Chapel Specification 0.4

Cray Inc
411 First Ave S, Suite 600
Seattle, WA 98104

February 4, 2005

1 Overview

This document is a working definition of the *Chapel* programming language.[1] It covers the fundamental elements at an intuitive level rather than formal one, with some rationale for choices that we have made.

The overriding goal of Chapel is to provide a language that bridges between algorithm development and production deployment within the domain of high-performance, parallel computing. This means that while incorporating many aspects of modern programming practice from the broader market, some special features are needed and some compromises are made to ensure high-performance can be achieved without reverting to a low-lever language.

To accomplish the broad goal above, Chapel strives to improve programmer productivity with the following techniques:

Locality-aware Multi-threading The target machine for Chapel is a parallel system with any number of processors, all of which have the same functional access to program state. Programmers are responsible for describing the available concurrency in their program, but the system will manage the details of implementing that work for specific environments. This is frequently called a *multi-threaded* programming model. This model is then augmented with a notion of *locale* that supports expression of the affinity between computations and the data they access.

[1] This language is part of the Cascade Project at Cray Inc. funded by DARPA Contract No. NBCH3039003 as part of their High Productivity Computing Systems initiative.

The expectation is that there is a performance advantage by exploiting this affinity. We call this *locality-aware multi-threading*. This concept represents the consensus view of high-performance system architecture: multiple processors with non-uniform access to system memory, and higher local memory bandwidth than global memory bandwidth.

The locality-aware multi-threaded programming model has significant advantages over existing programming models by removing from the programming model the need to manage processors explicitly. In Chapel the more abstract, problem-oriented concepts of concurrency and affinity are the concerns of the programmer. The assumption of a global name space for program state frees programmers from explicit management of communication in a system with physically distributed memory while retaining the ability to express the affinity necessary to minimize such communication. The multi-threaded model further provides a mechanism to tolerate latency of remote memory accesses via concurrency although not all architectures can efficiently exploit this capability.

New concepts in Chapel are the *domain* and its *distribution* to locales. These are tools to describe collections of problem data and their decomposition onto physically distributed memory. These are integrated into control constructs to allow convenient description of distributed computation.

Generic Programming High-productivity requires that software components be reusable. This is often at odds with the stringent performance requirements of high-performance computing where data structures are often intertwined with algorithms in ways that make adapting them to new contexts difficult. Chapel will mitigate this problem by adding language concepts to abstract primary aspects of data structures to allow algorithms to be expressed in a *generic* manner and then automatically customized to context-specific data structures choices. Chapel specifically looks at three aspects of data structures and provides abstraction tools:

Type Variables Many data structures implement generic concepts parameterized by the type of element they manipulate. Chapel allows program fragments to be parameterized by variable type. Rather than relying on dynamic polymorphism to support this,

Chapel specializes the fragments to specific bindings of type variables. This way Chapel programs enjoy the traditional benefits of compile-time type checking, both for error checking and for higher performance, without sacrificing expressiveness. This mechanism augments popular object-oriented techniques to allow parameterization of behavior.

The power of type variables is amplified by using a structural notion of type for function parameters. In this model, any data object can be used as an actual parameter to a function as long as the types of its component fields are consistent with the constraints on type variables specified by the function. This differs from types systems like Java where subtype relationships are related to the specification of the object. Chapel provides syntactic tools for constructing derived types that support this typing model.

Iteration Over Sets A major use of data structures is to describe ordered sets of values and to control the iteration over these sets. In Chapel, there is language support for such sequences both as a derived type constructor and with special control constructs to allow clean separation of data structure traversal from client algorithm code that is only interested in the generated sequence of values. This extends the generic programming by abstracting from the details of data structures to the concept of iteration.

Mapping Values to State Variables The second major use of data structures is to implement mappings from sets of values to associated state variables. This concept encompasses both Fortran-style multi-dimensional arrays as well as more explicitly programmed data structures that implement various kinds of dictionaries, some times called associative arrays. Chapel builds on the array concept and extends it by generalizing the kinds of values that can be used as "subscripts".

The sequence and array concepts are integrated with the generic programming techniques to allow application-specific specializations for performance-critical kernels.

This specification is separated into the following major chapters which describe orthogonal features and gradually introduce new concepts not commonly found in standard languages.

Base Language Features We first describe a number of standard programming concepts common to most imperative languages. This includes primitive types, expression syntax, statement syntax, function definition, function invocation and function closures. Our goal is to build on existing concepts when appropriate, so many aspects, such as the detailed behavior of floating point arithmetic, are discussed only briefly. This does not reflect indifference but merely our focus on problems more in need of attention and new language support. Where we omit details now, the reader should assume that they will be filled in later based on existing practice.

Structured Types We expand on the primitive types by adding rules for construction of structured types. This supports both the structural typing discussed above but also the object-oriented paradigms.

Sequences and Iterators Chapel's sequence data type is an ordered sequence of values of some base type. A special function form called an iterator provides a convenient bridge between simple sequences and more complex data structures.

Type Unions Type unions are a derived type that permits variables to hold a more heterogeneous collection of values. Chapel provides type-safe unions by permitting access to values only after the specific component types have been resolved.

Type Parameters and Determination We introduce the notion of type variables and the rules for determining the types of data variables from usage. The rules for function invocation and the resolution of overloaded functions are defined. While type variables provide a form of polymorphic program specification, we expect programs to be completely type checked at compile-time. As with C++ templates, we will instantiate specialized function and type definitions as needed to resolve types so that the only "method dispatch" from the object-oriented sublanguage and explicit type tests remain as runtime costs.

Domains and Arrays The Chapel domain describes a collection of names to which data can be associated. An array is a mapping from a domain

to a collection of variables[2]. These concepts generalize Fortran-style multi-dimensional arrays as well as various kinds of dictionaries.

Parallelism and Synchronization Chapel is an explicitly parallel language with additional control constructs to identify concurrent sub-computations and features to coordinate those computations. Chapel also defines a simple mechanism to facilitate multi-word updates against shared data structures.

Locales and Distribution The Chapel memory model incorporates a notion of *locale* that reflects the distributed nature of large scale computers. Both data and computation can be associated with locales. Chapel provides a mechanism to allow domains, and thereby the arrays defined over them, to be decomposed across locales. The Chapel programming model presumes a performance advantage to co-locating data and computation in the same locale but does not require this in the definition of the behavior of primitive operations.

Aggregate Expressions Chapel extends the base expression interpretation to allow element-wise operation on data aggregates in the form of arrays and sequences. The element-wise operator set is extended with special operators on aggregates that effect ordering and perform reductions and recurrences. All of these operations are implicitly parallel and interact with the locality and distribution aspects of the language.

Contents

[2]The idea of separating domains and arrays is most directly modeled after ZPL, The domain's use as a mechanism to implement domain decomposition is derivative of High Performance Fortran (HPF).

2 Base Language Concepts

This section describes concepts common to most imperative languages. These include literal constants, expressions, primitive types, variable declarations, statements, function definitions, function invocation, function closures and modules. Later sections will describe derived types such as records, classes, sequences, and unions. We provide an incomplete description with the understanding that definitions will evolve as the language matures.

2.1 Requirements

The base language must allow the basic data types and operators standard in programming languages in familiar ways. We choose to use an imperative language as a base rather than a functional or declarative language because it is more straightforward to make performance guarantees.

Beyond this basic framework we have some additional requirements:

1. Allow type information to be omitted to allow generic interpretation. This means we need to be able to specify scoping for variables without specifying type. This include function arguments, results, and fields in derived types. However, it should be possible to specify explicit types to facilitate clarity of interfaces.

To further the goal of productivity, we have additional targets:

1. Support strings as a primitive data types, and provide built-in mechanisms to convert between other types.

2. Support automatic storage management, and provide options for manual storage management.

3. Support name-space management, appropriate for large programs.

2.2 Reading Chapel

Some broad guidelines to reading Chapel are:

1. Curly braces ("{}") delimit structure for both complex type definitions and compound statements. Each such occurrence usually defines a lexical scope.

2. The scope of a symbol is typically the entirety of an enclosing scope. In particular, a function name can be used before it is defined and function definitions can have free variables, which are defined later in an enclosing scope.

3. The colon operator, "`:`", introduces type designators. Where a type is easily determined from context, the designator is optional.

4. The operator "`=>`" indicates that the right operand is not fully evaluated. This can be used to generate an alias for a variable or for a subset of an array's range. It can also be used to curry a function.

5. The token "`_`" serves as a wild card in contexts where a symbol, type, or expression is omitted but must be accounted for positionally.

6. Classes and method invocation are largely similar to C++ using operator "`.`" with an object instance as its left operand and a field name as its right operand.

7. The keyword "`do`" is optional in many contexts where visual separation between an expression and a following statement improves readability.

8. The keyword "`nil`" defines an undefined value or an empty object.

The table in Figure 1 on page 17 lists the operators used in expressions and Figure 3 on page 39 summarizes the statement syntax.

Notation In the following text, examples will use a `fixed width font` for literal tokens, operators, and keywords. Italics inside angle brackets, *e.g.,* "⟨*term*⟩", denote a lexical token like a symbol or a non-terminal that can be replaced with some multi-token expansions. Square brackets, *e.g.,* "[⟨*term*⟩]", denote optional terms. A list of one or more terms is indicated by following the term with an ellipsis inside brackets ("[...]"). If the list has a separator such as a comma, that is added inside the brackets and before the ellipsis, as in "[, ...]".

2.3 Program Structure

A Chapel program is organized as a collection of named modules. Each module associates symbols with types, variables, and functions. The syntactic

entities that specify these associations are called *declarations*. The bodies of functions can have additional declarations but also include sequences of statements that define the behavior of that function. We say declarations that are not inside a function are "at module scope". Statements that are not declarations generally appear only inside of function bodies are referred to as *executable statements.*

Executable statements specify changes in program state by altering values stored in variables, and they control the sequencing of these state changes. The description of new values is through the use of standard infix notation to build expressions from literal constants, variables, and function invocations. This base is augmented in later sections with extensions for dealing with sequences.

2.4 Lexical Elements

The lexical structure of Chapel is similar to most conventional languages with operators modeled after C and Fortran.

White Space White-space characters are spaces, tabs and newlines. Aside from delimiting other lexical elements, they are largely ignored.

Symbols Symbol characters include upper and lower case letters, digits, and the characters `$`, `?` and `_`. The character `?` is also an operand and so cannot begin a symbol. By convention, we use `?` at the end of a symbol to indicate a side-effect free function that returns a boolean value. ⟨*symbol*⟩ denotes a string of symbol characters.

Operators and Grouping Remaining characters form one and two-character operators:

`+`	`-`	`*`	`/`	`**`		arithmetic
`==`	`!=`	`<`	`<=`	`>`	`>=`	comparison
`^`	`~`	`&`	`|`			bitwise
`#`	`..`					sequence
`=`	`=>`	`:`	`;`	`,`	`_`	other

and grouping tokens:

`{`	`}`	structures and statements
`(`	`)`	tuples, argument lists
`[`	`]`	iteration specification
`(/`	`/)`	sequence constructor

Comments Non-semantic comments begin with either "`--`" or "`//`" when they are not inside a string literal and continue to the next newline character. The token "`/*`" begins a comment that ends at a matching "`*/`" which may not be inside a nested comment or a string literal. Aside from separating other tokens, comments are ignored.

Reserved Words The following symbols are reserved as keywords and operators[3].

`and`	`array`	`atomic`	`break`	`call`
`class`	`cobegin`	`config`	`const`	`constructor`
`continue`	`distribute`	`do`	`domain`	`enum`
`except`	`_expect`	`for`	`forall`	`function`
`goto`	`if`	`implements`	`in`	`inout`
`_invariant`	`iterator`	`let`	`like`	`_local`
`module`	`nil`	`not`	`or`	`otherwise`
`out`	`parameter`	`_private`	`private`	`public`
`_release`	`repeat`	`return`	`select`	`subtype`
`to`	`type`	`typeselect`	`union`	`until`
`_unordered`	`var`	`_view`	`when`	`where`
`while`	`with`	`yield`		

Reserved words that begin with an underscore ("`_`") are for optimization and can be removed from the program without affecting the intended interpretation of a program.

2.5 Primitive Types and Literal Constants

Arithmetic Types Chapel supports a number of bounded range integer types with the standard two's complement representation. They are charac-

[3]This list is rather long. How hard would it be to make some of the words reserved only in context?

terized by a byte-width k and can represent integers in the range of -2^{8k-1} to $2^{8k-1}-1$ inclusive. The default width is 8 bytes.[4]

Integer constants are represented by strings of decimal digits, a string of hexadecimal digits prefixed with `0x`, or a string of binary digits prefixed with `0b`. The strings are interpreted in the usual positional manner for bases 10, 16 and 2 respectively. Alternate literals for integer value 12 are this: `12`, `0xc` or `0b1100`.

There is no unsigned integer type and no shift operators. Instead, predefined shift functions (page 153) implement left shift as well as signed and unsigned variants of right shift.

Chapel's floating point arithmetic follows the IEEE 754 standard. Again there are various sizes of representation characterized by a byte-width. The default width is 8 bytes.

Floating point constants must include a single decimal point at any place and/or be followed by a suffix beginning with `e`, an optional `+` or `-`, and then a required integer constant. The suffix indicates scaling by 10 raised to the power of the signed integer value. For example, `6.673e-11` represents the value 6.673 multiplied by 10^{-11}.

Chapel also supports complex data types following the usual Cartesian representation consisting of two floating point numbers. The representation width of each of those floating point numbers is half of the representation width of the complex type. An imaginary complex literal is represented by adding a "`i`" as a suffix with no intervening white space. Examples are: `12i`, `45.2i`, `0.45e12i`. A real complex literal can be formed by converting a real literal to a complex (`14.0:complex`) or implicitly by adding an imaginary 0 (`14.0+0i`).

These arithmetic types are designated by the predefined symbols `integer`, `float`, and `complex`. Each of these can be modified with an explicit size value, for example `integer(4)` or `float(size=4)` that specifies the representation width.

Representation width of arithmetic literals is the larger of the default width or the smallest width that can represent the value. This may be adjusted by following the literal with `:` and a type designator that identifies a primitive arithmetic type. For example, a 4-byte floating point constant

[4]The choice of 8 here assumes that by 2010 64-bit processors will be the norm. However, it is still not clear that 4-byte integers are not the right default. The usual motivation is to index large tables but we could have a different default width for integer domain indices than for vanilla integers.

could be specified as `1.5:float(4)`.

Boolean Type Chapel defines a boolean data type designated by the symbol `boole` with two predefined values, `true`, and `false`. Comparison operators return boolean results, logical operators have boolean values for operands and results, and some statement contexts require boolean values.

String Type Chapel supports strings of unbounded extent as a primitive type designated by the predefined symbol `string`. Chapel has no separate character type for the special case of strings of length 1. String literals are represented as in C using `"` as a delimiter and using `\` as an escape character. Single quotes, `'`, may be used as an alternative to `"` where convenient. Standard C conventions apply for representing non-printing characters such as `\n` for ASCII newline.

Strings are defined over one of a number of system defined alphabets. The symbol `ASCII` is associated with the ASCII standard encodings to 7-bit values stored in bytes. In the initial implementations strings default to `string(alphabet=ASCII)`. The construction

```
10:ASCII
```

will convert an integer to a one-character string according to the encoding rules of the associated alphabet. A predefined function is used to map a one-character string back to the integer value based on the underlying alphabet.

Enumerated Types As in C, an enumerated collection of symbols defines a type. The syntax is the same as in C. Each symbol has an associated integer value and may be used anywhere an integer can. For example:

```
enum ExprTypes { ADD, MUL, DIV, NEG=-1 }
```

Here `ExprTypes` is the designator for the type and the other symbols, like `ADD`, can be used as literal values of this type.

2.6 Expressions

Chapel expressions consist of a number of primitives plus a number of prefix and infix operators. This is extended with a general notion of function invocation described later. The precedence of operators is given in the table in

Operator	Purpose	Operand Types
`:`, `like`	conversion	
`.`	field selection	
`**`	exponentiation	arithmetic
`~`	bitwise complement	integer
`+`,`-`	unary additive	arithmetic
`*`,`/`	multiplication	arithmetic
`+`,`-`	addition	string or arithmetic
`&`	bitwise conjunction	integer
`^`	bitwise exclusive-or	integer
`|`	bitwise disjunction	integer
`<`, `<=`, `>`, `>=`,	ordered comparison	any, result boole
`==`, `!=`	equality comparison	any, result boole
`not`	boolean complement	boole
`and`	conditional conjunction	boole
`or`	conditional disjunction	boole
`#`	sequence concatenation	sequence types
`if`, `let`		
`[ ]`	iteration	any
`,`	list separator	

Figure 1: Operator Precedence. Operators listed earlier have higher precedence than those that are listed later.

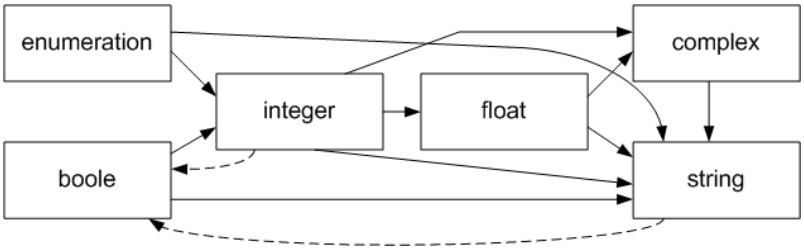

Figure 2: Type Conversions. Solid arrows indicate implicit conversions that may be performed when operators have different operand types or when a value is assigned to a variable. Only one implicit conversion is allowed so `boole` may not be implicitly converted to `float`. Dashed arcs indicate conversions that are legal when a boolean value is required by a statement.

Figure 1. Earlier operators have higher precedence. Except for exponentiation, amongst operators of equal precedence, operations are left-associative so `a+b+c` is interpreted as `(a+b)+c`. Exponentiation is right-associative.

The interpretation of operator symbols is subject to the types of the operands, except where those operations are values of primitive types described above. Then default interpretations are defined by the language. Chapel adopts normal interpretation following the current C, Fortran and IEEE floating point standards with clarifications stated here. Chapel defines that within an expression operand terms are evaluated left to right.

Generally the result type of an expression involving primitive types will be the same as the operand type. The conversion and comparison operators are an exception to this. The conversion operator is an infix operator whose left operand is a value and whose right operand is a type designator. The type of the expression is the specified type. The result of a comparison of primitive types is a `boole`.

Generally, we expect the operands of binary operators to agree in both kind and representation width. When they disagree, the smaller operand is first converted to the representation width of the larger. Types are then implicitly converted according to the solid arrows in Figure 2. Booleans are converted to integers by mapping `true` to 1 and `false` to 0. Booleans are mapped to strings as `"true"` and `"false"` respectively. Enumeration values can be converted directly to strings in which case they are represented by their symbolic names. Thus `ADD:string` is the same as `"ADD"`. For conditional

boolean operators and other contexts that require a boolean value, integers are converted to boolean by mapping 0 to `false` and non-zero values to `true`. Strings can also be mapped to boolean where the empty string and the string `"false"` map to `false` and all other strings map to `true`.

The above conversions are collectively called *standard promotions*. Conversions that do not correspond to a solid arrows in the diagram require an explicit conversion operator. The dashed arrows represent implicit conversion to `boole` when required for statements that require boolean values.

Integer division is defined as returning the floor of the corresponding real result.

The operator `+` is used to indicate string concatenation as well as addition for the arithmetic types.

The boolean operators `and` and `or` evaluate the left operand and then only evaluate the right operands if the left is `true` or `false` respectively.

The representation of arithmetic types is not generally visible to the programmer. There are be predefined functions to support converting between floating point and integers values of the same representation width without changing the bit-level representation. These provide access to the low-level representation of the floating point numbers.

Conditional Expressions The construction

`(if` ⟨*expr*$_1$⟩ `then` ⟨*expr*$_2$⟩ `else` ⟨*expr*$_3$⟩`)`

is also an expression. The parentheses are required in this expression.[5] The first expression is evaluated as a boolean and selects either the second or third expression. The unselected expression is not evaluated. The result types is the type corresponding to applying default promotions to ⟨*expr*$_2$⟩ and ⟨*expr*$_3$⟩.

Let Bindings A `let` expression associates a symbolic name with an intermediate result in an expression. The syntax involves a list of variable bindings and an expression:

`let` ⟨*binding*⟩ [`,` ...] `in` ⟨*expr*$_2$⟩

where each ⟨*binding*⟩ has the form

[5]There is a syntactic ambiguity when the first token in a statement is an `if`. Is it an expression statement or an `if`-statement?

⟨*symbol*⟩ [:⟨*type*⟩] = ⟨*expr*$_1$⟩

where ⟨*expr*$_1$⟩ is evaluated and its value associated with ⟨*symbol*⟩ for the balance of the expression. When ⟨*type*⟩ is specified, the standard promotions are applied to ⟨*expr*$_1$⟩ when the value is not the same as ⟨*type*⟩. When a list of bindings is specified, expressions can refer to earlier symbols, as in:

```
let x:float = a*b, y = x*x in 1/y
```

The value determined by `a*b` is computed and promoted to a default float if it is not one already. The square of that value is stored in `y` and the result of the expression is the reciprocal of that value.

2.7 Statements

The statement sub-language allows for declaration of variables, types and functions and the sequencing of program execution.

Declarations They keyword `var` is used to introduce a new variable that can hold values of some specific type. The basic syntax is:

`var` ⟨*symbol*⟩ :⟨*type*⟩ [= ⟨*expr*⟩];

where ⟨*type*⟩ is a type designator indicating the types of values stored in the variable. The optional initializer clause specifies the value stored in the variable when it comes into scope. The term ⟨*expr*⟩ is an expression that evaluates to a value of the designated type or an arithmetic value that can be converted to that type by default promotion. Scopes and variable lifetimes are discussed later but the basic rule is that the scope of a variable begins at the point of declaration and continues to the end of the lexically enclosing context.

In general, `var` can be followed by a comma-separated list of declarations. For example:

```
var x :integer = 4, y :float = 2*x, z :float;
```

Initializer terms are evaluated left-to-right so the value of `x` in the initializer for `y` is 4. (See discussion in section 2.15.1).

The keyword `const` can be used before or instead of the keyword `var` Such a declaration requires an initializing expression and is an assertion that that variable may not be modified.

```
const var NUM_DIMS :integer = x+y;
```

The keyword `parameter` can be used before or instead of the keyword `var`. A parameter must be bound to a compile-time known value. If a parameter has an initializing expression, that expression will be evaluated at compile time and there will be restrictions on the subset of the language that can be used in that expression.

```
parameter PRECISION = 4;
var x :float(size=PRECISION);
```

The representation width of arithmetic types is an example of a value that must be specified as a parameter.

The keyword `config` can be used before or instead of the keyword `var` for declarations at module scope. This identifies a set of variables whose initial value can be specified via implementation dependent means, such as command line switches or environment variables. Such variables may also have initializers which are evaluated only if no value is specified from the environment.

```
config var logfile :string = "logfile";
```

The `var` is optional here and keyword `const` may also be used. There will also be facilities to read and write configuration data into and out of files. A `config` variable can be also be a `parameter` indicating that its value is determined at compile time under the influence of compilation switches or other aspect of system environment. Examples might be:

```
config parameter target :string = "Mac OS X";
config parameter debug_level :integer = 0;
```

A symbolic name can be associated with a type with the construct

`type` ⟨*symbol*⟩ `:`⟨*type*⟩ [`=` ⟨*expr*⟩]

and then ⟨*symbol*⟩ may be used as synonym for the specified type expression for the balance of the current scope. Again, a comma-separated list of such bindings may follow the `type` keyword. When an ⟨*expr*⟩ is present, that expression is evaluated once and then becomes the default initial value for

variables of the specified type. When ⟨*expr*⟩ is the keyword `nil` then this removes the default value from ⟨*type*⟩ for declarations that refer to ⟨*symbol*⟩, causing such declarations to remain uninitialized.[6]

When the ⟨*type*⟩ term has parameters such as a representation width, those will be bound to default values by this declaration unless they are made explicit parameters. For example:

```
type int(bitsize:integer) = integer(size=(bitsize+7)/8);
```

This defines the symbol `int` as a parameterized type that corresponds to integers where the representation width is described in bits rather than bytes.[7]

The above type declarations simply provide a symbolic name for a type specification. If the `subtype` keyword is used instead of the `type` keyword, then a new type is identified: A value that has subtype ⟨*symbol*⟩ can be used anywhere a value of type ⟨*type*⟩ is needed, but to convert a value of ⟨*type*⟩ to have type ⟨*symbol*⟩ requires an explicit conversion. For example:

```
subtype Index = integer;
var i :Index = 4:Index;
```

Here every `Index` is considered an integer, but to convert integers to `Index` values requires explicit conversion. This relationship is called a *nominal subtype*.

A type can be also specified by the syntax `like <v>` where ⟨*v*⟩ identifies a variable. This expression can equivalently be written as `: <v>.type` where `.type` identifies the type of an arbitrary expression which is not actually evaluated. These constructions can be used in declarations:

```
var x like y;
```

and in type conversions: `(a+b) like y` . The `like` operator has the same precedence as ":".

[6] Let's consider alternatives to `nil` in this context.

[7] Since the `size` symbol for `integer` has `parameter` attribute, we can infer that `bitsize` also has that attribute. We probably do not need to be explicit about this.

Block In Chapel, a ⟨*block*⟩ is a statement or list of statements that forms a separate scope. Within the block, symbols bound to program entities such as variables and types are distinct from other such bindings of the symbol in other parts of the program. Blocks can nest, in which case bindings of symbols in the inner block have precedence over those of outer blocks for references inside the inner block. When "⟨*block*⟩" appears in the syntax descriptions below, it indicates a statement that is in a separate scope.

Assignment Statements Assignment statements combine a variable reference with an expression. The expression is evaluated and the value stored in the variable. Subsequent references to the variable in expressions return the new value until it is redefined, as in this example:

⟨*var*⟩ `=` ⟨*expr*⟩ `;`

Default promotions may be applied if the type of value determined by ⟨*expr*⟩ is not the same as that of ⟨*var*⟩. The semantics of assignment for more complex types will be discussed as they are introduced. In general, ⟨*var*⟩ can be a computation, in which case that computation is generally done after ⟨*expr*⟩ is evaluated.

Given some type t, we say another type t' *conforms* with t if values of type t' can be promoted to type t or assigned to a variable of type t without an explicit conversion operator. A stronger relation is that t' is a *subtype* of t, which means that values of type t' can be stored in variables of type t without conversion.

In addition to simple assignment, Chapel supports the compound assignment operators similar to C adjusted for Chapel's operator set: `+=`, `-=`, `*=`, `/=`, `**=`, `&=`, `|=`, `^=`, and `#=`. These operators evaluate the value expression, the location of a variable, and then read the value from the variable, perform the specified operation and store the value back to the variable. Any computation that can be performed to determine the variable is performed exactly once. The same type promotion rules apply to the value as in simple assignment.[8]

Expression Statements An expression that is evaluated for side-effects against variables can be a statement. It has the following syntax:

[8]If the operator is overloaded, then the corresponding instance of the operator as determined by the types of the operands is invoked. The application does not separately overload `+=` once an overloaded definition of `+` is defined.

[⟨*expr*⟩] `;`

The expression is optional which means a lone semi-column is also considered a statement.

Compound Statements A compound statement is a sequence of statements. These statements are delimited with braces, `{` and `}` and are executed in order. The following compound statement consists of three component statements.

```
{
    var x :integer;
    x = 4;
    y = x;
}
```

When a compound statement is used as a ⟨*block*⟩, then symbols bound by declarations in the statement list are scoped to just this list. The variable `x` above is a variable of default integer type. The scope of this variable is limited to this compound statement and is distinct from all other variables in the program. Variable `y` must be declared in an outer scope.

If-Statements The `if` statement has a guard expression and one or two controlled statements, as shown here:

`if` ⟨*expr*⟩ [`then`] ⟨$block_1$⟩
[`else` ⟨$block_2$⟩]

To evaluate this statement, the guard expression is evaluated and converted to a boolean value if necessary. If the value is `true`, then ⟨$block_1$⟩ is executed, otherwise ⟨$block_2$⟩ is executed, if present. Control then continues with the statement following the `if`. As in many languages, there is a syntactic ambiguity when the ⟨$block_1$⟩ is also an `if`. Chapel follows the common practice of associating a "dangling" `else` with the nearest preceding `if` without an `else`.

Label and Goto Any statement can be prefixed by a label definition of the form:

`label` ⟨*symbol*⟩ ⟨*statement*⟩

This associates ⟨*symbol*⟩ with the control point. That symbol may only be used in a `goto`, `break`, or `continue` statement. Unlike many constructs, `label` does not introduce a scope boundary.

The `goto` statement has this syntax:

`goto` ⟨*symbol*⟩`;`

This transfers control directly to the statement associated with the symbol. The scope of this binding is the innermost containing function definition.

There are restrictions on where branching can occur. In the base language, branching cannot cross a function definition boundary. Additional restrictions will be described later.

Loops, Break, and Continue Two serial loop constructs have the form:

`while` ⟨*expr*⟩ [`do`] ⟨*block*⟩

and

`repeat` ⟨*block*⟩ `until` ⟨*expr*⟩`;`

In both forms, ⟨*expr*⟩ evaluates to a boolean value which determines whether ⟨*block*⟩ is executed. In the first form when ⟨*expr*⟩ evaluates to `false`, control continues with the statement that follows the loop. In the second form, ⟨*block*⟩ is executed once unconditionally, and then when ⟨*expr*⟩ evaluates to `true`, control continues with the statement that follows the loop.

Another loop form will be introduced after we have introduced sequences in Section 5 and its parallel variant in Section 9.

The statement `break` transfers control immediately to the same point as when ⟨*expr*⟩ evaluates to `false` of the innermost containing loop. Similarly, the statement `continue` transfers control immediately to the point just before evaluation of ⟨*expr*⟩ of the innermost containing loop. Both `break` and `continue` can be followed by a ⟨*symbol*⟩ that is a label prefixing and enclosing loop. In this case the interpretation is to exit all intermediate loops and then exit or continue the identified loop, as in this example:

```
label outer
    while (⟨e0⟩) {
        ⟨statement0⟩
        label inner
            repeat
                ⟨statement2⟩
                if(⟨t0⟩) break;
                if(⟨t1⟩) continue outer;
            until (⟨e1⟩)
        ⟨statement1⟩
    }
⟨statement2⟩
```

If $\langle t_0 \rangle$ evaluates to true, then control exits the innermost loop and continues with $\langle statement_1 \rangle$. The "`break;`" could be also be written as "`break inner;`". If $\langle t_1 \rangle$ evaluates to true, then control returns to the next iteration of the outer loop and evaluates $\langle e_0 \rangle$ to determine if execution continues with the loop body or $\langle statement_2 \rangle$.

Select Statement The select statement provides a multi-way variant of the `if` statement. Here is the syntax:

```
select ⟨expr⟩ {
    ⟨case⟩ [ ...]
}
```

where each element $\langle case \rangle$ has the form:

`when` $\langle case\ values \rangle$ [`,` ...] [`do`] $\langle statement \rangle$ [...]

or

`otherwise` $\langle statement \rangle$ [...]

Let t denote the type of the expression $\langle expr \rangle$. The term $\langle case\ values \rangle$ is a common separated list of expressions that can be compared to a value of type t using operator "`==`". The first case clause that contains an expression where that comparison is `true` will be selected and control transferred to the associated list of statements. For both `when` and `otherwise` clauses, the

associated list of statements is considered as separate ⟨*block*⟩ and hence a scope. After evaluation of this list of statements is complete, control continues with the statement following the `select` statement unless changed by explicit branch. Each list of statements is a separate scope for variable declarations.

2.8 Functions

Definitions and Invocation As in most languages, Chapel allows the action of a sequence of statements to be abstracted into a *function*. This is the basic definition:

`function` ⟨*symbol*⟩ [`(`⟨*param*⟩ [`,` ...]`)`] [`:`⟨*type*⟩]
⟨*block*⟩

The term ⟨*symbol*⟩ is the name of the function. The term ⟨*type*⟩ describes the type of values returned by the function. ⟨*block*⟩ describes the actions of the function and is called the *body* of the function. When the return type is omitted, the function can be evaluated only for side-effects and not return any value.

Function definitions may appear at the top level of the program or may be nested inside any ⟨*compound statement*⟩. The association of this function with ⟨*symbol*⟩ follows the same basic scoping rules as variable declarations accept that the binding holds for the entire enclosing context rather than from the point of declaration to the end of that scope. Variable references inside a function definition and not bound to local declarations are *free variables* and are resolved against enclosing scopes statically.

The parameter list is an optional, comma-separated list of variable declarations of the form:

[⟨*intent*⟩] ⟨*symbol*⟩ `:`⟨*type*⟩ [`=` ⟨*expr*⟩]

Each of these symbols is called a *formal argument* of the function. Here ⟨*intent*⟩ is one of `in`, `out`, `inout`, or `const` and if omitted defaults to `const`. The symbols defined on this list are treated as variable declarations in a ⟨*compound statement*⟩ and their scope is the entire function definition.

A function *invocation* is a expression where the name of a function is mentioned followed by a parenthesized list of *actual arguments*, as follows:

⟨*fun expr*⟩`(`⟨*actual param*⟩ [`, ...`]`)`

Generally, ⟨*fun expr*⟩ is an expression that evaluates to a function value. A function that has no arguments may be invoked simply by using its name or with a empty list of actual parameters, such as:

```
function next_prime { ...  };
p1 = next_prime;
p2 = next_prime();
```

Both statements invoke function `next_prime` and store the result in a variable.

In the common case, each ⟨*actual param*⟩ is an expression or a variable. When the corresponding formal argument does not have intent `out`, then the argument is evaluated to a value and arithmetic promotions performed if the type of the expression differs from the type of the formal. The values are assigned to new instances of the formal arguments and control is transfered to the body of the function.

An intent attribute of `out` and `inout` indicates the value in the formal argument variable are copied back to the actual arguments when control transfers back to the invoking context. This implies the actual arguments must be variables of suitable type.

The intent value `const` is similar to `in` but further prohibits assignments to the associated variable.

Actual arguments can also be specified using a keyword notation. For example:

```
f(x=4, y=10)
```

where `x` and `y` are the names of formal arguments of the function associated with symbol `f`. Keyword-style binding allows argument values to be specified and evaluated in any order. Positional and keyword arguments can be mixed. The list of formal arguments not bound by keywords and respecting their original order are bound to the positional arguments.

A function can return a reference to a variable. This is indicated by using the syntax `ref(`⟨*type*⟩`)` to indicate a variable of the indicated type is returned. For example:

```
function f(...)  :ref(integer) ...
```

The function `f` returns a reference to an integer. The value of a function invocation would normally be the value stored in that variable, thus `f(...)+1` would evaluate to one added to the value in the variable returned by the function. Such a function may also be used as the target of an assignment or the right operand of a reference assignment, as follows:

```
– store 4 into the variable returned by f
f(...)  = 4;
```

Default values In addition to intent and type, a formal argument may specify a default value, as follows:

```
function f(x :integer = 4, z:integer) ...
```

Here 4 is an example of a default value. If no value is specified for `x` when the function is invoked, then the value 4 is used. When an argument following a default value does not have a default value, keyword bindings may be needed to use the default value. Thus `f(z=3)` is a valid invocation while `f(3)` is the same as `f(x=3)` and has too few arguments.

The default value for an `out` parameter is always evaluated and becomes the initial value of the new instance of the formal argument. In general, the default value can reference symbols from the enclosing scope, literal constants, and previously declared arguments. This expression is evaluated at the time the function is invoked. When there is no actual argument corresponding to a formal with intent `out` or `inout`, we discard the value of the corresponding formal at the end of execution of the function. For example, consider a hash function that returns a reference to an integer and also returns an optional `out` value that indicates if a new entry was added to the hash table:

```
function hash(key :float, out new :boole) :ref(integer)...
hash(key,new) = value
if(new) items += 1;
...  hash(key)
```

The first invocation tests the output value to keep a count of new items while the second call discards this value without naming it.

Function Values There are situations when a function reference does not imply an invocation. Special syntax indicates that a function value should be captured rather than evaluating the function. This is done by using "`=>`" instead of "`=`" in an assignment statement. In these cases, some of the arguments to the function may be omitted and any of the arguments may be the special token "`_`". This construct builds a new function by capturing the values and variables in the specified arguments. The new function has a reduced type signature corresponding to the unspecified values. An example is:

```
function f(a :integer, b:integer) { ...  }
bar => f(x+y);
...  bar(z) ...
```

Here the reference to `f` is not evaluated but expression `x+y` is evaluated and the resulting value captured. The result is a new unnamed function of one parameter which we store in variable `bar`. The subsequent reference to `bar` provides the missing second argument at which point we eventually invoke function `f` with the two values. This action is frequently called *currying*. Such values may be stored in variables and passed to functions as an argument.

To retain an optional argument when a function is curried, the formal name must be bound to "`_`" either positionally or by keyword. For example:

```
function f(x :integer, y :integer =4) { ...}
...  => f(y=_,3) ...
```

Here we curry a function `f` of two arguments yielding a function with a single integer argument named `y` with default value 4. If instead we had simply written:

```
...  => f(3) ...
```

then this would bind the value of `y` to its default value, 4, and yield a function of zero arguments. The type of the resulting function value is a reduced form of the original function type where parameters whose values are bound are omitted.

Function Types A *function prototype* is specified by the information in a function declaration excluding the body. The relevant information is the number, names, intents, and types of the arguments and any default value information, and the return type. The reduced information that excludes the names and default value information is called the *type signature*. As much as the full type and as little as the type signature can be specified for a function type. When only the function signature is specified, the ability to use keyword arguments and default and optional values is lost.

A *function type* can be declared by using a function prototype but omitting the function name. For example:

```
type IntFun :function(:integer, y:boole =true) :integer;
function bar(x:integer, z:boole) :integer {...}
var f :IntFun => bar;
    ...  f(-5)
    ...  f(3, y=false)
```

The first line defines a function type where: the function has two arguments; the second argument is named `y` and has default value `true`; and the function returns an integer. The next line defines a function `bar` and the third line defines a variable `f` that has function type that is bound to the function `bar`. Variables with function types are references to function values and are defined by the binding operator "`=>`". The fourth line is an invocation of function `f` which is equivalent to an invocation `bar(-5,true)`. The last line illustrates that it is the formal names from the function type that are used for argument binding rather than the names from the function.

When we bind a function value to a function variable like `f`, we require that the type signature of the function value agree with the type signature of the type of the variable. Argument names and default values are based on the type of the variable and override those in the value. In the example above, `f` has a default value for its second argument which is not the case for `bar`.

Return Statement A statement of the form:

`return` [⟨*expr*⟩]`;`

terminates the innermost function body and immediately returns the value determined by ⟨*expr*⟩. Control returns to the calling context where that

function was invoked. A function might not return a value. Rather it could execute for side-effects to other variables. This is indicated by omitting the return type and then omitting expressions from `return` statements.

When a function returns a value of reference type, it is permitted but not required to indicate this by adding operator "`=>`" after `return` and before the ⟨*expr*⟩.

Call Statement A statement whose only effect is to invoke a function is a `call` statement. This is the syntax:

[`call`] ⟨*fun expr*⟩ `;`

In this example ⟨*fun expr*⟩ is a function invocation. The keyword `call` is optional. Any value returned by this function is discarded.

2.9 Tuple Types

A tuple value is constructed as a comma-separated list of expressions inside parentheses[9] and a tuple type designator is an analogous list of types. For example

```
var pair :(integer,float);
pair = (3,4.0);
```

or

```
type PairType = (integer,float);
var p :PairType = (5,6);
```

Assignment between tuples is done positionally. In the first of these examples, 3 is stored to the first field of `pair` and 4.0 is stored to the second field. Default promotions may apply for primitive types. In this example, the integer value 6 will be promoted to a floating point value when assigned to the second component of `p`. Functions may have tuple types as arguments and return values.

The values in a tuple may be accessed by destructuring into a group of variables. This can be done as an initializer, by assignment, or by argument passing. For example:

[9]Parentheses are required to continue to use comma as a list separator in variable declarations, formal argument lists, and and argument lists.

```
var (i,j):PairType = pair;
var (i:integer, j:integer) = pair;
(i,j) = pair;
function f( (i:integer, j:integer)) ...
call f(pair);
call f( (i+1,j) );  – parens to form a tuple and
                    – to invoke the function
```

In these situations, the special token "_" can be used to discard some values that are not needed. For example:

```
var (i:integer, _:integer) = pair;
(i,_) = pair;
function f( (_:integer, j:integer)) ...
call f(pair);
```

Additional syntax is provide to support tuples whose components are all the same type. For example:

```
var index :  ⟨k⟩ * integer;
```

Here $\langle k \rangle$ is an integer `parameter` expression, so the value is known at program specification time just like the representation width for arithmetic types. Variable `index` has type of a tuple with $\langle k \rangle$ components all of which are integers. The type `2 * integer` is identical to `(integer,integer)`.

A variable of tuple type many be used in an expression that looks like a function invocation. If `t` has tuple type than `t(1)` is interpreted as the first component of that tuple. There is a restriction that if the types of the components of `t` are different, then the "argument" to this expression must be a `parameter` and hence known before execution of the program. When `t`'s components are all the same type, the argument may be any expression resulting in a value that conforms with `integer`, and whose value is in the range of 1 to the number of components in the tuple.

2.10 Modules

Chapel supports a simple module structure to manage the space of names. Every name is logically part of some module where the default module is

named `main`. Module names consists of a sequence of symbol separated by "." operators. These are denoted as ⟨*module name*⟩ below.

The `module` statement identifies the name of the current module. It has the form:

```
module ⟨module name⟩ [version=⟨string⟩] {
    module-level definitions
}
```

where *module-level definitions* is a sequence of type, variable, and function definitions whose names are considered part of the module. The same module name might appear on multiple module statements.

Names in a particular module can be explicitly referenced in other modules by prefixing them with their module name. For example, if `libmsl.blas` is a module name including a function named `saxpy`, then that function may be referenced as `libmsl.blas.saxpy` anywhere in the program. We will use ⟨*name*⟩ to indicate a ⟨*symbol*⟩ or, recursively, term of the form ⟨*name*⟩.⟨*symbol*⟩ where ⟨*name*⟩ identifies a module and ⟨*symbol*⟩ is defined in that module.

For Chapel implementations that separate the input program into multiple files, we expect the name of the file will encode the module and version information for the symbol defined if the first statement in the file is not a `module` statement. We assume file names are divided into two or three component names where the last is ignored. The first identifies the module and the second of three identifies the version. For example, assuming that "." is used as in UNIX as a file name component separator, the file named `string.chp` would contain names for module `string` and `string.dec04.chp` would contain names for module `string` with `version` equal to `"dec04"`.

The `use` statement is used to incorporate names from another module into the current scope. It has this syntax:

```
use ⟨module name⟩[version=⟨string⟩]
        [[only] ⟨rename list⟩]
        [except ⟨symbol⟩ [, ...]];
```

where the `version` clause optionally selects a particular version. The ⟨*string*⟩ term must be a `parameter` expression. The ⟨*rename list*⟩ is a possibly empty, comma-separated list of pairs of the form:

[⟨*symbol*$_1$⟩ `=>`] ⟨*symbol*$_2$⟩

where ⟨*symbol*$_2$⟩ is a name in the specified module and ⟨*symbol*$_1$⟩ is the name that will be used to refer to that symbol in the local context. If the optional `only` keyword is used, then only symbols on the rename list are imported and in this case items on the rename list may be just symbols rather than pairs. This list identifies symbols included in the current scope while other symbols in the module are not so included. For example:

```
use libmsl; – all of libmsl
use blas only saxpy, inner_product => sdot;
```

Here we add to the current scope all of the symbols defined in module `libmsl` which happens to include the module name `blas`. Note that the module name `blas` will shadow any top-level module with the same name. From that second module we add symbol `saxpy` to the current scope and symbol `sdot` is added under the alias `inner_product`. The `except` clause specifies a list of symbols defined in the module that are not added to the current scope.

It is an error for a symbol to be defined both by a `use` statement and by a declaration in the same scope.

A `use` statement may be used inside of function definitions when that is convenient, as in this example:

```
function f ...  {
    use libmsl.blas;
    ...  sdot ...
}
```

Here, the symbols from `libmsl.blas` are added to the scope of the innermost compound statement.

We say as symbol is *exported* if it would be included in another scope by a `use` statement. By default, all symbols in a module are exported but a separate statement can be used to identify those symbols that are included by a `use` statement. It has syntax:

```
public ⟨symbol⟩ [, ...];
```

The `public` keyword can be used as a modifier on any top level declaration for types, variables or functions and has the same effect as adding the defined symbols to a `public` list. We say such symbols are *exported* from the scope. When any symbol is explicitly exported, other symbols are by default not exported.

A `public` modifier may precede a `use` statement as well. In this case, a symbols introduced by the `use` statement are included as exported symbols of the current module. For example, if module `libmsl` included:

```
public use blas;
```

where `blas` is a module then all symbols exported by `blas` would be included by any `use libmsl;` statement.

Symbols not exported can still be referenced via an explicit module name prefix.[10]

When a module statement is nested inside another module, public symbols from the inner modules are implicitly exported to the outer module. A `public` keyword on the inner `module` statement indicates symbols exported from the inner module are also exported by the outer module. For example:

```
module libmsl {
    public module blas {
        public function sdot ...
```

Symbol `sdot` can be used inside of `libmsl` as if it were defined in that context because of the `public` keyword on its declaration. Further, the that symbol is exported from `libmsl` as well because of the `public` keyword on the `module blas` statement.

2.11 The `with` Statement

The `with` statement is syntactically similar to the `use` statement. Where the `use` statement simply changes the scoping of symbols, the `with` statement

[10] There is no strongly enforced notion of "private" in Chapel because of the interest in exploratory programming. Under suitable compilation switches we might disallow references to non-exported symbols.

indicates that the contents of the source module are logically copied into the current scope. Symbols that are defined in the module but also defined in the current scope are replaced with those of the current scope. This allows a variant of one module to be constructed where substitutions are made to either implementation or interface. Definitions of symbols not imported into the current scope are still copied but those symbols must be referred to with fully qualified names. An example:

```
module MyLib version="debug" {
    var debug_level :integer = 2;
    public with MyLib;
}
```

This builds a variant of `MyLib` where the symbol `debug_level` is redefined from its definition in `MyLib` which might be:

```
parameter debug_level:integer = 0;
```

There are no restrictions on how symbols may change when they are shadowed. Variables could change to 0-argument functions that return a reference, types could change, and attributes of variables could change.

2.12 Conditional Declarations

When an `if` statement appears at module scope, then its boolean guard must be an expression constructed from `parameter`'s. This expression is evaluated once and determines whether declarations from the `then` clause or, if present, the `else` clause are included. For example:

```
config parameter target :string = "";
use libmsl(version=target);
if target == "Cray X1" then {
    parameter cray_fft :boole = true;
    use cray_fftlib;
}
else {
    parameter cray_fft :boole = false;
    use fftlib;
}
```

Setting the `target` variable to the string `"Cray X1"` affects the version of `libmsl` that is used as well as selecting a different variant of an FFT package and the binding of the `cray_fft` flag. Note that the "compound statements" used at the module level do not introduce a new scope. They are simply used to group declarations under the control of a compile-time decision.a

2.13 Program Execution

Program execution begins by processing the module scope declarations in the `main` module. We say we "execute a declaration" to mean that we evaluate any initializer for a variable or default values for types and functions.[11] Initializers for configuration variables are only evaluated if values for those variables are not set by some other implementation dependent means.

When a `use` statement is "executed", we execute the declarations for the identified module unless they have already been executed. Note that all declarations in the module are executed regardless of the scoping clauses on the `use` statement. After all the declarations in a module have been executed, if that module defines a function `initialize` that has no arguments, then that function is invoked.

After execution of the declarations for module `main`, if that module defines a function without arguments called `main`, that function is invoked. Upon return of that function, the program terminates. As part of the termination process, for each module whose declarations were evaluated, if that module defines a function `finalize` without arguments, than that function is invoked. These functions are executed in the reverse order in which the module declarations where executed.

A predefined variable is used to capture additional arguments to the execution of a program. It has this declaration:

```
const config var argv :  seq of string = nil;
```

2.14 Summary

Figure 3 summarizes all of the statements in Chapel including some not yet introduced.

[11] More actions are taken for arrays and domains introduced in Section 8.

```
enum ⟨symbol⟩ { ⟨symbol⟩ [, ...] }
record ⟨symbol⟩ { ⟨definition⟩ [, ...] }
class ⟨symbol⟩ { ⟨definition⟩ [, ...] }
union ⟨symbol⟩ { ⟨definition⟩ [, ...] }
type ⟨symbol⟩ = ⟨type⟩;
subtype ⟨symbol⟩ = ⟨type⟩;
var ⟨symbol⟩ [:⟨type⟩] [= ⟨expr⟩];
⟨variable⟩ = ⟨expr⟩;
[call] ⟨expr⟩;
if ⟨expr⟩ [then] ⟨block1⟩ [else ⟨block2⟩];
while ⟨expr⟩ ⟨block⟩
repeat ⟨block⟩ until⟨expr⟩;
for ⟨variable⟩ in ⟨expr⟩ ⟨block⟩
forall ⟨variable⟩ in ⟨expr⟩ ⟨block⟩
cobegin { ⟨stmt⟩ [ ...] }
begin ⟨block⟩
serial ⟨expr⟩ ⟨block⟩
atomic ⟨block⟩
⟨symbol⟩:  ⟨stmt⟩
goto ⟨symbol⟩;
break [⟨symbol⟩];
continue [⟨symbol⟩];
return [⟨expr⟩];
yield [⟨expr⟩];
select(⟨expr⟩) { ⟨when clause⟩ [ ...] }
typeselect(⟨expr⟩) { ⟨when clause⟩ [ ...] }
module ⟨name⟩ [version ⟨string⟩];
{ ⟨stmt⟩ [ ...] }
function ⟨name⟩ [(⟨parameters⟩)] [:⟨type⟩] ⟨block⟩;
iterator ⟨name⟩ [(⟨parameters⟩)] [:⟨type⟩] ⟨block⟩;
constructor ⟨name⟩ [(⟨parameters⟩)] [:⟨type⟩] ⟨block⟩;
module ⟨name⟩ [version=⟨string⟩] { ...  }
use ⟨name⟩  [[only] ⟨rename list⟩] [except ⟨symbol⟩ [, ...]];
with ⟨name⟩ [[only] ⟨rename list⟩] [except ⟨symbol⟩ [, ...]];
```

Figure 3: Statement Summary

2.15 Base Language Notes

This section has discussion about open issues from above.

2.15.1 Factoring Type Specifications

The inability to factor complex types across a collection of variables is a concern. Later when types are expected to element types and some kind of additional type constructor, a separated type designator for a collection of variables may become onerous.

The ability to simply reuse a type specification can be accommodated either by creating a type name or by handling a `typeof` operator or possibly a `like` attribute.

```
var a :  ⟨some long messy type⟩;
var b like a;
var c :  a.type;
```

David: the current choice reflects the belief that many types can be inferred and that we expect many variable declarations to have initializers, which either requires one type per declaration or multiple declarations.

2.15.2 Parameter Intents and Type Inference

We have outstanding questions about the interactions of parameter intents with both reference variables and arrays. There are also some issues regarding the interpretation of `const` in those two cases.

Brad asks the question, given this function declaration:

```
function f(x) { ...  x = 3; ...  }
```

where we have a formal parameter `x` with no specified intent but to which we assign. If the rule is that parameters are by default `const`, then is this program illegal or should we assume it is only legal when `x` is an array passed by reference or when `x` has reference type?

David's position begins with these broad ideas:

1. The purpose of type variables is to allow program fragments to be reused in many contexts by reinterpreting operations in the context of specific type instantiations.

2. The purpose of intent attributes are to allow the writer of a function to define how values are communicated to the caller.

3. The purpose of `const` is to aide the reader in understanding intended usage as well as assist the client of a function who may only see the function prototype in documentation.

4. The purpose of default `const` intent is to make sure the function writer declares his intent when a parameter is modified in some way. This is not something that should depend on context or type instantiation.

5. The purpose of reference variables is to allow the choice of variable to define to be computed. In particular it allows functions to return *lvalue's* but also allows interacts with having "views" into an array.

From these I would suggest:

1. We never infer parameter intents and further a `const` parameter regardless of type may not be the subject of assignment, either value or reference.

2. We only infer a reference type if we see an explicit "`=>`" operator. I think will help users better understand what's going on.

2.15.3 Evaluation of default values

The text above states that expressions that define default values for function arguments are evaluated at the point of function invocation. This implies that we must build little function closures that get invoked at that point. Is this overkill? Should we say instead expressions for default values are evaluated when the `function` definition is processed? The difference is that that the definition as written allows default values for some arguments to depend on the actual values of other arguments.

A different approach is to not allow default values but rather use overloaded function to simulate them. Yet another approach might be to require that parameters with default values be bound by keyword and then apply overloading to only the positional parameters.

Another restriction might be to require that functions with the same name must be distinguishable based on arguments without default values.

2.15.4 Marking output parameters

3 Structured Types

Chapel supports two different kinds of structured type, records and classes. These two types are very similar in their specification and the various tools available to derive new types. However, they differ in their semantics with respect to assignment and argument passing.

Record types are used to create groups of variables where values can be read and written to them as a group rather than as individuals. *Class* types are used to represent program objects that have some notion of identity. These objects are manipulated by reference where records are manipulated by copying the values.

3.1 Requirements

Derived types are important both to simply aggregate related sets of values, associate operators and functions with such as set to build abstract data types, and to provide the basis of the object-oriented features. Chapel has additional goals to facilitate code reuse both by the common method of specialization of behavior and by structured substitution of definitions of variables and methods. This second technique is motivated by the desire to support *structural typing* of function argument which means that any binding of those variables is valid as long as all fields, methods, and invoked functions are type correct.

We therefore have the following requirements:

1. Support for describing groups of variables with value semantics where the group also has value semantics.

2. Support for describing groups of variables where the group has reference semantics (objects).

3. Support for construction of new descriptions by structured extension and redefinition of existing elements.

4. Support for associating a function with such a description.

5. Support for associating a function with the type of an object to support dynamic polymorphism.

6. Support for structural typing of function arguments and return values.

Here a "description of a group of variables" is also called a *derived type*.

Of course, we desire as much syntactic similarity between the value and object types as possible.

3.2 Record Types

A record type is declared using the syntax:

```
record ⟨symbol⟩ {
    [⟨base records⟩]
    [⟨definitions⟩ [...]]
}
```

The definitions in the body of the record are types, variables and functions using the same syntax as already presented, although the interpretation of initial expressions and function definitions are different. These are referred to as *fields* of the record. The term ⟨*symbol*⟩ is the name of the record and it becomes a type designator following the same scoping rules as for functions. The term ⟨*base records*⟩ will be describe later. Here is a simple example:

```
record ClientData {
    var name :string;
    var id :integer;
}
```

A record is fundamentally just a collection of variables that can be manipulated as a group. Individual elements within a record are accessed by using the "." operator that combines a reference to a record variable and a name of a variable defined within that record. For example:

```
var c :ClientData;  – declare a client variable
c.name = "Fred";    – define the name field
...  c.id ...       – reference the value in the id field
```

Assignment of record values to variables of record type is done *by name*. For example:

```
var c2 :OtherClientData;
c = c2
```

is equivalent to to

```
c.name = c2.name;
c.id = c2.id;
```

If type `OtherClientData` does not have one of these fields then this is an error. Any fields in the target with the `const` attribute are not effected by this assignment and need not be in the source value.

This interpretation applies to argument passing when a record variable is used as an actual argument. If the intent of the argument is `inout`, then any fields not present in the type of the formal argument are simply not modified by the function call. Otherwise, the behavior is just like simple assignment. Values of record type may be returned from functions.

3.3 Bound Functions

A *bound function* is a function definition that is associated with some more record type. This is the syntax:

`function` ⟨*symbol*$_1$⟩`.`⟨*symbol*$_2$⟩
`(`⟨*argument list*⟩`)` [`:`⟨*type*⟩]
`{ ... }`

⟨*symbol*$_1$⟩ identifies a record type while ⟨*symbol*$_2$⟩ is the name of the function. The balance of the function definition is like other functions. Free variables from default argument value expressions or the body of the function are resolved against field names of the record type. For example:

```
function ClientData.set_id(newid :integer) {
    id = newid;
}
```

Since `id` is a field of type `ClientData`, the reference to `id` inside the body of the function is bound to that field name

A bound function is invoked by providing both the actual arguments and an instance of the record type. The latter is specified as the left operand of a "`.`" operator where the right operand is the function name. For example:

```
var c :ClientData;
c.set_id(4);
```

The instance of the record is passed *by reference* so inside the invocation of `set_id`, the symbol `id` refers to the actual variable within `c` rather than a copy.

A bound function may be invoked with a different record type but only after that type is appropriately converted. For example:

```
var c2 :OtherClientData;
c2:ClientData.set_id(5);
```

This is a valid expression as long as every field of `ClientData` appears as a field in `OtherClientData` and all of the operations applied to fields inside the function are valid for types of those fields. For example, the `id` field could have different representation widths in the two classes.

A function definition that is syntactically placed inside a record is bound to that record. For example:

```
record ClientData {
    var name :string;
    var id :integer;
    function set_id(newid :integer) {
        id = newid;
    }
}
```

Here `set_id` is bound to record type `ClientData` just as in the previous example. Functions defined inside of records are called *primary methods* while those outside are called *secondary methods*. This distinction only effects subtyping issues described below.

Within a bound function, the reserved word `this` can be used to reference the instance of the record type used to invoke the function.

Functions can be bound to other types besides record types. For example, they could be bound to primitive types or enumerated types. In this case `this` is a reference to a variable holding the value with which the function is invoked. When applied to an expression, a temporary variable is created to hold the transient value.

3.4 Constructors

A constructor is a special kind of function that is used to initialize a new instance of a record. The syntax is similar to a bound function but uses the keyword `constructor` rather than `function` and has no return value. For example:

```
constructor ClientData.create(s :string, i :integer) {
    name = s;
    id = i;
}
```

Since the purpose of a constructor is to create a new instance of the type, it is invoked using the type name rather an instance variable. The new record instance can still be referenced as `this` and is the return value of the constructor. For example:

```
var cr :ClientData = ClientData.create(cname, cid);
```

A constructor may be specified inside a record declaration for convenience. Thus the above definition of `ClientData.create` is exactly the same as this definition:

```
record ClientData {
    var name :string;
    var id :integer;
    constructor create(s :string, i :integer) {
        name = s;
        id = i;
    }
}
```

When initializers are specified for fields in a record, those are evaluated in a *default constructor*. The name of the type can be used to explicitly invoke the default constructor. All of the field names become arguments to the default constructor and initializer terms are simply default values for the corresponding arguments. Thus the default constructor for the above class has this signature:

```
constructor ClientData(name:strings, id:integer);
```

and might be used

```
var c :ClientData = ClientData(name="fred", id=4);
```

or

```
var c :ClientData = ClientData(id=4,name="fred");
```

There is no predefined order for default constructor arguments so they must be specified by name. A bound function with the name `initialize` will be invoked instead of the default constructor if it is defined. When such a function is defined, arguments may be specified by position as well as by name. For example:

```
constructor ClientData.initialize(n:string, i:id) {
    name=n;
    id=i;
}
...  ClientData("sally", 21) ...
```

The last line invokes the `initialize` constructor and with its specified argument order.

Constructors are special kinds of functions but may be curried like normal function so the expression `ClientData("fred")` creates a new function value that takes a single integer argument and returns value of type `ClientData`.

3.5 Anonymous Records

A modified form of the tuple syntax is used to create values of anonymous records. The syntax has the form:

(⟨*symbol*⟩ = ⟨*expr*⟩ [, ...])

This creates an anonymous record value with field names specified by the symbols whose types and values are derived from the corresponding ⟨*expr*⟩. For example:

```
var c :ClientData;
c = (id = 4, name = "fred");
```

Note that assignment is *by name* so the order of the elements in such a list is immaterial.

For type contexts, the name of a record may be omitted when defining anonymous record types. For example:

```
var c :ClientData;
var c2 :record {
    var name :string;
    var id :integer; };
c2 = (name = "fred", id = 4);
c = c2;
var c3 :record {
    var id :integer = 5;
    var name :string = 'sally'; };
```

The types of `c2` and `c3` are anonymous records that are structurally the same as `ClientData`. Initial values are permitted for anonymous records as illustrated in declaration of `c3` above.

The operator "`+`" is overloaded for record types and is used to combine two records. For example:

```
record Name{
    var name :string;
    var age :integer;
}
function get_name(x :T) :Name {...}
c2 = get_name(t) + (id = 4);
```

Here the record determined by the right hand side consists of three fields, `name`, `age`, and `id`. Two of these, `name` and `id` are stored into the left hand side, `c2`. For operator "`+`" we have the requirement that no symbol be defined by both operands.

3.6 Derived Record Types

The term ⟨*base records*⟩ has the form analogous to that used for copying modules:

```
with ⟨name⟩ [[only] [⟨rename list⟩]]
             [except ⟨symbol⟩ [, ...]]
```

where each ⟨*name*⟩ identifies another record type. This type is referred to as a *base type* where the type in which this is used is called the *derived type*. The interpretation of this clause is that all of the fields including primary methods of the base types are also fields of the derived type as if they had been textually included in the derived type. Secondary methods defined outside of the base record are not included. For example:

```
record OtherClientData {
    with ClientData ;
    var address :Address;
}
```

which is semantically equivalent to:

```
record OtherClientData {
    var name :string;
    var id :integer;
    var address :Address;
}
```

where we have simply included the fields from the (first) definition of `ClientData` above.

It is possible for a field defined in a base class to be replaced by subsequent definition. For example:

```
record DifferentClientData {
    with ClientData ;
    var id :(integer, integer);
    var address :Address;
}
```

The `id` field from `ClientData` is replaced by a new definition and effectively omitted from the derived type definition. This is therefore equivalent to:

```
record DifferentClientData {
    var name :string;
    var id :(integer, integer);
    var address :Address;
}
```

In the presence of multiple base types, later types in the specification generally override earlier definitions. For example:

```
record C { with A; with B; ...}
```

For any fields defined in both `A` and `B` the ones in `A` are shadowed and the ones from `B` survive into the final definitions.

When a different behavior is desired, a rename clause can be used which has the syntax:

[[`only`] [⟨*rename list*⟩]] [`except` ⟨*symbol*⟩ [`,` ...]]

where ⟨$symbol_2$⟩ is a field in the base class and ⟨$symbol_1$⟩ becomes an alias for it in the derived scope. The optional `except` clause can also be used exclude some symbols from the identified record.

When a base record includes a function definition, that definition can have free variables. Those free variables are resolved against the fields of the derived class and so are subject to the affects of renaming and shadowing. This is the same thing that happens to bound functions declared outside of the class when they are invoked with an instance of the derived class. Examples of this will be provided later after we introduce support for generic programming in Section 7.

This behavior is modified by the presence of a `private` attribute on a field or type name. Symbols with this attribute are not directly added as symbols in the derived type. They can be referenced by explicitly prefixing the name with its containing type. For example, if `x` is a private field within base type `A`, it can be referenced as `A.x` in the derived class unless explicitly renamed. All references to private symbols in initializers and function definitions in the base class are implicitly rewritten to use the `A.x` form unless explicitly renamed or excluded.

3.7 Class Types

A class type is syntactically similar to a record type but have different assignment semantics. This is the syntax for a class definition:

```
class ⟨symbol⟩ {
    ⟨base classes⟩
    [⟨definitions⟩ [...]]
}
```

Again, ⟨*symbol*⟩ becomes the name of the type. The term ⟨*base classes*⟩ differs from the previous ⟨*base records*⟩ only in that it permits names of class types as well as record types.

The fundamental difference between classes and all previously described types is that assignment, including argument passing, is *by reference* rather than *by value*. A variable of class type stores a reference to an instance of the class and it is that reference which is copied rather than the object itself. This allows sharing of groups of variables, which now has an object identify rather than just value.

```
class Client {
    with ClientData ;
    var backup :Client;
}
var c :Client = ...,
    b :Client = c;
```

This type includes the same fields as the record `ClientData` but the initialization of `b` creates an alias so that `b.backup` and `c.backup` are aliases for the same variable.

Constructors for classes create new instances of the class and return a reference to that instance rather than a copy of the values.

Variables of class type may not be configuration variables.

If there is an assignment between a class and record types then that assignment is implemented by name as in the case of records.

3.8 Derived Classes

For class declarations, the `with` statement may be used to construct derived classes as for derived record types. For classes, the keyword `implements` can be used instead of the keyword `with` to specify a derived class type as well. For example:

```
class A { with ClientData; ...  }
class B {
    implements A;
    ...
}
```

Here `B` has all of the fields of `A` just as in the case of a `with` clause. The same rules and options apply for name management as for the `with` statement. The difference between these two clauses is that when `implements` is used the derived type is a nominal subtype of the base type. This is similar to the distinction made for the `subtype` declaration (page 22). The importance of this distinction is discussed in Section 6 where function overloading is described.

A variable that is declared to hold a reference to a class may in fact hold a reference to any nominal subtype of that class. Thus we may have:

```
class MyClient {
    implements Client;
    var index :integer
}
var c:Client = MyClient(...);
```

Class `MyClient` is a subtype of `Client` so it is legal to assign a reference to an instance of `MyClient` to a variable whose declared type is a nominal supertype of `MyClient`

3.9 The `use` Statement

The `use` statement is used to give access to the fields in a record or class in an executable context. This is the syntax:

`use` ⟨*expr*⟩ [[`only` ⟨*rename list*⟩]]
[`except` ⟨*symbol*⟩ [`,` ...]]

where ⟨*expr*⟩ evaluates to an instance of a class or record type. In this case the symbols of the record are incorporated into the enclosing scope. For example:

```
var c :ClientData;
use c;
id = 4;
name = "fred";
```

Here, the symbol `id` refers to `c.id` for the balance of the scope.

It is an error for a symbol to be defined both by a `use` statement and by a declaration in the same scope.

3.10 Nested Type Definitions

Chapel permits classes to be defined within other classes. For example, an abstraction that holds a list of integers might look like the code in Figure 4. The definition of `remove` is bound to an instance of `List.Elt` and free variables in the function body are first resolved against the fields in `Elt` and against the fields in `List`.

Outer class instances can be used by combining the type name with the reserved word `this`. In this example, the reference to `head` in `remove` could be more explicitly accessed as `this.List.head`.[12]

3.11 Variable Sharing

When an class instance is constructed, there is an additional by-name mechanism available that permits variable fields inside the class to be aliases to variables external to the class. This uses the "`=>`" operator instead of the "`=`". For example:

```
var myname :string = "harry";
var c :Client = Client(name=>myname);
myname = "foo";
```

[12]Implementation note: by default I'm assuming that an instance of an inner class will probably have an explicit reference to the associated outer class instance. While we might optimize this away, I think in general it is syntactically awkward and error prone to try and let the user manage this relationship. If they are bothered by the memory cost, then they can move the inner class outside and make the outer reference explicit and use the `with` statement to convenient access outer class fields.

```
class List {
    var head :Elt = nil;
    class Elt {
        var value :integer;
        var next :Elt;
    }
    function prepend(v :integer) :Elt {
        head = Elt(value=v, next=head)
        return head;
    }
}
function List.Elt.remove() {
    if(head == this) {
        head = next;
        return;
    }
    var link :Elt = head;
    while(link != nil) {
        if(link.next == this) {
            link.next = next;
            break;
        }
        link = link.next;
    }
}
```

Figure 4: Example of nested classes

Here the variables `myname` and `c.name` are the same variable, so assignments to one are reflected in the other.[13]

3.12 Interface Classes

A function definition whose body is omitted and is terminated with a semicolon is called a *function prototype*. Class declarations may contain function prototypes as a convenience for documentation where the function is declared elsewhere. Alternately, the function may have no definition in which case we say the class is an *interface class*.

3.13 Notes

Custom Sequence Implementations The reserved word `seq` may be listed as the super-type for a class. Thus

```
class MySeq { implements seq; ...  }
```

This requires that function `MySeq` satisfy at least a subset of the sequence interface. This interface will be defined later but at a minimum defines construction, concatenation and iteration.

Other special things We could use an iterator named `_iterator` to be the iterator used when an class object appears in an aggregate context. Thus for object reference `p` in one of these contexts:

`for i in p` ⟨*stmt*⟩
`...  [i in p]` ⟨*expr*⟩
⟨*seq expr*⟩ ⟨*op*⟩ `p`

In these case where `p` should be considered a sequence, we invoke the `_iterator` method.

A similar technique could be used with `_function` when it is used in a function context with explicit arguments: `p(...)`. This is interpreted as `p._function(...)`.

An array is very close to something that has the form:

[13]The purpose of variable sharing is to make it easy to build ad hoc wrappers around existing state to facilitate invoking functions with particular interface requirements.

```
class F {
    type e;
    iterator _iterator :e ...;
    function _function(:e) :ref(...)  ...  ;
}
```

This gives us a domain and a function defined over that domain. Some of the other attributes may need to be provided as well.

Storage Reclamation Would it be valuable to be able to bind a function to type where the function will be invoked when the storage for the type is reclaimed? Imagine that we have two kinds of references. One kind of reference is counted and the other is not. Then, when the reference count goes to zero, we invoke this function which might cleanup the other references. This way we could for example have lists and hash tables that hold references that "don't count" but can be cleaned up.

Another application might be that we have the an object that holds a `sync` variable bound to some computation. If the object goes dead, then we could try to garbage collect the associated computation even though it technically has a reference to the object as well.

The fact that we ignore some references for garbage collection could be a property of either the type or the references.

Using `use` for arguments Perhaps we should allow `use` to prefix a formal argument declaration as in

```
function f(use c:ClientData) { ...  }
```

where now the fields of `c` are available inside the body of the function.

4 Union Types

Chapel is a completely type safe language. No variable can be accessed in a manner inconsistent with the type of the value stored in that variable. However, a type can be formed as a union of distinct types. A variable of such a type will hold a value of one of the component types and the `typeselect` statement described below can be used to access that value.

4.1 Requirements

The basic requirements:

1. Support for defining a variables that may hold values that are one of a finite number of distinct types.

2. Support for accessing those values in a type safe manner.

We want to implement these requirements exploiting the mechanisms already in place for structured types with similar syntax.

4.2 Declarations

The syntax for a union type is modeled after a record. Rather than defining a collection of variables, this is interpreted as defining a set of possible types. Here is an example of a union type:

```
union arithmetic {
    type i :integer;
    type f :float;
}
```

The type denoted `arithmetic` can hold either an integer or a floating point value. Values of any of the component types can be assigned to variables of the union type, but we must also identify which component type is intended. For example:

```
var x :arithmetic;
x.i = 1;                 – integer
x = 1.0:arithmetic.f;    – floating point
```

The last line illustrates conversion of a component type into an element of the union type. It is illegal to assign a value of a component type into a union type without specifying the component. We cannot say:

```
x = 1;  – illegal when x has union type
```

since we do not know which component type is intended. The component types such as `arithmetic.f` are treated as nominal subtypes of the associated types as if they had been declared with a `subtype` declaration (page 22).

A union type may have a component type that has no specified type information. For example:

```
union Tree {
    type Leaf;
    type Node :record { var left :Tree, right :Tree };
}
```

An instance of `Tree` might have component type `Leaf`, in which case it has no additional values, or it might have component type `Node`, in which case it is a record with fields `left` and `right`. For union type fields with no values, we use the type name as a literal value. In this example, we would use `Tree.Leaf` as a literal value of this union type:

```
var u :Tree = Tree.Leaf;
```

The general syntax for union types is:

`union` ⟨*symbol*⟩ { ⟨*declaration*⟩ [...] }

where each ⟨*declaration*⟩ has the form:

`type` ⟨*symbol*⟩ [:⟨*type*⟩];

where omitting the ⟨*type*⟩ indicates that there is no value associated with this field. The symbols used as names for components are local to the type and can only be used with the type name as a prefix as in `Tree.Node` or to access a component value as described below.

The `with` declaration may be used with unions but the referenced types must also be unions. The result is a new union type that shares some or all components with the base type. For example, we may have:

```
union data {
    with arithmetic;
    type s :string;
}
```

Here, the two fields of `arithmetic` are also part of this union exactly as if the definitions had been copied here. Values of type `arithmetic` may be assigned to variables of type `data` but not vice-versa since `arithmetic` does not have an `s` component. Assignment between union types should be thought of as a switch over the components of the right hand value which assign by name into components of the left-hand value, with suitable promotions as needed. Function definitions inside a union are similar to those of records: they are simply bound functions associated with the type.

The structural subtype relation for union types is different than for record types in the sense that a type u is a subtype of v when v the components of u are a subset of the components of v. In this example `arithmetic` is a structural subtype of `data` despite it being a "base" type.

4.3 `typeselect` Statement

A `typeselect` statement is used to access the value in a type-safe manner. For example:

```
function incr(a :arithmetic) :arithmetic {
    typeselect(a) {
        when x:arithmetic.i do return (x+1):arithmetic.i;
        when y:arithmetic.f do return (y+1.0):arithmetic.f;
    }
}
```

In this function, the argument to the `typeselect` is a expression that evaluates to a value whose type is a union. The `when` clauses select one or more of the component names. Each such clause guards a list of statements that form a ⟨*block*⟩ as in the `select` statement. The variables introduced in the `when` clause have scope limited to the list of statements with the specified type. Thus `x` is an integer and `y` is a floating point value but they retain the fact that they are components of the type union. `goto` statements may not branch into one of the blocks of a `typeselect` statement from outside of that block.

A `when` clause can identify a structural subtype of the union type. For example, where `data`, `arithmetic`, and `incr` are defined as above:

```
var x :data;
typeselect(x) {
    when a:arithmetic do a = incr(a);
    when b:data.s     do a = incr(b:arithmetic.i);
}
```

The first `when` clause selects the type components corresponding to the `arithmetic` type, so `a` may be passed as an argument to `incr`. In the second clause, `b` is a string which is explicitly converted to an integer value by converting it to a component of union `arithmetic`. When a particular component appears in multiple `when` clauses, then the first applicable clause is selected.

More features of the `typeselect` statement are described in section 6.7 after we have discussed mechanisms to resolve overloaded functions.

4.4 Accessing Union Components

If `U` is a union type that includes a component type `f`, then we may form an expression of the form ⟨*u*⟩`.f` where ⟨*u*⟩ is either a variable of declared union type or an expression converted to union type as in: ⟨u_1⟩`:U`. In this case, `f` is treated as a bound function that returns the value of the corresponding type when ⟨*u*⟩ holds a value of that component type. This expression may generate an error if the value determined by ⟨*u*⟩ does not correspond to component type `f`.

5 Sequences and Iterators

Chapel supports a derived type called *sequence*. A sequence is an ordered collection of values of some homogeneous type. Such a sequence is denoted as either:

`seq of` ⟨*type*⟩

or as:

`seq (`⟨*type*⟩`)`

where ⟨*type*⟩ is some type designator which is called the *element type* of the sequence.

5.1 Requirements

One of the key roles of data structures that Chapel expects to provide increased abstraction support for is iteration over sets. Another goal is to exploit the conciseness of aggregate expressions. These generate the requirements:

1. Support for sequences as a primitive type that is fully generic with respect to the type of elements in the sequence.

2. Support for element-wise application of primitive operations and user-level functions to sequences arguments and eventually inter-operation with arrays.

3. Language support for construction and iteration over sequences that can be implemented using user-defined objects and methods with only changes in variable type specifications.

This list will be defined in Section 11.

5.2 Sequence Operations

The fundamental operations on sequences is to construct them, copy them via assignment, and to iterate over their elements.

The reserved word `nil` is used to represent an empty sequence of any element type. The other mechanism to construct a sequence is to use the form:

`(/` $\langle expr_1 \rangle$`, ...,` $\langle expr_k \rangle$ `/)`

where all instances of $\langle expr_i \rangle$ evaluate to values of the same type after default promotions. The length of the sequence is the number of expressions, and their order is the left to right order of their specification.[14]

The binary operator `#` can be used to concatenate two sequences or to add a new element to either end of a sequence. One operand of the operator must be of sequence type. If t denotes the element type of that sequence, then the other operand must conform with type t, be a sequence of t, or be `nil`. If s is a sequence of type t and y is a value conforming with t, then:

```
y # s
```

yields a sequence whose first element is y and whose remaining elements are the elements of s in the same order as in s. Similarly:

```
s # y
```

yields a sequence whose last element is y and whose initial elements are the elements of s in their original order. Finally if s' is also a sequence of type t, then:

```
s # s'
```

is the sequence whose initial values are the same as those of s in the same order as s, and whose remaining values are the same as the the values of s' in the same order as in s'.

When $\langle t_0 \rangle$ and $\langle t_1 \rangle$ are expressions of conforming non-sequence types, then we define:

$\langle t_0 \rangle$ `#` $\langle t_1 \rangle$

to be a sequence of two elements, the same as:

`(/` $\langle t_0 \rangle$ `,` $\langle t_1 \rangle$ `/)`

When these terms are sequence types, it is necessary to use the second form if a sequence of sequences is the desired result.

The construction:

[14] I choose to use this Fortran 90 style rather than `#(...)` because I think the latter is visually ambiguous if we are building a sequence of tuples.

```
s(e)
```

where s evaluates to a sequence and e to an integer is used to select a particular value in the sequence. The result is undefined if $e = 0$, or if the absolute value of e is greater than the length of the sequence s. Let $x_1, \ldots, x_k$ denote the elements of the sequence s. If e is positive, then $s(e)$ is the value x_e. If e is negative, then it is the value x_{k-e+1}.

When `e` is a tuple, then we require `s` to be a sequence of sequences. The first component of the tuple selects an element of `s` and the remaining components then recursively select an element of that sequence.

The length of a sequence can be determined using the predefined function `length`. The length of `nil` is 0. A predefined function, `reverse`, takes a sequence and returns a new sequence containing the same values in reverse order.

Arithmetic Sequences Chapel provides a special operator to create an arithmetic sequence:

$\langle expr_1 \rangle$ `..` $\langle expr_2 \rangle$

which defines a sequence of integer values. Let n be the value of the first expression, and m be the value of the second expression. If $n > m$, then the sequence is empty. Otherwise, the sequences consists of all integer values from n to m inclusive.

Another operator, `by`, can be used to produce a subsequence. The expression:

$\langle s \rangle$ `by` $\langle expr \rangle$

where $\langle s \rangle$ is a sequence and $\langle expr \rangle$ evaluates to an integer we will denote by e. Let n denote the length of s, $x_1, \ldots, x_n$ denote the values of s, and s' denote the resulting sequence. If $e = 0$, the operation is an error. If e is positive, then the length of the resulting sequence will be $p = \lceil n/e \rceil$ and s' consists of the values: $x_1, x_{n+e}, \ldots, x_{(p-1)*e+1}$. If n is negative, then $p = \lceil n/(-e) \rceil$ and the sequence is reversed starting at the final value of s: $x_n, x_{n+e}, \ldots, x_{n+(p-1)*e}$.

In this context, if $\langle s \rangle$ is an arithmetic sequence than so is the result. Such sequences can be efficiently represented by initial value, length, and stride.

The expression `a .. b` where `a` and `b` are strings of length 1 over the same alphabet generates a sequence of strings of length 1. Each element in this sequence corresponds to a character in the underlying alphabet between the first characters of `a` and `b`. For example, `'A'..'Z''` is the sequence of strings corresponding to capital letters in the default alphabet.

5.3 Sequence Assignment

Sequence assignment is defined to be by value, so if s and s' are variables of type sequence-of-t, then the assignment:

```
s = s'
```

logically constructs a new sequence which is a copy of the values in s'. We say "logically" because in most cases sequence values cannot be modified in-place, so we can copy a pointer to the sequence rather than performing a deep copy on the values.

5.4 For Loops

A new loop construct is implemented in Chapel that has the form:

`for` ⟨*variable*⟩ `in` ⟨*expr*⟩
    ⟨*statement*⟩

Here ⟨*expr*⟩ has type sequence of t for some element type t. ⟨*variable*⟩ becomes a new `const` variable of type t whose scope is limited to this construct. The ⟨*statement*⟩ is called the body of the loop and is executed once for each value in the sequence, with the variable ⟨*symbol*⟩ initialized to that value prior to execution. For example, the loop:

```
for i in 1..n
    f(i);
```

iterates over the integer values from 1 to n, with `i` bound to each value in turn when the invocation of `f` is evaluated.

This loop is generalized to allow multiple sequences to be listed. These are enclosed in either round parentheses or square brackets. The construction:

```
for (i,j) in (s1,s2)
    ...
```

evaluates for all (i, j) pairs in the *zipper product* of s_1 and s_2. This means that the sequences must have the same length and corresponding elements are selected to form the pairs. If k is the length of the sequences, then the above is equivalent to:

```
for p in 1..k {
    const var i = s1(p);
    const var j = s2(p);
    ...
}
```

Alternately:

```
for (i,j) in [s1,s2]
    ...
```

evaluates for all (i, j) pairs in the *cross product* and is equivalent to a nested loop:

```
for i in s1
    for j in s2
        ...
```

assuming there are no side-effects to evaluation of `s2`.

An additional restriction on the `goto` statement is that the target may not be associated with a statement inside a `for` loop that does not also contain the `goto`.

5.5 Expression Iterator

Chapel provides a mechanism to allow naming elements of a sequence inside an expression. The construct is

```
[i in S] f(i)
```

where `S` is a sequence. This expression yields a sequence corresponding to the values of evaluating `f` for each value of `S` denoted by `i`. For example to form a sequence of squares of a sequence we could use:

```
[y in X] y*y
```

The precedence of this expression form is lower than any other operator and they are right-associative so:

```
[i in S1] [j in S2] f(x,i,j)
```

has the same iteration order as:

```
for (i,j) in [S1,S2]
```

In this example, if t the type of `f(x,i,j)` then the type of the above expression is `seq(seq(`t`))`.

An expression iterator may also be used to control an assignment statement:

```
[i in S] f(i) = x(i)
```

Here `f` is, for example, a sequence of references to which values are assigned element-wise. This construct differs from:

```
([i in S] f(i)) = [i in S] x(i)
```

where the second fragment will evaluate all of the right hand side before assigning into the references on the left hand side.

5.6 Sequence Promotion of Scalar Functions

We define the *rank* of a type t to be 0 of t is non-sequence type. The rank of the type `seq(`t`)` is one more than the rank of t.

For a sequence of rank 1, the *leaf elements* are simply the elements of the sequence. For a sequence s of rank greater than 1, the leaf elements of s are the sequences formed by concatenating the leaf elements of the elements of s. We will denote this as $leaves(s)$. This order of elements in $leaves(s)$ is called the *normal order* for the sequence.

We say two sequences are conforming if they are both rank 0 or they are both sequences of equal length where corresponding elements are conforming. Necessarily, conforming sequences have the same rank and the same number of leaf elements. The *shape* of a rank-1 sequence is its length. The shape of a rank-k sequence is the rank-$(k-1)$ sequence formed by replacing all the rank-1 sequences with their lengths. The shapes of conforming sequences will be the same.

Consider now a function application of the form:

`f(`$\langle e_1 \rangle$`, ..., `$\langle e_k \rangle$`)`

where one or more of the actual arguments have sequence type where the corresponding formal parameter has the same leaf element type but lower rank than the argument. We require that all of these differences either be zero or the same value r. The interpretation of such a function application will be to build a rank-r sequence whose elements are the results of applications of `f`.

If there is only one argument, say $\langle e_1 \rangle$, whose rank is r more than the corresponding formal. Then we reduce r by rewriting the above expression to:

`[x in `$\langle e_1 \rangle$`] f(x, ..., `$\langle e_k \rangle$`)`

When there are two or more such expressions, say $\langle e_1 \rangle$ and $\langle e_2 \rangle$, then we require those expressions to have the same length and and `f` is applied to the zipper-product:

`[ (x,y) in (`$\langle e_1 \rangle$`, `$\langle e_2 \rangle$`)] f(x, y, ..., `$\langle e_k \rangle$`)`

This process is applied recursively until all actual arguments to `f` have the same rank as the corresponding formals. This is called the *zipper* interpretation. The zipper interpretation is also used for all binary operators. Thus the expression:

$\langle s_1 \rangle$`+`$\langle s_2 \rangle$

yields the sequence of values

`[ (x,y) in (`$\langle s_1 \rangle$`, `$\langle s_2 \rangle$`)] x+y`

An alternate construction using square brackets:

```
f[e1, ..., e2]
```

returns a sequence formed by taking the cross product of the values of the two sequences. For example, if `s1` has length `n1` and `s2` has length `n2`, then the expression:

`f[x,`⟨*s1*⟩`,`⟨*s2*⟩`]`

yields the sequence of values:

```
let s1 = ⟨s1⟩, s2=⟨s2⟩
    in [i in s1] [j in s2] f(x,i,j)
```

In general, the above recursive process is applied using cross-products instead of zipper-products. The rank of the resulting sequence is the sum of the non-zero differences between the ranks of actual and formal parameter types.

In the special case where `f` is a sequence and `s` is a sequence of integers, the resulting sequence `f(s)` is referred to as a *subsequence*.

5.7 Sequence Equality

The binary operators "`==`" and "`!=`" are like any other scalar operator when applied to sequence operands. Their result will be a sequence of boolean values based on pairwise comparisons of elements of the zipper product. The built-in functions `any` and `all` reduce a sequence of boolean values to a single value. Function `any` returns `true` if, and only if, the sequence is non-empty and any element is true. Function `all` returns `true` if, and only if, the sequence is empty or all elements in the sequence are `true`. These functions are not required to evaluate any more of their sequence inputs than are necessary to determine the result.[15]

The primitives `any` and `all` are examples of *reductions*. They produce scalar results regardless of the rank of their inputs. For example:

```
var x :  seq(seq(integer)) = ...;
var y :  like y = ...;
if(all(x == y)) ...
```

The type of `x==y` is `seq(seq(boole))`. Its result is reduced by `all` to a single boolean value used to determine the `if`.

When a sequence value is used as the test for an `if` statement or expression, then that sequence is promoted to boolean values and the boolean values reduced to a scalar implicitly by the `all` primitive. Thus we could write the above simply as:

[15]There may be faster ways to determine sequence equality depending on the implementation of the sequence. When we define the structural interface for sequences this is something we should be sure to expose.

```
if(x == y) ...
```

which would behave the same as above. This is also true for controlling expressions for `while` and `repeat` loops.

The interpretation is different in a select statement (page 26) such as:

```
select(⟨e_s⟩) ...
    when ⟨e_w⟩ ...
```

We permit two additional interpretations. When $\langle e_s \rangle$ is a sequence, then $\langle e_w \rangle$ must be a sequence of the same rank and the test to determine if this clause is selected can be expressed as `all(`$\langle e_s \rangle$ `==` $\langle e_t \rangle$`)`. If $\langle e_s \rangle$ is a scalar and $\langle e_w \rangle$ is an sequence, the test becomes: `any(`$\langle e_s \rangle$ `==` $\langle e_w \rangle$`)`.

5.8 Filtering Predicates

An `if` expression inside a sequence expression is not required to have an `else`-clause. The resulting sequence consists of the values where the predicate is true. For example, the following construct selects the odd elements in a sequence:

```
[i in 1..n] (if (mod(i,2) == 1) then s(i))
```

Regardless of the rank of the expression context, the rank of the result of a filtering predicate is always 1. Thus:

```
[i in 1..n] [j in 1..m] (if (mod(i+j,2) == 1) then (i,j))
```

The result of this expression is a rank-1 sequence of integer pairs.

5.9 Indefinite Sequences

An arithmetic sequence using either of the forms:

```
n..
..n
```

(with missing upper or lower bounds) is said to be an *indefinite arithmetic sequence*. An indefinite arithmetic sequence may appear in expression contexts but only when it is "zippered" with a definite sequence. Thus we may have:

```
f(x, ⟨s1⟩, m..)
```

which is interpreted as the expression:

```
let x1 = ⟨s1⟩, k = length(x1)
    in [i in 1..k] f(x, x1(i), m+i-1))
```

The other sequences provide bounding information.

The zipper operation need not occur in the immediate context of an indefinite sequence but could be higher in the expression. Thus:

```
f(x, ⟨s1⟩, ⟨s2⟩(m..))
```

is a valid expression where the term ⟨s_2⟩`(m..)` selects a subsequence of ⟨s_2⟩ beginning with the m^{th} element and continuing for the length of ⟨s_1⟩.

An indefinite sequence may also be used as a subscript for another sequence, in which case the extent of that sequence provides the necessary bounding information if there is no other constraint. For example, if `s` is a sequence of length n, then `s(2..)` is equivalent to `s(2..n)` unless the entire expression is in a context where other shape information is available, such as: `s(2..) + f(n..m)`.

The predefined function `fill` takes two or three parameters. When there are three parameters, `fill(`⟨s_1⟩`,`⟨e⟩`,`⟨n⟩`)`, the first is a sequence, the second is a value conforming to the element type of the sequence, and the third is an integer. The result of this function is a sequence of length at least ⟨n⟩ where copies of the value determined by ⟨e⟩ are appended as necessary. Express ⟨e⟩ is evaluated once regardless of the length of ⟨s_1⟩.

When there are only two parameters, they are the sequence and the element value. The result is an indefinite sequence whose extent is determined from context.

5.10 Arithmetic Sequences and Strings

For a variable, `s`, of string type, the expression `s(i)` returns a length-1 string corresponding to the i^{th} character in the string. When `i` is an arithmetic sequence, the result is a new string corresponding to the concatenation of the selected elements. If the arithmetic string is indefinite, then it becomes limited by 1 as a lower bound and the length of the string as an upper bound. Strings are not sequences. The expression `a+b` is interpreted as string concatenation, not an element-wise operation over the component characters.

5.11 Iterators

An *iterator* is a function that returns a sequence of values rather than just one. An iterator is defined with the syntax:

```
iterator ⟨symbol⟩(⟨parameter list⟩) :⟨type⟩ {
    ...
}
```

The ⟨*type*⟩ here is the element type of the sequence of values that will be returned.

The assignment of values, the syntax for invocation, default and optional parameters and interpretation of currying are all the same as for functions.

A new statement is permitted inside an iterator with the syntax:

```
yield ⟨expr⟩;
```

where ⟨*expr*⟩ evaluates to a value of type ⟨*type*⟩ or sequence of ⟨*type*⟩. In the form case, the value of ⟨*expr*⟩ becomes the next value in the sequence returned by the iterator. If ⟨*expr*⟩ is a sequence, the the values of that sequence, in sequence order, are the next values returned by the iterator. A `return` statement is similar but the iterator terminates after the return. The iterator also terminates when the last statement in the body has been executed.

Any actual parameters to an iterator with `out` or `inout` attributes are not modified until the iterator completes either with a `return` or by finishing its body.

An iterator that returns variable references and that appears as the target of an assignment will update each variable location with the corresponding elements from the right hand side. For example, assume that `dfs` is an iterator that returns variables associated with a tree in depth first traversal order. We may then assign a depth-first numbering to those variables by:

```
dfs(tree) = 1..;
```

or[16]

[16] So are we inferring that `t` has reference type because we assign to it coupled with the fact that the iterator returns references?

```
for (t,i) in (dfs(tree), 1..)  do
    t = i;
```

5.12 Arithmetic Index Sets

An *arithmetic index set* is a representation of a cross-product of a tuple of arithmetic sequences. This type is denoted by `index(k)` where `k` is a parameter corresponding to number of components. The `..` is overloaded to allow k-tuples of integers to be operands and k-tuples and simple integers may be mixed. Scalar inputs are promoted into k-tuples by creating k-copies of the value. The result has type `index(k)`. The `by` operator is similarly overloaded. A tuple type of the form:

`( :index(`$\langle k_1 \rangle$`)` ,..., `:index`$\langle k_n \rangle$`)`

is considered interchangeable with a value of type `index<k>` where $k = \sum_{i=1}^{n} k_i$.

The following are examples of such index sets:

```
var lb :  2*integer = (1,1);
var ub :  2*integer = (n,m);
1 ..  ub
lb ..  ub
(1..n, 1..m)
```

All three expressions represent the same value and have the same type.

For a value of type `index(k)`, there are several predefined methods that are listed in Figure 11 on page 152.

5.13 Sequence Primitives

Order and Shape A number of functions are defined that preserve the leaf values of a sequence but change their order or shape.

In what follows, assume s is a sequence of rank-k. $(i_1, \ldots, i_k)$ denotes a k-tuple of integer values. The result of functions will be denoted s'. We will define the shape and values of s', defining value-preserving relationships between selected elements.

`reverse(:seq,dim=1)` Function `reverse(s)` returns the elements of `s` in reverse order. An optional second argument is an integer or list of integers. If the value of this parameter is d, then the rank of the first parameter must be at least d. When d is one, then the sequence is reversed. When d is greater than one, the result is equal to this expression:

```
[x in s] reverse(x,d-1)
```

When the second argument is a list, then the reversal process is applied for each dimension value in turn.

Here are some examples involving rank-2 sequences:

```
var s :  seq(seq(integer,integer))
                = [i in 1..n] [j in 1..i] (i,j);
reverse(s)                  – [i in 1..n by -1] [j in 1..i] (i,j)
reverse(s,dim=2)            – [i in 1..n ] [j in 1..i by -1] (i,j)
reverse(s,dim=(/1,2/))      – [i in 1..n by -1] [j in 1..i by -1] (i,j)
```

The initializer for `s` builds a rank-2 sequence in integer pairs that has a triangular structure. The next three lines show the results of various `reverse` operations applied to that sequence in terms of the how you might specified the result directly.

`spread(:seq,dim=1,[length])` Function `spread` takes a sequence of rank k and returns a new sequence of rank $k+1$. There are two optional arguments that maybe specified by name. Let s denote the input sequence. When `dim` is equal to 1 , the result is a sequence where every element is equal to s. The length of this sequence is specified by `length` or is indefinite. When `dim` is greater than one, we generate the sequence:

```
[x in s] spread(x,dim=dim-1,length=length)
```

`transpose(:seq, dims=(2,1))` The `transpose` function will reorder both the values and change the shape of the sequence. The `dims` argument evaluates to either a tuple or a list of integers that corresponds to a permutation of the values $1..p$ where p is less than or equal to the rank of the input sequence. This list defines a permutation of the subscripts

such that the following relationship holds between the input and output sequences:

$$s'(i'_1, \ldots, i'_k) = s(i_1, \ldots, i_k)$$

where

$$i_j = \begin{cases} i'_{dims(j)} & \text{if } j \leq p \\ i'_j & \text{otherwise} \end{cases}$$

There is a somewhat complex requirement in the shape of the input sequence so that this relation is well-defined. In the simple case of a rank-2 input, we require that all elements of the sequence have the same length. In the general case we require all sequences selected by a q-tuple to have the same length whenever an index position q is less than p.

`reshape(:seq,` ⟨*shape*⟩`,` [`fill=`⟨e⟩`])` This returns a sequence whose leaves are the same as the first parameter, in the same order but whose shape matches that of ⟨*shape*⟩. The ⟨*shape*⟩ parameter might be a sequence or it might be a tuple of integers. When it is a sequence, the output will conform to that sequence. When the shape is a tuple, then the shape of the output conforms to the shape of the arithmetic index set that would be determined by `1..<shape>`.

If present, the `fill` parameter specifies a value to be used to pad the sequence if the number of leaf values in that sequence is too few to conform with ⟨*shape*⟩. If the input sequence has too many values, it is truncated.

The operations `reverse`, `transpose` and `reshape` can be used as the targets of assignment when the leaves of the input sequence are references.

Conversion between lists and tuples A value of tuple-type, `k*<t>` can be converted to a value of sequence types, `seq(<t>)` by using the `:seq` as a type conversion. Similarly, the conversion `:k` where `k` is an integer `parameter` can be used to construct a tuple from the first `k` elements of the list. It is an error if the list does not have length at least equal to `k`.

Reductions The reserved word `reduce` is used to indicate a *reduction*, which is a situation in which a binary operator is used to reduce the values of a sequence down to a single scalar. If the sequence has element type t, then

the common case is that the operator is a binary function with signature: `function (:t,:t) :t`. A `reduce` expression looks like this:

`[ordered] reduce` ⟨*s*⟩ `by` ⟨*op*⟩[`else` ⟨*expr*⟩])

The third parameter is the return value when ⟨*s*⟩ is empty. If ⟨*s*⟩ has length 1, then the value of its first element is returned. Otherwise ⟨*op*⟩ is replied repeatedly to combine elements. The order of element combining is not defined but will be consistent for sequence of a given length in a given execution environment as discussed later. If the `ordered` keyword is present then elements are reduced by applying ⟨*op*⟩ to reduce the first two elements to a single value, and then using ⟨*op*⟩ to accumulate remaining values from the list into that result.

An `ordered scan` is just like a reduce except that the result consists of a sequence of values. The first element is the first element in the original sequence followed by the result of the accumulations. An unordered `scan` assumes that ⟨*op*⟩ is associative and allows different algorithms to be used based on that assumption. The pattern of application is again determined by the length of the input sequence and the execution environment.

More complex forms of reductions are described in Section 11.1.

5.14 Cursor

A *cursor* is a reference to a sequence and a position in that sequence. A cursor technically identifies a point between elements and can be at the tail or the head of a list. A cursor can be created using one of these methods defined for sequences:

head A cursor before the first element.

tail A cursor after the last element.

before(index:integer) A cursor before the value at the specified index. It is an error for the index to be greater than one more than length of the sequence or less than 0.

after(index:integer) A cursor after the value at the specified index. It is an error for the index to be greater than the length of the sequence or less than 1.

The predefined type `cursor` identifies a generic cursor type for which we have the following fields and methods:

`prev` The value in the sequence before the cursor.

`next` The value in the sequence after the cursor.

`forward(amount=1)` A new cursor that is `amount` positions earlier in the sequence.

`backward(amount=1)` A new cursor that is `amount` positions later in a sequence.

`sequence` The sequence associated with a cursor.

`index` The integer index corresponding to the position of the `prev` field.

`tail?` A predicate that returns `true` when the cursor is positioned after the last element in a sequence or the sequence is empty.

`head?` A predicate that returns `true` when the cursor is positioned before the first element in a sequence or the sequence is empty.

The following methods of type `cursor` all return new cursors that refer to new sequences constructed from the sequence associated with the base cursor. They vary in how the new sequence is constructed:

`truncate` The new sequence consists of all elements before the position of the cursor.

`remainder` The new sequence consists of all elements after the position of the cursor.

`insert_after(value)` The new sequence consists of all elements of the original sequence with a new value inserted at the position of the cursor. The new cursor is positioned before the new element.

`insert_before(value)` The new sequence consists of all elements of the original sequence with a new value inserted at the position of the cursor. The new cursor is positioned after the new element.

`remove_next` The new sequence is formed by removing the `next` element from the base sequence. The new cursor is position before the element that followed `next`.

`remove_prev` The new sequence is formed by removing the `prev` element from the base sequence. The new cursor is position after the element that preceded `prev`.

In all of these cases the base sequence and any other cursors are not effected by these actions.

Iteration can be performed over cursors rather than list value by specifying that the type of the control variable is a cursor. In a `for` loop this looks like:

```
for i:cursor in S ...
```

and in an expression context:

```
[i:cursor in S] ...
```

A cursor may be used in a subscript position for a sequence in which case it returns the `next` value.

An iterator may be defined to return a `cursor` rather than an instance of the element type of the sequence. Such references are implicitly converted to their `next` values if they are not bound to `cursor` variables.

5.15 Notes

Enumerated sequences Given an integer enumeration. Can the "`..`" operator be used to define a sequence of enumerated values? If there are corresponding integer values, when is the conversion made?

Complex Arithmetic Index Spaces We can generalize the notion of an arithmetic index set to include more complex structures. Consider a sequence of the form:

`[`i_1 `in` a_1`]` ... `[`i_k `in` a_k`]` `(`$i_1, \ldots, i_k$`)`

where each term a_j is an arithmetic sequence operator: L_j `..` U_j `by` S_j where each expression that terms this sequence is a linear combination of enclosing i_j plus a constant. Such a sequence can be compactly represented by retaining the coefficients. The interesting common cases are triangular spaces such as `[i in 1..n][j in 1..i] (i,j)`.

We might need to also allow `min` and `max` operations to allow representation of banded spaces such as:

```
[i in 1..n][j in max(1,i-d)..max(i+d,n)] (i,j)
```

I don't think special syntax is important but whether such sets are compactly represented and those representations closed under `reverse`, `spread` `transpose`, slicing and projection.

6 Function Overloading

Chapel permits multiple functions to be associated with the same symbolic name. This is frequently called *function overloading*. When a function is invoked, the types of the actual arguments are used to determine which of a set of candidate function definitions is the one that is actually invoked. This section discusses the rules that govern this process.

6.1 Requirements

The basic purpose of overloading is to support polymorphism so that one algorithm specification can be specialized to different implementations of constituent functions.

Requirements:

1. Support for the object-oriented paradigm where functions may be specialized to object subtypes.

2. Support for specialization of generic behavior for special cases. This includes parameterization of function by structural types as well as by nominal types.

3. Support for modular programming so that implementations of different portions of an application may be developed independently once interfaces are defined.

Features:

1. Treat all arguments to a function equally with respect to determining which function to call.

6.2 Type Constraints on Function Arguments

When a function is declared, a specified type on a formal argument is interpreted as a *constraint* on the actual arguments. At a function invocation, the actual arguments must be a subtype of all of the constraints of the formal arguments of the function that is selected for execution.

Chapel supports two kinds of constraints on function arguments, *nominal* and *structural*. When the type of a formal argument is the name of a class, or a symbol bound by a `subtype` declaration, then it is a nominal constraint.

Otherwise, when the constraint is specified by a primitive type, record type, union type or sequence type, then it is a structural constraint.

For a nominal constraint, the actual argument must be an instance of the specified type or a nominal subtype of that type. Nominal subtypes are created with the `subtype` declaration (page 22) or the `implements` clause (page 52) for derived types. For example:

```
class A { ...  }
class B { implements A; ...  }
class C { implements A; ...  }
function lookup(b :B,...)  ...
function lookup(c :C,...)  ...
lookup(x,...);
```

Here we have three class definitions where both classes `B` and `C` are nominal subtypes of class `A`. There are also two definitions of function `lookup` with different constraints on the first argument. The particular instance of `lookup` that will be invoked by the invocation on the last line depends on the dynamic type of the value in variable `x`. If `x` refers to an instance of `B`, then the first function is invoked, while if `x` refers to an instance of `C`, then the second instance will be invoked. Should `x` hold an instance of class `A`, then the constraints of neither definition of `lookup` are satisfied, resulting in an error (one that may not be detected until runtime).

Structural constraints depend on the structure of the types rather than their names. When the formal argument identifies a primitive type, then the corresponding formal must be the same primitive type. Thus we can have functions:

```
function abs(x:integer) ...
function abs(x:float) ...
```

Here the primitive type of the actual argument determines which of these function will be invoked regardless of the name associated with that type. This idea is generalized to record types. For example:

```
function insert(x :record { var name :string;
                            var id :integer; }) ...
function insert(x :record { var name :string;
                            var age :integer;
                            var id :integer; }) ...
```

Here we have two functions called `insert` both of which require the actual argument to have fields `name` and `id`, while the second also requires the field `age`. As in record assignment, the order of fields does not matter for structural constraints, only their names and types. This interpretation is unchanged if the actual argument is specified with a named record type instead of an anonymous type. For example:

```
record ClientData {
    var name :string;
    var id :integer;
}
function insert(x :ClientData) ...
```

This declaration of `insert` is the same as the first one in the previous example: the actual argument is structurally constrained to have the two specified fields. Then name `ClientData` is irrelevant to determining which version of `insert` is invoked.

Union types are handled with similar structural rules:

```
union arithmetic {
    type i :integer;
    type f :float;
}
function incr(a:arithmetic) ...
```

The function `incr` defined in this example will be invoked for any union type all of whose components are included within type `arithmetic`.

For types with structure, such as tuples, records, unions and sequences, the component types for formal arguments are interpreted similarly as either structural or nominal types. For example:

```
class A {...}
function lookup( x :(integer, A)) ...
```

This defines a function with a structural constraint that the actual argument be a tuple whose first component is an integer and whose second component is constrained to be an instance of class `A` or a nominal subtype of `A`. This applies similarly to records and sequences. For tuples, a component type may be specified by "`_`" to indicate that component is unconstrained.

The component types in a union are treated as nominal subtypes of the types to which they are bound (page 60). They may be used as constraints on formal arguments. For example:

```
union Mat {
    type full :Matrix;
    type lower :Matrix;
}
function transpose(a :Mat.full) ...
function transpose(a :Mat.lower) ...
```

These define functions that require the actual argument to be a structural subtype of `arithmetic` and then use nominal rules to distinguish between the two variants.

It is possible to describe a structural constraint on an object argument. This is done by using the keyword `like` in place of the colon ("`:`") when we specify the constraint on a formal argument. For example we might have:

```
class A {
    var f :integer;
    function eval(:integer) :float;
}
function search(x like A) ...
```

Here we specify a class definition `A` that consists of an integer variable `f` and the prototype for a bound function `eval`. The function `search` is declared to apply to any class that is "like" `A` in that it has such fields `f` and `eval`. The name `A` is immaterial to this structural constraint.

6.3 Most Specific Definition

For a particular actual argument, there may be multiple function definitions whose constraints are satisfied. For example:

```
class A { ...  }
class B { implements A; ...  }
function lookup(a :A,...)  ...
function lookup(b :B,...)  ...
lookup(x,...);
```

If the variable `x` holds a reference to an instance of `b`, then this satisfies the constraints for both of the definitions of `lookup` since the first definition is valid for any instance of `A` or nominal subtype of `A`.

The subtype relation is a partial order on types in that instances of `B` are also instances of `A` but not vice-versa. In this case we say that the second declaration of `lookup` is the *most specific*. Given two constraints x and y, x is more specific than y if every value instance that satisfies x also satisfies y, while other values may satisfy y but not satisfy x. The function definition with the most specific constraints is the one selected at a particular call site from among candidate function definitions whose constraints are satisfied.

The most specific definition for nominal constraints is determined strictly from the declared subtype relationship. For structural constraints the relationship is inferred. For example, for a call site `f(r)` where `r` has a record value, the most specific call site is the one that mentions the most field names available in `r`. To continue the example from page 82, a call:

```
insert( (name="fred", id=4, age=20) )
```

would invoke the second version of `insert` because all three components of the actual argument are required by that version.

Given a call site `f(u)`, where `u` is a union type, the most specific constraint will include all the components of `u` but exclude some component mentioned in each other acceptable candidate. For example, if we augment the example on page 83 with:

```
union data {
    with arithmetic;
    type s :string;
}
function incr(a:data) ...
```

The previous definition is more specific for any call `incr(u)` where the type of `u` is a union with only `i` and `f` fields.

The most specific rule may be inadequate to uniquely identify which definition to invoke. For example:

```
class A {...} class B {...}
class C { implements A; implements B; }
function search(x :A) ...
function search(x :B) ...
```

Since class `C` is a nominal subtype of both `A` and `B`, neither definition of `search` is more specific than the other when the actual argument is an instance of `C`. In such a situation the invocation is said to be *ambiguous* and is treated as an error. Such an error may not be diagnosed until program execution, but the possibility can be diagnosed at compile time.

The default promotion rules defined on page 18 have the effect of inducing a partial order on primitive types that is respected for the purpose of determining which function is invoked. For example:

```
function hash(x :integer) ...
function hash(x :float) ...
...  hash(6) ...
```

Even though we can promote an integer value to a floating point value and so it would be valid to invoke the second definition of `hash`, we treat integers as a structural subtype of `float` for the purpose of determining which function is called and hence the first definition is the more specific.

For functions with multiple arguments, constraints must be simultaneously satisfied for all arguments. This also induces a notion of most specific constraint on the collection of formal arguments. For example:

```
subtype B:A; subtype D:C;
function hash(x :A, y :C) ...
function hash(x :B, y :C) ...
function hash(x :A, y :D) ...
```

Given an invocation `hash(p,q)` where `p` is an instance of `B` and `q` is an instance of `C`, then the constraints of the first two definitions are satisfied and between them the second is the more specific. However, if `p` is an instance of `B` and `q` is an instance of `D`, then the constraints of all three definitions are satisfied and while both the second and third are more specific than the first, neither is more specific than the other and the invocation is therefore ambiguous.

The rules for promoting scalar functions in sequence contexts (page 68) can lead to ambiguities. This happens when a function is overloaded where some argument may be a scalar type or a sequence with that scalar as element type. For example:

```
function find(x:integer) ...
function find(x:seq of integer) ...
var s :seq of integer;
...  find(s) ...
...  [i in s] find(i) ...
```

Either definition of `find` could be used to satisfy the first invocation, while only the first may be used to implement the second. To resolve the ambiguity for the first call, we treat the sequence definition as more specific than the scalar definition. Thus, the first invocation of `find` above will invoke the second definition which has a sequence formal argument.

The actual arguments at a call site may satisfy the constraints of a function definition that include default values for optional arguments. In this case, we require that there be a most specific definition as determined by the specified actual arguments. For example:

```
function hash(x :integer, y:integer=2)
function hash(x :integer, z:float=2.0)
...  hash(5) ...
```

Here the invocation of `hash` is valid for both definition but since neither definition is more specific than the other with respect to the actual argument, then the function invocation is ambiguous.

6.4 Function Candidates

The above rules discuss determining which of the candidate definitions is selected based on the types of actual arguments. Here we discuss the rules that identify the set of candidates.

Given a call site to a function `f`, we say a function definition for symbol `f` is *lexically visible* if that definition appears in an enclosing scope either directly or by effect of a `use` or `with` statement.

When all formal arguments to a function definition have structural constraints, then we only consider that definition to be a candidate where it is

lexically visible. For example, consider a function defined over integers in some module:

```
module MyLib {
function fib(x:integer) ...
```

Here `fib` is a candidate only inside `MyLib` and in scopes that explicitly `use MyLib`.

When a formal argument involves a nominal constraint however, then the definition is a candidate at a larger set of call sites. We define the *program* to be the `main` module and any module that is mentioned directly or indirectly via a `use` statement. Any function in the program with the same name `f` and at least one nominal constraint is considered as a candidate for any call to `f`. In this case, we rely on the scoping of type names to limit the set of candidates. For example, when we consider a function call `f(x)` where `x` is bound to an instance of a class `C`, then the candidate function definitions are all functions whose first argument is nominally constrained to be `C` or a nominal supertype of `C`. Neither `C` nor any of its superclasses needs be lexically visible at the point of invocation of `f`.

The above rule is modified when not all functions are exported by a module. A function defined in a module must be exported (page 35) by the module to be invoked by a call site at which it is not lexically visible. Similarly, no definition not lexically visible is a candidate at a call site that names a function which is private to a module. This allows function behavior to be modified by a module but also allows a module to protect its local name space to avoid inadvertent overloading with other modules.

6.5 Bound Functions

A bound function is invoked by specifying an instance of the type to which the function is bound. This instance is passed to the function "by reference" and that argument is treated as a type constraint following the usual rules to determine if it is a structural or nominal constraint.

This basic rule is modified for functions bound to record and union types where we treat the bound argument as a nominal constraint rather than a structural constraint. For example:

```
record A { int a; function lookup ...  }
function A.verify ...
record B { with A; }
var x :B;
...  x.lookup ...
```

By the definition of `with`, we have created two function definitions for `lookup`. For the purpose of determining which to call, we use the nominal type of `x`, which is `B`. If the nominal type of a record variable or value does not adequately determine the function, then it can be converted to a different nominal type. For example, if we have a record `C` that is a structural subtype of `A`, we can invoke `A`'s lookup method via:

```
var c :record { ...  };
...  c:A.lookup ...
```

This use of the conversion operator does not construct a new value but rather changes the interpretation of the value in `c` to be an instance of record `A` for the purpose of determining which instance of `lookup` to call.

Bound functions are a separate name space from other functions and a single call site will consider either bound or unbound functions as candidates. Thus an invocation `lookup(x)` never considers the definition `A.lookup`.

6.6 Function Results

The types of function results do not effect the choice of which function is invoked. This includes not only the return value of the function but also arguments with intent `out`. For example:

```
function search(p :A, out new:boole) :C ...
function search(p :B, out index:integer) :D...
...  search(x,y) ...   – determined x only
```

In this example, only the type of the first argument is used to disambiguate the candidates. The type of `y` and the return types do not effect the choice of which function is invoked.

Similarly, an argument with type `inout` generates a constraint on the type of the actual argument, but the requirement that an assignment back to the actual argument be valid is not used to select functions.[17] For example,

[17] I think this is consistent with our previous discussion but I'm not sure I understand the rationale. An `inout` intent is not simply sugar if we require a modifiable variable as

```
function lookup(in x :float) ...
function lookup(inout x :float) ...
...  lookup(5.0) ...   – ambiguous
```

The call to `lookup` is ambiguous even though we can not assign to `5.0` and so a call to the second `lookup` is not legal.

6.7 `typeselect` Statements

A `typeselect` statement may also be used to determine the types of one or more values using the same mechanisms used to disambiguate function definitions. For example:

```
class Rectangle { implements Shape; var height };
var s :Shape;
...
typeselect(s) {
    when r:Rectangle do r.height = 10;
    otherwise        call error("expected Rectangle");
}
```

Assume here that `height` is not a field of `Shape`. The reference to `r.height` is valid once we have determined that the object that `s` refers is in fact a `Rectangle`. This example also shows the `otherwise` clause which guards a list of statements executed in the event that there is no `when` clause corresponding to the value stored in the variable.

Syntactically, the primary expression in a `typeselect` may be a comma-separated list of expressions, and each `when` clause has a comma-separated list of constraints with local names which mimics a formal argument list for a function without intents.

Thus, we may have:

```
typeselect(x,y) {
    when (a:Square, b:Square) ...
    when (a:Rectangle, b:_) ...
    ...
}
```

---

the actual argument.

Here the values in `x` and `y` must both satisfy the constraint of the `when` clause. The wild-card "`_`" is used to indicate the type of the corresponding value is unconstrained. Like function calls, the most specific set of constraints is selected based on the types of the values during execution. In the absence of a most significant choice, the lexically first among equal candidates is the clause that is selected. The `otherwise` clause is selected only in the case that there are no `when` clauses whose constraints are satisfied.

6.8 Function Values

When an expression forms a function value (page 30) rather than invoking the function, the set of candidate definitions for this function is also identified. The set of candidates is determined as if the function were invoked at the point where the value is formed. The particular choice within this set is not determined until all actual arguments are specified at the point of invocation of the function. For example, we might capture a function into a class variable:

```
class A ...
function create(x :A, y :integer) ...
c.action => create(y=4);
```

Here we capture the value of `y` and the result is a function of one parameter constrained to be a nominal subtype of `A`. The set of candidate function definitions is determined at this point while a subsequent invocation:

```
call c.action(a);
```

will provide the actual argument that determines which instance of `create` is actually invoked. The point of invocation does not influence the set of candidates.

6.9 Nested Function Definitions

A function definition may be nested inside of another function definition. By default such functions are *public* but they may be explicitly declared to be *private*.

```
class A ...
function hash1(a:A)...
function search(x:A) { ...  hash(x) ...}
function phase2 {
    var x :integer;
    class B { implements A; ...  };
    private function hash2(b:B) ...  x ...
    ...  hash(y) ...
    ...  search(y) ...
}
```

Figure 5: An example of private nested functions. The definition of $\texttt{hash}_2$ shadows the outer scope definitions so they are not candidates at call sites inside function `phase2`. In this example, the invocation of `hash` inside of function `search` will see only $\texttt{hash}_1$ as a candidate.

Private Nested Definitions A private function declaration is treated as a separate name from the ambient scope and that name shadows all definitions not in the same scope. See the example in Figure 5. The local definition of function `hash` excludes any non-local definitions from being candidates to the call to `hash` inside of function `phase2`. Should that function be called with an instance of class `A`, then the definition $\texttt{hash}_1$ would not be a candidate. Such a case might be an error since the constraints for $\texttt{hash}_2$, the only candidate, are not satisfied. The local definition, denoted $\texttt{hash}_2$, is not a candidate for any call site outside of `phase2`.

Private functions in a scope can be captured as a function value. For example, we might extend the example of Figure 5 by storing a function in an object:

```
b.action => hash;
```

Since function values determine candidate definitions from the point of capture, the local function $\texttt{hash}_2$ might now be invoked from a call site outside of the function `phase2` by way of the function value `b.action`.

The bodies of private functions can make references to variables declared in the containing function scope. This is true even after the containing function has returned if the function was captured as a function value. This

```
class A ...
function hash1(a:A)...
function search(x:A) { ...  hash(x) ...}
function phase2() :A {
    var x :integer;
    class B { implements A; ...  };
    var y:B = B();
    function hash3(b:B) ...
    ...  hash(y) ...
    ...  search(y) ...
    return y;
}
function driver {
    var z :A = phase2();
    ...  search(z)...
}
```

Figure 6: An example of a public nested functions. The definition of `hash`$_3$ is public, hence is a candidate for the call site inside `search`. In the two cases illustrated `hash`$_3$ is also the most specific definition and so will be the one invoked.

will have the effect of extending the lifetime of variables such as `x` as long as the function value is still live.

Public Nested Definitions By default, a function with nominal type constraints[18] is public and may be invoked from any point in the program where it is the most specific candidate. Such a function definition is essentially the same as a function defined at module scope, except that it has access to nominal subtype definitions whose scope is limited to the containing function. An example is shown in Figure 6. This differs from the previous function in that `hash`$_3$ is public and so the function invocation inside of `search` now has two candidate definitions and both calls shown will invoke function `hash`$_3$ as the one with the most specific constraints.

[18]Functions with only structural type constraints are effectively private since such functions are only candidates at call sites in which they are lexically visible.

Since public functions can be invoked at any time, they do not have access to variables in the containing lexical scope and so may not reference them. They may still reference types and other public functions in that scope.[19]

6.10 Operator Overloading

Chapel allows most of the prefix and infix operators shown in Figure 1 on page 17 to be overloaded with application specific function or iterator definition. The syntax to declare such an "overloaded operators" is just like other functions or iterators but an operator is used instead of a ⟨*symbol*⟩ to name the function. For example we might have:

```
record Polar { var r :float=0.0, theta :float=0.0 }
function * (x :Polar, y:Polar) :Polar {
    return Polar(r=x.r*y.r, theta=x.theta+y.theta);
}
function * (x :Polar, y:float) :Polar {
    return Polar(r=x.r*y, theta=x.theta);
}
```

Here we have a representation for complex numbers in polar coordinates and examples of defining overloaded definitions multiplication which are selected based on structural constraints.

Operators that have unary forms may have definitions with only a single formal argument while those that have a binary form may have two arguments. The set of unary operators that may be overloaded this way are:

```
~  +  -  not
```

and the binary operators are:

```
**   *   /  &   ^   |
<    <=  >  >=  ==  !=
and  or  #
```

[19]When a "public" function has an invalid up-level reference, we will generate a compile-time diagnostic either as an error as a warning and the treat the function as implicitly private.

6.11 Notes

Conversion Operators The syntax `a:B` has two usages above as an operator above. The first is where `B` is a primitive type and `a` a value of some other primitive type and we want to force a conversion such as converting a floating point number to a string. The second usage is to convert a record or union *lvalue* to have a specific nominal type for the purpose of bound function dispatch.

When `B` is a class, we can define the following semantics. If the value in `a` is a nominal subtype of `B`, then no action is taken. Otherwise, the constructor `B.initialize(a)` is invoked to create a new instance of `B`.

This suggestion is motivated mostly to perform generic initialization such as `0:T` where `T` might be arithmetic or a user defined class.

The bound function case seems like an exception. May be we should use the syntax `x.(B.lookup)` to force invocation of the particular bound function `B.lookup` when it is needed.

Additional Constraints for function Values Do we need to be able to constrain the types of actual arguments when we form a function value:

```
bar => foo(_, :integer)
```

the idea is to constrain the second argument to `foo` to be an integer but say nothing else.

7 Type Parameterization

Chapel permits functions, variables, and types to be specified with *type variables*. Such constructs represent specification of behavior that is parameterized by the specific semantics of data and operations associated with some type. Program fragments that are parameterized this way are said to be *generic*. A program fragment with no type variables is sometimes called *concrete*.

7.1 Requirements

Requirements:

1. Allow functions and types to be parameterized by other types to allow reuse of algorithms specified over value as well as object types.

2. Extend the function overloading mechanism to functions with type variables. In particular, we allow subtype constraints on type variables to allow selection of different implementations for a function.

Features:

1. Use constraints on type variables to allow accurate error messages when a program is not type correct.

7.2 Type Variables

A type variable can be explicitly introduced in any context where a type is expected. The marker "`?`" is used as a prefix to indicate a new symbol is being bound. The scope of this symbol is like a variable: from the point of introduction to the end of the enclosing scope. Type variables can appear in many contexts:

Function argument types A common case for using type variables is in the specification of formal arguments to a function definition. For example:

```
function f(x :  ?t , y :t) :  t { ...  }
```

Here, `?t` introduces a type variable named `t`. Note that this is not a constraint on the actual argument that can be used to determine which definition of `f` will be used. Rather, it is an assertion that the type of `x` is the same as the type of actual argument. The subsequent uses of `t` indicate that both arguments to `f` must have this type. This usage is a constraint on the actual argument `y`, and that constraint is either nominal or structural depending on the actual type. Like `C++ templates`, we expect the implementation of `f` to be specialized to the various specific types for `t` used when `f` is invoked.

Variable declarations Type variables can also be used for variable declarations, as in:

```
var x :  ?t = f(a,b);
```

Where type variables for function parameters are determined by the calling context, types for variable declarations are inferred from the values assigned to the variable. Assuming only the above definition of `f`, we determine that `t` is the same as the type of `a`.

Structured types Inside a `class` or `record`, a type variable can also be specified using an abbreviated type definition:

```
class F {
    type elt_type;
    ...
}
```

Here we declare a type identifier, `elt_type`, but provide no binding. This symbol is a type parameter. It may be used inside the balance of the class specification and within bound functions. Such a parameter may be bound by name in a constructor, as in:

```
var f = F(elt_type=integer);
```

or it may be omitted from the constructor and determined from context.

Element Types Type variables can also appear inside of structural type constraint. For example, we might have:

```
function g(x :  seq(?t)) ...
```

The parameter to function `g` is a sequence type whose elements have type `t`. Type variables can also appear as components in tuples, records, classes, and unions, as well as sequences. If a type `T` has a component type variable `t` then a type constraint can be specified of the form `T(t=?a)` where `a` is bound to the corresponding type value of the actual argument type.

Parameter Variables The "`?`" notation can also be used to name integer parameters in types as well. For example, arithmetic types have `size` as the name of compile-time value that determines the representation width. This value can be given a name as in:

```
function h(x :  integer(size=?k),
           y :  float(size=k)) { ...  }
```

Anonymous Type Variables A type specification may be omitted for variables, function parameters, function return types, and fields. This is equivalent to having an unnamed but unique type variable. The token "`_`" may also be used as a type designator, as in `seq(_)`, which is also treated as an unnamed type variable.

The type specification may be completely omitted in any case where a type variable can be specified. This is equivalent to having the type specified by an anonymous type variable.

Formal type parameters We permit a function to have a parameter that is just a type variable. This is denoted by using `type` as a prefix. Thus:

```
function mkStack(type etype) {
    return Stack(elt_type=etype);
}
```

Here we have a function that has no data parameters but has a type parameter. When a function value is formed, all type parameters must be bound so that the resulting expression has a concrete type.

7.3 Type Constraints

The legal values for a type variable can be constrained by use of a `where` clause. A `where` adds a constraint just like formal arguments to a function:

`where` ⟨*symbol*⟩ `:`⟨*type*⟩`;`

The term ⟨*symbol*⟩ is a previously identified type variable associated with a type definition or a formal argument of a function. Constraints are not permitted on type variables associated with variable declarations. A `where` statement may be placed anywhere in the scope of the type variable it constrains.

These constraints might be either nominal or structural. For example, we might require that it be a union type with particular components or a sequence type:

```
where t = (i :  integer or
           f :  float); where t = seq(?s);
```

The first indicates that `t` is a type union that includes named fields `i` and `f` that are integers and floats. The second requires that `t` be some sequence type whose element type is bound to `s`. When the `where` specifies `record` or `class`, additional requirements may be placed on the fields defined by that type. For example:

```
where t :class {
    var id :  integer;
    function name() :  string ;
}
```

This statement constrains `t` to be a class that has a variable `id` of type integer and a function `name` of no parameters that returns a `string`.

Type constraints are sometimes needed to properly scope certain identifiers, such as enumeration values and type union components. For example, we might have:

```
function f(x :  ?T) {
    where T: record { var name, id };
    use x;
    ...name...;
```

One effect of the type constraint on `T` is to identify that symbol `name` is associated with `x`. Therefore, after the `use` statement, a reference to `name` is interpreted as `x.name`. Only fields that associated with explicit types or implied by explicit constraints are included by the `use` statement.

7.4 Overloading

A function definition involving type variables may be overloaded. In this case, the constraints on the type variables are used like other argument constraints to resolve which function definition is selected. An unconstrained type variable is always considered less specific than any other constraint.

Once a generic function has been selected, the value of the type variable is determined based on the actual argument. This differs from the non-generic case for value types such as records where the type of the formal argument is independent of the type of the actual argument. For example:

```
record ClientData  { var name :string,
                          id   :integer; }
record ClientData2 { var name :string,
                          id   :(integer,integer); }
function print(x :integer) ...
function print(x :(integer,integer)) ...
function print(x :?client)
     where client:  record { var name, id; } {
     call print(x.id);
```

Here we have two similar record definitions that have no structural relationship except a common set of names. The two `print` functions are suitable for the different `id` types, and the generic function `print` has a structural constraint on a type variable that is satisfied by both record types. For a particular call, the type variable `client` might be bound to either of the two record definitions. The type of `x.id` determines which version of `print` to call.

7.5 Type Constructors

The types of variables in generic types must be bound when an object of that type is created. The most direct way is to bind the types by name when the object is constructed. For example:

```
class Client { type id_type; var id :id_type }
...  Client(id_type=integer);
```

Here we specify the `id_type` type variable by name in the default constructor. If there was a non-default constructor it might invoked as:

```
...  Client(id_type=integer).create(...);
...  Client.create(id_type=integer,...);
```

These are equivalent when `create` does not have an explicit `id_type` formal argument.

An explicit constructor may also use a class type variable in the argument list of the constructor. For example

```
constructor Client.new(x :id_type) ...
...  Client.new((4,5))
```

This function definition uses a type variable from the class `Client`. An invocation of this function then provides a binding for that type variable. The example invocation would bind `id_type` to the tuple type `(integer,integer)`.

7.6 Variables with Variable Types

Chapel permits variables to be declared with type variables including simply omitting the type. For example:

```
var id = (4,5);
var x = Client.new(id));
```

The type `id` is inferred to be `(integer,integer)`, which is the type of the initial value assigned to `id`. Using that value in the constructor for `Client` determines the type of that expression, which defines the type of `x`. Explicitly, this would look like:

```
var id :(integer,integer) = (4,5);
var x :  Client(id_type=(integer,integer)) =
        Client(id_type=(integer,integer)).new(id);
```

In the first specification, if we changed the type of `id`, that change would propagate through the next statement.

Chapel allows variables without initializers to be declared with type variables as well. However, if that uninitialized variable is used as an actual argument to a function call, then the implementation may not be able to determine which function is called or the result type. This may generate

an error message at compile time indicating additional type information is required.

Variables in class declarations must have explicit types, although those types may be specified with type variables.[20]

7.7 Example

As an example of the use of type variables, Figure 7 shows a generic description of a stack class and its specialization for integers. The class `Stack` has a type variable `elt_type` indicated by the incomplete `type` declaration. That symbol is also used inside the support methods and those references bind to `Stack.elt_type`, just like other field references in a bound function. The instantiation of an instance of the class provides an opportunity to bind type variables to specific type. That binding allows concrete interpretation of the various object and function definitions.

Another generic function might be:

```
function total(s :  ?List) List.elt_type {
     var t = 0:List.elt_type;
     for e in s.elements
         t += e;
     return t;
}
```

This function is parameterized by a structured type locally referred to as `List` which has an field that designates a type, `elt_type`. It's further assumed the `List` type includes either a sequence or an iterator named `elements`. The syntax of the `for` statement permits either. The `Stack` class in Figure 7 satisfies these requirements so this function can be used with instances of that class when `Stack.elt_type` is bound to an arithmetic type. For example:

```
var t :integer = total(s);
```
– *sum all integers on the stack*

The bindings of `List` to `Stack(elt_type=integer)`.

The type of a variable can be accessed by treating `type` as a field name for that variable. This permits type variables to remain unnamed and the above example can be written:

[20] How are ways we can weaken this if we wanted to?

```
class Stack {
     type elt_type ;
     class Elt {
           var value :elt_type;
           var next :Elt;
     }
     var top :elt_type = nil;
}
function Stack.empty :boole { return top == nil; }
function Stack.push(v :elt_type) {
   top = Elt(value=v, next=top);
}
function Stack.pop :elt_type {
   var v :elt_type = top.value
   top = top.next;
   return v;
}
iterator Stack.elements :elt_type {
   var h :Elt = head;
   while(h != nil) {
       yield(h.value);
       h = h.next;
   }
}
var s = Stack(elt_type=integer);
```

Figure 7: Generic stack and a specialization

```
function total(s) :s.type.elt_type {
     var t :s.type.elt_type = 0:s.type.elt_type;
     for e in s.elements
         t += e;
     return t;
}
```

Here `s.type` is functionally similar to naming the type variable.

7.8 Element Types

The common idiom of parameterizing a collection-oriented data type by a single element type has special syntactic support. The construction `<type>of <type>` is used as short hand binding the second type expression by name as the `elt_type` of the first type expression. For example, we can say:

```
var s :  Stack of integer;
```

which is equivalent to:

```
var s :  Stack(elt_type=integer);
```

7.9 Notes

Unions We might want to expand the type inference to allow the assignment of component types to union types without explicit component identification when that assignment is unambiguous. For example, given:

```
union U  type i:  integer, f:  float;
var x :  U;
```

we might interpret an assignment `x = ⟨e⟩` as this:

```
typeselect(⟨e⟩) {
    when e:integer do x.i = e;
    when e:float do x.f = e;
}
```

but with a requirement that this choice be unambiguous.

Explicit type bindings We might allow introduction of type variables for return values but require them to be fully defined. For example:

```
function total(s) :  ?rtype {
     where rtype :s.type.elt_type;
     var t : rtype = 0;
     for e in s
         t += e;
     return t;
}
```

The use of `where` is simply to allow the variable `rtype` to be used before it is defined.

8 Arrays and Domains

8.1 Requirements

A *domain* is a description of a collection of names for data. These names are referred to as the *indices* of the domain. All indices for a particular domain are values with some common type. Valid types for indices are primitive types and class references or unions, tuples or records whose fields are valid types for indices.[21] Like sequences, domains have a rank and a total order on their elements. An *array* is generically a function that maps from a *domain* to a collection of variables. Chapel supports a variety of kinds of domains and arrays defined over those domains as well as a mechanism to allow application specific implementations of arrays.

Arrays and domains are mechanisms to implement two Chapel goals. Arrays are abstractions of the mapping from sets of values to variables. This is one of the key uses of data structures and by putting additional emphasis on this function including generic syntax support, we will increase the reusability of software. The second issue is the high-level distribution of data collections. This is a topic that we return to in a later section, but by making domains a separate and first class entity, we enable distribution at the collection rather than the object level.

Our requirements are:

1. Support a generic notion of array that encompasses any mapping from a set to variables and includes Fortran arrays as a specific instance.

2. Unify arrays with sequences for the purpose of aggregate expressions.

3. Extend the argument passing and reference mechanisms to handle arrays.

4. Support domains as a first-class concept with specialized syntax for the most common cases.

and a productivity goal is:

1. Support general *sparse* domains that provide both support for sparse linear algebra but also a limited form of generic *set* data type.

[21] This list excludes: references, sequences, functions, domains, and arrays.

8.2 Arrays

An array is declared as:

`var` ⟨*symbol*⟩ `[` ⟨*domain spec*⟩ `]` [`:`⟨*type*⟩] [`=` ⟨*expr*⟩]`;`

or equivalently:

`var` ⟨*symbol*⟩ `array` ⟨*domain spec*⟩ `of` ⟨*type*⟩ [`=` ⟨*expr*⟩]`;`

In each of these cases, ⟨*symbol*⟩ is an array that maps the specified domain to a set of variables holding values of the specified type. For example,

```
var x[1..n] :integer;
```

This defines an array `x` that maps integers in the range of 1 to `n` inclusive to integers variables. Chapel arrays generalize traditional notions of arrays found in C and Fortran. A declaration such as this also implies the creation of new variables that initially have no other references.

An individual variable is accessed by treating the array syntactically like a function and "invoking" it on an element of the domain. This `x(i)` where $1 \leq i \leq n$, is the variable corresponding to element i. Following tradition, the argument is called a *subscript*. Such an expression may be used anywhere a variable reference may be used. When used as the argument to a function, the subscript value is computed once, prior to execution of the function. This is true even for arguments with intent `out` where the selected variable is defined after execution of the function body.[22]

The usual rules for promoting functions with sequence arguments apply to subscripts as well. Thus, the expression "`x(s)`" where `s` has type `seq of integer` will yield a sequence of variable references, one for each index value in `s`. Such sequences may be dereferenced to yield a sequence of values or may be assigned into element-wise. The rank and order of the elements is the same as for the sequence `s`.

When an array is used without a subscript, it is implicitly converted to a sequence of variable references, one per domain element in domain order. The rank, shape and order is the same as for the domain.

[22]Evaluating `out` arguments before the function is invoked is consistent with the general left-to-right evaluation rule but does leave the possibility that if `x` is an array and `x(i)` is used as an actual argument where intent is `out`, then if the function changes `x` in some way then our reference may be stale. This is a problem that arises with *call by reference* argument passing as well.

8.3 Domains

Chapel provides a rich set of primitive domains and techniques to synthesize compound domains from these primitives. Domains are first class concepts in the language in that variables of domain type can be declared and can be fields in classes. Because of their complex structure and special semantics, Chapel provides special syntax for declaring variables of domain type:

[`const`] `var` ⟨*symbol*⟩ `:domain` [⟨*domain spec*⟩]
[`=` ⟨*range information*⟩]`;`

where the details of ⟨*domain spec*⟩ and ⟨*range information*⟩ vary with the specific kind of domain as described below. When ⟨*domain spec*⟩ is omitted, it will be inferred from range information or from assignments to the variable. The optional `const` attribute indicates that range information may not be modified once it is set.

Like sequences, a domain may have a specific bounded set of indices, in which case it said to be *definite* or it may have an unbounded set of potential index values in which case it said to be *indefinite*.

Domains can be used in contexts in which sequences are expected, such as `for` loops. In this case, they evaluate to the underlying set of index values in domain order. For example:

```
var a [D] : float;
var b [D] : float;
forall i in D do
    a(i) = 2*b(i);
```

or

```
[i in D] a(i) = b(i);
```

For indefinite domains, the iteration will be defined to be over the set of indices that is determined by previous usage of the domain as described later.

Index Types For each domain, there is a corresponding *index* type. Such an index type is a subtype of the type of the values used to index the domain and can be used in appropriate contexts. Variables of this type can be declared using the `index of` generic type constructor. In the above example this might be:

```
var j :  index of D;
```

If the indices of `D` are integers, then the type of `j` is a subtype of integer. This is the type of control variables of `for` loops that iterate over domains. We also permit the use of `index` in a parameterized form:

```
var j :index(D);
```

which is equivalent to the above.

Each domain variable includes an `index` method that maps values into instances of the corresponding index type. For example:

```
j = D.index(k)
```

Values of index type are known to be valid and may have specialized representations to facilitate accessing arrays defined for that domain. It may therefore be less expensive to access arrays using values of appropriate index type rather than values of the more general type the domain is defined over.

Arithmetic Domains An arithmetic domain has tuples of integers as the index type. The values of these tuples are constrained to lie within a bounding box with strides specified as a tuple of arithmetic sequences. At the point of declaration, we permit either the rank of the domain to be specified or the range information. The latter is specified via an arithmetic index set of like rank. For example

```
var w :  domain(1);
var x :  domain(1) = 1..n;
var y :  domain = 1..n;
var z :  domain(1..n)
```

All these define domains that are indexed simply by integers. The first does not specify the bounding information while the second and last do. The last three are equivalent.[23] Other examples of arithmetic domains are:

[23]It might be argued that the definition of `y` is ambiguous since we could also use an indefinite domain `domain(:integer)`. How should this be resolved?

```
var x1 :  domain(3); – rank-3
var y1 :  domain(2) = (1..n, 1..m);
var z1 :  domain(1..n, 1..m, 1..k by 2);
```

The shape of arithmetic domains is the same as shape of the cross-product of the arithmetic sequences that define the bounding information. The rank of an arithmetic domain must be a compile-time constant (`parameter`).

The bounding information for an arithmetic domain is called its *range*. The range can be specified by initializer, by assignment to the domain variable, or by an explicit call to `set_range` with such a value. This information can be explicitly accessed via the `range` method as well. Thus `y.range` for `y` as defined above is the value of type `index(1)` denoted `1..n`. This can be changed by assignment or by `set_range` method:

```
y = 2..m;
call y.set_range(3..100);
```

The difference between these is described below in section 8.5.

Assignment between two domain variables is defined to be a transfer of range information. Thus an assignment, `D1 = D2`, is interpreted as:

```
call D1.set_range(D2.range);
```

This is true for function argument passing as well, and it applies to other classes of definite domains described below.

Enumerated Domains Enumerated types and the type `boole` can also be used to define a domain. For example:

```
enum Colors { red,blue,green,yellow }
var x :  domain(Colors);
var y :  domain(boole);
```

Here `x` is a domain whose indices are values from the enumerated type `Colors` and `y` is a domain indexed by values `false` and `true`. The left-to-right order of enumeration values defines the domain order and `false` is before `true`. Such domains have no separate range information.

Indefinite Domains Scalar types are primitive types, enumerated types, class references, and tuples, unions, or records of scalar types. Any scalar type may be used to define an indefinite domain. Arrays of such domains are simply dictionaries mapping from values to variables. For example:

```
var people :  domain(string, integer);
var age [people] :integer;
...
age('fred', 4) = 42;
```

Here `people` is a domain whose index values consist of a string and an integer. There is no bounding information for such a domain and the elements of the domain vary as the program evolves. A new index is logically added to such a domain when an array element corresponding to that index is defined. In the example above, `('fred',4)` is added to `people` as an effect of the assignment to `age`. Index values can also be added with an explicit `add` method as in:

```
call people.add('fred',4);
```

The `add` method can be invoked with components of the tuple type, an instance of the tuple type or a sequence whose elements have this tuple type.

When an indefinite domain is used in a context where a sequence is expected, then the sequence of values is determined by the set of index values used so far in the program.

```
for p in people do
    age(p) += 1;
```

The order of elements in an indefinite domain is not defined but is consistent as long as no elements are added to or removed from the domain.

Indefinite domains support a `remove` method that removes index values from the domain. This method has a single argument that is either a value of index type or a sequence of such values. When an indefinite domain is defined over a tuple type, then `remove` may also be invoked with arguments corresponding to the components of the tuple:

```
var p = ('fred', 4);
call people.remove(p);            – remove an element
call people.remove('sally', 2);   – specified as components
```

It is legal to call `remove` when its argument identifies a value not currently in the index set. Such an operation has no effects.

For indefinite domains, the `+=` and `-=` operators are simply sugar over calls to `add` and `remove`. The right hand operands may also be either values or sequences of values of suitable scalar type:

```
people -= ('fred', 3); – remove an element
people += ('fred', 4); – add an element
```

An indefinite domain also supports a `member?` function that can test whether a particular value is part of the index set. It can also be invoked with either a tuple or the components of the tuple and it returns a `boole`. For example:

```
var s :  seq of (string, int);
var new = [p in s] if (not people.member?(p)) then p;
...
if(people.member?('sally', 3)) ...
```

Opaque Domains An *opaque* domain is a form of indefinite domain where there is no algebra on the types of the indices. These are denoted by the keyword `opaque` in the domain specification:

```
var D : domain opaque;
```

New index values for this domain are explicitly requested via a *new* method. For example

```
var d [1..n] :index of D = D.new();
```

Opaque domains permit more efficient implementations than indefinite domains under the assumption that creation of new domain index values is rare.[24]

[24]They also don't have to support the `index` method.

The `remove` method can be invoked for a single index value or a sequence of index values. Those values will be removed from the domain and any values associated with them reclaimed. Given the definitions above, we might remove the first 5 elements created above: `D.remove(d(1..5))`

Sub-domains A sub-domain is a domain whose values are elements of a *base domain*. A sub-domain is specified by identifying the base domain in the ⟨*domain spec*⟩. For example:

```
var D : domain (1..n,1..m);
var Interior :  domain (D) = (2..n-1, 2..m-1);
```

Sequencing of the index values in the sub-domain is consistent with the order of those values in the base domain.

The index values for a sub-domain are a subtype of the index values of the base domain and can be used where base domain index values can be used. To convert from base domain index to sub-domain index requires explicit use of the sub-domain `index` method: For example:

```
var d = D.index(4,4) ;       – base domain index
var i = Interior.index(d); – sub-domain index
var A [D] : float;
var B [I] : float;
   ...  A(i) ...             – valid, i is a subtype of d
   ...  B(d) ...             – valid, but perhaps more expensive
```

The last expression is valid because `d` is a subtype of `(integer,integer)` and such a tuple can be used to index into a domain but there may be greater cost to verify that it is a valid index value.

In the case of arithmetic domains, the sub-domain range information may be specified as a separate strided bounding box. That box may involve indefinite sequences which are constrained by the bounding box of the base domain.

Product Domains There are two ways to build compound domains from base domains. One is to form a *regular product domain* where index values in the product are tuples of index values from the component domains. The syntax for this uses tuple notation for the component domains:

⟨*domain spec*⟩ = ⟨*domain spec*$_1$⟩ , ..., ⟨*domain spec*$_k$⟩

where an example might be:

```
var A : domain opaque ;
var B : domain (2);
var C : domain (A,B);
```

Here C is a domain consisting of tuples of values where the first term is an element of A and the second term is an element in B. In the case of regular products, if x and y are index values of A and (x, i) is an index in C then (y, i) is also an index in C. Arithmetic domains with rank greater than 1 are merely special cases of regular product domains.

Range information for a regular product domain is specified as a tuple of the range information for the component domains for which range information may be specified.

The syntax for irregular domains uses the array syntax:

⟨*domain spec*⟩ = [⟨*domain spec*⟩] ⟨*domain spec*⟩

where an example might be

```
var D : domain [A] B;
```

where the interpretation of this declaration is that the domain D is an array of sub-domains of B. Each of those sub-domains may have different subsets of the base domain B. Here, the domain A is referred to as the *outer domain* and B is called the *inner domain*.

Range information for irregular domain consists of a tuple where the first component is the range information for the outer domain, A in this example, and the second is a sequence of range information for the inner domains. The set_range method may be called with just the first component of these, after which subscript notation can be used to set ranges for element domains:

```
call D.set_range(A range information);
call D(i).set_range(B range information);
```

set_range may also be called with complete information. For example:

```
var Triangle :  domain [ (1) ] (1);
Triangle.set_range(1..n, [i in 1..n] (1..i));
```

This declares a domain that is a rank-1 array of rank-1 domains. The arguments to `set_range` first define the outer domain and then each domain in that outer domain. When the outer domain is `opaque`, the range for that domain consists of the number of elements.

The order of elements in a product domain is the lexical-graphic order induced by the orders of the component domains.

Sparse Domains A sparse domain is a general kind of sub-domain where the set of sub-domain index values is arbitrary rather than structured as in the arithmetic case. A sparse domain is indicated by a `sparse` keyword and the set of index values may be specified by explicit enumeration. For example:

```
var Node :  domain (1..n) ;
var active :  seq of (Node, Node);
...
var Edges :  domain sparse (Node,Node) = active;
var weight [Edges, 1..3] :real;
call Edges.add(...);
call Edges.remove(...);
```

Here `Edges` is a sparse sub-domain of the regular product domain consisting of pairs of integers in the range 1 to `n`. For each element in `Edges`, there is a short 3-vector of floating point values designated by `weight`. These variables are only defined for elements in the sparse sub-domain, not for all elements in the dense base domain. The range information is specified as a sequence of pairs of base domain values. When the range information changes, so does the places where `weight` variables are allocated.

The operators `+=` and `-=`, when their target is a sparse domain, are interpreted as adding or removing elements of the domain respectively and are simply a different way to invoke the `add` and `remove` methods.

Anonymous Domains An array declaration can identify a domain by name or as a domain specification. In the later case, specification are permitted corresponding to regular products of definite domains, and previously

defined domains. This can be nested to indicate irregular products. Some examples are:

```
var w [1..n,1..m] :integer;
var x [1..n, Colors] :float;
var D :domain ...;
var y [1..n, D] :float;
var z [i in D] [1..f(i)] :float;
```

The domain for variable `w` is is the rank-2 arithmetic domain with the specified bounding information. The domain for `x` is the product of the rank-1 domain with specified range and the enumeration domain implied by `Colors`. The domain for `y` is the regular product of the rank-1 domain and the domain `D`. The domain of `z` is the irregular product of `D` and the collection of rank-1 arithmetic domains with specified range information. These anonymous domains implicitly have the `const` attribute and may not be modified.

8.4 Domain Arguments

When a formal argument of a function is specified as being a domain, then the actual argument must also be a domain. In this case, the actual is not treated as a sequence. Rather, the formal is bound to a reference to the actual as is done for instances of classes. If an explicit ⟨*intent*⟩ is specified, then a new domain is created and range information is communicated between formal and actual. For example:

```
var d :  domain (1); function foo( x :  domain) {
    ...
    x = 1..n;
    ...
}
function bar(in x :  domain) {...}
foo(d);
bar(d);
```

Function `foo` receives a reference to `d` and so the assignment to `x` will change the range information for `d`. Variable `x` in variable `bar` is a separate domain that is initialized to the range information from `d`.

8.5 Array-Domain Association

When the ⟨*domain spec*⟩ of an array identifies a specific domain variable, then the array is said to be *associated* with the domain. This association means that changes in the set of names in the domain are reflected in the set of variables mapped by the array and any array aliases. Sub-domains are similarly associated with the base domain and changes in the base domain may remove elements from the sub-domain.

When an arithmetic domain is specified without bounds information, or a sparse domain is specified without an initial set of elements, the domain is said to be *unallocated*. This information may later be specified using the `set_range` method of the domain after which the `allocate` method may be called. No variables are created for arrays when they are associated with unallocated domains. When the `allocate` method is invoked, variables are created for these arrays.

Arrays that are associated with unallocated domains may not have initializers[25] and may not refer to the elements unallocated domains in the specification of range information in irregular products.

The range information for a domain may be changed by a call to `set_range`, or for sparse domains by calls to `add` and `remove`. After range information has been changed, the method `reallocate(preserve_data:boole = true)` may be called to update the associated arrays. The argument to this methods determines whether existing values in those arrays are preserved where possible. When range information for a domain is modified by assignment to the domain, this is equivalent to a call to `set_range` followed by a call to `reallocate(preserve_data=true)`.

In this context, when we preserve data values, it means that any value that was a valid index before the reallocation and remains a valid index after the allocation will maintain its value, yet the index may refer to a different variable.

If the base domain has not been allocated when the sub-domain is declared, then the sub-domain is not allocated. After the base domain is allocated, a separate `allocate` method must be invoked on each sub-domain. A `reallocate` method must be used on each sub-domain when the base domains change.

[25] If the type of the elements of the array have a default initial value, that will be used when the array is allocated.

8.6 Expressions and Domains

In an expression context, a reference to an array `X` over domain `D` is treated as the expression: `[i in D] X(i)`. In this last expression, the reference to `D` is used to indicate a sequence of all of the indices of `D` in their natural order.

A bare "_" operator may be used as a subscript for a domain or an array defined over a domain. This is interpreted as selecting all legal values for the subscript position regardless of the index type for that domain. A domain or array reference involving such a selection is called a *slice* of the object.

A bare "*" operator may also be used as a subscript for references to domains and arrays but only in iterator control contexts. This operator is used to identify a set of non-empty slices by filtering an input sequence. Here is a simple example:

```
var D : domain sparse [A] B;
for i in D(*,50) do
    ...
```

The values assumed by `i` are those where `(i,50)` is a valid index in `D`. We call this a *projection* of `D`.

This example is generalized as follows: consider an expression of the form `X[`$i_1, \ldots, i_p$`]` where some of these index values are "*". Of the remaining subscripts, some are scalar and some are sequence expressions. Let S denote the cross product of those sequences. The effect of this expression is to filter `S` such that replacing the "*" with "_" and the sequence subscripts with the value from `S` yields an expression that is a non-empty sequences. We generalize the previous example to:

```
var cols :  seq of B;
for i in D(*, cols) do
```

Here, `i` assumes a sequence of values that would be returned by:

```
[j in D] if any([c in cols] D.member?(j,c)) then j
```

The initializer expression for an array declaration is a sequence expression determined by the underlying domain for iteration control. Like other domain references, element values may be given names for use in the expression context. For example:

```
var D : domain(1..n);
var iota [i in D] : integer = i;
class F {
    constructor new();
}
var objects [D] = F.new();
```

Here we define a domain `D` and two arrays over that domain. The first is an array of integers where symbol `i` is bound to each element of `D` as we evaluate the initializer expressions. The array `objects` holds references to instances of class `F`. The constructor `F.new` is invoked once per element of `D`, but this time we don't need a name for that value.

Domains of expressions .

8.7 Array Aliasing

An array may declared that reuses variables from another array. The syntax for this is:

`var` ⟨*symbol*⟩ [`[`⟨*domain spec*⟩`]`] `=>` ⟨*expr*⟩`;`

Here ⟨*expr*⟩ must evaluate to an array reference or a sequence of variables that has the same shape as ⟨*domain spec*⟩. Those variables are not evaluated, rather a new mapping to those variables is established. This new mapping is called an *alias* since the same variable is now known by different names. For example:

```
var xi[1..98] => x[2..99];
```

now `xi(2)` is the same variable as `x(3)` but `xi` has a different domain than `x`. If the ⟨*domain spec*⟩ is omitted, then it is assumed to be the same as the domain of ⟨*expr*⟩. In the case of arithmetic domains, an indefinite arithmetic sequence may also be used as long as bounding information for each dimension of the domain is well-defined.

The implementation of such an alias may be done by translating index values from the target domain into index values in the source domain and then using the source array. This will be done whenever the source and

target domains are simple cross-products of arithmetic sequences, possibly transformed by `translate`, `spread` and `reshape` operations. In this case we use the term '*view*" to describe the relationship. The keyword `_view` may precede the keyword `var`. This will generate a compile-time error if a view can not be used for such an implementation.

When a view is not used, then the alias is implemented as an array of references declared over the target domain.

8.8 Array Arguments

Aliasing is the argument passing mechanism when arrays are formal argument of a function. The formal argument may have a different domain from the actual or may inherit its domain from the actual argument. The extended syntax for arguments includes:

$$[\langle intent \rangle]\ \langle symbol \rangle\ \texttt{[}\ [\langle param\ \ domain \rangle]\ \texttt{]}\ \texttt{]}\ [\texttt{:}\langle type \rangle]$$

where $\langle intent \rangle$ now refers to the variables that are referred to by the domain. When an $\langle intent \rangle$ is specified as `in`, `inout`, or `out`, the new variables are created and values are copied to or from the actual argument as for scalars. If the intent is `const`, then no copies are made and as for scalars, assignment to array elements is not permitted. Unlike scalars, assignment to array elements to do not require specification of intent since the default is to pass elements "by reference".

If $\langle param\ domain \rangle$ is or has the form `?` $\langle symbol \rangle$, then the domain specified in the actual argument is used here. Given:

```
function foo(f:[?D]) ...
var D1 :  domain(2) ;
var x [D1] :  integer;
call foo(x);              – f => x[D1]
call foo(x[1..n, 1..m]);  – f => x[1..n,1..m]
call foo(x[1..n,2]);      – f[1..n] => x[1..n,2]
```

The first call to `foo` will associate `D` with `D1`. In the second `D` will be a sub-domain of `D1` corresponding to the specification range. In the third case, `D1` is a rank-1 arithmetic domain with range `1..n`.

A ⟨*param domain*⟩ may be specified as a domain variable from the enclosing lexical context or from a formal argument. Such a specification requires that the actual argument conform to that domain:

```
var D : domain(2) ;
function foo(f :[D]) ...
var D1 :  domain(2);
var x[D1];
call foo(x) ;              – f[D] => x[D1];
```

In these cases, an alias is established between the formal argument and the actual and no new variables are instantiated. A `_view` keyword may precede the ⟨*intent*⟩ and represents a requirement that argument passing for this argument be implemented with a view.

If a sequence is an actual argument passed to a formal array argument, the domain of the actual is determined by the shape of the sequence. This will be the domain of integers or integer tuples that are value subscript of the sequence.

8.9 Array Functions and Objects

An array may be defined by associating a domain with a function. The syntax for this is similar to other function definitions:

`function` ⟨*symbol*$_1$⟩ `[` ⟨*symbol*$_2$⟩ `:`⟨*domain*⟩`]` [`:`⟨*type*⟩] ⟨*block*⟩

The square brackets indicate an array definition. ⟨*symbol*$_1$⟩ is the name of the array and ⟨*symbol*$_2$⟩ a variable of domain index type that is local to the function and referred to as the formal argument. The term ⟨*domain*⟩ identifies a domain variable in the enclosing lexical context. ⟨*type*⟩ is the type of values stored in the array and ⟨*statement*⟩ computes an instance of that type or a variable of that type.

Note that ⟨*domain*⟩ is a variable and while it is specified like an argument type, a reference to it is part of the function value bound to ⟨*symbol*$_1$⟩, This reference is consulted when a function is used in an aggregate context.

When ⟨*domain*⟩ has a rank greater than 1, then ⟨*symbol*$_2$⟩ may be replaced with a comma separated list of symbols that will be bound to the components of the tuples that make up the actual argument. In this case, the actual argument may be a tuple or a list of values of component types.

8.10 Notes on Arrays and Domains

Default Values Maybe an initializer for an indefinite domain can be interpreted as the value associated with a newly created element if it has not been previously defined.

```
var dict [ string ] = '';
```

We also want to allow such values to be specified for sparse domains even when the sparse domain is unallocated.

Another approach would be to use the default value of the underlying type. For example:

```
type String :string = ''; var dict [ string ] :  String;
```

In this case, `dict[s]` returns `''` when `not dict.member?(s)`.

Distribution for indefinite domains This might be done with hashing function defined over the infinite space. Load balancing might become an issue...

Implicit Subscripts We want a mechanism to allow subscripts to be implicit when we iterate over domains.

```
[S] A = B;
```

where if `A` or `B` is defined over `S`, a subdomain of `S` or a domain that is a product where a term is `S`, then we use the "current value" of the iteration to select the appropriate slice of the array.

Other things

1. How to specify non-represented implicit value in sparse arrays
2. Some sort of ability to associate an "out-of-bounds" function with an array which can print errors or implement boundary conditions.
3. More information on array subscripting, especially for arrays of arrays – can we use integers, tuples, tuples of integers, do they have to correspond, what does it mean to partially subscript an array, etc.

4. I'd obviously like to see some sort of concept similar to ZPL's flood/grid dimensions, especially if we allow implicit subscripts.

5. reductions/scans on arrays

6. Along with the implicit subscripts, I think some sort of relative indexing would be very convenient to keep expressions shorter, sweeter:

   ```
   [(i, j) in S] A(i, j) = B(i-1, j) + B(i+1, j) + B(i, j-1) + B(i, j+1)
   ```

   or

   ```
   [ind in S] A(ind) = B(ind+(-1,0)) + B(ind+(1,0)) + B(ind+(0,1)) + B(i
   ```

   vs.

   ```
   [S] A = B@(-1,0) + B@(1, 0) + B@(0,-1) + B@(0,1)
   ```

   it's sugary, but very convenient, I think.

7. can one index into arrays using arrays? Is the interpretation zipper vs. cross-product? (I suspect we support both depending on the style of brackets used?).

9 Parallelism and Synchronization

Chapel is an explicitly parallel programming language. Parallelism is introduced into a program via three constructs and is coordinated via a small number of synchronization mechanisms. To avoid any unintended implications, we use the terms "computation" and "sub-computation" to refer to distinct, concurrently executing portions of a program.

9.1 Parallel Expressions

A sequence expression of the general form:

```
[i in S] f(i)
```

where `S` is a sequence, array, or domain is normally executed in parallel even though the results have a well defined sequential order. Unordered `reduce` and `scan` operations are also parallel by nature.

An assignment statement such as:

```
X(S) = Y;
```

where `X` is an array and `S` a sequence of index values is also executed concurrently in an element wise manner. If there are duplicate values in the sequence `S`, then the order of writes to the selected variables of `X` is non-determined.

The `ordered` keyword can be used as a unary operator to suppress parallel execution of a sequence expression that can involve side-effects to memory. This expression:

```
...  ordered f(⟨s1⟩
```

where `f` is some function and $\langle s \rangle$ a sequence will evaluate each instance of `f` one at a time and in sequence order. The `ordered` keyword does not inhibit parallelism inside of `f`.

9.2 Parallel Statements

Forall Statements Chapel supports a variant of `for` that allows concurrent execution:

`[ordered] forall` ⟨*var*⟩ `in` ⟨*expr*⟩ ⟨*block*⟩

Here ⟨*var*⟩ is either a symbol or a tuple of symbols and ⟨*expr*⟩ evaluates to a sequence of suitable type. This statement evaluates ⟨*block*⟩ once for each element in the sequence. The evaluation of each of these statements can be executed concurrently and is considered a separate computation.

The ordered keyword has no effect if ⟨*expr*⟩ is a sequence value but affects order of evaluation when ⟨*expr*⟩ is an iterator.

Without the `ordered` keyword, the evaluation of ⟨*expr*⟩ can proceed concurrently with evaluation of the statements as well. Alternately, ⟨*expr*⟩ could be fully evaluated before any ⟨*block*⟩ is evaluated.

When the `ordered` keyword is present, the only concurrency between statements is the concurrency that is explicitly specified in the iterator. This allows an iterator to not only define a sequence of values, but to impose a partial order on that sequence. An example will follow.

In either case, control continues with the statement following the `forall` only after all statement instances have been completely evaluated.

Control transfers such as `goto`, `break`, `continue`, and `return` are not permitted either into or out of the body of a `forall` statement. Return statements are also not permitted inside a `forall`. A `yield` statement is permitted, however.

Cobegin Statements A second form of parallelism is the `cobegin` statement which has syntax:

`cobegin` ⟨*compound statement*⟩

Here, every statement in the list that makes up the ⟨*compound statement*⟩ is executed concurrently with every other statement and is considered a separate computation.

Control continues after all of these statements have been evaluated. As with a `forall`, control transfers are not permitted either into or out of the body of a `cobegin` statement.

Begin Statements The `begin` statement is an unstructured way to create a new computation executed only for its side-effects. This is the syntax:

`begin` ⟨*block*⟩ `;`

The specified statement is executed in parallel with the balance of the initiating computation which continues with the statement following the `begin`. Control transfers in to or out of ⟨*block*⟩ are prohibited, including `return` and `yield` statements.

Serial Statement The `serial` statement is used to control the width or safety of a parallel computation. It has this syntax:

`serial` ⟨*expr*⟩ ⟨*block*⟩ `;`

where ⟨*expr*⟩ evaluates to a boolean value. Regardless of that value, ⟨*block*⟩ is evaluated. If the value evaluates to `true` however, any dynamically encountered `forall` or `cobegin` statement is executed sequentially. For example, we might have a recursive tree iterator that uses tree height to determine where concurrency should be used:

```
class Tree {
    var is_leaf :  boolean;
    var left :  Tree;
    var right :  Tree;
}
iterator Tree.walk {
    if(is_leaf) yield(this);
    else
        serial(height <= 10)
            cobegin {
                yield(left.walk);
                yield(right.walk);
            }
}
```

This iterator could be used in conjunction with an `ordered forall` to aggregate work to avoid parallelism overhead.[26]

[26]My expectation is that we clone functions that may be executed in a serial context so we can avoid the overhead of testing and suppressing parallelism.

TODO: Specialized Scheduling

9.3 Synchronization

Synchronization coordinates accesses to shared variables. One form of synchronization is the execute order requirements associated with `forall` and `cobegin` statements. Here introduce synchronization directly associated with variables. We introduce two kinds of *synchronized variables* whose semantics include execution order and memory consistency requirements.

Single Assignment Variables Single assignment variables are declared by adding the keyword `single` at the beginning of a variable declaration. For example:

```
single [var] x;
```

Such variables may only be defined once during their dynamic lifetime. Any reference to the variable before it is defined causes the computation to suspend execution and wait for the value. Thus a recursive algorithm to sum values in a tree might appear as follows:

```
function Tree.sum() {
    if(is_leaf) return value;
    single var x ;
    begin x = left.sum;
    var y = right.sum;
    return x+y;
}
```

While a `cobegin` might be a more suitable formulation, this fragment creates an asynchronous computation to compute the sum of the left sub-tree while the main computation continues with the right sub-tree. The final reference to `x` will be delayed until the assignment completes and that value will be used as a summand.

A `single` variable can have `record` type but in this case all fields must be assigned at the same time. Assignments to individual fields are not permitted.

When a `single` variable has an initializer, the evaluation of that initializer is implicitly performed as an asynchronous computation. Thus:

```
single var x = left.sum;
```

is equivalent to the above where the declaration and assignment are separated.

Any variable declaration in a `cobegin` is implicitly treated as a `single` variable for references in other statements of the `cobegin`. The above example might then be written as:

```
function Tree.sum() {
    if(is_leaf) return value;
    var z;
    cobegin {
        var x = left.sum;
        var y = right.sum;
        z = x+y;
    }
    return z;
}
```

The assignment to `z` waits for `x` and `y` to be available and then produces the value `z`.

`sync` Variable A synchronized variable generalizes the single assignment variable to permit multiple definitions. Synchronized variables are declared with the attribute `sync`, as in this example:

```
sync [var] buffer ;
```

A sync variable is logically either defined or not. When it is not defined, computations that attempt to read that variable are delayed until it becomes defined by the next assignment to it, which atomically changes the state to defined. When the variable is defined, a use of the variable consumes the value and atomically transitions the state back to undefined. If there are multiple computations waiting, one is non-deterministically selected to receive the value and the rest wait for the next value. If a computation attempts to assign into a `sync` variable that is currently defined, the effect of that assignment is delayed until the variable becomes undefined again. If there are multiple computations attempting such an assignment, one is non-deterministically selected and the result continue to wait.

A `sync` variable can hold a value of `record` type but in this case the entire record must be read and written. Reads and writes of individual non-`const` fields are not permitted.

`sync` variables allow a sequence of values to be communicated between computations using a single shared variable. They also can be used as building blocks for more traditional synchronization primitives such as semaphores and monitors.

Synchronized variables support a number of special methods. These include:

```
sync var s;
s.purge          – force s to be undefined.
s.read           – wait for the variable to be defined and
                 – return the value but don't reset to undefined
s.write(x)       – force s to be defined with value 'x' even if it
                 – is already defined.
```

Memory Consistency The Chapel implementation permits holding the values of variables in alternate locations which we will refer to as *caches*. The implementation may *prefetch* by copying a value from a variable to a cache before an access to that variable is executed. It may store a value into a cache after it is assigned but not write it to the variable in which case we say the cache is *dirty*.

There are restrictions on caching to ensure effective communication between concurrent computations. These restrictions are defined in terms of *input* and *output synchronization points*.

An input synchronization point occurs when a computation performs an operation, such as reading a synchronized variable, that may cause that computation to be delayed. This also includes waiting for sub-computations associated with `forall` and `cobegin` statements. Any cached value of a variable that might have been modified by concurrent computation must be treated as invalid at the input synchronization point. This is called *invalidating* the cache.

An output synchronization point occurs when a computation performs an operation that may enable another computation to begin or resume execution. This includes starting a `forall` or `cobegin` or accessing a synchronized

variable. Any dirty value of a variable that might be accessed by another computation must be *flushed* from cache back to the location of the variable.

Chapel does not guarantee communication of values between concurrent computations unless there is appropriate synchronization to coordinate producer and consumer.

Because of their role in coordinating threads, synchronized variables can not generally be cached.

Unordered Variables Both *single* and *sync* variables can be further attributed as *unordered*. This attribute suppresses the input and output synchronization points associated with accesses to these variables. Use of this attribute therefore increases the effectiveness of caching by allowing more state to be preserved in various caches. Unordered variables are appropriate when the variable will hold the entire effect associated with a subcomputation. For example:

```
function Tree.sum() {
    if(is_leaf) return value;
    _unordered single var x = left.sum;
    var y = right.sum;
    return x+y;
}
```

The `_unordered` attribute on `x` allows any cached state to be retained when the value of `x` is subsequently required.

9.4 Atomic Transactions

An *atomic transaction* is a region of the program that appears to execute as if all other computations in the program are suspended. It is indicated as:

`atomic` ⟨*block*⟩ `;`

where ⟨*block*⟩ is executed as if it were serialized by a `serial` statement.

The behavior of an atomic statement is defined operationally in terms of acquiring ownership of variables. Each variable in the program may have an *owning* computation. Inside an atomic statement, before each variable reference, the current computation attempts to acquire ownership of the

variable. If the variable is owned by another computation, a deadlock might occur if the computation naively waits. At such a point the implementation may *abort* the statement. This means we release ownership of all variables owned by the current computation without changing any of their values, and we restart execution at the beginning of ⟨*block*⟩. Visible side-effects to variables are delayed until ⟨*block*⟩ completes. At this point we say the atomic statement *commits*, and all variables it owns are released and any modifications to variables are made.

The implementation of the atomic statement must insure forward progress, but the details of how ownership information is maintained and when computations are aborted is not defined here.

Here is an example of an atomic transaction:

```
var found = false;
atomic {
    if(head == obj) {
        found = true;
        head = obj.next;
    }
    else {
        var last = head;
        while(last != null) {
            if(last.next == obj) {
                found = true;
                last.next = object.next;
                break;
            }
            last = last.next;
        }
}
```

Inside the atomic statement is a simple sequential implementation of removing a particular object denoted by `obj` from a singly linked list. This is an operation that is well-defined, assuming only one computation is attempting it at a time. The atomic statement insures that, for example, the value of `head` does not change after it is first in the first comparison and subsequently read to initialize `last`. The variables eventually owned by this computation are `found`, `head`, `obj`, and the various `next` fields on examined objects.

The effect of an atomic statement extends into called functions. Thus if we have some method associated with a list that removes an object, that method may not be parallel safe but could be invoked inside an atomic statement for safety:

```
var found ;
atomic found = head.remove(obj);
```

Chapel defines three operations that are used to optimize atomic statements to avoid overheads. They are the `_invariant`, `_private`, and `_release` operations. The first two are similar in that they identify variables that are either known to be invariant over a program interval, or known to not be accessed by any other computation. In either case, the protocol to assert ownership and delay side-effects can be avoided. The `_release` operator identifies a variable currently owned by the computation that has not been modified and will not be accessed again. The current computation's ownership is revoked and another computation can assert ownership.

These operations might be used to tune the above code to reduce overheads. For example:

```
var found;
atomic {
    _private found = false;
    if(head == _invariant obj) {
        found = true;
        head = _invariant obj.next;
    }
    else {
        var last = _release head;
        while(last != null) {
            if(last.next == obj) {
                found = true;
                last.next = _invariant obj.next;
                break;
            }
            last = _release last.next;
        }
    }
}
```

These avoid overheads on the computation's private variables `obj` and `found`. It also allows a list-removal operation to be pipelined by releasing ownership of the list head and various link fields as soon as they will no longer be referenced. The assertion that `obj.next` is invariant reflects the fact that no other variables need to be acquired to allow the transaction to complete, not that the field is actually invariant. The implementation is free to prove these attributes and apply them automatically. For example, we would expect the implementation to identify `last` as a private variable without programmer assertion.

Chapel allows variables to have an `atomic` attribute to allow optimization of ownership. For example:

```
class ObjType {
    atomic var next :  ObjType;
    ...
}
```

This declaration treats a definitions of such fields as if they are in short atomic sections. Thus, a statement of the form:

```
e1.x = e2;
```

where `x` is atomic would be evaluated as:

```
var t1 = e1;
var t2 = e2;
atomic (_invariant t1).x = _invariant t2;
```

The declaration also encourages the implementation to allocate any extra storage needed to maintain ownership adjacent to the variable itself, avoiding the overhead of mapping the variable to sparsely maintained information.

9.5 TODO

Refer to previous discussion about concurrency execution of sequences expressions. If they are parallel, do we need a way to serialize?

Can we garbage collection computations associated with single variables when the single variable is otherwise unreferenced?

Do we need a mechanism to name computations like we name objects? If so, we can we do with such a name? Maybe use it for hashing to test for equality so we can determine nested lock acquisition.

10 Locality and Distribution

This section discusses mechanisms available in Chapel to exploit locality by allowing a programmer to describe affinity between data and computation. This is accomplished by associating both data objects and computations with abstract *locales*. To provide a higher-level mechanism, Chapel allows a mapping from domains to locales to be specified. This mapping is called a *distribution* and it guides that placement of variables associated with arrays and the placement of subcomputations defined over the domain.

10.1 Requirements

Here the basic requirements and target features for locality and distribution mechanisms. Requirements:

1. The programmer must be able to express the affinity of threads and data.

2. The placement of data objects is not part of their type and in general have no impact on the meaning of any expression that manipulates the object.

3. The programmer must be able to decompose a collection of objects across a collection of locales. Specifically, a domain may have a *distribution* associated with it which is a mapping from index values to locales.

4. The distribution of a domain influences the decomposition of arrays defined over that domain and the computations that access those arrays. These influences can be overridden by programmer direction.

We use the term *local* to refer to data objects that are associated with the locale that a computation is running on and *remote* for data objects that are not. We assume that there is some overhead associated with accessing data that may be remote compared to data known to be local.

Features:

1. Default strategies should be available for distribution of domains where the programmer simply suggests decomposition is appropriate.

2. A programmer should be able to determine the locale with which an object is associated, the locale on which a computation is running, and determine whether these are the same.

3. It should be possible to change the distribution of a domain while preserving all other aspects, including the values stored in arrays. We expect to use facts about the structure of a data distribution to improve performance, and will limit the scope of changes allowed to distributions to facilitate optimization.

4. A protocol should be available to allow caching of remote data objects for local use. This protocol should be integrated with the distribution mechanism, but also be available for less structured use.

5. It should be possible for a programmer to assert that data is local.

10.2 Locales

Chapel provides a predefined data type called *locale*. Both data and computations can be associated with an instance of this type. The only operation defined for this type is equality comparison.

A predefined configuration variable defines the *execution environment* for a program. This environment is defined by these variables:

```
const config num_locales:integer;
const Locales [1..num_locales] :locale;
const Global :locale;
```

Notice that these are constants and are not subject to change during execution of the program. The variable `Global` holds a special value of `locale` type that can be distinct from the values stored in `Locales`. This value is used to denote an object or computation that has no defined affinity.

Every variable will be associated with some locale which can be queried as $\langle v \rangle$`.locale`. When $\langle v \rangle$ is a reference type or has class type, the locale is where the referenced variable or object is located rather than where $\langle v \rangle$ may be located. Every computation will also be associated with some locale and this can be queried with the function `this_locale`.

10.3 The `on` Statement

The most direct tool available to a programmer is to assert that a computation identified by a statement should be executed on a particular locale. This is the syntax:

`on` ⟨*locale spec*⟩ [`do`] ⟨*statement*⟩

Here ⟨*locale spec*⟩ is either a value of `locale` type, such as `locales(1)` or a variable that is implicitly mapped to its `locale`. The evaluation of ⟨*statement*⟩ is done in the specified locale. After that execution, the containing computation will continue in the same locale as before executing the statement. If `locale` equals `Global` then ⟨*statement*⟩ can be executed on any locale.

A common situation will be to use the `on` statement in conjunction with a `forall` loop accessing an array decomposed over multiple locales. For example:

```
forall i in D on(a(i)) do ...
```

where the body of the loop will be a computation accessing `a(i)` and related variables.

When a loop iterates over a sequence specified by an iterator, `on` statements inside the iterator control where the corresponding loop body is executed. For example, an iterator over a distributed tree might include:

```
class Tree {
    var left :Tree, right :Tree;
    iterator nodes {
        yield(this);
        if(bound?(left))  on(left)  yield(left.nodes);
        if(bound?(right)) on(right) yield(right.nodes);
    }
...
forall t in tree.nodes do ...
```

Here, each instance of the body of the `forall` loop is executed on the locale where the corresponding object `t` is located. The location of execution is undefined in the case of sequence products with conflicting specifications. In such a case an explicit `on` statement should be used.

By default, when new variables and data objects are created, they are created in the locale where the computation is running.

10.4 Domain Decomposition

A function that maps from domain index values to locales is called a *distribution*. A domain for which a distribution function is specified is referred to as a *distributed domain*. A domain supports a method, `locale`, that maps index values to locales that correspond to this function. For indefinite domains, this function can be invoked on values that are not part of the index set. For other domains, the method is undefined for arguments not in the index set.

Arrays defined over a distributed domain will have the element variables stored on the locale determined by the distribution. Thus, if `d` is an index of distributed domain `D` and `A` is an array defined over that domain, then `A(d).locale` is the same as `D(d).locale`.

Iteration over a distributed domain implicitly executes the control computation in the domain of the associated locale. If `D` is a distributed domain, then given the following:

```
forall d in D do ⟨statement⟩
...  [d in D] ⟨expr⟩
```

both ⟨*statement*⟩ and ⟨*expr*⟩ will be executed in locale `D.locale(d)`. Similarly, when iterating over the elements of an array defined over a distributed domain, the controlled computations are determined by the distribution of the domain. If there are conflicting distributions in product iterations, the locale of the computation is not defined.

10.5 Specifying Distributions [Outline]

The syntax to specify a distribution extends the domain declaration. The specification is somewhat complex in the general case: we want to identify a tuple of projections of the domain to a tuple of elemental distribution functions that map elements of the projection to target domains. The product of these target domains is then used as the domain for an array of locales that is used to complete the mapping.

Outline of this section:

1. Define *elementary* distributions that map a *source domain* entirely to an arithmetic index set that is used as the *target domain*. The target domain is the domain for an array of `locale` values to which the data is distributed.

2. Define protocols for defining the distribution of opaque and indefinite domains.

3. Extend elementary distributions to *product* distributions that map k distributions to a rank-k target domain that is the product of the target domains of the elementary distributions.

4. Define the default distribution for arithmetic domains and products of distributed domains.

5. Define methods to specify projections of the source domain to be used as the source domain for each of the elementary dimensions.

6. Identify parameters to distributions that can be modified and a protocol for changing them analogous to how we change range-information.

10.6 Object Caching [Outline]

Objectives: define a protocol to identify which objects are subject to local caching, when stale objects become invalid, when dirty objects are flushed, how updates are combined.

Extend this protocol to collections of objects by building on the the distribution machinery.

11 Structural Interfaces [Requirements]

Two of Chapel's goals are to provide high-level features that improve productivity for experimental programming and to encourage reuse. We also want to provide a path for extension and customization of those features that exploit the structural typing and other generic programming features of the language.

We define a *structural interface* to be a generic class definition where the fields of this class are separated into mandatory and optional. The mandatory fields characterize core behavior, while optional fields standardize common special cases. A class *implements* the interface by providing consistent bindings for the mandatory fields and a subset of the optional fields.

Requirements

1. Syntax for identification of mandatory and optional fields.

2. Each of the high-level concepts in Chapel, should have a structural interface. The concepts currently are: sequence, array, domain, and distribution.

3. Description of how implementation of control constructs that interact with these concepts may exploit optional elements of the interfaces. This includes `for` and `forall` and expression-level iteration, especially the parallel construction of new sequences.

For example, programmers should be able to write their own distribution with well defined rules about how `forall` statements over sections of that domain will be implemented in terms of the particular subset of the interface that is implemented.

11.1 Reductions

The syntax for a reduction may identify a class name. For example:

[`ordered`] `reduce` ⟨*seq*⟩ `by` ⟨*class*⟩ [`else` ⟨*expr*⟩]

where ⟨*class*⟩ identifies a class that `implements` the interface `reduce`. The fields for this interface are:

```
class reduce {
    type input ;
    type output :input:
    type state :input;
    _optional var zero_val :  state;
    function input (t:input) :state return t;
    function combine(t1:state, t2:state) :state;
    function output(t:state) :output return t;
    function combine1(t1:state, t2:input) :state
        return combine(t1,input(t2));
    function combine2(t1:input, t2:input) :state
        return combine1(input(t1),t2);
}
```

The type `input_type` is the element type of the sequence. The type `output_type` is the element type of the result which by default is the same as the input type. The type `state_type` represents the intermediate state which is also by default the same as the input type. The function `input` converts an input value into a state value. The function `combine` takes two state values and combines then into one value. This function is assumed to be associative unless the `ordered` keyword is present in which case it need not be. The function `output` converts the final state to an output value. The `else` clause may be omitted in which case `zero_val` will be returned if it is defined and the input sequence is empty.

A partial implementation of the reduction interface that finds the k largest elements in a sequence is given in Figures 8 and 9. The element type of the sequence is unspecified and the only requirements are that comparison be supported and that there is a `max` value for the type. The output is a sequence of the three largest elements possibly filled with max values.

11.2 Scans

Similar to reductions, there is an interface for defining application specific `scan` semantics. ...

```
class MaxK implements reduce {
   parameter var k :  integer;
   type elt_type;
   type input:  elt_type;
   record state {
       var values[1..k] :elt_type;
       create(x :  elt_type) {
             values(1..k-1) = elt_type.max;
             values(k) = x;
       }
   }
   type output :seq(elt_type);
   function input(x:elt_type) return state.create(x);
   function combine(a :state, b:state) {
       var start = 1;
       for x in b.values {
          -- insert 'x' into a
          for i in start..k
             if(a.values[i] < x) {
                 a.values[i+1..k] = a.values[i..];
                 a.values[i] = x;
                 start = i+1;
                 break;
             }
          return a;
       }
   }
   function output(y:state) return y.values;
   ... helper functions from Figure 9 go here
}
```

Figure 8: An example reduction that find the k largest values in an input sequence.

```
function combine1(a:state, b:input) {
   for i in 1..k
      if(a.values[i] < x) {
          a.values[i+1..k] = a.values[i..];
          a.values[i] = x;
          break;
      }
   return a;
}
function combine2(a:input, b:input) {
   var s = state();
   if(a < b)
     (a,b) = (b,a);
   s.values(1) = elt_type.max;
   s.values(2) = a;
   s.values(2) = b;
   return a;
}
```

Figure 9: Definitions of some secondary functions needed in Figure 8.

12 Input and Output Functions [Requirements]

Here is a short discussion of requirements and possible features for Chapel I/O capabilities.

Requirements

1. The I/O mechanisms must be integrated into all other aspects of the language including sequences and parallelism. *Files* must be object instances with extensible semantics.

   This implies the ability to read and write data aggregates plus the ability to divide up a file so that data can be accessed by separate sub-computations. It also requires that the results of those computations can be combined in a well-defined manner for output.

2. The I/O mechanism must allow inter-operation with data files written by C and Fortran or to be read by C and Fortran.

   We will need a notion of *record* and some notion of both variable and fixed record length files. Notions of both formatted and unformatted files will be supported. It is undesirable to require each I/O read or write to consume a complete record.

   It is not necessary to support the precise capabilities of either C or Fortran formating rules but building one or the other would facilitate learning. Default formating rules for structured data should be specified as part of the type (like bound functions).

3. The I/O implementation must support stream-oriented files like sockets as well as disk-oriented files.

   This affects not only how promptly buffer data is written but also when read-ahead is allowed to occur.

4. The lack of type-safety in this I/O model must be tolerated. The machinery that packs and unpacks typed data must be exposed so that such "binary" data can be manipulated by the program outside of a file.

5. Chapel should also support type-safe files, both formatted and unformatted.

6. File operations should be atomic.

Target Features

1. It should be possible to create persistent data structures, including object references, that can be written to and read from files.

2. It should be possible to read and write `config` variables to a file.

3. Can we associate static data with a file to provide a simple path to persistent data? Can such data be stored in a database instead of a flat file system?

4. The pre-fetching and buffering policies should be modifiable by the application. Techniques include derived classes and configuration variables.

5. The ability to deal with binary data should be integrated into frameworks for explicit message passing and for interfaces with external language functions. These are both situations where type-safety can not be enforced much as for I/O.

12.1 Files

A file is a sequence of *records*. The records may either be fixed or variable length and may contain either only strings or some alphabet or binary data. The former are called *formatted* while the later are called *unformatted*. Binary data files may be *untyped* or *typed*.

The system provided module `stdio` defines the interface class type `File`. This class has the two subtypes, `Formatted` and `Unformatted`. The default constructor for any file includes this generic prototype

```
constructor File(system_name :string = '',
                 record_length :integer = 0,
                 read=true, write=true);
```

The constructor may have additional implementation specific parameters. For example, on a UNIX system there is an additional integer argument `permission` that specifies permission attributes when a new file is created.

The `system_name` field is used to created an implementation dependent association with this file and some persistent storage object or communication stream. The default is an empty string which indicates a file whose existent is limited to the lifetime of program execution and may be stored entirely in memory. `record_length` indicates the fixed length of records where a non-positive value indicates variable length records. The boolean parameters `read` and `write` indicate which operations will be applied to the file.

The default constructor for `Formatted` files has this generic prototype of a `File` plus these parameters:

```
constructor Formatted(file arguments,
                      type alphabet = ASCII,
                      pad  :string(alphabet=alphabet) = ' ',
                      trim :boole = true,
                      eor  :string(alphabet=alphabet) = '\n');
```

The `alphabet` identifies the alphabet of strings that are stored in the file. The the first character of the `pad` argument is used to fill incomplete records for fixed record length files and `eor` is used as record terminator. If the `pad` string is empty, then it is an error to write incomplete records when a positive record length is specified. The `eor` may be empty only when `record_length` is positive.

The default constructor for `Unformatted` files is similarly an extension of `File`:

```
constructor Unformatted(file arguments,
                        rec_size_width :integer = 4
                        indexed        :boole = false,
                        typed          :boole = false);
```

The `indexed` attributes indicates that additional information is stored in the file to allow specific records in the file to be efficiently accessed. When `index` is false and `record_length` is 0, then each record has its sizes written into the file using an integer with representation width `rec_size_width`. The default value of 4 is intended to provide compatibility with existing Fortran variable length unformatted files.

When a file is associated with a system name, after construction, the method `open_status` returns an enumeration value. This enumeration is defined inside of the base `File` class and includes these values:

```
enum OpenStatus { OK, NOT_FOUND, INVALID_NAME, ...  }
```

The value `OK` indicates the file was successfully opened and associated with the system name. `NOT_FOUND` indicates a file opened with `write=false` where there system has no object with this name. `INVALID_NAME` indicate the specified system name was not valid for implementation dependent reasons. This enumeration can include implementation dependent values. The method `open_error` returns a descriptive character string related to the status suitable for use in diagnostic messages.

12.2 Parallelism and Files

A *section* of a file is a sequence of records that is treated as a separate file.

For untyped files, data stored in the file consist only of integers, floating point values, complex values and strings. Boolean values are converted to type `integer(size=1)` implicitly but enumerated values must be explicitly converted to some integral type. Data stored in typed files can be any primitive or derived types including classes, sequences, arrays and domains.

12.3 Notes

Function Values Can we store a function closure in a typed file? There are a set of captured values plus the *name* of the function to be invoked. Can we build tables to allow mapping of these names back to functions? This mapping might fail but we might also not have support for a data type.

Formats for variable length records It would be nice to support a format for variable length files that is more parallel-friendly. For example, we could maintain record lengths in some table that would allow more efficient mapping from record number to data. This could be done as some kind of subtyping or by having a type parameter that describes how this mapping should be implemented.

Maybe this is not programmable but rather we just provide the option of indexing the file.

A Predefined Methods and Functions

⟨*arithmetic type*⟩`.min`	Minimum value of type
⟨*arithmetic type*⟩`.max`	Maximum value of type
⟨*arithmetic type*⟩`.size`	Representation width in bytes
`string.alphabet`	Alphabet a string is defined over
⟨*float type*⟩`.nan`	An IEEE not-a-number value for the type
⟨*float type*⟩`.infinity`	A positive IEEE infinity value
⟨*float type*⟩`.precision`	Binary precision of the representation
⟨*float type*⟩`.max_exp`	Maximum binary Exponent
⟨*float type*⟩`.epsilon`	Machine epsilon
⟨*float type*⟩`.floor`	integer floor
⟨*float type*⟩`.ceil`	integer ceiling
⟨*type*⟩`.initial`	Initial value for unparameterized type
⟨*seq type*⟩`.rank`	rank of a sequence, array, or domain
⟨*seq type*⟩`.elt_type`	Element type of a sequence or array
⟨*seq type*⟩`.leaf_type`	Leaf element type of a sequence
⟨*variable*⟩`.locale`	Locale associate with variable

Figure 10: Predefined methods. A type `complex(size=`$2k$`)` is treated like a type of `float(`k`)` for the inquiry methods `precision`, `max_exp`, and `epsilon`.

⟨*expr*⟩`.type`	Returns the type of the expression. The expression is not evaluated.
⟨*complex*⟩`.real`	real part of complex value
⟨*complex*⟩`.imag`	imaginary part of complex value
⟨*index*(*k*)⟩`.lbound`	lower bound k-tuple
⟨*index*(*k*)⟩`.ubound`	upper bound k-tuple
⟨*index*(*k*)⟩`.stride`	stride k-tuple
⟨*index*(*k*)⟩`.extent`	extent k-tuple
⟨*array*⟩`.domain`	Domain of an array
⟨*arithmetic domain*⟩`.range`	Bounding information of type ⟨*index*(*k*)⟩
⟨*domain*⟩`.first`	Lexically first element in domain
⟨*domain*⟩`.last`	Lexically last element in domain
⟨*domain*⟩`.locale`	Map index value to locale

Figure 11: Predefined methods for values. Generally, any type method may also be applied to a value and will implicitly refer to the type of the value.

`shift_left(v,k)`	shift `v` left `k` bits
`shift_right(v,k)`	shift `v` left `k` bits, unsigned
`shift_right_signed(v,k)`	shift `v` left `k` bits, signed
`abs(x)`	absolute value
`min(x,...)`	minimum of ordered values, reduction
`max(x,...)`	minimum of ordered values, reduction
`float2bits(v)`	convert floating point value to an integer without changing low-level representation
`bits2float(v)`	convert integer value to floating point without changing low-level representation
`length(s)`	length of sequence or string
`code(s)`	convert first character of string to integer according to the string's alphabet
`bound?(r)`	determine if a reference is bound to a variable
`this_locale()`	return current locale associated with a computation

Figure 12: Predefined functions. Generally all functions can be promoted to sequence arguments. Those marked with "reduction" consume sequence arguments and return scalars.

B Index

Index