

Proposal for Constrained Generics and Module Extensions for Chapel

Jeremy G. Siek

February 22, 2011

1 Introduction

Generics, in its basic form, is a programming language feature that enables the parameterization of software components with respect to type. For example, a `List` class can be parameterized on the type of its elements, enabling the same `List` class to be used for lists of integers, lists of strings, etc. Generics, in its more advanced form, enables parameterization with respect to types and operations on those types. The advanced form of generics facilitates *generic programming*, a software development methodology for creating highly reusable and efficient components [2, 8]. Generic programming entered mainstream programming in 1994 when Stepanov and Lee brought the Standard Template Library to the C++ language [16]. Since then, many other developers have successfully applied generic programming to create highly reusable software libraries [1, 3, 7, 9, 12]. The author of this proposal was the lead developer of the Boost Graph Library [13] and the Matrix Template Library [14].

Chapel currently supports generic programming by providing generic functions, classes, and records. That is, Chapel provides facilities for parameterizing these entities with respect to one or more types. Compared to other programming languages, Chapel's current design for generics most closely resembles that of templates in C++. Chapel does not provide explicit support for constrained generics, which negatively impacts its ease of programming and modularity. The next three subsections describe examples of the negative impacts. We then propose extending Chapel with a notion of *constrained generics*.

Toward improving the modularity of Chapel, we also propose to extend the module feature, which we describe in Section 6. While modules are largely orthogonal to generics, there are a few interactions, and it makes sense to flesh out Chapel's story with respect to modules at the same time as the story for constrained generics, as the goal with both of these features is increased modularity.

1.1 Non-Modular Error Messages

Many misuses of a generic function result in error messages from the compiler that are difficult to decipher. For example, consider the following program that erroneously attempts to sort an array of objects of class `C`.

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

The Chapel compiler emits the following error message:

```
/modules/standard/Sort.chpl:4: In function 'InsertionSort':
/modules/standard/Sort.chpl:10: error: unresolved call '<(C, C)'  
/modules/internal/ChapelBase.chpl:162: note: candidates are: <(a: int(32), b: int(32))  
...
```

The main problem with this error message is that it points to line 10 of `Sort.chpl` as the origin of the problem, when in fact the problem is the use of `QuickSort` with an array element type that does not support less-than comparison. The

error message exposes the internals of the `QuickSort` function to the user, reducing the modularity of the `QuickSort` function. In more complex library functions, the error may originate from within deeply nested auxiliary functions, resulting in exceedingly long and impenetrable error messages. (C++ template libraries have become infamous for this problem [4].)

1.2 Reliability of Generic Libraries

The current design for generics in Chapel negatively impacts the reliability of generic libraries. The body of a generic function is not type checked until it is used. This means that type errors can lie dormant inside generic functions, only to be discovered after they are distributed to users. (The same is true for non-generic functions in Chapel, and this proposal includes improvements for those as well.)

C++ templates are also not fully type checked until they are used, and the author found several bugs that could have been caught if the body of C++ templates were instead type checked at the point of definition. For many years the `replace_copy` function template in the Standard Template Library used a conditional expression to choose between two values of potentially different type [15]. Conditional expressions require that the type of one alternative be convertible to the other alternative, but this was not documented as a requirement of the `replace_copy` template. The fix is to change the conditional expression into an `if` statement, thereby removing the need for convertibility.

Over the years, there were likely many users who tried to use `replace_copy` with types that were not convertible and received an impenetrable error message from the compiler. Such an error is doubly confusing because the user does not know whose fault it is. The user could be looking at a situation in which they misused the generic function, as was the previous case with `QuickSort`, or it could be the fault of the generic function's author, as was the case with `replace_copy`.

1.3 Hijacked Function Calls Inside Generics

There is also a problem concerning the visibility of functions from inside a generic function in Chapel. Suppose that a library developer creates the following module in which the generic function named `print_hello_world` makes a call to another auxiliary generic function named `helper`.

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

Then suppose that an application programmer decides to use `M1` and writes the following code. It just so happens that somewhere in the application, there is another function named `helper`.

```
proc helper(x : int) {
  writeln("you've been hijacked!");
}
use M1;
proc main() {
  M1.print_hello_world(1);
}
```

With the current Chapel function visibility rules, the result of this program is:

```
you've been hijacked!
```

The above is a toy example, but this problem has come up in large C++ applications that use the Boost template libraries. The most troubling aspect of this problem is that there may be no immediate indication that something has

gone wrong, and the programmer may only find out much later and after lots of debugging, that things are not as they seem.

1.4 Discussion

The above examples show how Chapel’s design for generics shares in the some of the pitfalls of C++ templates. These pitfalls have been known for some time in the C++ community and, starting in 2003, there has been active research to improve C++ templates, although a proposal to improve C++ templates has not yet become part of the standard. The author of this proposal played a primary role in that effort [5, 6, 10, 11]. The key idea behind that effort was to provide explicit support for defining interfaces (“concepts” in C++ lingo) and using interfaces to express constraints on templates.

2 Overview of Proposal for Constrained Generics

The core language support for constrained generics requires additions and changes to four areas of the Chapel language. Here we give an overview of these areas before discussing each of them in more detail in the following sections.

Interface definitions provide a mechanism for grouping and naming requirements on types.

Implements statements establish that a type implements the requirements of an interface.

Where clauses will be extended to express constraints on generic functions and generic types. The main kind of constraint is requiring a type parameter to implement an interface.

Instantiation of generic functions, classes, and records changes to include checking that the constraints in the **where** clause are satisfied.

An important aspect of the proposed design is that it provides *modular type checking*. Most languages provide modular type checking for functions, which means that the body of a function is type checked independently of any call to the function, and the type checking of each call to a function only needs to refer to the function signature (the parameter and return types) and not the body of the function. However, the current design of Chapel does not provide modular type checking for functions and generics. Instead, the body is type checked at each call or point of instantiation. This is the root of the problem that results in the non-modular error messages (Section 1.1) and the decreased reliability of generics (Section 1.2). The proposed design enables the modular type checking of both function and generics, thereby solving these two problems.

In addition, because **where** clauses introduce type-specific operations into scope, there is no need for special function visibility rules for constrained generics, thereby avoiding the function hijacking problem discussed in Section 1.3.

Example. Here we walk through a small but complete example that demonstrates the four changes to Chapel to support constrained generics. This example shows a version of `QuickSort` using constrained generics. To start, we define `EqualityComparable` and `LessThanComparable` interfaces that will be used to constrain the element type of the array passed to `QuickSort`.

```
interface EqualityComparable {
    proc ==(x : self, y: self): bool;
}
interface LessThanComparable : EqualityComparable {
    proc <(x : self, y: self): bool;
}
```

The `LessThanComparable` interface extends the interface `EqualityComparable`. The `EqualityComparable` interface requires an implementing type to provide an equality operator and the `LessThanComparable` interface requires an implementing type to also provide a less-than operator. The `self` type is a place

holder for the implementing type. Note that procedure prototypes inside an interface do not have an implicit “this” parameter.

A program asserts that a particular type implements an interface via an `implements` statement. For example, here is a `Person` class that implements the `LessThanComparable` interface.

```
class Person {
  var firstName : string;
  var lastName : string;

  proc ==(other: Person): bool {
    return this.firstName == other.firstName
           && this.lastName == other.lastName;
  }
  proc <(other: Person): bool {
    return this.firstName < other.firstName ||
           (this.firstName == other.firstName
            && this.lastName < other.lastName);
  }
}

Person implements EqualityComparable;
Person implements LessThanComparable;
```

The first `implements` statement is valid because the `Person` class provides an equality operator, and the second `implements` statement is valid because the `Person` class provides a less-than operator and there is a prior `implements` statement that asserts that `Person` implements `EqualityComparable`. A compiler diagnostic error message is printed if an `implements` statement is invalid.

Next we turn our attention to the `QuickSort` function. To place a constraint on the element type of the array, we add to the `where` clause, requiring that the element type implement the `LessThanComparable` interface.

```
proc QuickSort(Data: [?Dom])
  where Dom.rank == 1, Data.elType implements LessThanComparable
{
  ...
  if (Data(mid) < Data(lo)) then Data(mid) <=> Data(lo);
  ...
}
```

When type constraints are added to the `where` clause of a procedure, such as in the `implements` clause above, the procedure is type checked at its point of definition. Any generic types, such as `Data.elType` are considered by the type checker to be unique types, only equal to themselves (unless otherwise specified by type equality constraints). The body of the function may only make use of generic types in ways permitted by the constraints in the `where` clause. For example, in the above `QuickSort`, it would be an error to use any operator other than equality and less-than on the element type of the array.

Last but not least, we consider a call to the `QuickSort` function.

```
var A: [1..1000] Person;
...
QuickSort(A);
```

At the point of the call, the Chapel implementation checks that the `where` clause is satisfied. In this case, it must check that the element type of the array, `Person`, has an `implements` statement for `LessThanComparable`, which it does.

3 Interface Declarations

The syntax for interface definitions is listed below.

interface-declaration-statement:

```
interface interface-name [( interface-formals )] [ : interface-inherit-list ] {  
    interface-statement* }
```

interface-name:

identifier

interface-formals:

identifier
identifier , *interface-formals*

interface-inherit-list:

implements-clause
implements-clause , *interface-inherit-list*

interface-statement:

type *identifier*
type-constraint ;
function-signature-statement
function-declaration-statement

implements-clause:

type-list **implements** *interface-name*

type-constraint:

implements-clause
type-equality

type-equality:

type-specifier == *type-specifier*

where-clause:

where *where-item-list*

where-item-list:

where-item
where-item , *where-item-list*

where-item:

expression
type-constraint

An interface definition consists of a name for the interface, an optional list of type parameters enclosed in parentheses that serve as place holders for the modeling types, a list of interfaces that the interface inherits from, and a body, which is a sequence of statements that express constraints on the implementing types. The type parameters are in scope for the body of the interface. The common case is for there to be only a single implementing type, so if the interface definition omits the list of type parameters, then the type name `self` is in scope for the body of the interface and is a place holder for the single implementing type.

Example. The following `Stack` interface requires that the implementing type provide three methods, `push`, `pop`, and `isEmpty`, and requires that the implementing type specify a type to play the role of the `itemType`.

```
interface Stack {
    type itemType;
    proc self.push(x : itemType);
    proc self.pop(): itemType;
    proc self.isEmpty(): bool;
}
```

The body of an interface consists of a list of constraints. In the following, we describe the various kinds of constraints.

Function signatures A function signature says that an implementation must provide a function with the specified name and compatible parameter and return types. What we mean by “compatible” is described in detail in Section 4.

Example. The following `Vector` interface demonstrates four different kinds of function signatures. The `norm` signature simply requires that the implementation provide a regular function definition for `norm`. The `+` operator can be provided by either a function definition or a method. The `self.size` signature requires a method, and the `self.these` signature requires an iterator method.

```
interface Vector {
    type eltType;
    proc norm(v : self): eltType;
    proc +(u : self, v : self): self;
    proc self.size(): int;
    iter self.these(): eltType;
}
```

Function definitions A function definition statement in an interface provides a default implementation. An implementation may provide an overriding definition of the function, but if not, the definition provided by the interface will be used.

Example. For convenience, the `LessThanComparable` interface could provide definitions for `less-or-equal`, `greater-than`, and `greater-or-equal` that are implemented in terms of the one required `less-than` operator.

```
interface LessThanComparable : EqualityComparable {
    proc <(x : self, y: self): bool;
    proc <=(x : self, y: self): bool
        { return !(y < x); }
    proc >(x : self, y: self): bool
        { return y < x; }
    proc >=(x : self, y: self): bool
        { return !(x < y); }
}
```

Associated types Associated types are types that play an auxiliary role in the implementation of an interface, such as the iterator of a container or the key type of a hash table. The `type` statement adds the requirement for a type definition in any implementation of the interface. The `itemType` in the above `Stack` interface and the `eltType` in the `Vector` interface are examples of requiring an associated type.

Open issue. The use of the keyword **type** for specifying associated types inside interfaces may be confusing, as it has a different meaning than the use of **type** inside a class, which in that context specifies a type parameter. With that in mind, we may wish to find a different keyword for specifying associated types.

Same-type constraints The `==` interface statement requires that the two type expressions refer to the same type. The type equality may be assumed in the body of the interface. When the interface is used in the **where** clause of a generic function, the type equality may be assumed in the body. In any implementation of the interface, the concrete versions of the type expressions must be the same type. We provide an example of same-type constraints with the below example for nested requirements.

Nested requirements Interfaces may be composed using **implements** statements. This composition is similar to composing interfaces using inheritance, but there is one important difference. Inheritance brings in the associated types from the base interface whereas **implements** does not.

Example.

```
interface LinearTransformation(Mat) {
  type Vec;
  type eltType;
  Vec implements VectorSpace;
  Vec.eltType == eltType;
  proc *(A : Mat, x : Vec): Vec;
  proc *(alpha : eltType, A : Mat): Mat;
}
```

4 Implements Statements

The syntax for implements statements is as follows.

implements-statement:

```
type-list implements interface-name [where-clause] ;
type-list implements interface-name [where-clause] { statement* }
```

The implements relation between a type and an interface is established by an implements statement. All the requirements of the interface must be satisfied at the point of the implements statement, either by definitions inside the implements statement, by constraints in the where clause of the implements statement, or by definitions in the lexical scope of the implements statement. The definitions do not have to be an exact match, but they must be coercible to the required function signature. An operator signature may be satisfied by either a regular function or a method definition. The process for finding function definitions is the same as for function name and overload resolution. Inherited interfaces and nested requirements must be satisfied by preceding implements statements. Requirements for associated types are satisfied by type alias statements inside the implements statement or in the class or record definition of the implementing type.

An implementation may itself be generic, which is why there is an optional **where** clause. A common use of generic implementation statements is for adapter classes, that is, classes that are parameterized over a type of some interface, and use that interface to implement another interface. The rules for type checking a generic implements statement are the same as for generic functions, which we discuss in Section 5.

Example. The following example shows some interfaces from abstract algebra and implements statements for integers.

```

interface Semigroup {
  proc binary_op(x : self, y : self):self;
}
interface Monoid : Semigroup {
  proc identity_elt(): self;
}
int implements Semigroup {
  proc binary_op(x : int, y : int):int { return x + y; }
}
int implements Monoid {
  proc identity_elt(x : int, y : int):int { return 0; }
}

```

Of course, there are other ways in which an integer can implement the `Monoid` interface, such as using 1 for the identity element and multiplication for the binary operation. In the proposed design, it is possible to provide multiple `implements` statements for the same type and interface so long as the `implements` statements reside in separate modules.

5 Functions and Where Clauses

function–signature–statement:

```
proc function-name [argument-list] [var-param-clause] [where-clause] ;
```

function–declaration–statement:

```
proc function-name [argument-list] [var-param-clause] [where-clause] [checked] function-body
```

A function signature statement is a forward declaration of a function. It states that the specified function will be provided later or in a separate compilation unit. All of the formal arguments in a function signature must have type annotations.

A function declaration that begins with the keyword **checked** is always type checked at its point of definition, regardless of whether it is called by the main program. Also, if a function declaration includes at least one `implements` clause or type equality constraint in its **where** clause, then the function is type checked at its point of definition. The rules for functional visibility in generic functions (Section 22.2 of the Chapel Language Specification 0.796) do not apply to checked functions.

Open issue. The use of the **checked** keyword is rather verbose, as we expect checked functions to be the common case. One alternative is to trigger separate type checking when all of the parameters of a function have a type annotation. However, we have not proposed that option here because it would not be backwards compatible, that is, some existing Chapel programs would become ill typed.

The *where-clause* of a generic function may include type constraints on the type arguments of the function. The type constraints in the where clause of a function play an important role in the type checking of the function body. Functions and methods in the required interfaces are considered visible in the function body. Furthermore, the required type equalities as stated in the interfaces or directly in the where clause, as well as the congruence closure of those equalities, are assumed to be equal during the type checking of the function body. We discuss issues regarding type equality in detail in Section 5.2.

Example. In the following example, the function `f` is well-typed because `S2 implements Stack`, so the `pop` method on `y` has return type `S2.itemType`, `S1 implements Stack` so the `push` method on `x` has return type `S1.itemType`, and we have required that `S1.itemType equal S2.itemType`.

```

interface Stack(X) {
  type itemType;
}

```

```

    proc X.push(x : itemType);
    proc X.pop(): itemType;
    proc X.isEmpty(): bool;
}

proc f(x: ?S1, y: ?S2)
  where S1 implements Stack,
        S2 implements Stack,
        S1.itemType == S2.itemType
{
  x.push(y.pop());
}

```

In more detail, suppose the where clause includes a requirement of the form T_1, \dots, T_m **implements** \mathbb{I} , interface \mathbb{I} has formal parameters X_1, \dots, X_m , and \mathbb{I} contains the following function signature.

```

proc N(x1 : A1, ..., xn : An):B;

```

Then function N is visible in the function body, except that T_i is substituted for X_i throughout the signature, for $i = 1 \dots m$. The notation $A[B_1 \dots B_n / X_1 \dots X_n]$ refers to the type A with all free occurrences of type variables $X_1 \dots X_n$ replaced by $B_1 \dots B_n$, respectively. So using this notation, the function signature made visible in the function body can be written as

```

proc N[T1...Tm/X1...Xm](x1 : A1[T1...Tm/X1...Xm],
    ...,
    xn : An[T1...Tm/X1...Xm]) : B[T1...Tm/X1...Xm];

```

Inside an interface \mathbb{I} , a declaration of the form **type** Y ; expresses the requirement for an associated type. That is, any type that implements \mathbb{I} must also specify a type to play the role of Y . (Section 4 describes how this is done.) The associated type Y may be referred to in any of the following forms:

1. Y
2. $x.Y$ (if x is the only formal parameter of interface \mathbb{I})
3. $\mathbb{I}(x).Y$

The first two forms are shorthand for the third form. It is an error to use the first or second form if it is ambiguous.

Example. Continuing the above example, in the function f , the type requirement $S1$ **implements** `Stack` causes the following methods to be visible:

```

proc S1.push(x : Stack(S1).itemType);
proc S1.pop(): Stack(S1).itemType;
proc S1.isEmpty(): bool;

```

Similarly, the type requirement $S2$ **implements** `Stack` causes the following methods to be visible:

```

proc S2.push(x : Stack(S2).itemType);
proc S2.pop(): Stack(S2).itemType;
proc S2.isEmpty(): bool;

```

The type equality $S1.itemType == S2.itemType$ is shorthand for

```

Stack(S1).itemType == Stack(S2).itemType

```

It would be an error for the where clause or function body of `f` to refer to `itemType` without any qualification. For example, if the type equality were written as `itemType == S2.itemType`, it would be an error.

As a short-hand for implements clauses in the where clause, an interface name may be used as the type for function parameter.

type-specifier:
interface-name

Example. Here is the example from above, but this time using interface names in the parameter types instead of implements in the where clause.

```
proc f(x: Stack, y: Stack)
  where x.itemType == y.itemType
{
  x.push(y.pop());
}
```

5.1 Overload Resolution for Checked Generic Functions

Generic checked functions participate in overload resolution in a similar way as normal generic functions. The two main differences are that type requirements in where clauses can cause a generic functions to be removed from the set of candidate functions and the type requirements play a role in determining whether one function is more specific than another.

To check whether a generic function is a candidate, first the type arguments for the query types are determined by pattern matching against the types of the arguments to the generic function. Then, the type arguments are substituted for the query types in the where clause. If the resulting type requirements are satisfied in the current scope, then the generic function is a candidate. The following rules specify when a type requirement is satisfied.

1. An implements clause is satisfied if there exists a most specific implements statement in the lexical scope of the point of instantiation.
2. A type equality is satisfied if the two type specifiers are equal in the lexical scope of the point of instantiation.

Example. In the following, the call to `g` resolves to the first function named `g` because that is the only candidate function. The second `g` is not a candidate because class `C` does not implement `J`.

```
interface I(X) { }
interface J(X) { }

class C { }
C implements I;

proc g(x : ?t) where t implements I { return x; }
proc g(x : ?t) where t implements J { return x; }

g(new C());
```

Next, we discuss how type requirements in where clauses affect whether one function is more specific than another function. Suppose that two functions are equally specific using the normal rules. We then consider the type requirements in the where clauses of each function; call them F_1 and F_2 . If the type requirements of F_1 can be satisfied inside the body of F_2 , but not vice-versa, then F_2 is more specific than F_1 .

Example. In the following, the call to `g` resolves to the second function named `g` because both functions are candidates but the second is more specific. In particular, the type requirement of the first `g`, `t implements I`, is satisfied by the where clause of the second `g`, but the type requirement `t implements J` of the second `g` is not satisfied by the where clause of the first `g`.

```

interface I(X) { }
interface J(X) { }

class C { }
C implements I;
C implements J;

proc g(x : ?t) where t implements I { return x; }
proc g(x : ?t) where t implements I, t implements J { return x; }

g(new C());

```

5.2 Type Equality

Type equality is a *congruence relation*, which means several things. First it means that type equality is an *equivalence relation*, so it is reflexive, transitive, and symmetric. Thus, for any types ρ, σ, τ we have

- $\tau = \tau$,
- $\sigma = \tau$ implies $\tau = \sigma$, and
- $\rho = \sigma$ and $\sigma = \tau$ implies $\rho = \tau$.

Example. The following function is well typed.

```

proc g(f : func(?T, ?S), a : ?R) : T where R == S, T == S {
  return f(a);
}

```

There are two things to check in the body of this function to determine whether it is well typed. First, for the call `f(a)` to be well typed, the argument `a` must have the same type as the parameter of function `f`. Thus, we need `R` equal to `T`. The type checker would use the following reasoning to prove this. The **where** clause gives us `T == S`, so by symmetry `S == T`. The **where** clause gives us `R == S`, so by transitivity, we have `R == T`, which is what we needed.

The second thing that needs to be checked is that the thing returned has the same type as the declared return type, so in this case we need to check that `S` is equal to `T`. Again, we have `T == S` from the **where** clause, so by symmetry `S == T`.

The second aspect of type equality being a congruence is that it propagates in certain ways with respect to type constructors (that is, ways of constructing larger types out of smaller types). For example, if we know that `S == T`, then we can deduce that `func(S, S) == func(T, T)`. Similarly, if we have a generic class such as

```

class C {
  type X;
}

```

then `S == T` implies `C(S) == C(T)`.

Example. The following function is well typed.

```

proc g(a : C(S), b : T) : C(T) where S == T {
  return a;
}

```

The only thing that needs to be checked is the return type, that is, we need to check whether $C(S) == C(T)$. But we know this is true using $S == T$ and the fact that equality is a congruence.

The propagation of equality can also go in the other direction. For example, $C(S) == C(T)$ implies that $S == T$.

Example. The following function is well typed.

```

proc g(a : S, b : T) : T where C(S) == C(T) {
  return a;
}

```

For this function, we need to check whether $S == T$. The **where** clause gives us $C(S) == C(T)$, so again by the fact that type equality is a congruence, we know $S == T$.

The congruence also extends to associated types. For example, given the following interface

```

interface I {
  type helper;
  proc self.get_helper() : helper;
}

```

then $S == T$ implies $I(S).helper == I(T).helper$. However, for associated types, the propagation does not go in the reverse direction. The equality $I(S).helper == I(T).helper$ does not imply that $S == T$.

Example. The following function is *not* well typed.

```

proc f(a : ?S, b : ?T) : ?T
  where S implements I, T implements I, I(S).helper == I(T).helper
{
  return a;
}

```

Just because $I(S).helper == I(T).helper$ does not mean that $S == T$.

Like type parameters, associated types are in general assumed to be different from one another.

Example. The following program is also *not* well typed.

```

proc f(s : ?S, t : ?T) where S implements I, T implements I {
  var x : ?S.helper = t.get_helper();
}

```

5.3 Query Types

We propose relaxing the restriction that a query type may only appear once in a function signature. Instead, a query type may appear any number of times. The query type resolves to the match for the first occurrence of the query type.

Example. The following function and function call are well typed, assuming that real numbers implement `LessThanComparable`. The query type `T` resolves to `real`, and the second argument in the function call is coerced to `real`.

```

proc min(x : ?T, y : ?T): T where T implements LessThanComparable {
  if y < x then
    return y;
  else
    return x;
}
min(1.0, 2);

```

6 Modules

We propose several extensions to the module feature of Chapel. Our changes are backwards compatible, that is, all the existing language rules regarding modules stay the same.

First, we propose visibility restrictions by adding `public` and `private` sections. The default visibility of a module is `public`. Any definitions in a private section of a module are not visible from outside the module.

visibility-statement:

```

public: statement*
private: statement*

```

If a module contains a `use` statement in a public section, then the imported names are also available for being imported from the module. On the other hand, a `use` statement in the private section of a module only makes those names available for use in the module.

We also propose several additions to the `use` statement. First, we introduce a new `use` statement to import implementations. The resolution of the `implements` clause is the same as that for satisfying requirements at the point of instantiation of a generic function.

use-statement:

```

use implements-clause from module-name ;

```

Example. The following example imports an `implements` statement from module A into module B. Module B calls function `f`, which requires that C implements I, which is satisfied by the use statement.

```

interface I { }
proc f(x : ?T) where T implements I { }

module A {
  class C { }
  C implements I;
}

module B {
  use A.C implements I from A;

  proc g(y : A.C) {
    f(y);
  }
}

```

While we're at it, we propose a fine-grained `use` statement for importing specific definitions and optionally renaming them.

use-statement:

use *import-clause-list* **from** *module-name* ;

import-clause-list:

import-clause

import-clause , *import-clause-list*

import-clause:

identifier

identifier **as** *identifier*

7 The Any Type

We propose to support dynamic dispatch and dynamic polymorphism for interfaces using the **any** type feature. There are three forms for specifying an **any** type. We start with the simplest.

type-specifier:

any *interface-name*

Any type *T* may be implicitly cast to the type **any** *I*, if *T* **implements** *I*. The methods and functions in interface *I* are available for use on objects of type **any** *I*.

Example.

```
interface I {
  proc self.f() { }
}
class C { }
C implements I;

var x : any I = new C();
x.f()
```

The next **any** form adds more expressiveness by including a **where** clause that includes one or more constraints on the **any** type. The type specifier is a pattern that specifies what the **any** type looks like.

type-specifier:

any *type-specifier* **where** *where-item-list*

Example. Continuing from the previous example, we have an any type that implements two interfaces, I and J.

```
interface J {
  proc self.g() { }
}
C implements J;

var y : any ?T where T implements I, T implements J = new C();
y.f();
y.g();
```

The next example demonstrates the use of an any type with a slightly more complex type specifier, in this case a tuple.

```
var t : any (?T, ?U) where T implements I, U implements J = (new C(), new C());
t(1).f();
t(2).g();
```

A Alphabetical Syntax Productions

formal-type:

: interface-name

function-declaration-statement:

proc *function-name* [*argument-list*] [*var-param-clause*] [*where-clause*] [**checked**] *function-body*

function-signature-statement:

proc *function-name* [*argument-list*] [*var-param-clause*] [*where-clause*];

implements-clause:

type-list **implements** *interface-name*

implements-declaration-statement:

type-list **implements** *interface-name* [*where-clause*] (*block-statement* | ;)

interface-declaration-statement:

interface *interface-name* [(*interface-formals*)] [: *interface-inherit-list*] {
 *interface-statement** }

interface-formals:

identifier

identifier , *interface-formals*

interface-inherit-list:

implements-clause

implements-clause , *interface-inherit-list*

interface-name:

identifier

interface-statement:

type *identifier*

type-constraint ;

function-signature-statement

function-declaration-statement

statement:

function-signature-statement

type-declaration-statement:

interface-declaration-statement

implements-declaration-statement

type-equality:

type-specifier == *type-specifier*

type-constraint:
implements-clause
type-equality

type-specifier:
any *interface-name*
any *type-specifier* **where** *where-item-list*

where-clause:
where *where-item-list*

where-item-list:
where-item
where-item , *where-item-list*

where-item:
expression
type-constraint

References

- [1] F. Alet, P. Dayal, A. Grzesik, A. Honecker, M. Koerner, A. Laeuchli, S. R. Manmana, I. P. McCulloch, F. Michel, R. M. Noack, G. Schmid, U. Schollwoeck, F. Stoeckli, S. Todo, S. Trebst, M. Troyer, P. Werner, and S. Wessel. The ALPS project: open source software for strongly correlated systems. *J.PHYS.SOC.JPN.*, 74:30, 2005.
- [2] M. H. Austern. *Generic programming and the STL: Using and extending the C++ Standard Template Library*. Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., 1998.
- [3] J.-D. Boissonnat, F. Cazals, F. Da, O. Devillers, S. Pion, F. Rebufat, M. Teillaud, and M. Yvinec. Programming with CGAL: the example of triangulations. In *Proceedings of the Fifteenth Annual Symposium on Computational Geometry*, pages 421–422. ACM Press, 1999.
- [4] R. Garcia, J. Järvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. In *OOPSLA '03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 115–134. ACM Press, 2003.
- [5] D. Gregor and J. G. Siek. Implementing concepts. Technical Report N1848=05-0108, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, August 2005.
- [6] D. Gregor, J. G. Siek, J. Willcock, J. Järvi, R. Garcia, and A. Lumsdaine. Concepts for C++0x (revision 1). Technical Report N1849=05-0109, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, August 2005.
- [7] U. Köthe. *Handbook on Computer Vision and Applications*, volume 3, chapter Reusable Software in Computer Vision. Academic Press, 1999.
- [8] D. R. Musser and A. Stepanov. Generic programming. In *ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation*, 1988.
- [9] W. R. Pitt, M. A. Williams, M. Steven, B. Sweeney, A. J. Bleasby, and D. S. Moss. The bioinformatics template library: generic components for biocomputing. *Bioinformatics*, 17(8):729–737, 2001.

- [10] J. G. Siek. *A Language for Generic Programming*. PhD thesis, Indiana University, August 2005.
- [11] J. G. Siek, D. Gregor, R. Garcia, J. Willcock, J. Järvi, and A. Lumsdaine. Concepts for C++0x. Technical Report N1758=05-0018, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, January 2005.
- [12] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., 2002.
- [13] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.
- [14] J. G. Siek and A. Lumsdaine. The Matrix Template Library: Generic components for high-performance scientific computing. *Computing in Science and Engineering*, 1(6):70–78, Nov/Dec 1999.
- [15] J. G. Siek and A. Lumsdaine. A language for generic programming in the large. *Science of Computer Programming*, September 2008.
- [16] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.