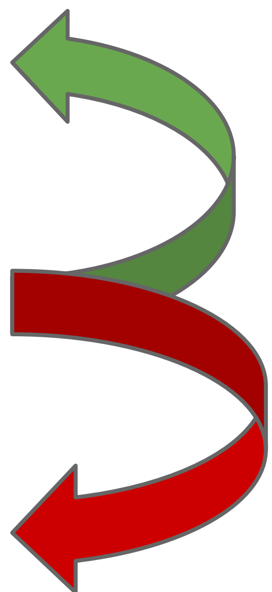
Constrained generics can be checked eagerly

```
proc InsertionSort(Data: [?Dom](?T)) where implements LessThan(T) {
     const lo = Dom.low;
     for i in Dom {
          const ithVal : T = Data(i);
          var inserted     = false;
          for j in lo..i-1 by -1 {
               if (ithVal < Data(j)) {
                    Data(j+1) = Data(j);
               } else {
                    Data(j+1) = ithVal;
                    inserted  = true;
                    break;
               }
          }
          if (!inserted) { Data(lo) = ithVal; }
     }
}
```

Resolves to function defined in LessThan interface

Function call hijacking is prevented

```
module M1 {
     proc helper() {
          writeln(“hello, world!”);
     }
     proc print_hello_world(x) {
          helper();
     }
}

proc helper() {
     writeln(“you’ve been hijacked!”);
}

proc main() {
     M1.print_hello_world(42);
}
```



you’ve been hijacked!

hello, world!

Compiler has less work to do

Unconstrained		Constrained	
`proc foo(x : T   )      { … }`		`proc foo(x : T   )      { … }`	✔
`proc foo(x : int)      { … }`	✔	`proc foo(x : int)      { … }`	
`proc foo(x : double)   { … }`	✔	`proc foo(x : double)   { … }`	
`proc foo(x : string)   { … }`	✔	`proc foo(x : string)   { … }`	
`proc foo(x : Car)      { … }`	✔	`proc foo(x : Car)      { … }`	
`proc foo(x : Account) { … }`	✔	`proc foo(x : Account) { … }`	

Why interfaces

How interfaces

Semantic changes

Compiler changes

Improving the compiler

Interfaces place requirements on types

```
interface Monoid(T) {
    proc binary_op(x:T, y:T):T;
    proc identity_element():T;
}

interface Comparable(T) {
    proc  <(x:T, y:T):bool;
    proc  >(x:T, y:T):bool;
    proc ==(x:T, y:T):bool;
}
```

Implements statements check a type against an interface's requirements

```
interface Monoid(T) {
    proc binary_op(x:T, y:T):T;
    proc identity_element():T;
}

proc binary_op(x:int, y:int):int { return x + y; }
proc identity_element():int      { return 0;     }

implements Monoid(int);
```

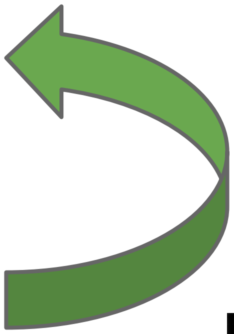
Interfaces allow us to resolve generics without instantiation

```
interface Loggable(T) {
    proc getID(T):int;
    proc getMessage(T):string;
}

interface Runnable(T) {
    proc run(T):bool;
}

proc runEvent(event : ?T) where
  implements Runnable(T) { … }

proc logEvent(id : int,
              message : string) { … }
```

```
proc processEvent(events : [](?T))
  where implements Loggable(T),
                   Runnable(T) {
  for i in 1..events.size {
    var event   : T      = events(i);
    var id      : int    = getID(event);
    var message : string = getMessage(event);

    if (runEvent(event)) {
      logEvent(id, message);
    }
  }
}
```

Eager resolution prevents function call hijacking

```
module M1 {
    proc helper() {
        writeln("hello, world!");
    }
    proc print_hello_world(x) {
        helper();
    }
}

proc helper() {
    writeln("you've been hijacked!");
}

proc main() {
    M1.print_hello_world(42);
}
```



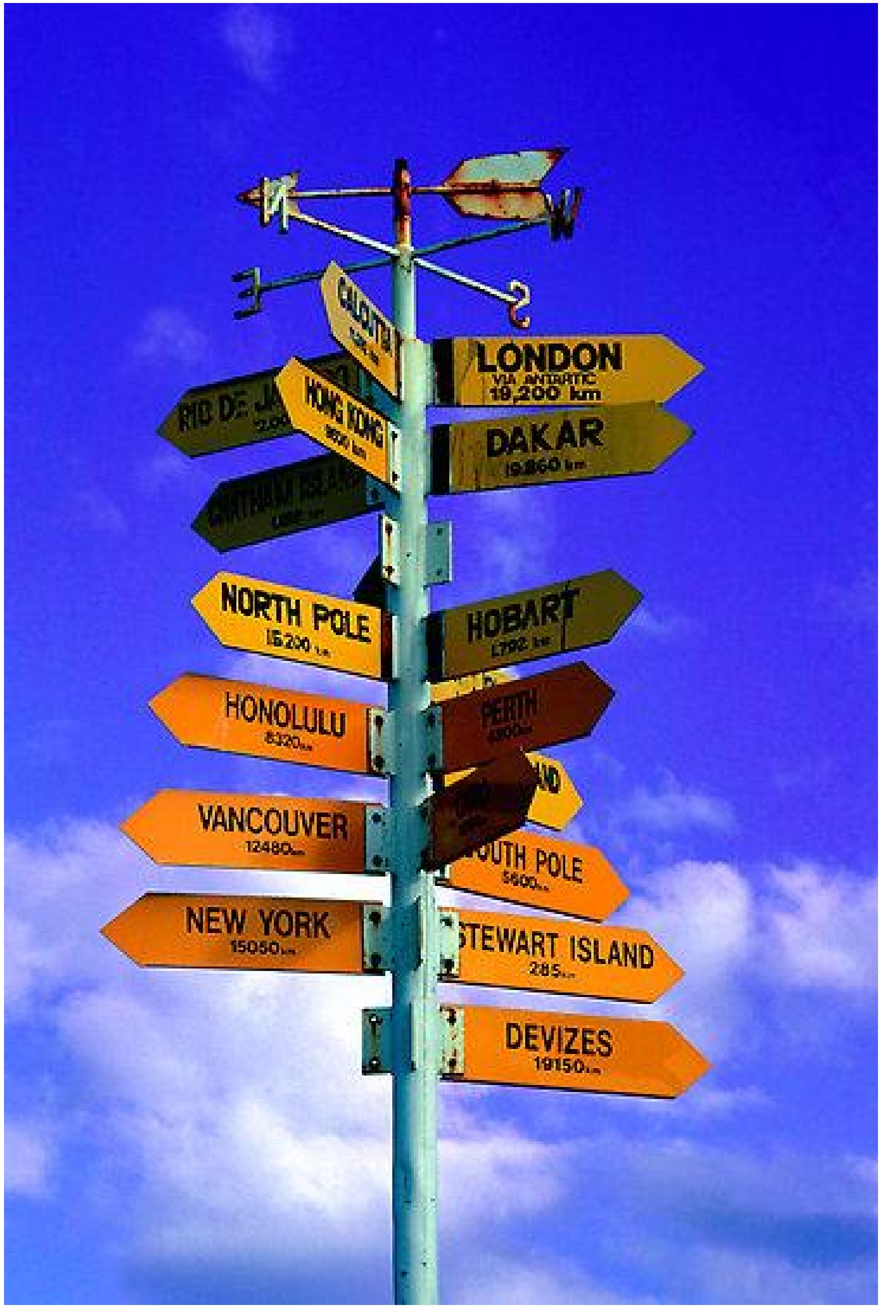
Why interfaces

How interfaces

Semantic changes

Compiler changes

Improving the compiler



Constrained generics are prefered over unconstrained generics

```
proc logEvents(e:[](?T)) { … }
```

```
proc logEvents(e:[](?T)) where
  implements Loggable(T) { … }
```

```
logEvents(getEvents());
```

Constraints can’t increase

```
proc logEvents(l : Log, events : [](?U)) where
     implements Loggable(U); Runnable(U); Copyable(U);

proc processEvents(l : Log, events : [](?T)) where
     implements Loggable(T), Runnable(T) {

     for i in i..events.size {
          run(events(i));
     }

     logEvents(l, events);
}
```

Unconstrained generics gain the caller’s constraints

```
proc logEvents(l : Log, events : [](?U));where
     implements Loggable(U), Runnable(U);

proc processEvents(l : Log, events : [](?T)) where
     implements Loggable(T), Runnable(T) {

     for i in i..events.size {
          run(events(i));
     }

     logEvents(l, events);
}
```

Implementations are passed from one generic to the next

```
module A {
    interface Incrementable(T) {
        proc inc(x:T):T;
    }

    proc inc(x:int):int { return x + 100; }
    implements Incrementable(int);

    proc helper(x:?T)   where implements Incrementable(T) { return    inc(x); }
    proc incOuter(x:?T) where implements Incrementable(T) { return helper(x); }
}

proc inc(x:int):int { return x + 2; }
implements A.Incrementable(int);

A.incOuter(40);
```

⟶ **42**

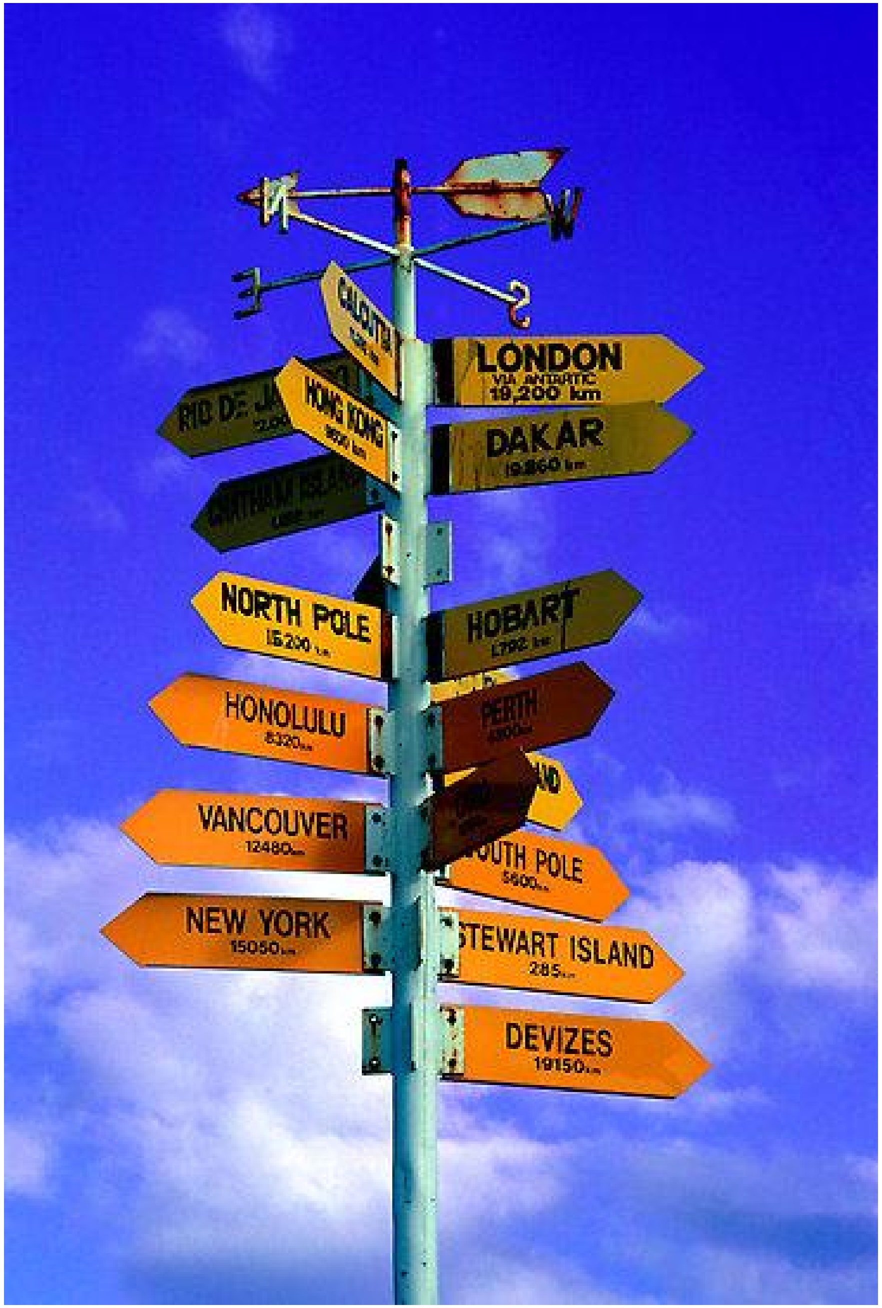
Why interfaces

How interfaces

Semantic changes

Compiler changes

Improving the compiler



Implements statements cause implementations to be built

```
interface Foo(T) {
    proc foo(a:T):T;
    proc zap(a:T, b:T):bool;
}

proc foo(a:int):int { … }
proc zap(a:int, b:int):bool { … }

implements Foo(int);
        foo(int);
        zap(int, int);
```

```
implementation Foo(int) {
    [0]: 0x6542f0
    [1]: 0x6c3570
}
```

Calls can resolve to implementation slots

```
proc constrainedGeneric(x:T, y:U)
    where implements Foo(T), Bar(U) {

    foo(x);       [0][0]

    bar(y);       [1][0]

    zap(x, x);    [0][1]

    baf(y, 42);   [1][1]

    bar(x);       Unresolved call
}
```

```
interface Foo(T) {
    proc foo(a:T):T;
    proc zap(a:T, b:T):bool;
}

interface Bar(U) {
    proc bar(x:U):U;
    proc baf(x:U, y:int):int;
}
```

Specialization replaces implementation slots with pointers

```
proc constrainedGeneric(x: Int, y: Real)
     where implements Foo(T), Bar(U) {

     foo(x);        0x6542f0

     bar(y);        0x8281e0

     zap(x, x);     0x6c3570

     baf(y, 42);    0x6b8e20

}
```

```
implementation Foo(int) {
     [0]: 0x6542f0
     [1]: 0x6c3570
}

implementation Bar(real) {
     [0]: 0x8281e0
     [1]: 0x6b8e20
}

constrainedGeneric(42, 100.0);
```

Why interfaces

How interfaces

Semantic changes

Compiler changes

Improving the compiler

The Chapel compiler has room for improvement

- Large and complicated passes

Function Resolution:
6841 SLOC!

- Mixing of subtyping and tagging
- Poorly documented invariants spread across large sections of code
- Using common C++ idioms and STL classes would make life easier

Chapel mixes subtyping and tagging

```
isFnSymbol(node)

fnSymbol->hasFlag(FLAG_GENERIC)

aggregateType->aggregateTag == AGGRAGATE_CLASS
```

More subtyping would result in less memory usage

```
FnSymbol*  instantiatedFrom;
SymbolMap  substitutions;
BlockStmt* instantiationPoint;
SymbolMap  partialCopyMap;
FnSymbol*  partialCopySource;
Symbol*    retSymbol;
int        numPreTupleFormals;
```

Invariants are not well documented

- Copying a node does not result in a well-formed AST
- remove_help doesn't adjust a node's list member

```
breakInvariant();

intermediateCall();
call();

breakAndFixInvariant();
intermediateCall();

call();
intermediateCall();

fixInvariant();
```

Invariant tracking is time consuming and difficult

```
if (newType) {
  map.put(fn->retType->symbol, newType->symbol);
}

FnSymbol* newFn = fn->partialCopy(&map);
fn->finalizeCopy();

addCache(genericsCache, root, newFn, &all_subs);

if (call) {
  newFn->instantiationPoint = getVisibilityBlock(call);
}

Expr* putBefore = fn->defPoint;
if( !putBefore->list ) {
  putBefore = call->parentSymbol->defPoint;
}

putBefore->insertBefore(new DefExpr(newFn));

for (int i = 0; i < subs.n; i++) {
  if (ArgSymbol* arg = toArgSymbol(subs.v[i].key)) {
    if (arg->intent == INTENT_PARAM) {
      Symbol* key = map.get(arg);
      Symbol* val = subs.v[i].value;
      if (!key || !val || isTypeSymbol(val))
        INT_FATAL("error building parameter map in instantiation");
      paramMap.put(key, val);
    }
  }
}

for_formals(arg, fn) {
  if (paramMap.get(arg)) {
    Symbol* key = map.get(arg);
    Symbol* val = paramMap.get(arg);
    if (!key || !val)
      INT_FATAL("error building parameter map in instantiation");
    paramMap.put(key, val);
  }
}
```

Using common C++ idioms and STL classes makes life easier

- Copy constructors
- operator[]
- Iterators
- vector, map, and list
- Methods over functions

Towards interfaces for Chapel

- Improved programmer experience
- New behaviours for constrained generics
- Function resolution in the presence of type variables
- Suggested improvement to the compiler
- Questions?