The Fortress Language Specification

Version 0.707

Eric Allen
David Chase
Victor Luchangco
Jan-Willem Maessen
Sukyoung Ryu
Guy L. Steele Jr.
Sam Tobin-Hochstadt

Additional contributors:
Joao Dias
Carl Eastlund
Joe Hallett
Yossi Lev
Cheryl McCosh

July 28, 2005

Contents

Chapter 1

Introduction

The Fortress^{TM} Programming Language is a general-purpose, statically typed, component-based programming language designed for producing robust high-performance software with high programmer productivity.

In many ways, Fortress is intended to be a “growable language”, i.e., a language that can be gracefully extended and applied in new and unanticipated contexts. Fortress supports state-of-the-art compiler optimization techniques, scaling to unprecedented levels of parallelism and of addressable memory. Fortress has an extensible component system, allowing separate program components to be independently developed, deployed, and linked in a modular and robust fashion. Fortress also supports modular and extensible parsing, allowing new notations and static analyses to be added to the language.

The name “Fortress” is derived from the intent to produce a “secure Fortran”, i.e., a language for high-performance computation that provides abstraction and type safety on par with modern programming language principles. Despite this etymology, the language is a new language with little relation to Fortran other than its intended domain of application. No attempt has been made to support backward compatibility with existing versions of Fortran; indeed, many new language features were invented during the design of Fortress. Many aspects of Fortress were inspired by other object-oriented and functional programming languages, including The Java^{TM} Programming Language [5], NextGen [6], Scala [20], Eiffel [16], Self [1], Standard ML [17], Objective Caml [14], Haskell [22], and Scheme [13]. The result is a language that employs cutting-edge features from the programming-language research community to achieve an unprecedented combination of performance and productivity.

1.1 Overview of Fortress

Two basic concepts in Fortress are that of *object* and of *trait*. An object consists of *fields* and *methods*. The fields of an object are specified in its definition. An object definition may also include method definitions.

Traits are named program constructs that declare sets of methods. They were introduced in the Self programming language, and their semantic properties (and advantages over conventional class inheritance) were analyzed by Schärli, Ducasse, Nierstrasz, and Black [23]. In Fortress, a method declared by a trait may be either *abstract* or *concrete*: abstract methods have only *headers*; concrete methods also have *definitions*. A trait may *extend* other traits: it *inherits* the methods declared by the traits it extends (except those that it *overrides*). A trait declares the methods that it inherits as well as those explicitly declared in its definition.

Every object has a set of traits; an object includes every method declared by any of its traits. An object inherits the concrete methods of its traits and must include a definition for every method declared but not defined by its traits. It is also allowed to override the definition of a concrete method inherited from a trait.

```
object traits {StarSystem, OrbitingObject}
  sun = Sol
  planets =
    { Mercury, Venus, Earth, Mars,
      Jupiter, Saturn, Uranus, Neptune, Pluto }

  position = Polar (25000 lightYear, 0 radian)
  ω: radian/s := 2 π radian / 226 million year in s

  accelerate(θ) = ω := ω + θ
end
```

In this example, the object `SolarSystem` is defined with the traits `StarSystem` and `OrbitingObject`. The fields ω and `position` are defined with appropriate quantities. The field `sun` is defined to be another object named `Sol`, and the field `planets` is defined to be a set of objects. The method `accelerate` is defined to take a single parameter θ, and update the ω field of the object. As this example illustrates, Fortress provides static checking of physical units and dimensions on quantities.

Note that the identifiers used in this example are not restricted to ASCII character sequences. Fortress allows the use of Unicode characters in program identifiers, as well as subscripts and superscripts. (See Appendix C for a discussion of Unicode and suggested input methods for Fortress program editors). Fortress also allows multiplication to be expressed by simple juxtaposition, as can be seen in the definitions of ω and `position`. Fortress also allows for operator overloading, as well as a facility for extending the syntax with domain-specific languages.

Although Fortress is statically and nominally typed, types are not specified for all fields, nor for all method parameters and return values. Instead, wherever possible, *type inference* is used to reconstruct types. In the examples throughout this specification, we often omit the types when they are clear from context. Additionally, types can be parametric with respect to other types and values (most notably natural numbers).

These design decisions are motivated in part by our goal of making the scientist/programmer's life as easy as possible without compromising good software engineering. In particular, they allow us to write Fortress programs that preserve the look of standard mathematical notation.

In addition to objects and traits, Fortress allows the programmer to define top-level functions. Functions are first-class values: They can be passed to and returned from functions, and assigned as values to fields and variables. Functions and methods can be overloaded, with calls to overloading methods resolved by multiple dynamic dispatch. Keyword parameters, variable size argument lists, and multiple return values are also supported.

Fortress programs are organized into *components*, which export and import *apis* and can be linked together. Apis describe the "shape" of a component, specifying the types in traits, objects and functions provided by a component. All external references within a component (i.e., references to traits, objects and functions implemented by other components) are to apis imported by the component. We discuss components and apis in detail in Chapter 4.

To address the needs of modern high-performance computation, Fortress also supports a rich set of operations for defining parallel execution and distribution of large data structures. This support is built into the core of the language. For example, `for` loops in Fortress are parallel by default.

1.2 Organization

This language specification is organized as follows. In Chapter 2, the basic concepts of the Fortress programming model are explained, including objects, types, and functions. Many examples illustrating the concrete syntax are provided. In Chapter 3, advanced language constructs are described. In particular, the Fortress model of parallelism and support for domain-specific languages are discussed. In Chapter 4, the compilation and deployment model is described, including a discussion of Fortress components and apis. In Chapter 5, the abstract syntax is explained. Finally, in Chapter 6, the Fortress concrete syntax is defined in BNF notation.

Chapter 2

Basic Concepts

2.1 Expressions, Values, and Types

Expressions are program constructs that reduce to *values*. Every value has a *location*. Every location holds a single value. Two values are identical iff they have the same location. Every expression has a *static type*. Every value has a *runtime type*. Some types have names; two types with the same name are identical. *Generic types* are parametric with respect to types and values. Two instantiations of a generic type are identical iff their arguments are identical.

Types are related by a subtyping relation, which is reflexive, transitive, and antisymmetric. Fortress programs are checked before they are executed to ensure that if an expression e reduces to a value v, the runtime type of v is a subtype of the static type of e.

Some types are defined by programs; others are built into the language. The built-in type `Any` is a supertype of all types. Every finite set of types $\{T_1, ..., T_n\}$ is itself a type, referred to as the *intersection* of $T_1, ..., T_n$. $\{T_1, ..., T_n\}$ is a subtype of all of its subsets. An intersection type is not a first-class type; intersection types are used for bounds on trait parameters, trait variables in `where` clauses as described in Section 2.5.2, and for type inference. They cannot appear in programs as explicit expression types and they cannot be nested.

For every finite set of types, there is also a type denoting a unique *least upper bound* of those types. The least upper bound of a set of types S is a supertype of every type $T \in S$ and of the least upper bound of every proper subset of S. Least upper bound types are not first-class types; they are used solely for type inference and they cannot be expressed directly in programs. In some circumstances, a named type is identified with a least upper bound type, as discussed in Section 2.2.

There are built-in types for `Bool`, `Char`, `String`, the special type `()` (pronounced "void"), and several numeric types. These types are mutually exclusive; no value has more than one of them. Values of these types are immutable. Many of them are identified with special expressions called *literals*.

The two values of type `Bool` are the literals `True` and `False`. Two expressions e and e' can be compared via an expression of the form e `=` e' (where the types of e and e' are not mutually exclusive). An expression of this form evaluates to `True` iff e and e' reduce to identical values.

Values of type `Char` are arbitrary characters in Unicode 4.0.0 [8], enclosed in single quotes (e.g., `'a'`, `'A'`, `'`α`'`). Values of type `String` are sequences of characters enclosed in quotation marks (e.g., `"`$\pi\, r^2$`"`). Escape sequences in strings abide by the conventions of the Java Programming Language.

The only value with type `()` is the literal `()`. References to the value `()` as opposed to the type `()` are determined by context.

The numeric types share the common supertype `Num`. Fortress includes types for arbitrary-precision integers (of type $\mathbb{Z}$), rational numbers (of type $\mathbb{Q}$), fixed-size representations for integers including the types $\mathbb{Z}8$, $\mathbb{Z}16$, $\mathbb{Z}32$, $\mathbb{Z}64$, $\mathbb{Z}128$, their unsigned equivalents $\mathbb{N}8$, $\mathbb{N}16$, $\mathbb{N}32$, $\mathbb{N}64$, $\mathbb{N}128$, floating-point numbers of various precisions (some having hardware support), intervals (of type `Interval⟦X⟧`, abbreviated as `⦇X⦈`, where `X` can be instantiated with any number type), and imaginary and complex numbers of fixed size (in rectangular form with types $\mathbb{C}n$ for $n = 16, 32, 64, 128, 256$ and polar form with type `Polar⟦X⟧` where `X` can be instantiated with any complex number type).

For floating-point numbers, Fortress supports types $\mathbb{R}32$ and $\mathbb{R}64$ to be 32 and 64-bit IEEE 754 floating-point numbers respectively, and defines two functions on types: `Double⟦F⟧` is a floating-point type twice the size of the floating-point type `F`, and `Extended⟦F⟧` is a floating-point type sufficiently larger than the floating-point type `F` to perform summations of "reasonable" size.[1] Other built-in types are introduced in this specification as they are needed.

2.1.1 Numerals

Every numeral is a non-empty sequence of digits and letters with an optional decimal point, starting with a digit (possibly zero), and an optional radix as a subscript.

Examples: 27 7fff_{16} 0fff_{16} 10101101_2 3.14159265 3.11037552_8

Numerals are not directly converted to any of the number types because, in common mathematical usage, we expect them to be polymorphic. As a simple example, consider the literal `3.1415926535897932384`; it is a bad idea to immediately convert it to a floating-point number because that may introduce a rounding error. If that literal is used in an expression involving floating-point intervals, it is better to convert it directly to an interval. Therefore literals that would be considered `REAL` in Fortran have their own types in Fortress, `Numeral⟦X⟧` (where `X` is the radix). This approach allows library designers to decide how literals should interact with other types of objects.

2.1.2 Aggregate expressions

Aggregate expressions reduce to values that are themselves homogeneous collections of values. Aggregate expressions in Fortress are provided for sets, maps, lists, tuples, matrices, and arrays.

Set expressions: Elements are enclosed in braces and separated by commas, e.g.,

```
{0, 1, 2, 3, 4, 5}          (* This set has six elements. *)
```

This expression evaluates to a set containing six elements, as explained in the comment immediately proceeding it. Comments in Fortress are delimited by tokens `(*` and `*)` and can be nested.

The type of a set expression is `Set⟦T⟧`, where `T` is the least upper bound of the types of all element expressions of the set. This type can be abbreviated as `{T}` in contexts where there is no ambiguity with intersection types.

Set containment is checked with the operator $\in$. For example:

```
3 ∈ {0, 1, 2, 3, 4, 5}
```

[1] This formulation of floating-point types follows a proposal under consideration by the IEEE 754 committee.

reduces to `True`. The subset relationship is checked with the operator ⊆. For example:

```
{0, 1, 2} ⊆ {0, 3, 2}
```

reduces to `False`.

Map expressions: Elements are enclosed in brackets, separated by commas, and matching pairs are separated by ↦, e.g.,

```
[0 ↦ 'a', 1 ↦ 'b', 2 ↦ 'c']
```

The type of a map expression is `Map[S,T]` where `S` is the least upper bound of the types of all domain element expressions, and `T` is the least upper bound of the types of all range element expressions. This type can be abbreviated as `[S ↦ T]`.

A map m is indexed by placing an element in the domain of m enclosed in brackets immediately after an expression evaluating to m. For example, if:

```
m = ['a' ↦ 0, 'b' ↦ 1, 'c' ↦ 2]
```

Then `m['b']` evaluates to `1`.

List expressions: Elements are enclosed in angle brackets ⟨ and ⟩ and are separated by commas, e.g.:

```
⟨0,1,2,3⟩
```

The type of a list expression is `List⟦T⟧` where `T` is the least upper bound of the types of all elements. This type can be abbreviated as `⟨T⟩`.

A list l is indexed by placing an index enclosed in angle brackets immediately after an expression evaluating to l. For example:

```
⟨3,2,1,0⟩⟨2⟩
```

evaluates to `1`.

Tuple expressions: Elements are enclosed in parentheses and separated by commas, e.g:

```
(0,1,2)
```

Unlike other aggregate expressions, tuple expressions do not evaluate to values; they evaluate to *tuples* of values. This distinction is subtle but important. For example, variables cannot be bound to a tuple of values (as discussed in Section 2.1.6). If an element e' of a tuple expression e evaluates to a tuple, the elements of e' are *flattened* into e. For example, the expression:

```
((0,1),(2,(3),4),5)
```

evaluates to the tuple of values:

```
(0,1,2,3,4,5)
```

A tuple of one element is flattened to its element. The expression `(0)` evaluates to the value `0`.

The static type of a tuple expression has a *tuple type*: `(`T_0`, ···, `T_n`)`. A well-formed tuple type does not contain tuple types itself. The type T of a tuple expression e is formed by flattening the types of all elements into T. For example , the type of:

```
((0,1),(2,(3),4),5)
```

is

```
(Numeral⟦10⟧, Numeral⟦10⟧, Numeral⟦10⟧, Numeral⟦10⟧, Numeral⟦10⟧, Numeral⟦10⟧)
```

Matrix expressions: Elements are enclosed in brackets. Elements along a row are separated only by whitespace, as in the following example:

```
[1 0 0]
```

All matrices have two or more elements. All matrices have two or more dimensions. Two dimensional matrices of size $1 \times n$ are *row vectors*. Two dimensional matrices of size $n \times 1$ are *column vectors*. Two dimensional matrix expressions are written by separating rows with newlines or semicolons. If a semicolon appears, whitespace before and after the semicolon is ignored, as in the following four examples, which are all equivalent:

```
[3 4
 5 6]
```

```
[3 4 ;
 5 6 ]
```

```
[3 4

; 5 6 ]
```

```
[3 4 ; 5 6]
```

The parts of higher-dimensional matrices are separated by repeated-semicolons, where the dimensionality of the result is equal to one plus the number of repeated semicolons. Here is a $3 \times 3 \times 3 \times 2$ matrix:

```
[ 1 0 0
  0 1 0
  0 0 1   ;;  0 1 0
              1 0 1
              0 1 0   ;;  1 0 1
                          0 1 0
                          1 0 1
   ;;;

   1 0 0
   0 1 0
   0 0 1   ;;  0 1 0
               1 0 1
               0 1 0   ;;  1 0 1
                           0 1 0
                           1 0 1 ]
```

The elements in a matrix expression may be either scalars or matrices themselves. If they are matrices, then they are "flattened" into the enclosing matrix, as discussed in Section 3.3. The elements along a row (or column) must have the same number of columns (or rows), though two elements in different rows (columns) need not have the same number of columns (rows).

The type of a k dimensional matrix expression is `Matrix`⟦`T`⟧`[`$n_0 \times ... \times n_{k-1}$`]`, where `T` is the least upper bound of the types of the elements and $n_0 \times ... \times n_{k-1}$ are the sizes of the matrix in each dimension. This type can be abbreviated as `T[`$n_0 \times ... \times n_{k-1}$`]`.

An n-dimensional matrix M is indexed by placing a sequence of n indices enclosed in brackets, and separated by commas, after an expression evaluating to M. For example:

```
M = [1 2 3; 4 5 6; 7 8 9]
```

then `M[1,0]` evaluates to 4.

Array expressions: Elements are enclosed in brackets. Elements along a row are separated by commas:

```
[1, 0, 0]
```

Elements of multidimensional arrays are separated by newlines and sequences of semicolons, as with matrices. (Note that there is no conflict with matrix notation because all matrices have at least two elements). The type of a k dimensional array expression is `Array`⟦`T`⟧`[`n_0`, `$\cdots$`, `n_{k-1}`]`, where `T` is the least upper bound of the types of the elements and $n_0, ..., n_{k-1}$ are the sizes of the array in each dimension. This type can be abbreviated as `T[`n_0`, `$\cdots$`, `n_{k-1}`]`. (Note that there is no conflict with matrix type notation because matrices must have at least two dimensions).

Arrays are indexed in the same manner as matrices.

2.1.3 If expressions

An `if` expression consists of the reserved word `if` followed by a test expression, followed by the reserved word `then`, a sequence of expressions, a sequence of `elif` clauses (each consisting of the reserved word `elif` followed by a test expression, the reserved word `then`, and a sequence of expressions), an optional `else` clause (consisting of the reserved word `else` followed by a sequence of expressions), and finally the reserved word `end`. For example,

```
if x ∈ {0, 1, 2} then 0
elif x ∈ {3, 4, 5} then 3
else 6 end
```

The type of an `if` expression is the least upper bound of the types of all clauses. If there is no `else` clause in an `if` expression, then the last expression in every clause must evaluate to `()`.

2.1.4 While loops

`while` loops are written as follows:

```
while expr do
  exprs
end
```

The value of a `while` loop is `()`.

2.1.5 For loops

`for` loops are written as follows:

```
for v1 ← g1,
    v2 ← g2,
    ...
    vn ← gn do
  exprs
end
```

The loop header is made up of a series of *generators*. Generators are described in Section 3.2.2. Each generator binds one or more loop variables. A loop variable scopes over the remaining generators and over the body of the loop. By default, loop iterations are assumed to run in parallel. The order of nesting of generators does not imply anything about the relative order of loop iterations. Multiple nested loops preserve the order of loop iterations. The value of a `for` loop is `()`.

2.1.6 Bindings

A binding is an expression that declares a variable. The name of a variable can be any valid Fortress *identifier*, which is a non-empty sequence of alphanumeric characters in Unicode 4.0.0 that begins with a letter, and that is not a *reserved word*. Throughout this text, reserved words are identified when they are first discussed.

The scope of a variable is the rest of the innermost *enclosing context* of its binding. Several Fortress language constructs define new enclosing contexts; we mention each such construct when we define it.

There are four forms of binding expression. The first form:

name `:` *type* = *expr*

declares *name* to be an immutable variable with static type *type* whose value is computed to be the value of the expression *expr*. The static type of *expr* must be a subtype of *type*.

The second (and most convenient) form:

name = *expr*

declares *name* to be an immutable variable whose value is computed to be the value of the expression *expr*; the static type of the variable is the static type of *expr*.

The third form:

`var` *name* `:` *type* = *expr*

declares *name* to be a mutable variable of type *type* whose initial value is computed to be the value of the expression *expr*. As before, the static type of *expr* must be a subtype of *type*. The modifier `var` is optional when '`:=`' is used instead of '=' as follows:

`[var]` *name* `:` *type* `:=` *expr*

The first three forms are referred to as *defined bindings*. The fourth form:

`[var]` *name* `:` *type*

declares a variable without giving it an initial value (where mutability is determined by the presence of the `var` modifier). It is a static error if the variable is referred to before it has been given a value either by another binding expression or by assignment. Whenever a variable bound in this manner is given a value, the type of that value must be a subtype of its declared type. This form allows declaration of the types of variables to be separated from definitions, and it allows programmers to delay assigning to a variable before a sensible value is known.

All forms can be used with *tuple notation* to bind multiple variables together. A tuple of variables to bind is enclosed in parentheses and separated by commas, as are the types declared for them:

`(`*name*`[,` *name*`]`*`) : (`*type*`[,` *type*`]`*`)`

Alternatively, the types can be included alongside the respective variables, optionally eliding types that can be inferred from context:

`(`*name*`[:` *type*`][,` *name*`[:` *type*`]]`*`)`

Alternatively, a single, non-tuple, type can be declared for all of the variables:

(*name*[, *name*]*) : *type*

This notation is especially helpful when a function application returns a tuple of values. Note, however, that tuples are *not* values in Fortress. In particular, a single variable cannot be bound to a tuple.

Here are some simple examples of binding expressions:

```
π = 3.141592653589793238462643383279502884197169399375108209749445923078
```

binds the variable π to an approximate representation of the mathematical object π. It is also legal to write:

```
π : Float = 3.141592653589793238462643383279502884197169399375108209749445923078
```

This definition enforces that π has static type `Float`.

In this example, the declaration of the type of a variable and its definition are separated:

```
π : Float
π = 3.141592653589793238462643383279502884197169399375108209749445923078
```

The following example binds multiple variables using tuple notation.

```
var (x, y): Int = (5, 6)
```

The following three expressions are equivalent:

```
(x, y, z): (Int, Int, Int) = (0, 1, 2)
(x:Int, y:Int, z:Int) = (0, 1, 2)
(x, y, z): Int = (0, 1, 2)
```

2.1.7 Comprehensions

Fortress provides "comprehension" syntax for several of the built-in aggregate types.

Set comprehensions are enclosed in braces, with a left-hand expression separated by a `|` from a sequence of boolean expressions and generators. The generators bind variables exactly as in a `for` loop. The boolean expressions act as filters. For example, the comprehension:

$\{$ `x`^{2} `| x` $\leftarrow$ $\{$`0,1,2,3,4,5`$\}$`, x MOD 2 = 0` $\}$

denotes the set

```
{0,4,16}
```

Array comprehensions are like set comprehensions (except that they are syntactically enclosed in brackets). However, an array comprehension may have multiple clauses as follows:

```
a = [ (x,y,1) = 0.0   | x ← 1:xSize, y ← 1:ySize
      (1,y,z) = 0.0   | y ← 1:ySize, z ← 2:zSize
      (x,1,z) = 0.0   | x ← 2:xSize, z ← 2:zSize
      (x,y,z) = x+y*z | x ← 2:xSize, y ← 2:ySize, z ← 2:zSize ]
```

Each clause conceptually corresponds to an independent loop. Clauses are run in order.

2.1.8 Function definitions

A function definition is similar to a binding expression for an immutable variable: it establishes a name for a function whose scope is its entire enclosing context. Function definitions can be mutually recursive.

Syntactically, a function definition consists of the name of the function, followed by all type parameters (described in Section 2.5), all value parameters with their (optionally) declared types, the optional types of all return values, the thrown exceptions, an optional contract for the function (discussed in Section 2.1.21), and finally the body. Value parameters cannot be mutated inside the function body. For example, here is a definition of a simple function:

```
swap(x:Any, y:Any):(Any, Any) = (y, x)
```

This function has no type parameters, throws no checked exceptions, and has no contract. It takes two parameters of type `Any` and returns a tuple of two values. Namely, it returns its parameters in reverse order. If the return type is elided, it is inferred to be the static type of the body. The following definition of `swap` has the same return type as the above definition:

```
swap(x:Any, y:Any) = (y, x)
```

Similarly, function parameter types can often be inferred from the body of the function. When a least upper bound can be inferred for a parameter from the body of the function, that parameter need not be declared explicitly. In the case of `swap`, the unique least upper bound of both `x` and `y` happens to be type `Any`. Thus, the following definition of `swap` has the same parameter types and return type as the above definitions:

```
swap(x, y) = (y, x)
```

When a function is called, the body of the function is evaluated in a new enclosing context, extending the enclosing context in which it is defined with all parameters bound to their arguments.

A function parameter is allowed to include a *default* expression, which is used when no argument is bound to the parameter explicitly. The default expression of a parameter x of function f is evaluated each time the function is called without a value provided for x at the call site. All parameters declared to the right-hand side of x must include default expressions as well and x scopes over the remaining parameters and over the body of the function. The default expression of x is evaluated in an environment extending the environment in which f is defined with all parameters textually preceding x bound to their arguments.

If no type is declared for a parameter with a default, the type is inferred from the static type of its default expression. Syntactically, this default value is specified after an = sign. For example, we can write:

```
wrap(xs, ys = xs) = [xs, ys]
```

The function `wrap` returns an array containing its parameters. If a value for only the parameter `xs` is given to `wrap` at a call site, the value of `xs` is bound to `ys` as well, and an array that contains `xs` as both of its indices is returned. Default parameters can be bound at a call site by keyword arguments, as described in Section 2.1.9.

The rightmost parameter of a function definition that does not have a default expression is allowed to have type `T...` for any type `T`. A parameter with this type is a *varargs* parameter; it is used to pass a variable number of arguments to a function as a single array. For example:

```
asArray(xs:Object...) = xs
```

takes an arbitrary number of arguments and returns an array containing them all.

If a function does not have a varargs parameter then the number of arguments is fixed by the function's type. A varargs parameter is not allowed to have a default expression.

Function definitions can be immediately preceded by the following special modifiers:

io: Functions and methods which perform externally visible effects such as I/O are said to be `io` functions. An `io` function must not be invoked from a non-`io` function.

pure: If a method has no visible side effects, it is said to be `pure`. This means that no side effects are performed to references. New objects may be allocated freely. A pure function invokes only other pure functions.

2.1.9 Function calls

A function value consists of three parts: the function's type, its body, and the environment in which it is defined.

As with languages such as Scheme and the Java Programming Language, function calls in Fortress are call-by-value. Each argument passed to a function is evaluated to a value before the function is applied. Arguments to a function can be specified at a call site in one of two ways, *positionally* or by as *keyword arguments*.

1. Positionally. If none of the parameters of a function definition include default expressions, the arguments are passed to the function as a tuple of expressions. The values of these expressions are bound to the parameters of the function in the order specified in the function declaration. If the last parameter `p` in the function declaration has type `T...` for some type `T`, then all arguments whose position is greater than or equal to the position of `p` are placed in an immutable array (i.e., an object of type `Array`⟦`T`⟧), and bound to `p`.
2. With keyword arguments. If a function definition consists of k parameters without default expressions (and no varargs parameter) followed by j parameters with default expressions, then there is a sequence of k expressions passed positionally, followed by a sequence of bindings to parameters with default values. The first k arguments are bound to the first k parameters of the function, as specified in its definition. The remaining arguments are passed as bindings v=e. For each binding v=e, the value of e is bound to the parameter with name v. If a boolean expression e_1=e_2 is passed as an argument, it must be parenthesized as `(`e_1=e_2`)`. If the function has k parameters

without default expressions followed by a varargs parameter `p`, then any arguments after the first k that are not passed as bindings are placed in an immutable array and bound to `p`. Parameters specified with default values can be bound only by keyword arguments.

Parameters not explicitly bound are bound to their default values. If a parameter that has no default value is not explicitly bound to an argument, a static error is signaled.

If the application of a function f ends by calling another function g, tail-call optimization must be applied. Storage used by the new environments constructed for the application of f must be reclaimed.

Examples:

```
sqrt(x)
atan(y, x)
makeColor(red=5, green=3, blue=43)
processString(s, start=5, end=43)
```

If the function takes a single argument, then the argument need not be parenthesized:

```
sqrt 2
sin x
log log n
```

Most function applications do not include explicit instantiations of type arguments; the type arguments are statically inferred from the context of the function application.

2.1.10 Operator applications

To support a rich mathematical notation, Fortress allows most Unicode characters that are specified to be mathematical operators to be used as operators in Fortress expressions, as well as these characters and character combinations:

```
!   @   #   $   %   *   +   -   =   |   :   <   >   /   ?
->   -->   =>   ==>   :->   <=   >=   /=   **   !!
```

In addition, a token that is made up of a mixture of uppercase letters and underscores (but no digits), does not begin or end with an underscore, and contains at least two different letters is also considered to be an operator:

```
MAX   MIN   SQRT   TIMES
```

Some of these uppercase tokens are considered to be equivalent to single Unicode characters, but even those that are not can still be used as operators.

All of the operators described above can be used as prefix, infix, postfix, or nofix operators as described in Section 2.7; the fixity of an operator is determined syntactically, and the same operator may have definitions for multiple fixities. A simple example is that ‘`-`’ may be either infix or prefix, as is conventional. As another example, the Fortress standard library (discussed in Chapter 4) defines ‘`!`’ to be a postfix operator that computes factorials when applied to integers.

Simple juxtaposition is also regarded as an infix operator in Fortress. When the left operand is a function, juxtaposition performs function application; when the left operand is a number, juxtaposition conventionally performs multiplication; when the left operand is a string, juxtaposition conventionally performs string concatenation.

Here are some examples of Fortress expressions where the brackets indicate subscripts:

```
(-b + sqrt(b^2 - 4 a c)) / 2 a
n^n e^(-n) sqrt(2 pi n)
a[k] b[n-k]
x[1] y[2] - x[2] y[1]
1/2 g t^2
n(n+1)/2
(j+k)!/(j!  k!)
1/3 3/5 5/7 7/9 9/11
17.3 meter/second
17.3 m_/s_
u DOT (v CROSS w)
u · (v × w)
(A UNION B) INTERSECT C
(A ∪ B) ∩ C
i < j <= k AND p PREC q
i < j ≤ k ∧ p ≺ q
print("The answers are " (p+q) " and " (p-q))
```

Another class of operators is always postfix: a ‘`^`’ followed by any ordinary operator (with no intervening whitespace) is considered to be a superscripted postfix operator. For example, ‘`^*`’ and ‘`^+`’ and ‘`^?`’ are available for use as part of the syntax of extended regular expressions. As a very special case, ‘`^T`’ is also considered to be a superscripted postfix operator, typically used to signify matrix transposition.

Certain infix mathematical operators that are traditionally regarded as *relational* operators, delivering boolean results, may be *chained*. For example, an expression such as $A \subseteq B \subset C \subseteq D$; it is treated as being equivalent to $(A \subseteq B) \wedge (B \subset C) \wedge (C \subseteq D)$ except that the expressions B and C are evaluated only once (which matters only if they have side effects). Fortress restricts such chaining to operators of the same kind and having the same sense of monotonicity; for example, neither $a \subseteq B \leq C$ nor $A \subseteq B \supset C$ is permitted.

Any infix operator that does not chain may be treated as *multifix*. If $n - 1$ occurrences of the same operator separate n operands where $n \geq 3$, then the compiler first checks to see whether there is a definition for that operator that will accept n arguments. If so, that definition is used; if not, then the operator is treated as left-associative and the compiler looks for a two-argument definition for the operator to use for each occurrence. As an example, the cartesian product $S_1 \times S_2 \times \cdots \times S_n$ of n sets may usefully be defined as a multifix operator, but ordinary addition `p + q + r + s` is normally treated as `((p + q) + r) + s`.

Finally, more than two dozen pairs of brackets are available that can be defined by the user as functions on any number of arguments. For example, angle brackets ⟨ ⟩ (not to be confused with the less-than and greater-than signs `< >`) may be used as a defined function of any desired number of arguments:

```
opr ⟨ x:Num               ⟩ :  Num = x^2
opr ⟨ x:Num, y:Num        ⟩ :  Num = x^2 + y^2
opr ⟨ x:Num, y:Num, z:Num ⟩ :  Num = x^2 + y^2 + z^2
⟨ 3       ⟩    (* evaluates to  9 *)
⟨ 3, 4    ⟩    (* evaluates to 25 *)
⟨ 2, 3, 4 ⟩    (* evaluates to 29 *)
```

Alternatively, we might have written a single definition to handle any number of arguments:

```
opr ⟨ x:Num...  ⟩ :  Num = SUM[a ∈ x] a^2
⟨             ⟩    (* evaluates to  0 *)
⟨ 3           ⟩    (* evaluates to  9 *)
⟨ 3, 4        ⟩    (* evaluates to 25 *)
⟨ 2, 3, 4     ⟩    (* evaluates to 29 *)
⟨ 2, 3, 4, 5 ⟩    (* evaluates to 54 *)
```

While the standard Fortress libraries are quite rich, there are many possible operators that are not predefined by the standard Fortress libraries and so are available for language extension by users.

Every operator application is equivalent in behavior to a function call. The behavior of every Fortress operator is defined by an explicit operator declaration. Frequently such a declaration will simply invoke an appropriate method. For example, the boolean operators `AND`, `OR`, `XOR`, and `NOT` are defined in the standard Fortress library as

```
opr AND(BoolOperators x, BoolOperators y) = x.and(y)
opr OR (BoolOperators x, BoolOperators y) = x.or (y)
opr XOR(BoolOperators x, BoolOperators y) = x.xor(y)
opr NOT(BoolOperators x                 ) = x.not()
```

(The arguments are of type `BoolOperators` so that these operator definitions may be shared by other types, such as `BoolInterval`, that support such operators. The trait `Bool` extends the trait `BoolOperators`. These are details that are of concern to library designers; application programmers need not be aware of them.)

2.1.11 Assignments

An assignment expression consists of a left-hand side indicating one or more variables to be updated, an assignment token, and a right-hand-side expression.

The assignment token may be ‘`:=`’, to indicate ordinary assignment; or may be any operator (other than ‘`:`’ or ‘`=`’ or ‘`<`’ or ‘`>`’ or ‘`/`’) followed by ‘`=`’ with no intervening whitespace, to indicate compound (updating) assignment.

A left-hand side may be a single variable or a tuple. If it is a tuple, then the right-hand side must be either a tuple of equal length or a funtion application that returns multiple values, equal in number of the length of the left-hand-side tuple.

If the left-hand side is a tuple and the assignment token is ‘`:=`’, then each element of the tuple may be a variable or a binding consisting of a variable, a colon, and a type (in which case the variable is declared as a local variable and initialized to a value rather than being assigned). If the left-hand side is a tuple and the assignment token is other than‘`:=`’, then each element of the tuple must be a variable; bindings are not permitted in this case. Examples:

```
x := f(0)
c[i,j] := c[i,j] + a[i,k] b[k,j]
(a, b, c) := (b, c, a)                          (* Permute a, b, and c *)
(q:Int, r:Int) := quotientAndRemainder(x, y)    (* Bind q and r *)
(q, s:Int) := quotientAndRemainder(x, y)        (* Assign q but bind s *)
x += 1
(x, y) += (delta_x, delta_y)
myBag = myBag ∪ newItems
myBag ∪= newItems
```

Variables updated in assignment expressions must be already declared.

The value of an assignment expression is `()`.

2.1.12 Block expressions

A block expression consists of the reserved word `do`, a series of expressions, and the reserved word `end`. The value of a block expression is the value of the last expression in the block. Some compound expressions have clauses that are implicitly block expressions. Here are examples of function definitions whose bodies are block expressions:

```
f(x:ℝ64) = do
  (sin(x) + 1)²
end
```

```
foo(x:ℝ64) = do
  y = x
  z = 2 x
  y + z
end
```

```
mySum(i:ℤ64):ℤ64 = do
  acc:Int := 0
  for j ← 0:i do
    acc := acc + j
  end
  acc
end
```

2.1.13 Labelled block expressions

Block expressions may be labelled with a name and any inner expression can exit the labelled block with an optional value. The same name is required after both `label` and `end`.

```
label I95
  if goingTo(Sun)
  then exit I95 with x32B
  end
end I95
```

2.1.14 Case expressions

A `case` expression evaluates a *test expression* and determines which of a set of case clauses applies to the test expression's value. When an applicable case clause is found (checking from left to right), the body of that case clause (and only that clause) is evaluated. If no applicable clause is found, an exception is thrown.

To find which case clause applies, the *guarding expression* of each case clause is evaluated in turn and compared to the value of the test expression. The two values are compared according to an optional binary method named by an identifier specified in the `case` expression immediately after the test expression. For example, we could write:

```
case planet ∈ of
  { Mercury, Venus, Earth, Mars } ⇒ "inner"
  { Jupiter, Saturn, Uranus, Neptune, Pluto } ⇒ "outer"
  else ⇒ "remote"
end
```

The special case clause `else` always applies; if it appears in a `case` expression, it must appear as the rightmost clause.

If the binary method is omitted, it defaults to = or ∈:

```
case 2 + 2 of
  4 ⇒ "it really is 4"
  5:7 ⇒ "we were wrong again"
end
```

The special reserved words `largest` and `smallest` may appear in a test expression context to select the largest (or smallest) quantity from a set of case clauses:

```
case largest of
  mile ⇒ "miles are larger"
  kilometer ⇒ "we were wrong again"
end
```

A more interesting example is described in Section 3.3.

2.1.15 Atomic expressions

An atomic expression consists of the reserved word `atomic` followed by a block expression. The block expression is executed as an atomic transaction. See Section 3.2.5 for a discussion of atomic memory transactions.

A function or method with modifier `atomic` acts as if its entire body were surrounded in an `atomic` expression.

2.1.16 Throw expressions

A `throw` expression consists of the reserved word `throw` followed by an expression which has the trait `Exception`. The thrown exception must be caught in an enclosing `try` expression or declared in the `throws` clause of the enclosing function definition.

2.1.17 Try expressions

`try` expressions start with the reserved word `try` followed by a sequence of expressions, and then `catch`, `forbid`, and `finally` clauses, as in the following example:

```
try
  do
    in = read(file)
    write(in, newFile)
  end
catch e
  IOException ⇒ throwException(e)
end
```

If a thrown exception matches the exception in a `forbid` clause, an exceptoin is thrown. For example, we could also write the above `try` expression as follows:

```
try
  do
    in = read(file)
    write(in, newFile)
  end
forbid IOException
end
```

`finally` clauses in `try` expressions are like `finally` blocks in the Java Programming Language. The `finally` clause is executed after the exception handler completes. For example,

```
try
  open(file)
  do
    in = read(file)
    write(in, newFile)
  end
catch e
  IOException ⇒ throwException(e)
finally
  close(file)
end
```

2.1.18 Function expressions

Function expressions denote functions. They start with the reserved word `fn` followed by a parameter list, optional return type, ⇒, and finally an expression. Unlike defined functions, function expressions are not allowed to include type parameters, `where` clauses, and contracts described in Section 2.5.8. Here is a simple example:

```
fn(x:double) ⇒ if x < 0 then -x else x end
```

2.1.19 Dispatch expressions

Fortress supports `dispatch` expressions, which provide a shorthand for multiple dispatch on a sequence of types. The form of these expressions is as follows:

```
dispatch ($v_1$=$e_1$,$v_2$=$e_2$,...,$v_n$=$e_n$) in
  ($t_{11}$,$t_{12}$,...,$t_{1n}$) $\Rightarrow$ $exprs_{v_1}$
  ($t_{21}$,$t_{22}$,...,$t_{2n}$) $\Rightarrow$ $exprs_{v_2}$
   ...
  ($t_{m1}$,$_{m2}$,...,$t_{mn}$) $\Rightarrow$ $exprs_{v_m}$
end
```

A dispatch expression evaluates the expressions $e_1, \cdots, e_n$ and then performs a type dispatch, exactly as if each clause were the header of an overloaded function described in Section 2.2.1. The most specific clause is chosen, and the corresponding value expressions *exprs*$_{v_j}$ are evaluated (and the value of the last expression of *exprs*$_{v_j}$ is the value of the construct). All the rules of function overloading apply; in particular, ambiguity is not allowed and the order of the clauses is irrelevant.

If $n = 1$, the parentheses may be elided, as in the following example:

```
dispatch x = myLoser.myField in
  String ⇒ x.append("foo")
  Num    ⇒ x + 3
  Thread ⇒ x.run()
  Object ⇒ yogiBerraAutograph
end
```

Note that “x” has a different type in each clause.

The syntactic sugar

```
dispatch $x$ in ...  end
```

(where x is a valid local identifier) is equivalent to:

```
dispatch $x$ = $x$ in ...  end
```

At least one clause’s type must be a supertype-or-equal of all the other clauses’ types.

2.1.20 Typecase expressions

A `typecase` expression has the same syntax as a `dispatch` expression except that the reserved word `typecase` occurs in place of `dispatch`. However, a `typecase` expression evaluates its clauses from top to bottom, and the first match is chosen. What would be forbidden ambiguity in a `dispatch` expression is allowed in a `typecase` expression.

2.1.21 Function contracts

Function contracts consist of three parts: a `requires` part, an `ensures` part, and an `invariant` part.

The `requires` part consists of a sequence of expressions of type `Bool`. The `requires` clause is evaluated during a function call before the body of the function. If any expression in a `requires` clause does not evaluate to `True`, an exception is thrown. For example, we can add a `requires` clause to our `factorial` function as follows:

```
factorial(n:Int)
  requires n ≥ 0
= if n = 0 then 1
  else n factorial(n-1) end
```

The `ensures` part consists of a sequence of `ensures` clauses. Each such clause consists of a sequence of expressions of type `Bool`, optionally followed by a `provided` clause. A `provided` clause begins with the reserved word `provided` followed by an expression of type `Bool`. For each clause in the `ensures` part of a contract, the `provided` clause is evaluated immediately after the `requires` clause during a function call (before the function is executed). If a `provided` clause evaluates to `True`, then the expressions preceding this `provided` clause are evaluated after the function is executed. If any of the expressions evaluated after function execution does not evaluate to `True`, an exception is thrown. The expressions preceding the `provided` clause can refer to the return value of the function. If there is a single return value for the function, a `result` variable is implicitly bound to the return value of the function. If there are multiple return values, an immutable array named `result` contains these values. A `result` variable scopes over the expressions preceding the `provided` clause. For example, we can write the following function:

```
print(input:List)
  ensures sorted(result) provided sorted(input)
= if x ≠ Empty then
    print(first(input))
    print(rest(input))
  end
```

The `invariant` clause consists of a sequence of expressions of *any type*. These expressions are evaluated before and after a function call. For each expression e in this sequence, if the value of e when evaluated before the function call is not equal to the value of e after the function call, an exception is thrown.

2.2 Traits

Programmers can define new types in their programs through *traits*. Traits are named collections of *methods*, which are functions that can be inherited and overridden. Methods are invoked on *objects*, which are values that have traits.

Syntactically, a trait definition starts with a sequence of modifiers followed by the reserved word `trait`, followed by the name of the trait, an optional set of *extended* traits, an optional set of *excluded* traits, an optional set of *bounds* on the trait, a list of method declarations and definitions, and the reserved word `end`. Syntactically, a method declaration is identical to a function declaration, except that a special `self` parameter is provided immediately before the name of the method. When a method is invoked, the `self` parameter is bound to the object on which it is invoked. If no `self` parameter is provided explicitly, it is implicitly a parameter with name `self`.

For example, the following trait definition:

```
trait Catalyst extends Molecule
  self.catalyze(reaction: Reaction): ()
end
```

defines a trait `Catalyst` with no modifiers, no `excludes` clauses, and no `bounds` clauses. Trait `Catalyst` extends a single trait named `Molecule`. A single method (named `catalyze`) is declared, which has a parameter of type `Reaction` and the return type `()`.

Methods are invoked with the following syntax:

receiver`.`*app*

where *receiver* evaluates to the receiver of the invocation (bound to the `self` parameter of the method). There must be no whitespace on either side of the '`.`', and there must be no whitespace between the method name and the left parenthesis of the argument list. *app* is syntactically identical to a function application, except that the non-`self` arguments must be parenthesized, even if there is only one of them. All non-`self` parameters are bound in a manner identical to function application. Examples:

```
myString.toUppercase()
myString.replace("foo", "few")
SolarSystem.accelerate((π/2 radian) / 452 million year)
```

Even if a method takes a single argument, it must nevertheless be parenthesized:

`myNum.add(otherNum)` (*not* `myNum.add otherNum`)

Every trait extends the built-in trait `Object`. Every trait with an `extends` clause extends every trait appearing in its `extends` clause. If a trait T extends trait U, we call T a subtrait of U and U a supertrait of T. Extension is transitive; if T extends U it also extends all supertraits of U. Extension is also reflexive: T extends itself. The extension relation induced by a program is the smallest relation satisfying these conditions. This relation must form an acyclic hierarchy rooted at trait `Object`.

We say that trait T *strictly extends* trait U if and only if (*i*) T extends U and (*ii*) T is not U. We say that trait T *immediately extends* trait U if and only if (*i*) T strictly extends U and (*ii*) there is no trait V such that T strictly extends V and V strictly extends U. We call U an *immediate supertrait* of T and T an *immediate subtrait* of U. If a trait definition of T includes a `bounds` clause, the trait must not be extended with immediate subtraits other than those that appear in its `bounds` clause. Furthermore, T serves as the least upper bound of the traits appearing in its `bounds` clause. For example, the trait:

```
trait Molecule
  bounds {OrganicMolecule, InorganicMolecule}
  mass(): Mass
end
```

is bounded by two traits: `OrganicMolecule` and `InorganicMolecule`. Therefore, the following trait definition is not allowed:

```
(* Not allowed! *)
trait ExclusiveMolecule extends Molecule end
```

If a trait T excludes a trait U, the two traits are mutually exclusive. No object can have them both, no third trait can extend them both, and neither can extend the other. U can optionally exclude T. For example, we define traits `OrganicMolecule` and `InorganicMolecule` as follows:

```
trait OrganicMolecule extends Molecule
  excludes {InorganicMolecule}
```

```
end

trait InorganicMolecule extends {Molecule} end
```

`OrganicMolecule` excludes `InorganicMolecule`. It does not matter that `InorganicMolecule` has no `excludes` clause; the traits are mutually exclusive solely on account of the definition of `OrganicMolecule`. For example, the following trait definition is not allowed:

```
(* Not allowed! *)
trait InclusiveMolecule extends {InorganicMolecule, OrganicMolecule} end
```

A trait is allowed to have multiple immediate supertraits. The following trait has two immediate supertraits:

```
trait Enzyme extends {OrganicMolecule, Catalyst}
  reactionSpeed(): Speed
  catalyze(reaction) = reaction.accelerate(reactionSpeed())
end
```

Traits inherit methods from their immediate supertraits; In fact, a trait inherits every method from every one of its immediate supertraits *except* for methods that are overridden by declarations in the trait itself. In our example, `Enzyme` inherits the abstract method `mass` from `OrganicMolecule` and overrides the abstract method `catalyze` from trait `Catalyst`.

We say that a declaration of a function or method *occurs* in a trait definition if and only if the trait definition either syntactically contains the declaration or inherits the declaration. If a declaration occurs in a trait definition, we say the trait definition *supplies* the declaration. For example, trait `Enzyme` supplies methods `mass` and `catalyze`, but it syntactically contains only the declaration of method `catalyze`.

2.2.1 Overriding and overloading

A *signature* of a method consists of the name of the method, the number and types of its formal parameters, and the names of keyword arguments. Note that the type of the receiver of a method and the return type of a method are not parts of its signature. A method declaration *overrides* a declaration in a supertrait if and only if the *signatures* of the two declarations match exactly. If a declaration with return type `T` is overridden by a declaration with return type `U`, then `U` must extend `T`. It is not permitted for an abstract method declaration to override a concrete method declaration.

If a trait inherits multiple methods with the same name, those declarations must conform to the restrictions explained in Section 2.6 on multiple dispatch.

For every trait, there is a corresponding static type with the same name, called a *trait type*. If trait S extends trait T, then the trait type of S is a subtype of the trait type of T. If an expression e has a trait type, then any method supplied by the trait can be invoked on e.

2.2.2 Method contracts

Contracts on methods are handled similarly to the manner described in [10]. In particular, substitutability under subtyping is preserved.

Evaluation of a call site $e.m(...)$, where e has static type T, proceeds as follows. First, e is reduced to a value v with runtime type U. Let C be the contract declared in the declaration of m determined by static type T. We call C the *pivot contract* of the call site. The `requires` clause of C is checked. If that `requires` clause fails, a `CallerViolation` exception is thrown.

Otherwise, consider every contract C' in every declaration of m that overrides the declaration of m determined by T, as well as declarations of m in any supertype of T. If C' occurs in a type V that is a supertype of e's runtime type U, then the `requires` clause of C' is checked. If any such `requires` clause fails, a `ContractHierarchy` exception is thrown.

Otherwise, consider every contract C'' in every declaration of m that occurs either in a supertype of U (including U itself). The `provided` clauses of C'' are evaluated. For every `provided` clause that evaluates to `True`, the corresponding `ensures` clause is recorded in a table E for later evaluation. Similarly, the `invariant` clauses of C'' are evaluated and the results are stored in E for later comparison.

Then the body of m (as determined by e's runtime type U) is evaluated. After evaluation of the body, all `ensures` clauses in E that are declared in the contract in U are checked, and all `invariant` clauses in E declared in U are checked to ensure that they reduce to values equal to the values they reduced to before evaluation of the body. If any such check fails, a `CalleeViolation` exception is thrown. Otherwise, all other `ensures` clauses and `invariant` clauses in E are checked. If any of these clauses fail, a `ContractHierarchy` exception is thrown.

2.3 Objects

Objects are values that have object types described in Section 2.5.1. Objects contain *fields* and *methods*, and have a set of traits from which they inherit methods.

Syntactically, an object definition begins with a sequence of modifiers followed by the reserved word `object`, followed by the name of the object, the traits of the object, the *fields* of the object, the methods of the object, and finally the reserved word `end`. The traits of an object are listed in an optional `traits` clause, which starts with the reserved word `traits` followed by a sequence of one or more trait references separated by commas and enclosed in braces '{' and '}'. If a `traits` clause contains only one trait, the enclosing braces may be elided. If an object definition has no `traits` clause, the object is understood to have only trait `Object`.

For example, we define an empty list object with trait `List` as follows:

```
object Empty traits {List}
  first() = throw Error
  rest() = throw Error
  cons(x) = Cons(x, self)
  append(xs) = xs
end
```

This object has no fields and four methods.

Fields are variables local to an object. They must not be referred to outside their enclosing object definitions. Field declarations in an object are syntactically identical to defined bindings, with the same meanings attached to the form of binding. However, special modifiers can precede a field declaration:

hidden: By default, a field declaration implicitly defines a *getter* method for the field. This method takes no arguments, has the same name as the field, and a return type equal to the field type. The implicitly defined getter returns the value of the field when called. A field with modifier `hidden` has no implicitly defined getter.

settable: A field with this modifier has an implicitly defined *setter*. This method takes a single parameter (with no default) whose type is the type of the field. It returns `()`. When called, the implicitly defined setter rebinds the corresponding field to its argument. A settable field must not be immutable.

Method declarations in objects are identical to their syntax in traits.

The implicitly defined getters and setters of fields can be overridden with methods with the appropriate signatures, names, and return types. An explicitly defined getter must include the modifier `getter`. An explicitly defined setter must include the modifier `setter`.

A getter method must be invoked with the syntax:

expr`.`*name*

where *name* is the name of the getter.

A setter method must be invoked with the *assignment syntax*:

$expr_1$`.`*name* `:=` $expr_2$

Getter and setter methods can be declared in traits as well. Syntactically, such definitions are bindings. If such a binding is a defined binding, a getter is defined with the expression in the defined binding. If the binding is not a defined binding, the getter is abstract. If the binding includes the modifier `settable` an abstract setter is also declared. If the binding includes the modifier `settable` and `hidden`, only an abstract setter is declared. Such a binding must not include the modifier `hidden` without `settable`, or a static error is signaled.

A getter must not be overridden by a method other than a getter. A setter must not be overridden by a method other than a setter.

2.3.1 Object expressions

Object expressions denote objects. They start with the reserved word `object` followed by the ordinary aspects of an object definition (except for the name). Unlike top-level object definitions, object expressions are not allowed to include type parameters, `where` clauses, and contracts. For example, the following is a valid Fortress expression:

```
object traits {StarSystem, OrbitingObject}
  sun = Sol
  planets =
    { Mercury, Venus, Earth, Mars,
      Jupiter, Saturn, Uranus, Neptune, Pluto }

  position = Polar (25000 lightYear, 0 radian)
  ω:radian/s := 2 π radian / 226 million year in s

  accelerate(θ) = ω := ω + θ
end
```

This expression evaluates to a new object, with traits `StarSystem` and `OrbitingObject`.

2.3.2 Parametric objects

Object definitions are also allowed to be parametric with respect to other objects. Object parameters are specified after an object's type parameters. They are enclosed in parentheses and are separated by commas. Syntactically, each object parameter is identical to the beginning of a field definition; it consists of a sequence of modifiers followed by the name of the parameter, followed by a `:`, a type, and, optionally, a default value. Here is an example of a parametric `Cons` object with trait `List⟦T⟧`:

```
object Cons⟦T⟧
  ( first: T,
    rest : List⟦T⟧ )
  traits List⟦T⟧
  cons  (x ) = Cons(x,self)
  append(xs) = Cons(first,rest.append(xs))
end
```

Note that this declaration implicitly introduces the "factory" function:

```
Cons⟦T⟧(first:T, rest:List⟦T⟧)
```

which is used in the body of the trait to define the `cons` and `append` methods. Multiple factory functions can be defined by overloading a parametric object with functions. For example:

```
Cons⟦T⟧(first:T) = Cons(first,Empty)
```

transient: A parameter to a parametric object can be declared `transient`, indicating that it doesn't correspond to a field in an instantiation of the object. `transient` parameters are in scope only in other field definitions of an object; they are not in scope in the object's method definitions.

Fields can be explicitly defined within a parametric object as usual. All fields of an object are initialized before that object is made available to subsequent computations.

As with functions, parametric object definitions are allowed to include contracts (`requires`, `ensures`, and `invariant` clauses). Syntactically, these contracts appear after the `traits` clause and before the field definitions of an object. They are called at the appropriate times during an instantiation of the object.

wrapped: If the field f with trait type T is declared to have modifier `wrapped`, then the enclosing object implicitly includes "forwarding methods" for all methods in T. Each of these methods simply calls the corresponding method on the object referred to by field f. If the object definition enclosing f explicitly defines a method m that conflicts with an implicitly defined forwarding method m', then the enclosing object contains only method m, not m'. The signature of m must be a valid overriding signature of m' or a static error is signaled.

For example, in the following definitions:

```
trait Dictionary⟦T⟧
  put(x:T):()
  get():T
end
```

```
object WrappedDictionary⟦T⟧
  ( wrapped val:Dictionary⟦T⟧ )
  traits Dictionary⟦T⟧
  get() = throw Error
end
```

the parametric object `WrappedDictionary` implicitly includes the following forwarding method:

```
  put(x) = val.put(x)
```

If `get` were not explicitly defined in `WrappedDictionary`, then `WrappedDictionary` would also include the forwarding method:

```
  get() = val.get()
```

2.4 Value Objects

There is a special modifier `value` in the language. Conceptually, an object definition with modifier `value` is understood to define what is called in many languages a *primitive* value. For example, here is a definition of a parametric, primitive, `Complex` number:

```
value object Complex(real:Double, imaginary:Double)
  opr +(other:Complex) = Complex(real + other.real(),
                                 imaginary + other.imaginary())
  ...
end
```

A variable defined with modifier `value` (including the name of an object definition) implicitly has modifier `pure`, indicating that it must not be assigned to. The fields of a `value` object are implicitly `pure`, indicating that they cannot be assigned to.

2.4.1 Value object types

If a trait `T` has modifier `value`, all objects with that trait are required to be value objects. The object type defined by a value object implicitly has the modifier `value`.

2.4.2 Predefined value objects

We are now in a position to expand our description of several of the built-in types, and, in some cases, how they might be implemented in a library. For Java there was a conscious attempt to minimize the number of distinct primitive types to reduce programmer confusion. Fortress has a richer set of types to address a richer set of programming situations.

Booleans

Booleans include the traditional `True`/`False` objects:

```
value trait  Bool  end
value object True  traits Bool end
value object False traits Bool end

opr ∧(b0:Bool, b1:Bool):Bool
opr ∨(b0:Bool, b1:Bool):Bool
opr ¬(b0:Bool):Bool
opr ⊕(b0:Bool, b1:Bool):Bool

opr ∧(b0:True, b1:Bool) = b1
opr ∨(b0:True, b1:Bool) = True
opr ¬(b0:True) = False
opr ⊕(b0:True, b1:Bool) = ¬b1

opr ∧(b0:False, b1:Bool) = False
opr ∨(b0:False, b1:Bool) = b1
opr ¬(b0:False) = True
opr ⊕(b0:False, b1:Bool) = b1
```

We also provide for `Bool` intervals and possibly other `Bool` algebras.

Characters

In addition to the `Char` and `String` types already described, Unicode also has the idea of a grapheme, which is sort of like a character but may be represented as a sequence of Unicode code points, typically a base character plus a set of combining marks such as accents. We may want to allow for “grapheme” and “grapheme string” data types. For this purpose, `Grapheme` and `String` should be traits, which various sorts of objects may have. Because users of other languages will expect `Char` to be small and cheap, we will use that to name the value type of Unicode characters, and it will have the trait `Grapheme`, but so will other objects that contain appropriate sequences of characters. Similarly, `UTF32String`, `UTF8String`, `GraphemeString`, and so on may have the trait `String`.

Numbers

In addition to the number types already described, Fortress allows various numeric (and other) types of traits in libraries that represent algebraic structures of interest such as monoids, groups, rings, and fields. For example, reduction operators generally can accept any type of a monoid trait with the appropriate binary operator.

2.5 Types

Types in Fortress include all built-in types, all trait types, and all object types. Additionally, Fortress supports several forms of parametric polymorphism, described in this section.

2.5.1 Object types

A defined object has an object type (of the same name). The object type defined by an object definition includes, as abstract methods, all of the public methods, including all implicitly defined getters and setters, introduced by the object definition (i.e., those methods not declared by any traits of the object). It extends all of the declared traits of the object. No other objects can have the object type and no traits can extend an object type

2.5.2 Trait parameters

A trait is allowed to be parametric with respect to other traits. These *trait parameters* are listed in white square brackets '⟦' and '⟧' immediately after the name of the trait. We use the term *naked trait variable* to refer to an occurrence of a trait variable as a stand-alone trait (rather than as a syntactic subcomponent of a larger trait reference). They are in scope in the entire body of the trait definition, and can appear in any context that an ordinary type can appear, except that a naked trait variable must not appear in the `extends` clause of the trait definition. Here is a parametric version of trait `List`:

```
trait List⟦T⟧
  first(): T
  rest (): List⟦T⟧
  cons (x: T): List⟦T⟧
  append(xs: List⟦T⟧): List⟦T⟧
end
```

Trait parameters are allowed to have bounds placed on them in a `where` clause. A `where` clause begins with the reserved word `where`, followed by a sequence of trait parameter constraints enclosed in braces '{' and '}' and separated by commas. For example, we could place a constraint on the trait parameter of `List` as follows:

```
trait List⟦T⟧ where {T extends Comparable}
  first(): T
  rest (): List⟦T⟧
  cons (x: T): List⟦T⟧
  append(xs: List⟦T⟧): List⟦T⟧
  sort (): List⟦T⟧
end
```

A `where` clause is allowed to introduce new trait *variables*, i.e., identifiers for traits that may not be trait parameters. Trait variables that are not also trait parameters are in scope only in the `extends` and `where` clauses of a trait. For example, we can write a trait definition like the following:

```
trait C⟦S⟧ extends D⟦T⟧
  where {S extends T}
end
```

In this example, for every subtype `S` of `T`, `C⟦S⟧` is a subtype of `D⟦T⟧`. For example, both `C⟦Object⟧` and `C⟦SolarSystem⟧` are subtypes of `D⟦Object⟧`.

Each trait constraint in a `where` clause is either a type alias (described in 2.5.6) or begins with the name of a naked trait variable, followed by the reserved word `extends`, followed by a trait reference. This trait reference is allowed to be any valid trait reference in the enclosing scope including a naked trait variable. Mutually recursive bounds are allowed in `where` clauses. A trait parameter that is not explicitly bound in a `where` clause is implicitly bound by trait `Object`. All trait variables in an object or trait definition must occur either as a trait parameter or as a bound trait variable in a `where` clause.

Trait definitions are allowed to extend other instantiations of themselves. For example, we can write:

```
trait C⟦S⟧ extends C⟦T⟧
  where {S extends T}
end
```

In this definition, for every subtype `S` of `T`, `C⟦S⟧` is a subtype of `C⟦T⟧`.[2] A trait parameter that is bound by a naked trait variable must not appear in the types of method parameters. A trait parameter that serves as the bound of a trait variable must not appear in the types of method return values. *These restrictions apply both to programmer-defined methods and to the implicitly defined methods such as getters.*

In fact, trait definitions need not have any trait parameters in order to have a `where` clause. For example, the following trait definition is legal:

```
trait C extends D⟦T⟧
  where {T extends Object}
end
```

In this definition, trait `C` is a subtrait of *every* instantiation of parametric trait `D`. Thus, trait `C` has all of the methods of every instantiation of `D`. By thinking of the declaration this way, we can see what limitations we need to impose on the body of trait `C` in order for it to be sensible. If trait `C` inherits a method definition that refers to `T`, it really contains infinitely many methods (one for each instantiation of `T`), so it must be possible to infer which method is referred to at a call site. If `C` inherits an abstract method definition, then an object with trait `C` must be able to define this method without referring to trait variable `T`, (which is not in scope in the definition of `C`, nor in any object definition with trait `C`). In Fortress, the only valid way to write such a method body is to throw an exception.

Object definitions are also allowed to include `where` clauses. Here is an alternative definition of an `Empty` list:

```
object Empty traits List⟦T⟧ where {T extends Object}
  first() = throw Error
  rest () = throw Error
  cons(x) = Cons(x,self)
  append(xs) = xs
end
```

where `Cons` is defined in Section 2.3.2 and `self` denotes the object itself.

2.5.3 Nat type parameters

Trait definitions are allowed to be parametric with respect to a sequence of `nat` type parameters. These parameters are instantiated at runtime with numeric values. They are allowed to be used to instantiate other `nat` type parameters, or to appear in any context that a variable of type `nat` can appear, except that they cannot be assigned to. Syntactically, a

[2]Effectively, we have expressed the fact that the type parameter `S` of `C` is covariant.

`nat` type parameter is declared along with other type parameters, and begins with the reserved word `nat` followed by a variable name. For example, the following function `f`:

```
f⟦nat n⟧(x:Length^{2n}): Length^n = sqrt(x)
```

declares a `nat` type parameter `n`, which appears in both the parameter type and return type of `f`.

The set of expressions allowed to instantiate a `nat` type parameter includes all `nat` constants along with all `nat` type parameters, and is closed under addition and multiplication. Static determination of the equivalence of such expressions is limited to a simple normalization process where all factors are distributed, the variables of each term are put in lexicographic order, and the normalized terms are put in lexicographic order. For example, the `nat` type:

```
(d + a)·(c + b)
```

is normalized to:

```
a·b + a·c + b·d + c·d
```

Method and function definitions are also allowed to be parametric with respect to a sequence of `nat` type parameters.

2.5.4 Dimensions

There are special types called *dimensions* that are separate from traits. Dimensions must be declared globally in a program component. For every two dimensions `D` and `E`, there is a dimension `D E`, corresponding to the product of the dimensions `D` and `E` and a dimension `D/E`, corresponding to the quotient of the dimensions `D` over `E`. There is also a dimension `D^n` (henceforth written $\mathtt{D}^{\mathtt{n}}$) for every `nat` type `n`. Instances of a given dimension are referred to as *quantities*. The set of declared dimensions have the algebraic structure of a free abelian group. The identity element of this group is dimension `Unity` which represents dimensionless quantity. The syntactic sugar `1/D` is equivalent to `Unity/D` for all dimensions `D`.

For each dimension `D` referred to in a Fortress program component other than dimension `Unity`, exactly one variable of that dimension must be declared globally as a `unit` variable and must not include a definition. This variable may appear in the definitions of other variables of the same dimension. For example, we might include the following declarations in a program:

```
dim Length
unit m : Length
k = 1000

circumference = 40075 k m
```

Although exactly one `unit` variable of each dimension must not include a definition, other variables are allowed to be `unit` variables along with a definition. If multiple imported variables of a given dimension are declared to be `unit`, all but one must be given a definition in the importing program component.

2.5.5 Dimension parameters

Trait definitions are allowed to be parametric with respect to a sequence of dimension parameters. Syntactically, a parameter begins with the reserved word `dim` followed by a variable name, and occurs alongside other type parameter declarations. For example, here is a function that is parametric with respect to a dimension:

```
trait Coordinates⟦dim D⟧
  nth(n:Int):D
end
```

These parameters are allowed to appear in any context that a dimension can appear.

2.5.6 Type aliases

Fortress allows names to serve as aliases for more complex type instantiations. The *type alias* begins with the reserved word `type` followed by the alias name, followed by `=`, followed by the type it stands for. Here are some examples:

```
type IntList = List⟦Int⟧
type Area    = Length²
type BinOp   = (Float, Float) → Float
```

All uses of type aliases are expanded before type checking. Type aliases do not define nominal equivalence relations among types. Type aliases must not be recursively defined.

2.5.7 Operator parameters on traits

Traits may be parameterized with respect to operator symbols and names of methods. Syntactically, these parameters occur along with other trait parameters and are prefixed with the reserved word `opr`. Here are some examples:

```
trait UnaryOperator⟦T, opr OPR⟧
  where { T extends UnaryOperator⟦T,OPR⟧ }
  OPR():T
end

trait BinaryOperator⟦T, opr OPR⟧
  where { T extends BinaryOperator⟦T,OPR⟧ }
  OPR(that:T):T
end

trait UnaryPredicate⟦T, opr OPR⟧
  where { T extends UnaryPredicate⟦T,OPR⟧ }
  OPR():Bool
end

trait BinaryPredicate⟦T, opr OPR⟧
  where { T extends BinaryPredicate⟦T,OPR⟧ }
  OPR(that:T):Bool
end
```

2.5.8 Parametric functions

Functions and methods are also allowed to be parametric with respect to a sequence of trait, `nat`, and dimension parameters. Syntactically, these trait parameters are listed in white square brackets immediately before a function's ordinary parameters. They are in scope in the entire body of the method definition, and are allowed to appear in all contexts that ordinary types appear. Bounds may be put on these parameters in a `where` clause occurring after all other parts of the header of a function. For example, here is a simple polymorphic function for creating lists:

```
List⟦T⟧(rest: T...) where {T extends Object} =
  do
    length = rest.length()
    if length = 0 then Empty
    else Cons(rest[0], List(rest.asTuple(1, length - 1)) end
  end
```

Here is a simple function that is parametric with respect to a dimension:

```
square⟦dim D⟧(x:D):D² = x²
```

2.5.9 Array types

Array types are written as `T[`e_0`, `e_1`, ..., `e_m`]` where `T` is the type of the elements and e_i ($0 \leq i \leq m$) is either a `nat` type corresponding to the size of the i-th dimension of the array or the range n_1`:`n_2 meaning that an index of the i-th dimension of the array is between n_1 (inclusive) and n_2 (exclusive).

2.5.10 Arrow types

Functions can be passed as arguments and returned as values. The types of function values are called *arrow types*. An arrow type specifies the types of parameters to the function, the types of return values, and (optionally) the checked exceptions of thrown values. Syntactically, an arrow type occurs in either of the following forms:

1. Positionally. The type consists of a sequence of parameter types in parentheses followed by the token $\rightarrow$, followed by a sequence of return types, and optionally a `throws` clause. Here are some examples:

   ```
   (Float, Float) → Float
   Int → (Int, Int) throws IOException
   ```

2. With keyword arguments. This form is like the positional form, except that some parameters have names. All parameters with names are default parameters that should be called with keyword arguments. For example:

   ```
   (Int, Int, p:Printer) → Int
   ```

2.6 Overloading and Multiple Dispatch

Fortress allows functions and methods to be overloaded in the context of a single lexical scope. Calls to overloaded functions and methods are resolved via dynamic dispatch. In this section, we define the mechanism for this dynamic dispatch, and the restrictions placed on overloaded definitions. First, we introduce some terminology.

Recall that two traits can also be defined to be disjoint, according to their `excludes` clauses. Therefore any two traits `A` and `B` are related by exactly one of the following relationships:

equality	$\mathtt{A} = \mathtt{B}$	`A` is `B`
subtrait	$\mathtt{A} \prec \mathtt{B}$	`A` strictly extends `B`
supertrait	$\mathtt{A} \succ \mathtt{B}$	`B` strictly extends `A`
incompatible	$\mathtt{A} \parallel \mathtt{B}$	`A` is disjoint from `B`
incomparable	$\mathtt{A} \sim \mathtt{B}$	none of the above

We write $\mathtt{A} \preceq \mathtt{B}$ to mean that `A` extends `B` (that is, $(\mathtt{A} \prec \mathtt{B}) \vee (\mathtt{A} = \mathtt{B})$); similarly, we write $\mathtt{A} \succeq \mathtt{B}$ to mean that `B` extends `A` (that is, $(\mathtt{A} \succ \mathtt{B}) \vee (\mathtt{A} = \mathtt{B})$).

Similarly, a trait can be defined to have a fixed set of immediate subtraits, according to its `bounds` clause.

We write $\mathtt{T}^+$ to mean a sequence of n types $\mathtt{T}_1, \mathtt{T}_2, \ldots, \mathtt{T}_n$, and we write $\mathtt{T}^*$ to mean a sequence of $n+1$ types $\mathtt{T}_0, \mathtt{T}_1, \mathtt{T}_2, \ldots, \mathtt{T}_n$. Henceforth we assume that n is the same for all sequences under discussion, restricting our attention to only functions and methods that have n parameters and to function calls and method calls that have n arguments. Everything that follows is true separately for each possible value of n. Method declarations, function declarations, method calls, and function calls do not interact at all if they have different values for n. Functions and methods with variable argument parameters must not be overloaded. Similarly, functions and methods with keyword parameters must not be overloaded. However, there can be a single function with keyword parameters and with the same name as a set of overloaded functions; calls to this function can be determined syntactically, as keyword arguments are always present. For brevity, we refer to functions and methods that can be overloaded as *dispatched functions*.

Any two sequences of types are related by one of five relationships:

equality	$\mathtt{T}^+ = \mathtt{U}^+$	$\forall 1 \leq i \leq n : \mathtt{T}_i = \mathtt{U}_i$
more specific	$\mathtt{T}^+ \sqsubset \mathtt{U}^+$	$(\forall 1 \leq i \leq n : \mathtt{T}_i \preceq \mathtt{U}_i)$ and not $\mathtt{T}^+ = \mathtt{U}^+$
less specific	$\mathtt{T}^+ \sqsupset \mathtt{U}^+$	$(\forall 1 \leq i \leq n : \mathtt{T}_i \succeq \mathtt{U}_i)$ and not $\mathtt{T}^+ = \mathtt{U}^+$
incompatible	$\mathtt{T}^+ \parallel \mathtt{U}^+$	$\exists 1 \leq i \leq n : \mathtt{T}_i \parallel \mathtt{U}_i$
incomparable	$\mathtt{T}^+ \sim \mathtt{U}^+$	none of the above

We write $\mathtt{A} \sqsubseteq \mathtt{B}$ to mean $(\mathtt{A} \sqsubset \mathtt{B}) \vee (\mathtt{A} = \mathtt{B})$; similarly, we write $\mathtt{A} \sqsupseteq \mathtt{B}$ to mean $(\mathtt{A} \sqsupset \mathtt{B}) \vee (\mathtt{A} = \mathtt{B})$.

A dispatched function is overloaded only with other dispatched functions whose definitions appear in the same lexical scope. If a dispatched function definition uses the same name as a function or method in an enclosing scope, all dispatched functions with that name in the enclosing scope are shadowed; only functions of that name in the new scope are accessible. When a subtrait *T* of a trait *S* defines a set of overloaded methods *N* with the same name as a set of overloaded methods *M* in *S*, the methods *N override* the methods *M* if and only if for every method `m` in *N* there is a method `m` in *M* with the same signature; a call to such a method on an object *O* with trait *T* dispatches to a method in *N*.

A declaration is *visible* from a given program point *Z* if it occurs in a trait definition or block *B* that lexically contains *Z*.

We write `f(`$\mathtt{P}^+$`)` to refer to a declaration for a function named `f` whose parameter types are the sequence $\mathtt{P}^+$. By an abuse of notation, we similarly write `f(`$\mathtt{A}^+$`)` to refer to a call to a function named `f` where the arguments given in the call have (static) types $\mathtt{A}^+$. By a further abuse of the notation, we write `f(`$\mathtt{X}^+$`)` to describe a call to a function for which the dynamic object types of the calculated actual arguments at run time are $\mathtt{X}^+$. (Note that if the type system is sound—which we certainly hope it is!—then $\mathtt{X}^+ \sqsubseteq \mathtt{A}^+ \sqsubseteq \mathtt{P}^+$.) Context will distinguish which of these three cases are meant.

A declaration `f(`$\mathtt{P}^+$`)` is *more specific* than another declaration `f(`$\mathtt{Q}^+$`)` iff $\mathtt{P}^+ \sqsubset \mathtt{Q}^+$.

A declaration $\mathtt{f(P^+)}$ is *dynamically applicable* to a function call $\mathtt{f(X^+)}$ with dynamic argument types $\mathtt{X^+}$ iff $\mathtt{P^+} \sqsupseteq \mathtt{X^+}$.

A declaration $\mathtt{f(P^+)}$ is *statically applicable* to a function call $\mathtt{f(A^+)}$ with static argument types $\mathtt{A^+}$ iff $\mathtt{P^+} \sqsupseteq \mathtt{A^+}$.

A declaration $\mathtt{f(P^+)}$ is *accessible* to a function call $\mathtt{f(X^+)}$ iff it is visible from the function call.

A declaration $\mathtt{f(P^+)}$ is *applicable* to a function call $\mathtt{f(X^+)}$ iff it is dynamically applicable to the function call.

The basic principle for a function call or method call, as in Java, is that we wish to identify a unique concrete declaration that is the most specific among all declarations that are both accessible and applicable at the point of the call. (However, the meanings of the terms “accessible” and “applicable” are slightly different for Fortress from their meanings for Java.) If there is no such concrete declaration, it is of course an error; moreover, if there are two or more such concrete declarations, no one of which is more specific than all the others, the call is said to be *ambiguous*, which is also an error.

Now we introduce a requirement on programs that is more stringent than in Java.

The Meet Rule (for functions): Suppose that two distinct declarations for a function named `f` are accessible at some point Z in a Fortress program (Z need not be the site of a function call); call these two declarations $\mathtt{f(P^+)}$: $\mathtt{R^+}$ and $\mathtt{f(Q^+)}$: $\mathtt{S^+}$, where $\mathtt{P^+}$ and $\mathtt{Q^+}$ are the sequences of parameter types and $\mathtt{R^+}$ and $\mathtt{S^+}$ are the sequences of return types for the two declarations. It is a static error if the following condition does not hold:

either

$\mathtt{P^+} \parallel \mathtt{Q^+}$

[parameter types are disjoint at some parameter position]

or

all three of

$\forall 1 \leq i \leq n : (\mathtt{P}_i \prec \mathtt{Q}_i \vee \mathtt{P}_i = \mathtt{Q}_i \vee \mathtt{P}_i \succ \mathtt{Q}_i)$

[parameter types are comparable at all positions]

and

$\exists 1 \leq i \leq n : \mathtt{P}_i \neq \mathtt{Q}_i$

[parameter types differ at some position]

and

there is a declaration visible from Z for $\mathtt{f(P^+ \sqcap Q^+):T^+}$, where $\mathtt{T^+} \sqsubseteq \mathtt{R^+} \wedge \mathtt{T^+} \sqsubseteq \mathtt{S^+}$

[if there is an ambiguity, a third declaration with more specific or equal return types must resolve it]

where the *meet* operator on sequences of types is defined as

$$(\mathtt{P^+} \sqcap \mathtt{Q^+})_i = \begin{cases} \mathtt{P}_i & \text{if } \mathtt{P}_i \prec \mathtt{Q}_i \\ \mathtt{P}_i & \text{if } \mathtt{P}_i = \mathtt{Q}_i \\ \mathtt{Q}_i & \text{if } \mathtt{P}_i \succ \mathtt{Q}_i \\ \text{undefined} & \text{if } \mathtt{P}_i \parallel \mathtt{Q}_i \\ \text{undefined} & \text{if } \mathtt{P}_i \sim \mathtt{Q}_i \end{cases}$$

Notice that this requirement makes no mention of any specific function call that might refer to such declarations. This is in contrast to Java, where the prohibition against ambiguity applies only to method calls that actually appear in the program.

Notice also that it may be that $\mathtt{P^+} \sqcap \mathtt{Q^+} = \mathtt{P^+}$ or $\mathtt{P^+} \sqcap \mathtt{Q^+} = \mathtt{Q^+}$, in which case the requirement that there be a declaration for $\mathtt{f(P^+ \sqcap Q^+)}$ is trivially satisfied, there is no ambiguity, and a separate third declaration is not needed after all.

To put this requirement simply: static overloading ambiguity is forbidden. If two function declarations create the potential for an ambiguous function call because neither is more specific than the other, then there must be a third function declaration that is more specific than either and covers all the ambiguous cases.

(This requirement should not be difficult to obey, especially because the compiler can give useful feedback. First example:

```
foo(x:Num, y:Integer) = ...
foo(x:Integer, y:Num) = ...
```

Assuming that `Integer` $\prec$ `Num`, the compiler reports that these two declarations are a problem because of ambiguity and suggests that a new declaration for `foo(Integer, Integer)` would resolve the ambiguity. Second example:

```
bar(x:Printable) = ...
bar(x:Throwable) = ...
```

Assuming that `Printable` $\sim$ `Throwable`, the compiler reports that these two declarations are a problem because `Printable` and `Throwable` are incomparable but possibly overlapping types.)

Now consider a function call `f(X`$^+$`)` at some program point Z. Let Δ be the set of parameter type sequences of function declarations of `f` that are visible at Z and dynamically applicable to the call, and let Σ be the set of parameter type sequences of function declarations of `f` that are visible from Z and statically applicable to the call. Moreover, let δ be the subset of Δ such that $\forall d \in \delta : \neg\exists d' \in \Delta \setminus \{d\} : d' \sqsubset d$ and let σ be the subset of Σ such that $\forall s \in \sigma : \neg\exists s' \in \Sigma \setminus \{s\} : s' \sqsubset s$.

Claims (to be proved):

1. $|\sigma| \leq 1$
2. $|\delta| \leq 1$
3. If $|\sigma| = 1$ then $|\delta| = 1$
4. If $\sigma = \{s\}$ and $\delta = \{d\}$ then $d \sqsubseteq s$

Put into words:

1. It is impossible for a function call to be statically ambiguous. (This is a consequence of the Meet Rule.)
2. It is impossible for a function call to be dynamically ambiguous. (This is also a consequence of the Meet Rule.)
3. If there is a statically most specific applicable declaration, then there is a dynamically most specific applicable declaration.
4. The parameter type sequence for the dynamically most specific applicable declaration is more specific than, or the same as, the parameter type sequence for the statically most specific applicable declaration.

Therefore an implementation strategy may be used in which the statically most specific applicable declaration is identified at compile time, and the run-time dispatch mechanism need only consider dispatching among that declaration plus declarations that are more specific than that declaration.

2.6.1 Overloaded methods

Now we discuss additional rules for overloaded methods and method calls.

A method call $\mathtt{X}_0$`.m(`$\mathtt{X}^+$`)` is first resolved based on the runtime type $\mathtt{X}_0$ of the receiver, and then by the runtime types $\mathtt{X}^+$ of the arguments. When a subtrait $\mathtt{Q}_0$ of a trait $\mathtt{P}_0$ defines a set of overloaded methods N with the same name as a set of overloaded methods M in $\mathtt{P}_0$, the methods N override the methods M if and only if for every method `m` in N there is a method `m` in M with the same signature. Furthermore, the signatures of overloaded methods in a trait definition must respect the same constraints as the signatures of a set of overloaded functions.

2.7 Operator Fixity

Most operators in Fortress can be used variously as prefix, postfix, infix, nofix, or multifix operators. (Some operators can be used in pairs as enclosing (bracketing) operators—see Section 2.8.) The Fortress language dictates only the rules of syntax; whether an operator has a meaning when used in a particular way depends only on whether there is a definition in the program for that operator when used in that particular way (see Section 3.4).

The fixity of a non-enclosing operator is determined by context. To the left of such an operator we may find (1) a primary expression, (2) another operator, or (3) a comma, semicolon, or left encloser. To the right we may find (1) a primary expression, (2) another operator, (3) a comma, semicolon, or right encloser, or (4) a line break. Considered in all combinations, this makes twelve possibilities. In some cases one must also consider whether or not whitespace separates the operator from what lies on either side. The rules of operator fixity are specified by Figure 2.1, where the center column indicates the fixity that results from the left and right context specified by the other columns:

left context	whitespace	operator fixity	whitespace	right context
primary	yes	**infix**	yes	primary
	yes	**error** (infix)	no	
	no	**postfix**	yes	
	no	**infix**	no	
primary	yes	**infix**	yes	operator
	yes	**error** (infix)	no	
	no	**postfix**	yes	
	no	**infix**	no	
primary	yes	**error** (postfix)		`,` `;` right encloser
	no	**postfix**		
primary	yes	**infix**		line break
	no	**postfix**		
operator		**prefix**		primary
operator		**prefix**		operator
operator		**error** (nofix)		`,` `;` right encloser
operator		**error** (nofix)		line break
`,` `;` left encloser		**prefix**		primary
`,` `;` left encloser		**prefix**		operator
`,` `;` left encloser		**nofix**		`,` `;` right encloser
`,` `;` left encloser		**error** (prefix)		line break

Figure 2.1: Operator Fixity (I)

A case described in the center column of the table as an **error** is a static error; for such cases, the fixity mentioned in parentheses is the recommended treatment of the operator for the purpose of attempting to continuing the parse in search of other errors.

The table may seem complicated, but it all boils down to a couple of practical rules of thumb:

1. *Any* operator can be prefix, postfix, infix, or nofix.
2. An infix operator can be *loose* (having whitespace on both sides) or *tight* (having whitespace on neither side), but it mustn't be *lopsided* (having whitspace on one side but not the other).
3. A postfix operator should have no whitespace before it and should be followed (possibly after some whitespace) by a comma, semicolon, right encloser, or line break.

See Section 2.1.10 for a discussion of how infix operators may be chained or treated as multifix operators.

2.8 Enclosing Operators

These operators are always used in pairs as enclosing operators:

```
                (/   /)        (\   \)
[   ]           [/   /]        [\   \]        [*   *]
{   }           {/   /}        {\   \}        {*   *}
                </   />        <\   \>
               <</   />>      <<\   \>>
```

(ASCII encodings are shown here; they all correspond to particular single Unicode characters.) There are other pairs as well, such as ⌊ ⌋ and ⌈ ⌉.

These operators may also be used as enclosing operators:

```
|   |           ||   ||        |||   |||
/   /           //   //        ///   ///
\   \           \\   \\        \\\   \\\
```

but there is a trick to it, because on the face of it you can't tell whether any given occurrence is a left encloser or a right encloser. Again, context is used to decide, this time according to Figure 2.2:

This is very similar to the table in Section 2.7; a rough rule of thumb is that if an ordinary operator would be considered a prefix operator, then one of these will be considered a left encloser; and if an ordinary operator would be considered a postfix operator, then one of these will be considered a right encloser.

In this manner, one may use `|   |` for absolute values, `||   ||` for matrix norms, and `//   //` for continued fractions.

2.9 Operator Precedence

Fortress specifies that certain operators have higher precedence than certain other operators, so that one need not use parentheses in all cases where operators are mixed in an expression. However, Fortress does not follow the practice of

left context	whitespace	operator fixity	whitespace	right context
primary	yes	**infix**	yes	primary
	yes	**left encloser**	no	
	no	**right encloser**	yes	
	no	**infix**	no	
primary	yes	**infix**	yes	operator
	yes	**left encloser**	no	
	no	**right encloser**	yes	
	no	**infix**	no	
primary	yes	**error** (right encloser)		`,` `;` right encloser
	no	**right encloser**		
primary	yes	**infix**		line break
	no	**postfix**		
operator		**error** (left encloser)	yes	primary
		left encloser	no	
operator		**error** (left encloser)	yes	operator
		left encloser	no	
operator		**error** (nofix)		`,` `;` right encloser
operator		**error** (nofix)		line break
`,` `;` left encloser		**left encloser**		primary
`,` `;` left encloser		**left encloser**		operator
`,` `;` left encloser		**nofix**		`,` `;` right encloser
`,` `;` left encloser		**error** (left encloser)		line break

Figure 2.2: Operator Fixity (II)

other programming languages in simply assigning an integer to each operator and then saying that the precedence of any two operators can be compared by comparing their assigned integers. Instead, Fortress relies on defining traditional groups of operators based on their meaning and shape, and specifies specific precedence relatonships between some of these groups. If there is no specific precedence relationship between two operators, then parentheses must be used. For example, Fortress does not accept the expression `a + b ∪ c`; one must write either `(a + b) ∪ c` or `a + (b ∪ c)`. (Whether or not the result then makes any sense depends on what definitions have been made for the `+` and `∪` operators—see Section 3.4.)

Here are the basic principles of operator precedence in Fortress:

- Subscripting (`[ ]`), superscripting (`^`), member selection (`.`), method invocation (`.`*name*`(...)`), and postfix operators have higher precedence than any operator listed below; within this group, these operations are left-associative (performed left-to-right).
- *Tight juxtaposition*, that is, juxtaposition without intervening whitespace, has higher precedence than any operator listed below. The associativity of tight juxtaposition is type-dependent; see Section 2.10.
- Next, *tight fractions*, that is, the use of the operator '`/`' with no whitespace on either side, have higher precedence than any operator listed below. The tight-fraction operator has no precedence compared with itself, so it is not permitted to be used more than once in a tight fraction without use of parentheses.
- *Loose juxtaposition*, that is, juxtaposition with intervening whitespace, has higher precedence than any operator listed below. The associativity of loose juxtaposition is type-dependent and is different from that for tight juxtaposition; see Section 2.10.
- Prefix operators have higher precedence than any operator listed below.

- The infix operators are partitioned into certain traditional groups, as explained below.
- Binding and assignment operators (`=`, `:=`, `+=`, `-=`, $\wedge$`=`, $\vee$`=`, $\cap$`=`, $\cup$`=`, and so on) have lower precedence than any operator listed above.

The majority of infix binary operators are divided into four general categories: arithmetic, relational, boolean, and other. The arithmetic operators are further categorized as multiplication/division/intersection, addition/subtraction/union, and other. The relational operators are further categorized as equivalence, inequivalence, chaining, and other. The boolean operators are further categorized as conjunctive, disjunctive, and other.

The arithmetic and relational operators are further divided into groups based on shape:

- "plain" operators: $+ - \cdot \times / \pm \mp \oplus \ominus \odot \otimes \oslash \boxplus \boxminus \boxdot \boxtimes < \leq \geq > \ll \lll \ggg \gg \not< \not\leq \not\geq \not>$ etc.
- "rounded" or "set" operators: $\cap \Cap \cup \Cup \uplus \subset \subseteq \supseteq \supset \Subset \Supset \not\subset \not\subseteq \not\supseteq \not\supset$ etc.
- "square" operators: $\sqcap \sqcup \sqsubset \sqsubseteq \sqsupseteq \sqsupset \not\sqsubseteq \not\sqsupseteq$ etc.
- "curly" operators: $\curlywedge \curlyvee \prec \preceq \succeq \succ \not\prec \not\preceq \not\succeq \not\succ$ etc.
- "triangular" relations: $\vartriangleleft \trianglelefteq \trianglerighteq \vartriangleright \ntriangleleft \ntrianglelefteq \ntrianglerighteq \ntriangleright$ etc.
- "chickenfoot" relations: $\lessdot \gtrdot$ etc.

The principles of precedence for binary operators are then as follows:

- A multiplication or division or intersection operator has higher precedence than any addition or subtraction or union operator that is in the same shape group.
- Certain addition and subtraction operators come in pairs, such as $+$ and $-$, or $\oplus$ and $\ominus$, which are considered to have the same precedence and so may be mixed within an expression and are grouped left-associatively. These addition-subtraction pairs are the *only* cases where two different operators are considered to have the same precedence.
- An arithmetic operator has higher precedence than any equivalence or inequivalence operator.
- An arithmetic operator has higher precedence than any relational operator that is in the same shape group.
- A relational operator has higher precedence than any boolean operator.
- A conjunctive boolean operator has higher precedence than any disjunctive boolean operator.

While the rules of precedence are complicated, they are intended to be both unsurprising and conservative. Note that operator precedence in Fotrress is not always transitive; for example, while + has higher precedence than < (so you can write `a + b < c` without parentheses), and < has higher precedence than `OR` (so you can write `a < b OR c < d` without parentheses), it is *not* true that + has higher precedence than `OR`—the expression `a OR b + c` is not permitted, and one must instead write `(a OR b) + c` or `a OR (b + c)`.

Another point is that the various multiplication and division operators do *not* have "the same precedence"; they may not be mixed freely with each other. For example, one cannot write `u` $\cdot$ `v` $\times$ `w`; one must write `(u` $\cdot$ `v)` $\times$ `w` or (more likely) `u` $\cdot$ `(v` $\times$ `w)`. Similarly, one cannot write `a` $\cdot$ `b / c` $\cdot$ `d`; but juxtaposition does bind more tightly than a loose (whitespace-surrounded) division slash, so one is allowed to write `a b / c d`, and this means the same as `(a b)/(c d)`. On the other hand, loose juxtaposition binds less tightly than a tight division slash, so that `a b/c d` means the same as `a (b/c) d`. On the other other hand, tight juxtaposition binds more tightly than tight division, so that `(n+1)/(n+2)(n+3)` means the same as `(n+1)/((n+2)(n+3))`.

There are two additional rules intended to catch misleading code: it is a static error for an operand of a tight infix operator to be a loose juxtaposition, and it is a static error if the rules of precedence determine that a use of infix operator a has higher precedence than a use of infix operator b, but that particular use of a is loose and that particular use of b is tight. Thus, for example, the expression `sin x + y` is permitted, but `sin x+y` is not permitted. Similarly, the expression `a · b + c` is permitted, as are `a·b + c` and `a·b+c`, but `a · b+c` is not permitted. (The rule detects only the presence or absence of whitespace, not the amount of whitespace, so `a   ·   b + c` is permitted. You have to draw the line somewhere.)

When in doubt, just use parentheses. If there's a problem, the compiler will (probably) let you know.

2.10 Interpretation of Juxtapositions

The manner in which a juxtaposition of three or more items should be associated requires type information and awareness of whitespace. (This is an inherent property of customary mathematical notation, which Fortress designed to emulate where feasible.) Therefore a Fortress compiler must produce a provisional parse in which such multi-element juxtapositions are held in abeyance, then perform a type analysis on each element and use that information to rewrite the n-ary juxtaposition into a tree of binary juxtapositions. All we need to know is whether each element of a juxtaposition is a function.

A loose juxtaposition is reassociated as follows:

- First the loose juxtaposition is broken into chunks; wherever there is a non-function element followed by a function element, the latter begins a new chunk. Thus a chunk consists of some number (possibly zero) of functions followed by some number (possibly zero) of non-functions.
- The non-functions in each chunk, if any, are replaced by a single element consisting of the non-functions grouped left-associatively into binary juxtapositions.
- What remains in each chunk is then grouped right-associatively.
- Finally, the sequence of rewritten chunks is grouped left-associatively.

(Notice that no analysis of the types of newly constructed chunks is needed during this process.)

Here is an example: `n (n+1) sin 3 n x log log x`. Assuming that `sin` and `log` name functions in the usual manner and that `n`, `(n+1)`, and `x` are not functions, this loose juxtaposition splits into three chunks: `n (n+1)` and `sin 3 n x` and `log log x`. The first chunk has only two elements and needs no further reassociation. In the second chunk, the non-functions `3 n x` are replaced by `((3 n) x)`. In the third chunk, there is only one non-function, so that remains unchanged; the chunk is the right-associated to form `(log (log x))`. Finally, the three chunks are left-associated, to produce the final interpretation `((n (n+1)) (sin ((3 n) x))) (log (log x))`. Now the original juxtaposition has been reduced to binary juxtaposition expressions.

A tight juxtaposition follows a different strategy:

- If the tight juxtaposition contains no function element, or if only the last element is a function, go on to the next step. Otherwise, consider the leftmost function element and examine the element that follows it. If that latter element is not parenthesized, it is a static error; otherwise, replace the two elements with a single element consisting of a new juxtaposition of the two elements (in the same order), and perform a type analysis on this new juxtaposition. Then repeat this step on the original juxtaposition (which is now one element shorter).
- Left-associate the remaining elements of the juxtaposition.

(Note that this process requires type analysis of newly created chunks along the way.)

Here is an (admittedly contrived) example: `reduce(f)(a)(x+1)sqrt(x+2)`. Suppose that `reduce` is a curried function that accepts a function `f` and returns a function that can be applied to an array `a` (the idea is to use the function `f`, which ought to take two arguments, to combine the elements of the array to produce an accumulated result).

The leftmost function is `reduce`, and the following element `(f)` is parenthesized, so the two elements are replaced with one: `(reduce(f))(a)(x+1)sqrt(x+2)`. Now type analysis determines that the element `(reduce(f))` is a function.

The leftmost function is `(reduce(f))`, and the following element `(a)` is parenthesized, so the two elements are replaced with one: `((reduce(f))(a))(x+1)sqrt(x+2)`. Now type analysis determines that the element `((reduce(f))(a))` is not a function.

The leftmost function is `(sqrt)`, and the following element `(x+2)` is parenthesized, so the two elements are replaced with one: `((reduce(f))(a))(x+1)(sqrt(x+2))`. Now type analysis determines that the element `(sqrt(x+2))` is not a function.

There are no functions remaining in the juxtaposition, so the remaining elements are left-associated:

```
(((reduce(f))(a))(x+1))(sqrt(x+2))
```

Now the original juxtaposition has been reduced to binary juxtaposition expressions.

2.11 Tests

The `test` modifier on a function or variable definition indicates that it is part of the test suite of a component, and can be referred to by other parts of the test suite. A test function that takes no arguments is run by default when a component is tested. For example, we can write the following (very short) test function for `factorial`:

```
test testFactorial() = do
  assert(factorial(0) = 1)
  assert(factorial(5) = 120)
end
```

(This function makes use of the function `assert`, provided in the Fortress standard library).

A test function may also be called directly on a component, with an appropriate set of arguments passed to it. *Calling specific test functions directly can be used to form smaller test suites.*

If a variable definition includes the modifier `test`, then the value of that variable is used as a test case. Test functions that have one or more parameters are called with every permutation of test cases whose types are compatible with the functions' parameters.

```
test zero = 0
test one = 1
test five = 5

test factorial(x,y) =
  if x > y then
```

```
    assert factorial(x) > factorial(y)
  end
```

If an object definition includes the modifier `test`, then the methods of that object with modifier `test` are run when the enclosing component is tested. The test cases applicable as arguments to the test methods of the object consist of all test cases in the enclosing scope along with all fields of the object with modifier `test`.

```
test object TestFactorial
  test zero = 0
  test one = 1
  test five = 5

  test factorial() = do
    assert(factorial(0) = 1)
    assert(factorial(5) = 120)
  end

  test factorial(x,y) =
    if x > y then
    assert factorial(x) > factorial(y)
  end
end
```

If the object definition is parametric, then it is instantiated with every valid permutation of test cases from the enclosing scope, and the test methods of each instantiation are run on all valid permutations of test case arguments.

The parts of a program without modifier `test` must not refer to those with the `test` modifier.

Chapter 3

Advanced Language Constructs

In this section, we build on the basic Fortress language elements to develop more advanced aspects of the language. In particular, we describe the semantics of parallelism and support for domain-specific languages. First, however, we define the context in which a Fortress program executes.

3.1 Execution Model

All Fortress programs are executed in the context of a *fortress*, which encompasses the functionality of a virtual machine, as well as handling the components system, as described in Chapter 4. Fortresses are responsible for managing the execution of processes, and can run multiple processes simultaneously.

3.1.1 Processes

A Fortress process is created whenever the `execute` operation is invoked within a fortress (see 4.3). This new process object executes the code in the `exec` method of the specified component.

In the execution of a Fortress process, there is a set of *threads* and a set of *regions*. Every Fortress object resides in some region; those objects are in close proximity with respect to communication cost.

Threads are objects, and thus every thread also resides in some region. A thread consists of a *continuation* P, describing the remainder of the computation that T must complete, and an environment which maps variables in P to objects.

Regions are objects which are grouped hierarchically to form a tree; this tree reflects the relative locality of the regions it contains. Every pair of regions has a common ancestor in the tree, reflecting the degree of locality those locations share. The different levels of this tree reflect underlying machine structure, such as threads within a CPU, memory shared by a group of processors, or resources distributed across the entire machine.

3.2 Parallelism and Locality

Fortress is designed to make parallel programming as simple and as painless as possible. We adopt a multi-tiered approach to parallelism:

- At the highest level, we provide libraries which allocate locality-aware distributed arrays (Section 3.2.1) and syntax to perform parallel looping (Section 3.2.2). Our use of parallel `for` loops is intended to maximize available parallelism; this leads to computations with lots of slack (Section 3.2.3) which are easy to load balance.
- Immediately below that, we provide syntax for spawning a parallel block as a new thread (Section 3.2.4), and for synchronization using transactional memory access (Section 3.2.5).
- There is an extensive library of *distributions*, which permit the programmer to specify locality and data distribution explicitly (Section 3.2.7).
- Finally, there are mechanisms for constructing new distributions via recursive subdivision (Section 3.2.8) of index spaces into tree structures with individual indices at the leaves. These mechanisms are grounded in fundamental data structures such as `Region` and `LinearStorage` (Section 3.2.9).

We approach these from the highest level to the lowest level. The lowest level is bare-metal programming and best left until the end.

3.2.1 Arrays are distributed by default

Arrays in Fortress are assumed to be spread out across the machine. Like arrays in Fortran, Fortress arrays are complex data structures; simple linear storage is encapsulated by the `LinearStorage` type, which is used in the implementation of arrays (see Section 3.2.9). The default distribution of an array is determined by the Fortress libraries; in general it will depend on the size of the array, and on the size and locality characteristics of the machine running the program. For advanced users, the distribution library (introduced in Section 3.2.7) provides a way of combining and pivoting distributions, or of re-distributing two arrays so that their distributions match. Arrays can be created by calling a factory function:

```
a = array(xSize, ySize, zSize)
```

Note that matrices and vectors are subtypes of arrays. They are allocated and distributed in the same way, but also define arithmetic operations such as multiplication and addition.

3.2.2 The `for` loop is parallel by default

`Generator` is a trait in Fortress. Some common generators include:

$l\#n$	n consecutive integers beginning with l
a`.indices()`	The index set of an array a

The `indices` generator is of particular interest. Given a multidimensional array, it returns multiple values. The parallelism of a loop on this generator follows the spatial distribution of the array as closely as possible.

By default, loop iterations are assumed to run in parallel. The `sequential` distribution can be used to change this behavior (Section 3.2.7). For a parallel loop, the order of nesting of generators does not imply anything about the relative order of nesting of loop iterations. In most cases, multiple generators are equivalent to multiple nested loops:

```
for $v_1 \leftarrow g_1$ do
  for $v_2 \leftarrow g_2$ do
    ...
      for $v_n \leftarrow g_n$ do
        exprs
      end
    ...
  end
end
```

The compiler will make an effort to choose the best possible iteration order it can for a multiple-generator loop. There may be no such guarantee for nested loops. Thus loops with multiple generators are preferable in general:

```
for $v_1 \leftarrow g_1$
    $v_2 \leftarrow g_2$
    ...
    $v_n \leftarrow g_n$ do
  exprs
end
```

In both cases generated variables v_i scope over subsequent generators v_{i+k} and over the loop body.

Iterations may be re-structured to eliminate colliding dependencies, reductions may be localized as described in Section 3.2.6, parallel iterations may be serialized, serial iterations may be parallelized, and so forth, so long as the compiled code executes *as if* it matched the given source code.

Any loop iteration may throw an exception. In this case, the loop as a whole throws an exception; every loop iteration either runs to completion, does not run at all, or runs until it encounters an exception. The exception thrown by the loop can be any one of the exceptions thrown by individual loop iterations. In this respect nested loops have very different exception behavior from a single multiple-generator loop.

3.2.3 Slack

Different iterations of a loop body may execute in very different amounts of time. A naively parallelized loop will cause processors to idle until every iteration finishes. The simplest way to mitigate this delay is to expose substantially more parallel units of work than there are threads to run them. Load balancing can move the resulting (smaller) units of work onto idle processors to balance load.

The ratio between available work and number of threads is dubbed *parallel slack* by Blumofe [3, 4]. With support for very lightweight threading and load balancing, slack in hundreds or thousands proves beneficial; very slack computations easily adapt to differences in the number of available processors. The Fortress programmer should be aware that slack is a desirable property, and endeavor to expose parallelism where possible.

Note that there is no particular need for slack in array layout except the desire to collocate data and computation. In general, we expect the structure of a distributed array to be considerably simpler (and coarser-grained) than the equivalent generator. The built-in distributions account for this difference of granularity.

3.2.4 Parallel threads

We can spawn a block of code in parallel as follows:

```
v = spawn do
        exprs
    end
```

Here the block of code represented by *exprs* is run in parallel with any succeeding computation. We refer to `v` as a *thread*. Every thread returns a value (though that value might be `()`). We write `v.value()` to obtain the value computed by *exprs*. If thread `v` has not yet completed execution, `v.value()` will wait until it has done so. When *exprs* do not return a value, but are executed purely for effect, we may optionally omit the binding for `v`—but note in this case that there will be no simple way to detect the termination of the block.

In the absence of sufficient parallel resources, the compiler executes *exprs* before continuing execution of the code in which the `spawn` occurred. We can imagine that it is actually the *rest* of the computation *after* the parallel block which is spawned off in parallel. This is a subtle technical point, but makes the sequential execution of parallel code simpler to understand, and avoids subtle problems with the asymptotic space behavior of parallel code [18, 11].

When a parallel block throws an exception, that exception is *deferred*. Any invocation of `v.value()` throws the deferred exception. If the value of the thread is discarded, the exception itself will be silently ignored.

Note that parallel loop iterations conceptually occur in separate threads. The necessary synchronization for these threads is performed by the compiler and runtime system.

3.2.5 Transactions

It is often convenient to imagine that a thread or a portion of a thread behaves *transactionally*: all reads and writes appear to occur simultaneously in a single atomic step. For this purpose, Fortress provides `atomic` blocks. For example:

```
arraySum ⟦N extends Additive, nat x⟧(a:N[x]):N = do
  sum:N := 0
  for i←a.indices() do
    atomic do sum:=sum+a[i] end
  end
  sum
end
```

Very long transactions can degrade performance. Two transactions *conflict* when one attempts to read or write state written by the other. When transactions conflict, their execution must be partially serialized. The exact mechanism by which this occurs will vary; the serialization is provided by the implementation of transactional memory. In general, the execution of one or both transactions may be abandoned, rolling back any state changes which might have occurred, and requiring that transaction to be re-run. The longer a transaction runs and the more memory it touches the greater the chance of conflict and the larger the bottleneck that conflict may impose.

Fortress provides a user-level `abort()` function which abandons execution of a transaction and rolls back its changes, again requiring the transaction to be re-run. This permits a transaction to perform consistency checks before committing.

Fortress also includes a `tryatomic` construct, which attempts to run its body atomically. If it succeeds, the result is returned; if the transaction aborts, either due to conflict or due to a call to `abort`, the `TransactionFailed` exception is thrown. Conceptually `atomic` can be defined in terms of `tryatomic` as follows:

```
label AtomicBlock
  while True do
    try
      result = tryatomic do body end
      exit AtomicBlock with result
    catch e
      TransactionFailed ⇒ () (* continue execution *)
    end
  end
  throw(UnreachableCode)
end AtomicBlock
```

Transactions may be nested arbitrarily; semantically, inner transactions appear atomic within the scope in which they occur. Unless `tryatomic` is used, this has no particular semantic impact: erasing an inner `atomic` block can affect the performance, but not the correctness, of a program.

When an exception of any kind is thrown from within an `atomic` block or a `tryatomic` block, and is not caught within the block, the transaction fails. The exception continues to propagate to the enclosing context—unless `TransactionFailed` is thrown from inside an `atomic` block, in which case the transaction retries. All side effects to previously-allocated objects are discarded. Side effects to newly-allocated objects are retained (these objects will be local; see the next section). A local variable reverts to the value it held before the transaction began.

We do not provide input and output in the context of a transaction; thus, we may only call an `io` function from outside an `atomic` block. Similarly, we do not provide nested parallelism in the context of a transaction. An interesting exception is within a `pure` function: since these functions have no visible side effects (see Chapter 2), such a function may contain arbitrary parallelism, even if it occurs within the scope of an `atomic` block. Thus, only `pure` and `io` functions may contain parallel blocks or parallel `for` loops.

It is not difficult to assign a semantics to arbitrary nestings of parallelism and transactions, permitting parallelism everywhere—even inside `atomic` blocks. However, at the moment no efficient implementation strategy is known. As a result, we defer transactions with non-`pure` nested parallelism to future work.

3.2.6 Shared and local data

Every datum (function or object) in a Fortress program is considered to be either *shared* or *local* (collectively referred to as the *sharedness* of the datum). A local datum is accessible to at most one running thread. It may be accessed more cheaply than a shared datum, particularly in the case of transactional reads and writes.

The following rules govern sharedness:

- Data are considered to be local by default.
- The sharedness of a datum can change on the fly.
- If a datum is transitively reachable from more than one thread at a time, it must be shared.
- When a reference to a local datum is stored into a shared datum (by field assignment to a shared objects, or by assigning to a mutable variable closed over by a shared function), the local datum must be *published*. Its sharedness is changed to shared, and all of the data to which it refers is also published.

- Local variables referenced by a thread must be published before that thread may be run in parallel with the thread which spawned it.
- Data in a field or closed-over variable of value type is assigned by copying, and thus has the sharedness of the containing object or closure.

The sharedness of a datum should only matter for performance purposes. Publishing can be expensive, particularly if the structure being broadcast is large and heavily nested; this can cause an apparently short transaction (a single write, say) to run arbitrarily long. To avoid this, the programmer can request that an object be allocated as shared by tagging a call to the factory:

```
x := shared Cons(x, xs)
```

A datum can be published early as follows:

```
publish(x)
```

A local copy of an object can be obtained by copying it:

```
localVar := sharedVar.copy()
```

Note that function closures may not be localized by copying.

The functionality described so far is solely a performance optimization; we can't tell whether a given datum is shared or local, and sharedness will not make a difference to programs written using only these constructs. Two additional methods are provided which *can* change program behavior based on the sharedness of objects:

- `o.isShared()` returns true when `o` is shared, and false when it is local. This permits the program to take different actions based on sharedness; it should be used with caution.
- `o.localizeNoCopy()` is equivalent to the following expression:

  ```
  if o.isShared() then o.copy() else o end
  ```

  `localizeNoCopy` can have unexpected behavior if there is a reference to `o` from another local object. Publishing that object will cause `o` to be published; updates to `o` will be visible through the other object. By contrast, if `o` was already shared, and referred to by another shared object, the newly-localized copy will be entirely distinct.

In order to perform computations as locally as possible, and avoid the need to serialize relatively simple `for` loops, Fortress gives special treatment to *reductions*. A reduction is a commutative, associative binary operation with an identity (an abelian monoid) and is captured by the following trait:

```
trait Reduction ⟦T⟧
  op(l : T, r : T) : T
  identity() : T
end
```

A loop body may contain as many of the following reductions as desired:

```
l := r.op(l,value)
l := r.op(value,l)
```

As long as every assignment uses the same reduction `r`, and the value of `l` is not otherwise used in the loop body, we say `l` is reduced using `r`.

Several common mathematical operators are also treated as reductions. These include `+`, `*`, `AND`, `OR`, and `XOR`. Note that since there are no guarantees on the order of execution of loop iterations, there are also no guarantees on the order of reduction.

Reductions are treated roughly as in OpenMP [21]. The local variable `l` is assigned `r.identity()` at the beginning of the loop body or block. At the end of the loop or block, the original variable value before the loop and the final variable values from each execution of the loop body are combined together using the reduction operator, in some arbitrarily-determined order.

Consider the `arraySum` example from the previous section:

```
arraySum⟦N extends Additive, nat x⟧(a:N[x]):N = do
  sum:N := 0
  for i←a.indices() do
    atomic do sum:=sum+a[i] end
  end
  sum
end
```

Here the variable `sum` is reduced, so this loop is equivalent to the following code:

```
arraySum⟦N extends Additive, nat x⟧(a:N[x]):N = do
  sum:N := 0
  for i←a.indices() do
    var temp:N
    atomic do
      temp:=a[i]
    end
    sum:=sum+temp
  end
  sum
end
```

3.2.7 Distributions

Most of the heavy lifting in Fortress is performed by *distributions* and parallel blocks. The job of a distribution is to impose parallel structure on generators, and to provide for the allocation and distribution of arrays on the machine.

An instance of trait `Distribution` describes the placement of data or computation on a machine. A `Distribution` acts as a transducer for generators and for arrays. It copies an array, re-distributing its elements as it does so. It organizes the data produced by a generator into the leaves of a tree whose inner nodes correspond (conceptually) to the levels of parallelism and locality on the underlying machine. Thus, a distribution does the hard work of splitting data up and distributing it over the machine.

The intention of distributions is to separate the task of data distribution and program correctness. That is, it should be possible to write and debug a perfectly acceptable parallel program using only the default data distribution provided by the system. Imposing a distribution on particular computations, or designing and implementing distributions from scratch, is a task left for performance tuning.

A distribution also acts as a factory for generators and arrays. We can think of these factories as being defined in terms of transducers and the built-in default factory methods. This is the default implementation provided by the `Distribution` trait; built-in distributions will usually override this implementation and construct arrays and generators directly.

There is a `default` distribution which is defined by the underlying system. This distribution is designed to be reasonably adaptable to different system scales and architectures, at the potential cost of some runtime efficiency. Arrays and generators which are not explicitly allocated through a distribution are given the `default` distribution. Thus `array` is merely a convenient shorthand for `default.array`.

We said in Section 3.2.2 that there is a generator, `indices`, associated with every array. This generator is distributed in the same way as the array itself. When we re-distribute an array, we also re-distribute the generator.

There are a number of built-in distributions:

`default`	Name for distribution chosen by system.
`sequential`	Sequential distribution. Arrays are allocated in one piece of memory.
`local`	Equivalent to `sequential`.
`par`	Blocked into chunks of size 1.
`blocked`	Blocked into roughly equal chunks.
`blocked(`n`)`	Blocked into n roughly equal chunks.
`subdivided`	Chopped into 2^k-sized chunks, recursively.
`interleaved(`d_1`, `d_2`,...`d_n`)`	The first n dimensions are distributed according to $d_1 \ldots d_n$, with subdivision alternating among dimensions.
`joined(`d_1`, `d_2`,...`d_n`)`	The first n dimensions are distributed according to $d_1 \ldots d_n$, subdividing completely in each dimension before proceeding to the next.

From these, a number of composed distributions are provided:

`morton`$_n$	Bit-interleaved Morton order [19], recursive subdivision in n dimensions. Local in remaining dimensions.
`blocked(`x_1`,`x_2`,...`x_n`)`	Blocked in n dimensions into x_i chunks in dimension i; remaining dimensions (if any) are local.

To allocate an array which is local to a single thread (and most likely allocated in contiguous storage), the `local` distribution can be used:

```
a = local.array(xSize, ySize, zSize)
```

Other distributions can be requested in a similar way.

A generator g can be made sequential simply by sequentializing the distribution as follows:

$v \leftarrow$ `sequential(`g`)`

Note that at the moment there is no way to tell the compiler that we really mean it when we ask for sequentiality, as opposed to saying that we should preserve sequential semantics. In future, we may distinguish `local` and `sequential` distributions for this purpose.

Distributions can be constructed and given names:

```
spatialDist = blocked(n,n,1) (* Pencils along the z axis *)
spaceVecs = spatialDist.array(n,n,n,5)  :Double[n, n, n, 5]
spaceMats = spatialDist.array(n,n,n,5,5):Double[n, n, n, 5, 5]
```

The system will lay out arrays with the same distribution in the same way in memory (as much as this is feasible), and will run loops with the same distribution in the same way (as much as this is feasible). By contrast, this code will likely divide up the arrays into the same-sized pieces as above, but these pieces need not be collocated:

```
spaceVecs = blocked(n,n,1).array(n,n,n,5)  :Double[n, n, n, 5]
spaceMats = blocked(n,n,1).array(n,n,n,5,5):Double[n, n, n, 5, 5]
```

3.2.8 Recursive subdivision

Internally, generators accomplish their task by *recursive subdivision*. This subdivision is guided by the distribution. It is possible to write computations which follow this recursive structure directly. We can view the pattern of recursive calls used by a generator as a tree with arbitrary fanout. At the leaves are sequential loops over index space. Interior nodes represent recursive subdivision. Thus, we can break a parallel generator into a series of sequential generators. Interior nodes generate a series of generators (children of the current generator). Leaf nodes generate the actual values produced by the iterator. Thus, the following code follows the structure of a generator recursively, and sums the generated values:

```
recSum(gen : Generator⟦Int⟧) : Int = do
  sum : Int := 0
  if (gen.isSequential) then
    for i ← gen do
      sum += i
    end
  else
    for childGen ← gen.children() do
      sum += recSum(childGen)
    end
  end
  sum
end
```

This can be parallelized as follows. Note the use of the `par` distribution to make every iteration of the sequential generator `gen.children()` run in parallel, and the use of a simple reduction on the variable `sum`.

```
recSum(gen : Generator⟦Int⟧) : Int = do
  sum : Int := 0
  if (gen.isSequential) then
    for i ← gen do
      sum += i
    end
  else
    for childGen ← par(gen.children()) do
      sum += recSum(childGen)
    end
  end
  sum
end
```

Properties of known distributions may be exploited in this way to do complex restructuring of generated traversals. This mechanism lies at the heart of the Fortress loop compilation strategy.

3.2.9 Primitives for constructing distributions

Every object reference, including a thread, and every array/index pair, has a corresponding region (see Section 3.1.1). For an array, the region of the array will contain the region of any element of that array. In an array of references the region of an array element may be different from the region of the object referred to by that element.

Non-array objects are allocated in a region which `isLocalTo` the region in which their constructor is run, unless they are produced by a factory with an appropriate region argument (in which case the factory itself embeds a parallel block which constructs the object in the appropriate region).

A thread can be placed in a particular region by providing that region as an argument to `spawn`:

```
v = spawn region(a,i) do
      a[i]
    end
w = spawn v.region() do
      v.value() * 17
    end
```

Here the spawned thread is sent to the indicated region. Computation continues locally immediately after the spawned region, regardless of the current load on the machine. By contrast, an ordinary unplaced spawn executes the spawned code first, and optionally ships the region after the spawn to another processor for execution.

Finally, Fortress provides the `LinearStorage` data type. `LinearStorage` represents contiguous, one-dimensional, zero-indexed memory. Arrays in Fortress are constructed from individual pieces of `LinearStorage`, plus objects representing dope vectors and so forth. Again, `LinearStorage` is allocated in the region from which it is requested.

Recall that regions are organized into a tree-structured hierarchy. Objects are placed at an appropriate level of that hierarchy when they are created. For example, the region of a thread might refer to the particular processor core on which it is run, or to the multi-threaded CPU which contains that core. The region of a data object may, by contrast, refer to the shared memory on one node of a large multiprocessor. Thus, while the memory is local to a particular thread, it might be local to many other threads as well. Thus, `ref.region` and `thread.region` need not be equal when `ref` is allocated by `thread`. However, it should be the case that `ref.region.isLocalTo(thread.region)`—that is, `ref.region` will be a transitive parent of `thread.region` in region hierarchy.

3.3 Matrix Unpasting

Matrix unpasting is an extension of variable declaration syntax as a shorthand for breaking a matrix into parts. On the left-hand-side of a declaration, what looks like a matrix pasting of unbound variables serves to break the right-hand side into pieces and bind the pieces to the variables. This syntax is concise, eliminates several opportunities for fencepost errors, guarantees unaliased parts, and avoids overspecification of how the matrix should be taken apart.

The motivating example for matrix unpasting is cache-oblivious matrix multiplication. The general plan in a cache oblivious algorithm is to break the input apart on its largest dimension, and recursively attack the resulting smaller and more compact problems.

```
mm⟦nat m, nat n, nat p⟧(left:T[m × n], right:T[n × p], result:T[m × p]):() = do
  case largest of
    1 ⇒ result[0,0] += (left[0,0] right[0,0])
    m ⇒ [ lefttop
           leftbottom   ] = left
         [ resulttop
           resultbottom ] = result
         t1 = spawn do mm(lefttop, right, resulttop) end
         mm(leftbottom, right, resultbottom)
         t1.wait()
    p ⇒ [ rightleft  rightright  ] = right
         [ resultleft resultright ] = result
         t1 = spawn do mm(left, rightleft, resultleft) end
         mm(left, rightright, resultright)
         t1.wait()
    n ⇒ [ leftleft leftright ] = left
         [ righttop
           rightbottom ] = right
         mm(leftleft , righttop   , result)
         mm(leftright, rightbottom, result)
  end
end
```

In unpasting, the element syntax is slightly enhanced both to permit some specification of the split location and to receive information about the split that was performed. For example, perhaps only the upper left square of a matrix is interesting. The programmer can add array bounds to the square unpasted element:

```
foo(A:T[m × n]):() = do
  if   m < n then
    [ square:[m × m] rest ] = A
    ...
  elif m > n then
    [ square:[n × n]
      rest           ] = A
    ...
  else (* A already square *)
    ...
  end
end
```

If an unpasting into explicitly sized pieces does not exactly cover the right-hand-side matrix, an exception is thrown.

An element's `low#high` extent specification establishes the origin for the parts from the array. The lower extent must be bound, either before the unpasting, or earlier (left-or-above) in the unpasting. For example, suppose that an algorithm chooses to break an array into 4 pieces, but retain the original indices for each piece:

```
bar⟦nat p, nat q⟧(X:T[r0#p × c0#q]):() = do
```

```
  [ A[r0#m     × c0#n] B[r0#m     × c0+n#q-n]
    C[r0+m#p-m × c0#n] D[r0+m#p-m × c0+n#q-n] ] = X
  ...
end
```

Unpasting currently does not directly support non-uniform decomposition, and does not provide any sort of constraint satisfaction between the extents of the parts. Thus, this decomposition would not be legal because it constrains the split sizes to be equal without specifying the actual size.

```
fubar⟦nat m, nat n⟧(X:T[m × n]):() = do
  (* p and q unbound *)
  [ A[p × q] B[p × q]
    C[p × q] D[p × q] ] = X
  ...
end
```

To get this effect, the programmer should compute the constrained values:

```
fubar⟦nat m, nat n⟧(X:T[m × n]):() = do
  [ A[m/2 × n/2] B[m/2 × n/2]
    C[m/2 × n/2] D[m/2 × n/2] ] = X
  ...
end
```

Some non-uniform unpastings can be obtained with composition, which can be expressed either by repeated unpasting:

```
unequalRows⟦nat m, nat n⟧(X:T[m × n]):() = do
  [ c1[m × n/2]  c2[m × n/2] ] = x
  [ A[ m/4 × n/2]
    C[3m/4 × n/2] ] = c1
  [ B[3m/4 × n/2]
    D[ m/4 × n/2] ] = c2
  ...
end
```

or simply by nesting matrices in the antipasting:

```
unequalColumns⟦nat m, nat n⟧(X:T[m × n]):() = do
  [ [ A[m/2 ×  n/4] B[m/2 × 3n/4] ]
    [ C[m/2 × 3n/4] D[m/2 ×  n/4] ] ] = X
  ...
end
```

3.4 Operator Definitions

An operator definition may appear anywhere a function definition may appear. Such definitions are like function definitions in all respects except that an operator definition has the reserved word `opr` and has an operator instead of an identifier. The precise placement of the operator within the definition depends on the fixity of the operator. Just as functions may be overloaded, so operators may have overloaded definitions, of the same or differing fixities.

An operator definition has one of five forms: infix/multifix operator definition, prefix operator definition, postfix operator definition, nofix operator definition, and bracketing operator definition. Each is invoked according to specific rules of syntax.

3.4.1 Infix/multifix operator definitions

An infix/multifix operator definition has the reserved word `opr` and then an operator where a function or method definition would have an identifier. The definition must not have any keyword parameters, and must be capable of accepting at least two arguments. It is permissible to use a `...` parameter; in fact, this is a good way to define a multifix operator. Type parameters may also be present, between the operator and the parameter list. Example:

```
opr MAX⟦T extends Rational⟧(x:T,y:T):T = if x > y then x else y end
```

An expression consisting of an infix operator applied to an expression will invoke an infix/multifix operator definition. The compiler considers all infix/multifix operator definitions for that operator that are both accessible and applicable, and the most specific operator definition is chosen according to the usual rules for functions. If the expression is actually multifix, the invocation will pass more than two arguments.

An infix/multifix operator definition may also be invoked by a prefix or nofix (but not a postfix) operator application if the definition is applicable.

Note that superscripting (ˆ) may be defined using an infix operator definition even though it has very high precedence and cannot be used as a multifix operator. (An operator definition for superscripting should have exactly two value parameters.)

3.4.2 Prefix operator definitions

A prefix operator definition has the keyword `opr` and then an operator where a function definition would have an identifier. The definition must have one value parameter, which must not be a keyword parameter or `...` parameter. Type parameters may also be present, between the operator and the parameter list. Example:

```
opr ˜(x:Widget):Widget = x.invert()
```

An expression consisting of a prefix operator applied to an expression will invoke a prefix operator definition. The compiler considers all prefix and infix/multifix operator definitions for that operator that are both accessible and applicable, and the most specific operator definition is chosen according to the usual rules for functions.

3.4.3 Postfix operator definitions

A postfix operator definition has the keyword `opr` where a function definition would have an identifier; the operator itself *follows* the parameter list. The definition must have one value parameter, which must not be a keyword parameter or `...` parameter. Type parameters may also be present, between the reserved word `opr` and the parameter list. Example:

```
opr (n:Integer)! = PRODUCT[i←1:n] i     (* factorial *)
```

An expression consisting of a postfix operator applied to a primary expression will invoke a postfix operator definition. The compiler considers all postfix operator definitions for that operator that are both accessible and applicable, and the most specific operator definition is chosen according to the usual rules for functions.

3.4.4 Nofix operator definitions

A nofix operator definition has the keyword `opr` and then an operator where a function definition would have an identifier. The definition must have no parameters. Example:

```
opr :() = ImplicitRange
```

An expression consisting only of a nofix operator will invoke a nofix operator definition. The compiler considers all nofix and infix/multifix operator definitions for that operator that are both accessible and applicable, and the most specific operator definition is chosen according to the usual rules for functions.

Uses for nofix operators are rare, but those rare examples are very useful. For example, the colon operator is used to construct subscripting ranges, and it is the nofix definition of `:` that a lone `:` to be used as a subscript.

3.4.5 Bracketing operator definitions

A bracketing operator definition has the reserved word `opr` where a function definition would have an identifier. The value parameter list, rather than being surrounded by parentheses, is surrounded by the brackets being defined. A bracketing operator definition may have any number of parameters, keyword parameters, and `...` parameters in the value parameter list. Type parameters may also be present, between the reserved word `opr` and the parameter list. Any paired Unicode brackets may be so defined *except* ordinary parentheses and white square brackets.

```
(* angle bracket notation for inner product *)
opr <| x:Vector, y:Vector |> = SUM[i ← x.indices()] x[i] * y[i]

(* vector space norm (may not be the most efficient) *)
opr ||x:Vector|| = sqrt <| x, x |>
```

An expression consisting of zero or more comma-separated expressions surrounded by a bracket pair will invoke a bracketing operator definition. The compiler considers all bracketing operator definitions for that type of bracket pair that are both accessible and applicable, and the most specific function is chosen according to the usual rules. For example, the expression `<|p,q|>` might invoke the sample bracketing method shown above.

3.5 Subscripting and Subscripted Assignment Operator Method Definitions

A subscripting or subscripted assignment operator method definition may appear anywhere a method definition may appear. Such definitions are like method definitions in all respects except that a subscripting or subscripted assignment operator method definition has the reserved word `opr` and has special syntax instead of an identifier. Just as methods may be overloaded, so subscripting and subscripted assignment operator methods may have overloaded definitions.

3.5.1 Subscripting operator method definition

A subscripting operator method definition has the reserved word `opr` where a method definition would have an identifier. The value parameter list, rather than being surrounded by parentheses, is surrounded by a pair of brackets. A subscripting operator method definition may have any number of value parameters within the brackets, keyword parameters, and `...` parameters in that value parameter list. Type parameters may also be present, between the reserved word `opr` and the parameter list. Any paired Unicode brackets may be so defined *except* ordinary parentheses and white square brackets; in particular, the ordinary square brackets ordinarily used for indexing may be used.

```
(* subscripting method *)
opr [x:BizarroIndex] = self.bizarroFetch(x)
```

An expression consisting of a subexpression immediately followed (with no intervening whitespace) by zero or more comma-separated expressions surrounded by brackets will invoke a subscripting operator method definition. Methods for the expression preceding the bracketed expression list are considered. The compiler considers all subscripting operator method definitions that are both accessible and applicable, and the most specific method is chosen according to the usual rules. For example, the expression `foo[p]` might invoke the sample subscripting method shown above.

3.5.2 Subscripted assignment operator method definition

A subscripted assignment operator method definition has the reserved word `opr` where a method definition would have an identifier. The value parameter list, rather than being surrounded by parentheses, is surrounded by a pair of brackets; this is then followed by the operator `:=` and then a second value parameter list in parentheses, which must contain exactly one non-keyword value parameter. A subscripted assignment operator method definition may have any number of value parameters within the brackets, keyword parameters, and `...` parameters in that value parameter list. A result type after the second value parameter list, but it must be `()`. Type parameters may also be present, between the reserved word `opr` and the first parameter list. Any paired Unicode brackets may be so defined *except* ordinary parentheses and white square brackets; in particular, the ordinary square brackets ordinarily used for indexing may be used.

```
(* subscripted assignment method *)
opr [x:BizarroIndex] := (newValue:Widget) = self.bizarroInstall(x, newValue)
```

An assignment statement consisting of an expression immediately followed (with no intervening whitespace) by zero or more comma-separated expressions surrounded by brackets, followed by the assignment operator `:=`, followed by another expression, will invoke a subscripted assignment operator method definition. Methods for the expression preceding the bracketed expression list are considered. The compiler considers all subscript operator method definitions that are both accessible and applicable, and the most specific method is chosen according to the usual rules. For example, the assignment `foo[p] := myWidget` might invoke the sample subscripted assignment method shown above.

3.6 Support for Domain-specific Languages

In order to support syntax for domain-specific languages, and to allow the Fortress language to grow with time, programmers are allowed to extend the basic syntax of Fortress in their programs. Extensions are allowed through the use of *syntax expanders*.

Syntax expanders must be defined in the top-level scope of a program component. There are three kinds of syntax expanders:

Simple syntax expanders

A *simple* syntax expander starts with the reserved word `syntax`, followed by an identifier, followed by an =, followed by an expression with type `fortress.ast.SyntaxTree`, as in the following example:

```
syntax Area =
  RaisedTypeRef
    (SimpleTypeRef
      (Identifier("Length")),
     SimpleTypeRef("2"))
```

A use site of a syntax expander consists solely of an occurrence of the expander's identifier. This identifier is expanded into the `SyntaxTree` specified in the definition of the expander.

Parametric syntax expanders

A *parametric* syntax expander starts with the reserved word `syntax`, followed by an *opening* identifier, followed by a *contents* parameter (implicitly of type `fortress.lang.SourceAssembly`, which is a sequence of Unicode characters and abstract syntax trees) and a *terminating* identifier. The terminating identifier is followed by an = and an expression of type `fortress.ast.SyntaxTree`. Here is an example:

```
syntax sql exp end = parseSQL(exp)
```

where `parseSQL` is a static function that takes a `SourceAssembly`, interprets it as an SQL query, and returns a `SyntaxTree` consisting of constructor calls to SQL syntax nodes (defined in some SQL library).

At a use site, all characters between the opening identifier and the terminating identifier are turned into a `SourceAssembly` and all escaped subsequences of this `SourceAssembly` are converted into abstract syntax trees (see Section 3.6.3 for a discussion of escaped subsequences). The resulting `SourceAssembly` is bound to the contents parameter of the parametric syntax expander. The use site is then expanded by evaluating the body of the expander.

For example, we could define `parseSQL` so that a use site such as:

```
 sql
   SELECT spectral_class FROM stars
 end
```

would be expanded into:

```
Call(Empty,
     List(VarRef(Identifier("SqlQuery")),
          Call(Empty,
               List(VarRef(Identifier("Select")),
                    String("spectral_class"))),
          Call(Empty,
               List(VarRef(Identifier("From")),
                    String("stars")))))
```

(The `Empty` lists passed to `Call`s are the lists of type parameters to these calls). Note that this `SyntaxTree` corresponds to the following Fortress concrete syntax:

```
SqlQuery(Select("spectral_class"), From("stars"))
```

Parenthesized syntax expanders

A *parenthesized* syntax expander starts with the reserved word `syntax`, followed by an identifier, followed by a sequence of parameters, each enclosed in parentheses of various forms. Each parameter enclosed in parentheses is implicitly of type `SourceAssembly`. After these parameters, an = and an expression with type `fortress.ast.SyntaxTree` is provided. Here is an example of a parenthesized syntax expander that allows for Java-style `for` loops in Fortress programs:

```
syntax jfor (inits) {bodyText} = do
  bindings  , inits = parseBindings  (inits)
  terminator, inits = parseTerminator(inits)
  increment , inits = parseIncrement (inits)
  body = parseForBody(bodyText)
  JavaFor(bindings, terminator, increment, body)
end
```

Depending on how we define the parse functions in this expander, we could parse the following use site of this expander:

```
  jfor (int i = 0; i < 10; i++) {
    System.out.println(i);
  }
```

into a `SyntaxTree` denoting instantiations of parametric “Java syntax” objects such as this:

```
Call(Empty,
     List("JavaFor",
          Call(Empty,
               List("JavaBinding",
                    Identifier("i"),
                    DecimalLiteral("0"))),
          Call(Empty,
               List("JavaOp",
                    Identifier("<"),
                    Identifier("i"),
                    Identifier("10"))),
          Call(Empty,
               List("JavaIncrement".
                    Identifier("i"))),
          Call(Empty,
               List("JavaCall",
                    Identifier("System.out.println"),
                    Identifier("i")))))
```

which corresponds to the Fortress concrete syntax:

```
JavaFor(JavaBinding("i","0"),
        JavaOp("<","i","10"),
        JavaIncrement("i"),
        JavaCall("System.out.println", "i"))
```

Alternatively, the parse functions could be defined so that use sites expand into a `SyntaxTree` for a Fortress `for` loop.

Syntax expanders must not call any functions or refer to any variables except those declared to be `static`. Additionally, a syntax expander must not refer to any variables or functions that have modifier `test`.

Because syntax expanders are defined at the top-level of program components, and because they are syntactically distinguished, they can be identified before scanning or parsing. Use sites are then identified and expanded before parsing occurs.

3.6.1 Introduced variable names

Often, when expanding concrete syntax for a domain-specific language, it is useful to introduce variable binding constructs into the resulting `SyntaxTree`. It is required that such bindings, in general, respect the rules of hygiene and referential transparency [7]. Therefore, our system automatically renames identifiers, following the `syntax-case` system of Dybvig et al. [9].

3.6.2 Expanders for Fortress

As the above examples demonstrate, it is often useful to denote Fortress abstract syntax using Fortress concrete syntax. A special set of parametric syntax expanders are defined in the api `fortress.syntax` for every nonterminal in Fortress concrete syntax. The name of each expander consists of the name of the nonterminal in lowercase. The terminating symbol for each nonterminal consists of its name prefixed with `end_`. For example, the expression:

```
expr
  x + y
end_expr
```

evaluates to the `SyntaxTree`:

```
  Call(Empty,
       VarRef(Identifier("+")),
       VarRef(Identifier("x")),
       VarRef(Identifier("y")))
```

When one of these syntax expanders parses a binding construct, the bound identifier is replaced with an identifier resulting from a call to `gensym`, and all variable references captured by the original identifier are replaced with references to the new identifier.

For convenience, there is a special parametric expander `</` that behaves identically to the `expr` expander. Uses of this expander are terminated with `/>` and escaped with $\sim$ (see Section 3.6.3 for a discussion of escape clauses).

Using the `fortress.syntax` expanders, we can rewrite our original example of a simple expander as follows:

```
syntax Area = </Length²/>
```

3.6.3 Escape clauses

Programmers are encouraged to declare *escape* clauses in their expanders. Escape clauses allow for nested program fragments in another concrete syntax. They occur immediately before the = sign of an expander. They start with the reserved word `escape` followed by a delimiter `String`. For example, we modify our SQL expander as follows:

```
syntax sql exp end escape ˜ = parseSQL(exp)
```

An occurrence of the escape character at a use site delimits either the immediately proceeding identifier or a `SourceAssembly` enclosed in any of the standard parentheses. The delimited `SourceAssembly` is parsed as a Fortress expression. *This expression is allowed to be a use site of a syntactic expander, which may itself have an escape clause*.

We can now embed program fragments in other domain-specific languages at a use site of this expander. For example, we could write the following function:

```
spectralClass(star:SourceAssembly) = sql
  SELECT spectral_class FROM
    SELECT ˜star FROM stars
end
```

Escapes at use-sites of expanders are processed from the leftmost-innermost clause outward.

Chapter 4

Program component compilation and linking

Fortress programs are developed, compiled, and deployed as *encapsulated upgradable components* that exist not only as programming language features, but also as self-contained run-time entities that are managed throughout the life of the software. The imported and exported references of a component are described with explicit *apis*, which can be thought of as interfaces of components. With components and apis, Fortress provides the stability benefits of static linking with the sharing and upgrading benefits of dynamic linking. [1]

4.1 Overview

Components are the fundamental structure of Fortress programs. They export and import apis, which serve as "interfaces" of the components. Components do not refer directly to other components. Rather, all external references are to apis imported by the component. These references are resolved by linking components together: the references of a component to an imported api are resolved to a component that exports that api. Linking components produces new components, whose *constituents* are the components that were linked together.

Components are similar to modules in other programming languages, such as those of ML and Scheme [17, 14, 13]. But, unlike modules in those languages, components are designed for use during both development and deployment of software. In addition to compilation and linking, components can be produced by upgrading one component using another component that exports some of the apis exported by the first component.

A key aspect of Fortress components is that they are encapsulated, so that upgrading one component does not affect any other component, even those produced by linking with the component that was upgraded. Abstractly, each component has its own copy of its constituents. However, implementations are expected to share common constituents when possible.

Users do not manipulate components directly. Instead, every component is installed in a persistent database on the system. We think of this database, which we call a *fortress*, as the agent that actually performs operations such as compilation, linking, upgrading, and execution of components: a virtual machine, a compiler, and a library registry all rolled into one. A fortress also maintains a list of apis that are installed on it. A fortress also provides a shell by which the user can issue commands to it.

[1]The system described in this chapter is based on that described in [2].

The ways in which fortresses are actually realized on particular platforms is beyond the scope of this specification. An implementor might choose to instantiate a fortress as a process, or as a persistent object database stored in a file system, with fortress operations being implemented as scripts that manipulate this database.

In addition to an informal description of the component system, we also formally specify key functionality of the system, and illustrate how we can reason about the correctness of the system. Components and apis are abstract immutable objects. A fortress maps names to components installed on the system. The fortress operations are modeled as methods of the fortress that change the mapping.

4.2 Source Code

We call the source code for a single software component a "project". Typically, when a project written in other programming languages is compiled, each file in the project is separately compiled. To ship an application, these files are linked together to form an application or library. Fortress uses a different model: a project is compiled directly into a single component, which is installed in the compiling fortress.

From the point of view of the compiler, all the source code for a project is contained in a single file. This approach simplifies the design, and gives a well-defined order for initialization of static elements of the component. However, this approach is unworkable for components of substantial size. Therefore, the compiler can be instructed to concatenate several source files together before compiling, while maintaining the original source location information.

After these components are compiled from source files, they can then be linked together to form larger components.

Components

In this specification, we will refer to components created by compiling a file as "simple components", while components created by linking components together will be known as "compound components".

The source code of a simple component definition begins with the reserved word `component` followed by an identifier, followed by a sequence of import and export declarations, and finally a sequence of declarations and definitions.

Each import or export declaration includes api names. An api serves as an interface of a component; it includes the declarations (but not definitions!) of top-level functions, objects, traits, and other values. In our examples, we use published descriptions of packages in the Java 6.0 API [24] as examples of apis expressible in our component system. We use, as names for these apis, the names of the corresponding Java packages, with `java` replaced with `fortress`. For example, the following is the beginning of a source file for a fictional application `IronCrypto`:

```
component com.sun.IronCrypto

import fortress.io
import fortress.security

export fortress.crypto
...
end
```

When a component is compiled, the apis it refers to must be present in the fortress. The import declarations in a component are not a way to abbreviate unqualified names of objects or functions. In our system, an import declaration merely allows references to the imported api to appear in the component definition. References to elements of an imported api must be fully qualified.

A key design choice we make is to require that components never refer to other components directly; all external references are to apis. This requirement allows programmers to extend and test existing components more easily, swapping new implementations of libraries in and out of programs at will.

For convenience, the following extended forms of import declarations are provided:

```
import {name+} from api
import * from api
```

The first declaration imports the given api and allows the listed elements (separated by commas) to be referred to with their unqualified names. The second imports the given api and allows all elements in that api to be referred to with unqualified names. If multiple elements with conflicting names are imported from separate apis, all references to those elements within the component definition must be fully qualified. Every component implicitly imports a set of *core apis* called *the Fortress standard library* (e.g., `fortress.lang` and other core apis to be determined); every fortress has at least one component implementing all of these apis. A *preferred* component exporting these apis (configurable by the user) is implicitly linked to every component installed in the fortress.

One important restriction on components is that no api may be both imported and exported by the same component. This restriction is necessary to make sense of the operations on components that we define in section 4.3. Formally, we introduce two functions on components, *imp* and *exp*, that return the imported and exported apis of the component, respectively. For any component c, $\mathit{imp}(c) \cap \mathit{exp}(c) = \emptyset$. This restriction is required throughout to ground the semantics of operations on components, as discussed in Section 4.3.

Every component has a unique name, used for the purposes of component linking. This name includes a user-provided identifier. In the case of a simple component, the identifier is determined by a component name given at the top of the source file from which it is compiled. A build script may keep a tally on version numbers and append them to the first line of a component, incrementing its tally on each compilation. The name of a compound component is specified as an argument to the `link` operation (described in section 4.3) that defines it.

Component equivalence is determined nominally to allow mutually recursive linking of components. By programmer convention, identifiers associated with components begin with the reverse of the URL of the development team. A fortress does not allow the installation of distinct components with the same name. Component names are used during `link` and `upgrade` operations to ensure that the restrictions on upgrades to a component are respected, as explained in Section 4.3.

Every component also includes a vendor name, the name of the fortress it is compiled on, and a timestamp, denoting the time of compilation. The time of compilation is measured by the compiling fortress, and the name of the fortress is provided by the fortress automatically. Every timestamp issued by a fortress must be unique. The vendor name typically remains the same throughout a significant portion of the life of a user account, and is best provided as a user environment variable.

Apis

Apis are compiled from special api definitions. These are source files which declare the entities defined by the api, the names of all apis referred to by those declarations, and prose documentation. In short, the source code of an api should specify all the information that is traditionally provided for the published apis of libraries in other languages.

The syntax of an api definition is identical to the syntax of a component definition, except that:

1. An api definition begins with the reserved word `api` rather than `component`. As with components, the identifiers associated with apis are prefixed with the reverse of the URL of the development team.
2. An api does not include `export` declarations. (However, it does include `import` declarations, which name the other apis used in the api definition.)
3. Only declarations are included in an api definition. All method bodies and variable definitions are elided. A method or field declaration may include the modifier `abstract`. (Whether a declaration includes the modifier `abstract` has a significant effect on its meaning, as discussed below).

For example, consider the apis `fortress.io`, `fortress.security`, and `fortress.crypto`, with declarations similar to those in their respective Java packages. These apis are interdependent. For example, both `PublicKey` in `fortress.security` and `SecretKey` in `fortress.crypto` have the trait `fortress.io.Serializable` and the trait `CipherSpi` in `fortress.crypto` has methods that return values of type `AlgorithmParameters` in `fortress.security`. So the header of api `fortress.crypto` is written as follows:

```
api fortress.crypto

import fortress.io
import fortress.security
...
end
```

For the sake of simplicity, every reference in an api definition must refer either to a declaration in a used api (i.e., an api named in an import declaration, or a core api, which is implicitly imported), or to a declaration in the api itself. In this way, apis differ from signatures in most module systems: they are not parametric in their external dependencies.

Every api has a unique name that consists of a user-provided identifier. As with components, api equivalence is determined nominally. Every api also includes a vendor name, the name of the fortress it is compiled on, and a timestamp.

Component and api names exist in separate namespaces. For convenience, a compiler can also produce an api directly from a project with the same name as the component it is derived from. Such an api includes *matching* declarations that include all the public definitions (and only the public definitions) of the component.

A component must include, for every api a it exports, matching definitions for all the declarations in a. A matching definition of a declaration D is a definition C with the same name as D that includes a public member for every member in D and that includes definitions for all members other than those declared `abstract` in D. C is allowed to include additional definitions not declared in D.

Other than its identity, the only relevant characteristic of an api a is the set of apis that it uses, denoted by $\mathit{uses}(a)$. Because an api a might expose types defined in $\mathit{uses}(a)$, we require that a component that exports a also exports all apis in $\mathit{uses}(a)$ that it does not import. Formally, the following condition holds on the exported apis of a component c:

$$a \in \mathit{exp}(c) \wedge a' \in \mathit{uses}(a) \implies a' \in \mathit{imp}(c) \cup \mathit{exp}(c)$$

4.3 Basic Fortress Operations

We now describe the operations that can be performed on a fortress by developers and end-users for developing, installing, and maintaining components. We can think of these operations as commands to an interactive shell provided by the fortress.

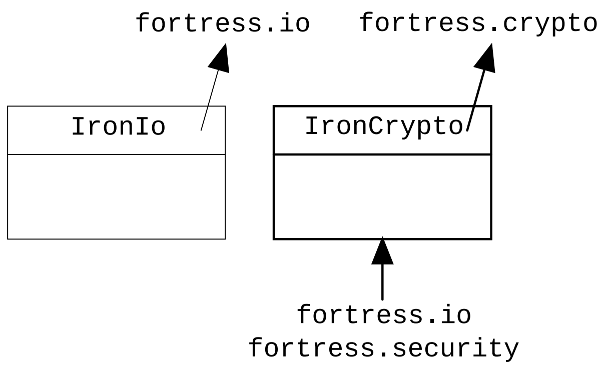

Figure 4.1: Simple components in box notation: A component is represented by a box, with the name of the component at the top of the box. The arrow protruding from the upper right corner of a box is labeled with the apis exported by the component. The arrow pointing into the bottom of a box is labeled with apis imported by the component. If no apis are imported, we elide the arrow.

In this section, we discuss operations on a fortress in their most basic form, postponing the discussion of more advanced options, including additional optional parameters, to Section 4.4. Although these more advanced options are critical to performing some real-world tasks with components, it is easier to describe their behavior after the basic forms of operations have been discussed.

Compile This operation takes the source code for a simple component (or api) definition and produces a new component object (or api object) that is installed on the fortress. Its type is as follows:

```
compile(file:String):()
```

For example, suppose `IronCrypto.fss` contains the source code for the aforementioned `IronCrypto` application, which imports `fortress.io` and `fortress.security`, and exports `fortress.crypto`. Suppose we also have source code, `IronIo.fss`, for another application, `IronIo`, which imports nothing and exports `fortress.io`. We generate these components by compiling the source files:

```
compile("IronIo.fss")
compile("IronCrypto.fss")
```

The results are depicted diagrammatically in Figure 4.1.

Formally, compilation takes a program and produces a new component with exported and imported apis as defined in the program. In the example above,

$$\mathit{imp}(\texttt{IronCrypto}) = \{\texttt{fortress.io}, \texttt{fortress.security}\}$$
$$\mathit{exp}(\texttt{IronCrypto}) = \{\texttt{fortress.crypto}\}$$

Link A collection of one or more components exporting different apis may be combined to form a new, compound, component by calling the `link` operation, passing the names of the components to link along with the name of the resulting compound component. Syntactically, a `link` operation is written as follows:[2]

[2]We present only the basic form of `link` here. `link` has additional optional arguments that we discuss in the Section 4.4.

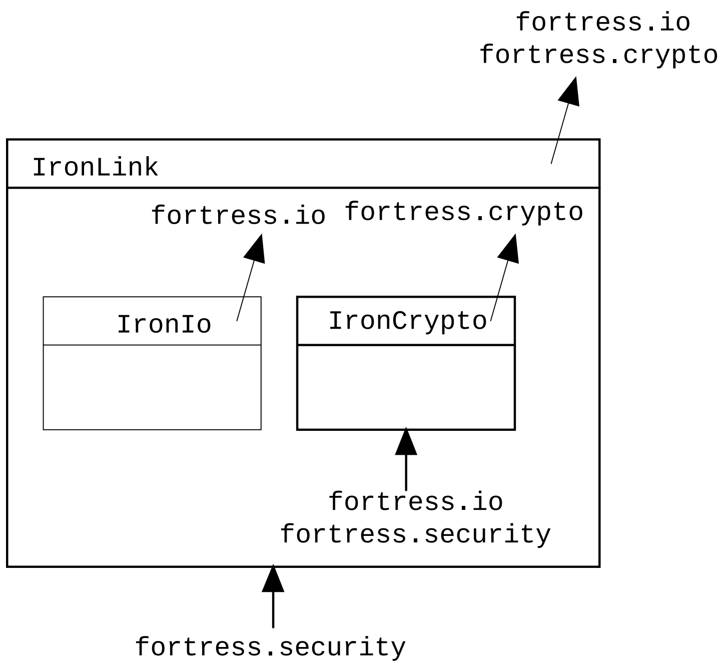

Figure 4.2: A compound component: A component inside another component is a constituent of the component that immediately encloses it.

```
link(result:String, constituents:String[]):()
```

The components being linked are called *constituents* of the resulting component, which exports all the apis exported by any of its constituents, and imports the apis imported by at least one of its constituents but not exported by any of them.

For example, we can link the `IronIo` and `IronCrypto` libraries compiled above:

```
link(IronLink, [IronIo, IronCrypto])
```

The resulting component, illustrated in Figure 4.2, imports `fortress.security` and exports `fortress.io` and `fortress.crypto`.

`link` does not distinguish between simple and compound components, so we can get arbitrarily nested components. For example, we can construct an application `CoolCryptoApp` by compiling another source code, `IronSecurity.fss`, for the library `IronSecurity` that imports `fortress.io` and exports `fortress.security`, and then linking the result with `IronLink`.

```
compile(IronSecurity.fss)
link(CoolCryptoApp, [IronSecurity, IronLink])
```

The resulting components are illustrated in Figure 4.3.

Formally, given a set $C = \{c_1, \ldots, c_k\}$ of components, we define a partial function $\mathit{link}(C)$ that returns the component resulting from c_1 through c_k. If $c = \mathit{link}(C)$, then $\mathit{exp}(c) = \bigcup_{c' \in C} \mathit{exp}(c')$ and $\mathit{imp}(c) = \bigcup_{c' \in C} \mathit{imp}(c') - \mathit{exp}(c)$.

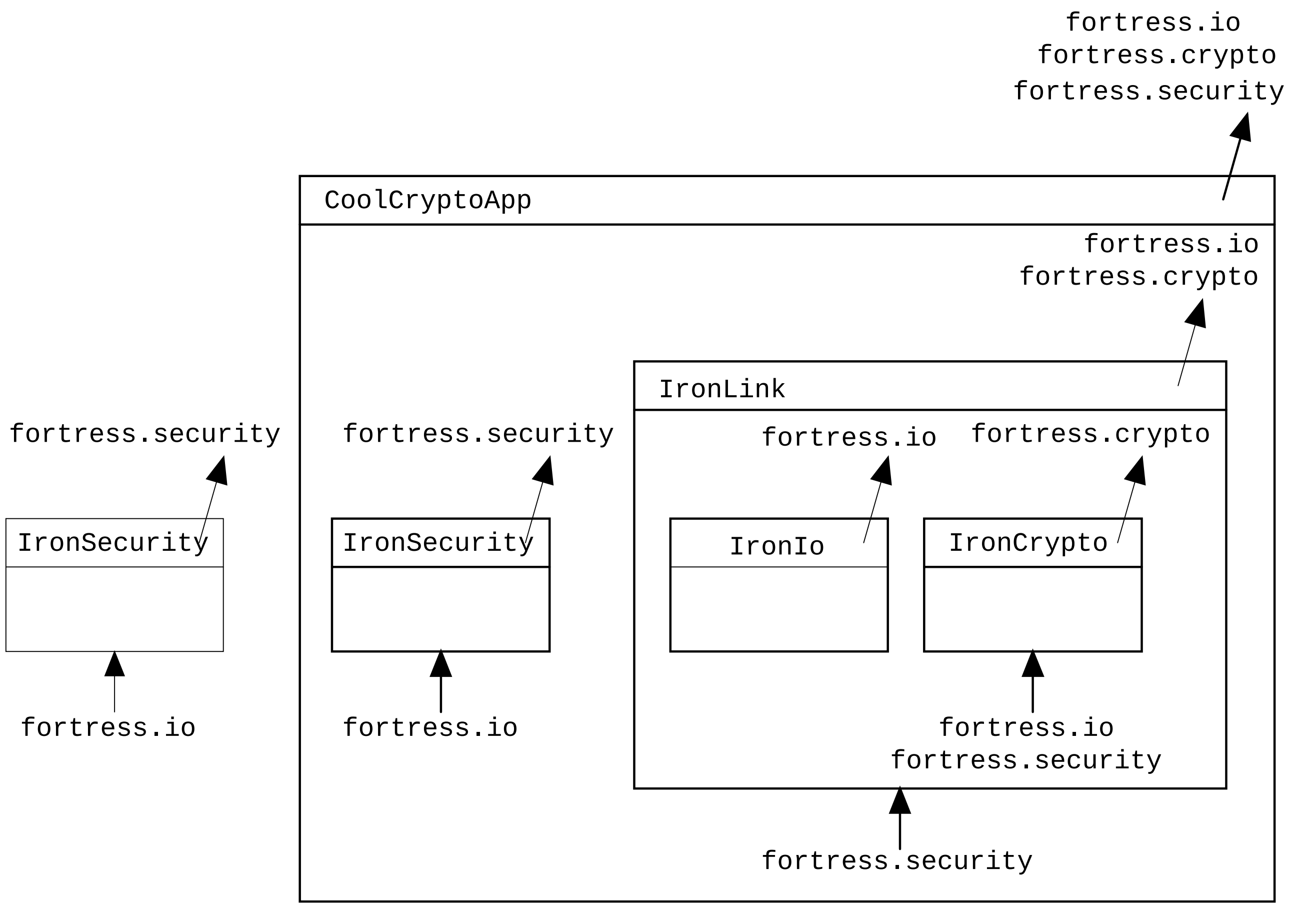

Figure 4.3: Repeated linking

The function *link* is partial because we do not allow arbitrary sets of components to be linked. In particular, two components cannot be linked if they export the same api.[3] This restriction is made for the sake of simplicity; it allows programmers to link a set of components without having to specify explicitly which constituent exporting an api a provides the implementation exported by the linked component, and which constituent connects to the constituents that import a: only one component exports a, so there is only one choice. Although we lose expressiveness with this design, the user interface to link is vastly simplified, and it is rare that including multiple components that export a given api in a set of linked components is even desirable. We discuss how even such rare cases can be supported in Section 4.4.

For a compound component, in addition to the exported and imported apis, we want to know what its constituents are. So we introduce another function *cns*, which takes a component and returns the set of its constituents. That is, $\mathit{cns}(\mathit{link}(C)) = C$. It is an invariant of the system that for any compound component c (i.e., $\mathit{cns}(c) \neq \emptyset$), any api imported by any of its constituents is either imported by c or exported by one of its constituents (i.e., $\bigcup_{c' \in \mathit{cns}(c)} \mathit{imp}(c') \subseteq \mathit{imp}(c) \cup \bigcup_{c' \in \mathit{cns}(c)} \mathit{exp}(c')$). This property is crucial for executing components, as we discuss below. A simple component c (i.e., one produced directly by compilation) has no constituents (i.e., $\mathit{cns}(c) = \emptyset$).

Execute Components provide implementations of the apis they export. A component is *executable* if it imports no apis and it exports the special api `executable`, defined as follows:

[3]There is one exception to this rule: the special api `upgradable`, which is used during upgrades discussed below.

```
api executable
public exec(args:String[]):()
```

An executable component may be *executed* by calling the `execute` operation, resulting in a call to the component's implementation of the `exec` function in a new process. Arguments to the `exec` function are passed to the shell:

```
execute(componentName:String, args:String[]):()
```

We say that a component is being executed when `execute` has been called on that component and has not yet returned, or if it is the constituent component of a component being executed. During an execution, references may be made to apis exported by a component being executed, which may in turn make references to apis that it imports.

For references to an api a exported by the component, if the component is simple, then it contains the code necessary to evaluate any reference to an api it exports, possibly making references to apis that it imports to do so. If the component is compound, then it contains a unique constituent that exports a; the reference is resolved to that constituent component.

For external references within a constituent component, recall that all such references in a component must be to apis that the component imports. A component being executed either does not import any api (and thus there are no external references to resolve), or else is a constituent of another component that is being executed. In the latter case, the constituent defers the reference to its enclosing component.

For example, suppose `CoolCryptoApp` above is the constituent of some executable component, and when that component is executed, it generates a reference to `SecretKey` in `fortress.crypto`, which it resolves to `CoolCryptoApp`. `CoolCryptoApp` resolves this reference to `IronLink`, which resolves it to `IronCrypto`, which is a simple component. Suppose that in evaluating this reference, `IronCrypto` generates a reference to `PublicKey` in `fortress.security`. Because `IronCrypto` imports `fortress.security`, it resolves this reference to its enclosing component, `IronLink`, which in turn resolves it to `CoolCryptoApp`, which resolves it to `IronSecurity`, which is a simple component.

Not all projects are compiled to components that export `executable`. For example, a library component does not usually export `executable`.

Upgrade Compound components may be upgraded with new constituent components by calling an `upgrade` operation, passing the name of the component to upgrade (the *target*), the name of a component to upgrade with (the *replacement*), and a name for the resulting component (which we call the *result*). The type of the `upgrade` operation is as follows:

```
upgrade(target:String, replacement:String, result = target):()
```

If no result name is provided, the result is bound to the name of the target, and the target is uninstalled (see below).

For example, we can upgrade `CoolCryptoApp` with a component `CoolSecurity`, which exports `fortress.security` and imports nothing to `CoolCryptoApp.2.0`.

```
upgrade(CoolCryptoApp, CoolSecurity, CoolCryptoApp.2.0)
```

The resulting component is illustrated in Figure 4.4. Notice that the constituent, `IronSecurity`, exporting `fortress.security` has been replaced.

A component can be upgraded only if it exports the special api `upgradable`, defined as follows:

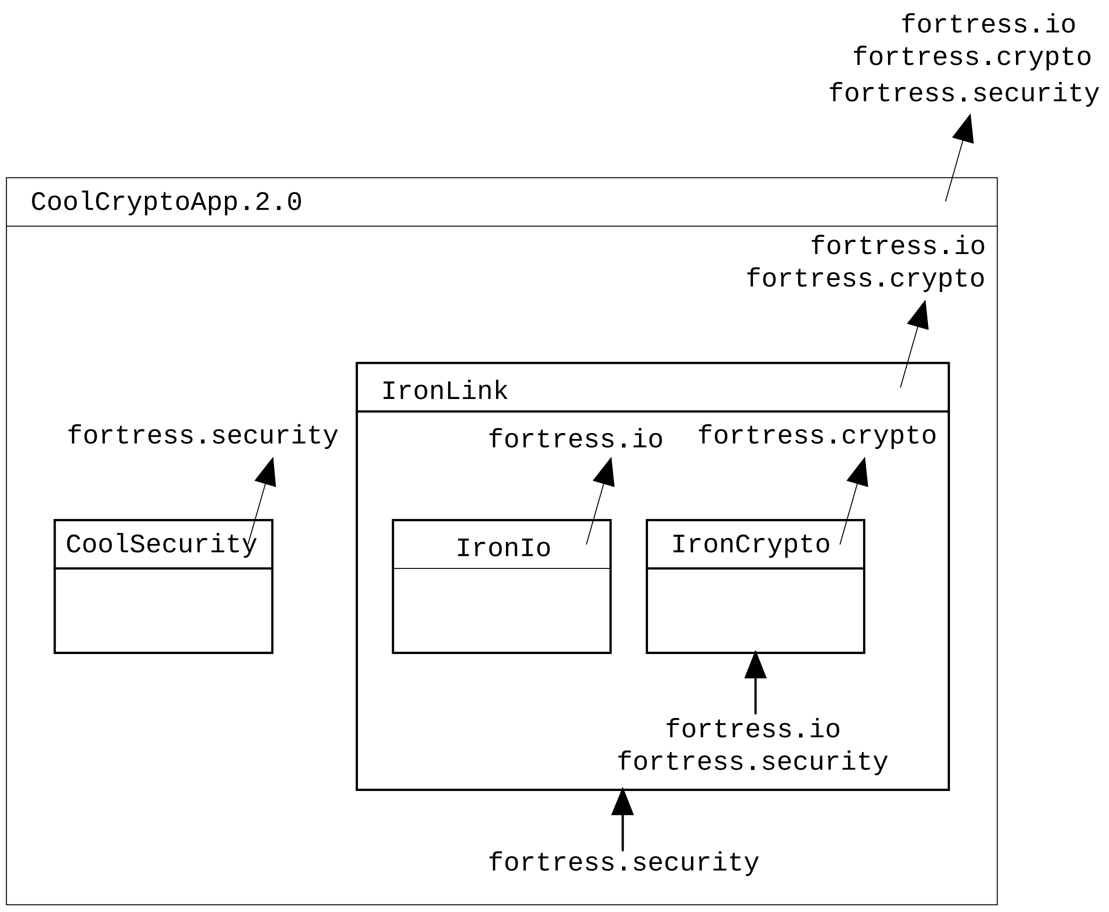

Figure 4.4: An upgraded component

```
api upgradable
import {Component, UpgradeException} from components

public isValidUpgrade(that:Component):Boolean
public upgrade(that:Component):Component throws UpgradeException

end
```

The `upgradable` api imports a special api `components` that provides handles on `Component` and `Api` objects. The `components` api is described in Appendix B.

An `upgrade` operation on a component invokes the `isValidUpgrade` method, as declared in the api `upgradable`. This function must take a component and return `True` iff it is legal to upgrade with respect to that component. The `upgrade` operation throws an exception if `isValidUpgrade` returns `False`. Developers can define their own versions of this component to restrict how their components can be upgraded. For example, they can prevent upgrades with older versions of a component, or with a matching component from an untrusted vendor.

The `upgradable` api presents a problem for our model. Its implementation by the various constituent components in a compound component must be accessed during an `upgrade` operation. However, because the exported apis of the constituent components must be disjoint, they cannot all export `upgradable` after linking.

We solve this problem by introducing an additional step during linking. In a `link` operation, a special component, called a *restriction component*, is constructed automatically, based on the provided constituents. This component exports the `upgradable` api; its implementation is a function of all the constituents provided to the `link` operation. The provided constituents are then used to construct a new set of constituents that are identical to the provided constituents except that they do not export `upgradable`. These new constituents are then combined, along with the restriction component, to form the constituents of a new compound component.

In addition to the constraints imposed by a component's `isValidUpgrade` method, there are several other conditions that must be met in order for an upgrade to be valid. These conditions are necessary to ensure that the resulting component is well-formed and imports and exports the same apis as the target: [4]

1. Every api imported by the replacement must be either imported or exported by the target.
2. The apis exported by the replacement must be a subset of those exported by the target.
3. If the replacement does not subsume a constituent then either the replacement and constituent do not export any apis in common or the constituent can be upgraded with the replacement.

The rationale for the first two conditions is straightforward: If an api is imported by the replacement but not imported or exported by the target, then references to that api cannot be resolved in the result (unless we also import that api in the result). If an api is exported by the replacement but not the target, then the result will export an api not exported by the target.

The third condition says that the constituents of the target can be partitioned into three sets: those that are subsumed by the replacement, those that are unaffected by the upgrade, and all the rest, which can be upgraded with the replacement. This condition enables recursive propagation of upgrades. That is, an upgrade not only replaces constituents at the top level of the the component, but is also propagated into any constituents with which it exports some apis in common. Thus, in the example above, we could have upgraded `CoolCryptoApp` with a component that exports `fortress.io`. However, we could not have upgraded `CoolCryptoApp` with a component that exports both `fortress.security` and `fortress.io` because `IronLink` exports `fortress.io` but not `fortress.security`. In Section 4.4, we show how hiding and constraining apis can help us get around many of the limitations that this condition imposes.

Formally, a predicate *upg?* takes two components and indicates whether the first can be upgraded with the second; that is, $\mathit{upg?}(c_\mathrm{t}, c_\mathrm{r})$ returns true if and only if c_t can be upgraded with c_r. This predicate captures both the constraints imposed by a component's `isValidUpgrade` method and the conditions that guarantee the well-formedness of the result. That is,

$$\begin{aligned}
\mathit{upg?}(c_\mathrm{t}, c_\mathrm{r}) \implies\ & c_\mathrm{t}.\mathtt{isValidUpgrade}(c_\mathrm{r}) \\
& \wedge \mathit{imp}(c_\mathrm{r}) \subseteq \mathit{exp}(c_\mathrm{t}) \cup \mathit{imp}(c_\mathrm{t}) \\
& \wedge \mathit{exp}(c_\mathrm{r}) \subset \mathit{exp}(c_\mathrm{t}) \\
& \wedge \forall c \in \mathit{cns}(c_\mathrm{t}).(\mathit{exp}(c) \subseteq \mathit{exp}(c_\mathrm{r}) \vee \mathit{exp}(c) \cap \mathit{exp}(c_\mathrm{r}) = \emptyset \vee \mathit{upg?}(c, c_\mathrm{r}))
\end{aligned}$$

Recall that in our system, unlike with dynamic linking, components are encapsulated so that an upgrade to one component does not affect any other component on the system. We can imagine that all operations on components copy the components that they operate on rather than share them. Because components are immutable, these two interpretations are semantically indistinguishable. Convenience operations that support mass upgrades are provided on fortresses (e.g., an `upgradeAll` operation that takes a component and upgrades all components in the fortress that can be upgraded with its argument).

[4] These conditions are sufficient provided there are no hidden or constrained apis, which are discussed in Section 4.4.

Extract and install A component installed on a fortress may be *extracted* by calling an `extract` operation on the fortress, passing the name of the component as an argument, along with an argument `prereqs`, denoting the names of all apis that must be installed on any fortress before this component can be installed.

```
extract(componentName:String, prereqs:{String} = {}):()
```

Furthermore, the destination fortress must have a component that exports these apis and is a valid upgrade of the extracted component. Intuitively, a `prereqs` argument allows a component to be serialized without having to include all of its libraries; new libraries can be provided when the component is installed at a destination fortress.

The `prereqs` argument is optional; if omitted, the extracted component can be installed on any fortress. Any component can be extracted; however only compound components can be extracted with a `prereqs` argument: because extracted components must be upgradable with respect to a component exporting their `prereqs`, no `prereqs` argument makes sense for a simple component.

The apis included in a `prereqs` argument must be the apis exported by some subset of the extracted component's constituents (or a subset of the constituents of one of its constituents, and so on, due to recursive updating).

The extracted component is serialized to a file, including all the apis it refers to (and, transitively, all apis they refer to) and all constituent components, except those that export the `prereqs`. This operation does not remove the extracted component from the fortress; there is a separate `uninstall` operation for that.

When the component is extracted, if no `prereqs` were passed to the `extract` operation, then the contents of the file can be deserialized by any fortress into the extracted component, which can be installed on the fortress. However, if `prereqs` were passed to `extract`, then the file must be deserialized into a component that exports only the `installable` api:

```
api installable
import Component from components
public reconstitute(candidate:Component):Component
end
```

The deserialized component is immediately linked with preferred implementations of all of its imported apis. (Preferred implementations of apis are maintained in a table by a fortress, which maps each api to a list of components that implements it, in order of preference). Because the deserialized and linked component exports the `installable` api, it has a `reconstitute` method that takes a *candidate* component, which exports the `prereq` apis, and checks whether the given component satisfies the `isValidUpgrade` condition of the extracted component. If so, it returns the extracted component upgraded with the given component. The `reconstitute` method is called by the fortress with a new component, formed by linking the preferred components for each api in the extracted components' `prereqs` argument.

Note that an extracted component with `prereqs` apis is *not* the same as an extracted component that imports the same apis but has no `prereqs` apis. The latter can always be installed on a fortress, and then can be subsequently linked with any component that exports the imported apis. In contrast, the fortress has no access to an extracted component with `prereqs` apis unless it has a component that exports these apis and satisfies the `isValidUpgrade` method of the extracted component. This difference provides a means for controlling access to the extracted component, for security, legal, or other reasons.

Syntactically, an `install` operation takes the name of a file constraining an extracted component. The `install` operation is overloaded with another operation that takes the name of a component to match `prereqs`. If this optional argument is provided, and the deserialized component exports the `installable` api, then the `reconstitute` method is called with the component denoted by the optional argument of `install`, rather than the fortress' preferred implementation of the `prereq` apis. Install operations are written as follows:

```
install(file:String):()
install(file:String, prereqs:{String}):()
```

By default, a fortress adds a newly installed component to the head of the “preferred” list for every api it exports. However, this default may be overridden by the end-user; an end-user may modify the table or even map some apis differently during a particular installation. If one or more of the apis required by an extracted component is not mapped to an api on the destination fortress, an exception is thrown.

There is a corresponding operation for apis, `installApi`, that takes a serialization of a set of apis and installs them into a fortress.

```
installApi(file:String):()
```

This set of apis must be closed under imports. If an api that is installed in this way is already installed on the fortress, the definitions must match exactly, or an exception is thrown.

Uninstall An `uninstall` operation takes the name of a component as an argument and removes the top-level binding of that component from a fortress. Note that the uninstalled component may have been linked to other components, or used as a replacement in an upgrade, and the result may still be installed; an `uninstall` operation will not affect these other components.

```
uninstall(file:String):()
```

There is a corresponding operation for apis, `uninstallApi`, that removes an api from a fortress.

```
uninstallApi(file:String):()
```

Typically, this operation is used only to remove apis that have been corrupted in some fashion.

4.4 Advanced Features of Fortress Operations

The system we have described thus far provides much of the desired functionality of a component system. However it has a few significant weaknesses:

1. It exposes to everyone all the apis used in the development of a project.
2. By allowing access to these apis, it inhibits significant cross-component optimization.
3. It prevents components that use two different implementations of the same api from being linked, even if they never actually pass references to that api between each other.
4. It restricts the upgradability of compound components, as described earlier.

We can mitigate all these shortcomings by providing two simple operations, `hide` and `constrain`. Informally, `hide` makes apis no longer visible from outside the component and `constrain` merely prevents them from being exported. An api that is constrained but not hidden can still be upgraded. There are other subtle consequences of this distinction, which we discuss as they arise.

Formally, we introduce two new functions on components: *vis*, which returns the apis of a component that have not been hidden; and *prov*, which returns those visible apis that are exported by some top-level constituent of the component (or all the exported apis of a simple component); we say these apis are *provided* by the component. We need to distinguish provided apis because they can be imported by the top-level constituents of a component, and thus by a replacement component in an upgrade, while other visible apis cannot be. Thus, for a compound component c, $\mathit{prov}(c) = \mathit{vis}(c) \cap \bigcup_{c' \in \mathit{cns}(c)} \mathit{exp}(c')$. For a simple component c, $\mathit{prov}(c) = \mathit{vis}(c) = \mathit{exp}(c)$.

Some of the properties about the apis exported by a component discussed in Section 4.3 are actually properties of apis that are visible or provided by a component. For example, apis visible in a component cannot be imported by that component, even if they are not exported. Other properties are really properties only of the exported apis. Most importantly, components that do not export any common apis can be linked, as can components that share only visible apis.

Constrain A `constrain` operation takes a component name of an installed component, a new component name, and a set of apis, and produces a new component that does not export any of the apis specified. Syntactically, we write:

```
constrain(source:String, destination = source, apis:{String}):()
```

If no `destination` name is provided, the name of the `source` is used.

The set of apis provided must be a subset of the apis exported by the component. Also, recall that every api used by an api exported by a component must be imported or exported by that component. Thus, if we constrain an api that is used by any other api exported by the component, then we must also constrain that other api.

If the component is a simple component, we first link it by itself, and then apply `constrain` to the result.

Formally, if c is a compound component and $A \subset \mathit{exp}(c)$ is a set of apis such that $a \in \mathit{exp}(c) \wedge a' \in \mathit{uses}(a) \cap A \implies a \in A$, we define $c' = \mathit{constrain}(c, A)$ such that $\mathit{exp}(c') = \mathit{exp}(c) - A$ and for any component c'', $\mathit{upg?}(c', c'') \iff \mathit{upg?}(c, c'') \wedge \mathit{exp}(c') \not\subseteq \mathit{exp}(c'')$. The *imp*, *vis*, *prov* and *cns* functions all have the same values for c and c'. The extra condition on the upgrade compatibility simply captures the restriction we mentioned above, that a replacement component should not export every api exported by the target.

Hide A `hide` operation is like a `constrain` operation, except that the given set of apis is subtracted from the visible and provided apis, along with the exported apis, in the resulting component.

```
hide(source:String, destination = source, apis:{String}):()
```

The requirement of apis being imported or exported whenever an api using them is exported also applies to visible apis. Thus, if we hide an api used by another exported api, we must hide that other api as well.

Formally, if c is a compound component and $A \subset \mathit{vis}(c)$ is a set of apis such that $\mathit{exp}(c) \not\subseteq A$ and $a \in \mathit{vis}(c) \wedge a' \in \mathit{uses}(a) \cap A \implies a \in A$, we define $c' = \mathit{hide}(c, A)$ such that $\mathit{vis}(c') = \mathit{vis}(c) - A$, $\mathit{prov}(c') = \mathit{prov}(c) - A$, $\mathit{exp}(c') = \mathit{exp}(c) - A$, and for any component c'', $\mathit{upg?}(c', c'') \iff \mathit{upg?}(c, c'') \wedge \mathit{exp}(c') \not\subseteq \mathit{exp}(c'') \wedge \mathit{vis}(c'') \subseteq \mathit{vis}(c')$. The additional clause in $\mathit{upg?}(c', c'')$ (compared with that of *constrain*) reflects the hiding of the apis: we can no longer upgrade apis that are hidden.

Link With constrained apis, there is a new restriction on link: Any api visible in one constituent and imported by another must be exported by some constituent. This restriction is necessary because an api visible in a component cannot be imported by that component. Thus, if one of the component's constituents imports that api, then the api must be provided by some other constituent. Other than that, the `link` operation is largely unchanged: the visible apis are just all the apis visible in any constituent, and the provided apis are just those exported by any constituent. There is a subtle additional restriction on how linked components can be upgraded, which we discuss below.

Rather than requiring users and developers to call `constrain` and `hide` directly, we provide optional parameters to the `link` operation to do these operations immediately. The `link` operation has the following type:

```
link(result:String, constituents:String[], export = {} , hide = {}):()
```

If the export clause is present, only those apis listed in the set following `export` are exported; the others are constrained. If the hide clause is present, those apis listed in the set following `hide` are hidden. An exception is thrown if the export clause contains any api not exported by any constituent, or if the hide clause contains any api not visible in any constituent.

Hiding enables us to handle the rare case in which programmers want to link multiple components that implement the same api without upgrading them to use the same implementation. Before linking, the programmer simply hides (or constrains) the api in every component that exports it except the one that should provide the implementation for the new compound component.

For example, suppose we wish to link the following two components:

- A component `NetApp` that imports `fortress.io` and exports the `fortress.net` api.
- A component `EditApp` that imports `fortress.io` and exports the `fortress.swing.textrf` api.

We want to link these two components to use in building an application for editing messages and sending them over a network. But we want to use different implementations of `fortress.io` (e.g., `IoApp1` and `IoApp2` for the two components). We simply perform the following operations:

```
link(temp1, [ NetApp, IoApp1], export = {fortress.net}, hide = {fortress.io})
link(temp2, [EditApp, IoApp2], export = {fortress.swing.textrf},
                               hide   = {fortress.io})
link(NetEdit, [temp1, temp2])
```

In this case, the `NetEdit` component does not export, or even make visible, `fortress.io` at all.

Upgrade For the `upgrade` operation, there is no change at all in the semantics. However, because hiding and constraining apis allow us to change the apis exported by a component, it is possible to do some upgrades that are not possible without these operations.

For example, suppose we have a component `IoSecurity` that exports `fortress.io` and `fortress.security`, and we want to upgrade `CoolCryptoApp` with `IoSecurity`. As discussed above, we cannot use `IoSecurity` directly because `IronLink` exports `fortress.io` but not `fortress.security`. We can get around this restriction by doing two upgrades, one with `fortress.security` hidden and the other with `fortress.io` hidden.

```
hide(IoSecurity, NewIo,       {fortress.security})
hide(IoSecurity, NewSecurity, {fortress.io})
upgrade(CoolCrytoApp,      NewSecurity, temp1)
upgrade(CoolCryptoApp.3.0, temp1,       NewIo)
```

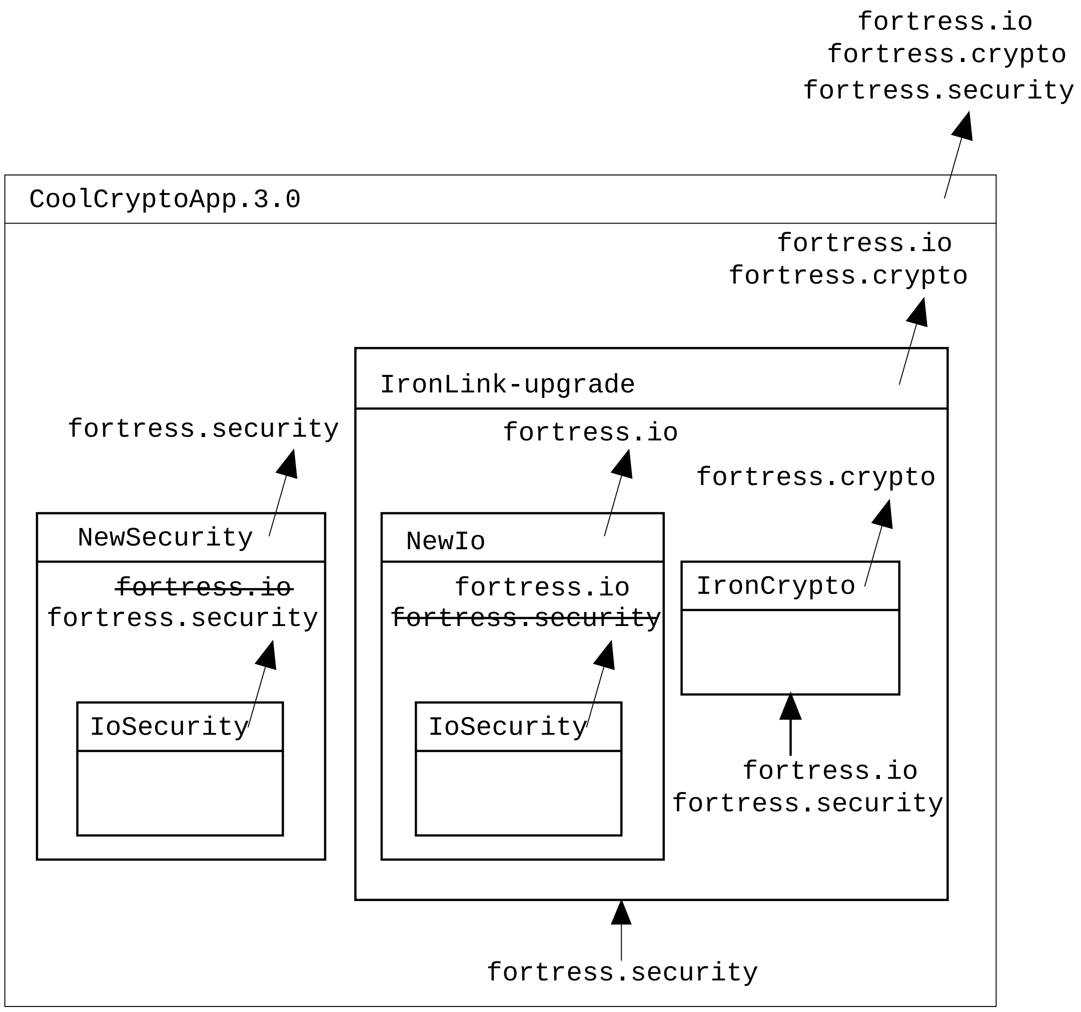

Figure 4.5: Upgrading with hidden apis: Crossed out apis are hidden.

The resulting component is shown in Figure 4.5.

The interplay between imported, exported, visible and provided apis introduces subtleties that not present in our discussion above. In particular, the last of the three conditions imposed for well-formedness of upgrades is modified to state that for any constituent that is not subsumed by a replacement component, either it can be upgraded with the replacement, or its *visible* apis are disjoint from the apis exported by the replacement (i.e., it is unaffected by the upgrade). To maintain the invariant that no two constituents export the same api, we need another condition, which was implied by the previous condition when no apis were constrained or hidden: if the replacement subsumes any constituents of the target, then its exported apis must exactly match the exported apis of some subset of the constituents of the target. That is, if $\mathit{upg?}(c_\mathrm{t}, c_\mathrm{r}) \wedge \exists c \in \mathit{cns}(c_\mathrm{t}).\ \mathit{exp}(c) \subseteq \mathit{exp}(c_\mathrm{r})$ then $\mathit{exp}(c_\mathrm{r}) = \bigcup_{c \in C} \mathit{exp}(c)$ for some $C \subset \mathit{cns}(c_\mathrm{t})$. In practice, this restriction is rarely a problem; in most cases, a user wishes to upgrade a target with a new version of a single constituent component, where the apis exported by the old and new versions are either an exact match, or there are new apis introduced by the new component that have no implementation in the target.

4.5 Component-related modifiers

The following modifiers are specific to the Fortress component system.

abstract

A method declaration in a trait declaration of an api might include an `abstract` modifier, indicating that an object of the trait *does not* inherit the definition of the method from that trait.

private

An object, function, variable, or trait declared as `private` must not be declared by any implementing api of a component. Apis are not allowed to include the `private` modifier on any of their constituents.

A method or field declared as `private` must not be referred to outside its enclosing object or trait definition.

Chapter 5

Abstract Syntax

In this chapter, we describe the abstract syntax of Fortress programs.

First, we provide some context about the role that the abstract syntax plays in the Fortress language. For the sake of interoperability of compilers and development tools, a Fortress compiler is required to be divided into the following top-level phases:

- Parser
- Type Annotator
- Code Generator

The Parser passes a syntactically well-formed abstract syntax tree to the Type Annotator, the Type Annotator passes a fully annotated type safe abstract syntax tree to the Code Generator, and the Code Generator produces a simple component. (Note that the output of the Type Annotator is a special case of the Parser output; that is, a fully annotated abstract syntax tree is just an abstract syntax tree.)

The only communication between these phases is the information passed in the data structures mentioned above. For example, there is no symbol table passed between modules; all information must be encapsulated in the abstract syntax trees.

Programmers are able to easily plug in their own modules for these various phases because the interfaces between modules are tightly defined. For example, a programmer could define his own Parser module that would work just fine with the rest of the compiler provided that the output of the Parser is an abstract syntax tree. Alternatively, a programmer may define his own Type Annotator and hook it up to a Parser and code generator.

The data structures passed between modules should have well-defined serializations into text. That way, programmers can write new modules in other languages and plug them into the rest of the system. Our goals with this design are to make it easy to write new diagnostic tools, and to standardize the abstract syntax and allow for specialized notations that parse into it.

The abstract syntax of Fortress programs in BNF notation is as follows:

Program	::=	*Component*
	|	*Api*
Component	::=	`Component` *DottedName Import* Export* Def**
Api	::=	`Api` *DottedName Import* Decl**
DottedName	::=	`DottedName` *Id**
Import	::=	*ImportApi*
	|	*ImportIds*
ImportApi	::=	`Import` *DottedName* [*DottedName*]
ImportIds	::=	`Import` *DottedName ImportFrom*
ImportFrom	::	`AllImports`
	|	`NamedImports` *Id**
Export	::=	`Export` *DottedName*
Def	::=	*VarDef*
	|	*FnDef*
	|	*TraitDef*
	|	*ObjectDef*
	|	*DefOrDecl*
Decl	::=	*VarDecl*
	|	*FnDecl*
	|	*TraitDecl*
	|	*ObjectDecl*
	|	*DefOrDecl*
DefOrDecl	::=	`Dimension` *Id*
	|	`UnitVar` *Id TypeRef*
	|	*TypeAlias*
TypeAlias	::=	`TypeAlias` *Id TypeRef*
VarDef	::=	`VarDef` *VarMod* Id* [*TypeRef*] *Expr*
VarDecl	::=	`VarDecl` *VarMod* Id TypeRef*
UniversalMod	::=	`Static` | `Test`
TraitMod	::=	`Private` | `Value` | *UniversalMod*
ObjectMod	::=	*TraitMod*
FnMod	::=	`Atomic` | `Io` | `Private` | `Pure` | *UniversalMod*
MdMod	::=	`Getter` | `Setter` | *FnMod*
VarMod	::=	`Unit` | `Var` | *UniversalMod*
FldMod	::=	`Hidden` | `Settable` | `Var` | `Wrapped` | *UniversalMod*
FnDef	::=	`FnDef` *FnMod* FnHeader Expr*
FnDecl	::=	`FnDecl` *FnMod* FnHeader*
FnHeader	::=	`FnHeader` *FnName TypeParam* Param** [*TypeRef*] *FnClauses*
FnName	::=	*Op*
	|	*Id*
FnClauses	::=	`FnClauses` *Throws Where Contract*
Throws	::=	`Throws` *TypeRef**
Where	::=	`Where` *WhereClause**
WhereClause	::=	*WhereExtends*
	|	*TypeAlias*
WhereExtends	::=	`WhereExtends` *Id TypeRef*
Contract	::=	`Contract` *Expr* Ensures Expr**
Ensures	::=	`Ensures` *EnsuresClause**
EnsuresClause	::=	`EnsuresClause` *Expr** [*Provided*]
Provided	::=	`Provided` *Expr*

```
TypeParam          ::=  SimpleTypeParam
                    |   DimensionParam
                    |   NatParam
                    |   OperatorParam
SimpleTypeParam    ::=  SimpleTypeParam Id [TypeRef*]
NatParam           ::=  NatParam Id
DimensionParam     ::=  DimensionParam Id
OperatorParam      ::=  OperatorParam Op
Param              ::=  Param Id [TypeRef] [Expr]
TraitDef           ::=  TraitDef TraitHeader (MdDef | MdDecl)*
TraitDecl          ::=  TraitDecl TraitHeader MdDecl*
TraitHeader        ::=  TraitClauses TraitMod* Id TypeParam* [Extends] Excludes [Bounds] Where
Extends            ::=  Extends TypeRef*
Excludes           ::=  Excludes TypeRef*
Bounds             ::=  Bounds TypeRef*
MdDef              ::=  MdDef MdMod* FnHeader Expr
MdDecl             ::=  MdDecl[Abstract] MdMod* FnHeader
ObjectDef          ::=  ObjectDef ObjectHeader FldDef* MdDef*
ObjectDecl         ::=  ObjectDecl ObjectHeader FldDecl* MdDecl*
ObjectHeader       ::=  ObjectHeader ObjMod* Id TypeParam* ValParam* [Traits] FnClauses
Traits             ::=  Traits TypeRef*
ValParam           ::=  ValParam[Transient] FldMod* Param
FldDef             ::=  FldDef FldMod* Id [TypeRef] Expr
FldDecl            ::=  FldDecl FldMod* Id TypeRef
TypeRef            ::=  IdType DottedName
                    |   ParamType DottedName TypeArg*
                    |   SetType TypeRef
                    |   MapType TypeRef TypeRef
                    |   ListType TypeRef
                    |   TupleType TypeRef*
                    |   MatrixType TypeRef Indices
                    |   ArrayType TypeRef Indices
                    |   ArrowType KeywordArgType* TypeRef* TypeRef Throws
                    |   RestType TypeRef
                    |   DimType
TypeArg            ::=  TypeRef
                    |   NatTypeArg
                    |   OprTypeArg
NatTypeArg         ::=  BaseNatTypeArg Number
                    |   IdNatTypeArg Id
                    |   SumNatTypeArg NatTypeArg*
                    |   ProductNatTypeArg NatTypeArg*
OprTypeArg         ::=  Id
                    |   Op
DimType            ::=  UnitDimType
                    |   NameDimType DottedName
                    |   ExponentDimType DimType NatTypeArg
                    |   ProductDimType DimType*
                    |   QuotientDimType DimType DimType
Indicies           ::=  FixedDim Range*
                    |   PolyDim
Range              ::=  Range [NatTypeArg] [NatTypeArg]
KeywordArgType     ::=  KeywordArg Id TypeRef
```

```
Expr           ::=  VarRef Id
                |   Let
                |   Flow
                |   Value
                |   Comprehension
                |   LooseJuxt Expr*
                |   TightJuxt Expr*
                |   OprExpr Op Expr*
                |   PostfixExpr Expr Op
                |   FieldSelection Expr Id
                |   Assignment Expr [Op] Expr
                |   Apply Expr Expr
                |   TypeApply Expr TypeArg*
                |   Subscript Expr Expr*
                |   ExternalSyntax
                |   ExpanderVarRef
Flow           ::=  Accumulator Accumulator Generator* Expr
                |   Throw Expr
                |   AtomicExpr Expr
                |   Tryatomic Expr
                |   Exit [Id] [Expr]
                |   Block Expr*
                |   If Expr Expr* Elif* Expr*
                |   Try Expr* [Catch] TypeRef* Expr*
                |   Case Expr [Id] CaseClause* Expr*
                |   Dispatch DispatchTypecase
                |   TypeCase DispatchTypecase
                |   Spawn [Expr] Expr
                |   For Generator* Expr
                |   While Expr Expr
                |   Label Id Expr*
Value          ::=  Number
                |   FnExpr Param* [TypeRef] Throws Expr
                |   ObjectExpr Traits FldDef* MdDef*
                |   Set Expr*
                |   Map Entry*
                |   List Expr*
                |   Tuple Expr*
                |   Matrix Number Extent* Expr*
                |   Array Number Extent* Expr*
                |   Interval Expr Expr
                |   Infinity
                |   String
                |   Void
Extent         ::=  Extent Number Number
Entry          ::=  Entry Expr Expr
Comprehension  ::=  SetComprehension Expr Expr* Generator*
                |   ArrayComprehension ArrayComprehensionClause*
Accumulator    ::=  AccumulatorId Id
                |   AccumulatorBig Op
Generator      ::=  Generator Id* Expr
```

Let	::=	`LetId` *Id** *TypeRef*
	|	`LetBinding` *L-Val** *Expr*
	|	`LetFn` *Id Param** [*TypeRef*] *Throws Expr*
L-Val	::=	`L-Scalars` *Id* [*TypeRef*]
	|	`L-Array` *Array*
Elif	::=	`Elif` *Expr Expr**
Catch	::=	`Catch` *Id CatchClause**
CatchClause	::=	`CatchClause` *TypeRef Expr**
CaseClause	::=	`CaseClause` *Expr Expr**
DispatchTypecase	::=	`DispatchTypecase` *Binding** *DispatchClause** *Expr**
DispatchClause	::=	`DispatchClause` *TypeRef** *Expr**
Binding	::=	`Binding` *Id Expr*
ArrayComprehensionClause	::=	`ArrayComprehensionClause` *Binding Expr** *Generator**
ExternalSyntax	::=	`ExternalSyntax` *Id SourceAssembly*
ExpanderVarRef	::=	`ExpanderVarRef` *DottedName*

5.1 Descriptions of Selected AST Constructs

Most of the nodes in the Fortress abstract syntax correspond directly to the program elements described in Chapter 2. However, there are a few special nodes, which we describe in this section.

5.1.1 External syntax

An *ExternalSyntax* tree includes the name of a `static` function and a `SourceAssembly` object (see Section 3.6). The static function referred to must take a `SourceAssembly` object as an argument. This function is called on the given `SourceAssembly` after parsing but before type checking. The result of this call must be a new abstract syntax tree, which is inserted in place of the external syntax treee.

There is a special tree type `ExpanderVarRef` used by syntactic expander functions for the sake of referential transparency. These nodes should not appear in an abstract syntax tree until that tree has been syntactically expanded to eliminate an `ExternalSyntax` node. An `ExpanderVarRef` node is introduced by a syntactic expander to refer to a variable in the scope of the *expander definition* itself. Note: because such a variable must be in scope during the static process of syntactic expansion, it must be a `static` variable.

Because a syntactic expansion does not occur until the reference to the corresponding syntactic expander is actually resolved, the variable referred to in an `ExpanderVarRef` node is known at expansion time. Thus, a fully qualified reference to this variable (including the name of the component in which it resides) is included in the `ExpanderVarRef` tree.

Chapter 6

Concrete Syntax

In this chapter, we describe the concrete syntax of Fortress programs in BNF notation. This syntax is “human-readable” in the sense that it does not describe uses of whitespaces and semicolons exactly. Instead, they are described as follows. Fortress has three different contexts influencing the whitespace-sensitivity of expressions:

statement Expressions appearing in a block other than the last expression are in a statement-like context. Multiple expressions can appear on a line if they are separated (or terminated) by semicolons. If an expression can legally end at the end of a line, it does; if it cannot, it does not. A prefix or infix operator that lacks its last operand prevents an expression from ending. For example,

```
an = expression +
     spanning +
     four +
     lines
a  = one-liner
four() ; on(); one(); line();
```

nested An expression or list of expressions appearing within parentheses or braces is nested. Multiple expressions are separated by commas, and the end of a line does not end an expression. Because of this effect, the meaning of a several lines of code can change if they are wrapped in parentheses. Parentheses can also be used to ensure that a multiline expression is not terminated prematurely without paying special attention to line endings.

```
lhs = rhs
- aSeparateExpression

postProfit(revenue
           - expenses)
```

pasted Fortress has special syntax for matrix pasting. Within square brackets, whitespace-separated expressions are treated (depending on their type) as either matrix elements or submatrices within a row. Because whitespace is the separator, it also ends expressions where possible. In addition, newline-or-semicolon-separated rows are pasted vertically along their columns. Higher-dimensional pasting is expressed with repeated semicolons, but repeated newlines do not have the same effect.

```
Id2a = [ 1 0 ; 0 1 ]
Id2b = [ 1 0 ;
         0 1 ]
Id2c = [ 1 0
         0 1 ]
Cube2 = [ 1 0 ; 0 1 ;; 1 -1 ; 1 1 ]
```

A restricted form of the pasting syntax can also be used on the left hand side of variable declarations to express both declaration and submatrix decomposition.

```
[ top
  bot ] = X
[ left right ] = Y
Z = [ top·left top·right ;
      bot·left bot·right ]
```

Section 3.3 describes matrix unpasting in detail and includes more examples.

Program	::=	*Component*
	|	*Api*
Component	::=	`component` *DottedName Import* Export* Def** `end`
Api	::=	`api` *DottedName Import* Decl** `end`
DottedName	::=	*Id* (`.` *Id*)*
Import	::=	`import` *ImportFrom* `from` *DottedName*
	|	`import` *AliasedNameList*
ImportFrom	::=	`*`
	|	*Id*
	|	{ *IdList* }
IdList	::=	*Id* (`,` *Id*)*
AliasedNameList	::=	*AliasedName* (`,` *AliasedName*)*
AliasedName	::=	*DottedName* [`as` *DottedName*]
Export	::=	`export` *DottedNameList*
DottedNameList	::=	*DottedName* (`,` *DottedName*)*
Def	::=	*VarDef*
	|	*FnDef*
	|	*TraitDef*
	|	*ObjectDef*
	|	*DefOrDecl*
Decl	::=	*VarDecl*
	|	*FnDecl*
	|	*TraitDecl*
	|	*ObjectDecl*
	|	*DefOrDecl*
DefOrDecl	::=	`dim` *Id*
	|	`unit` *Id* `:` *DimType*
	|	*TypeAlias*
TypeAlias	::=	`type` *Id* = *TypeRef*
VarDef	::=	*VarMod* Id* [*IsType*] = *Expr*
	|	*VarMod* Id IsType* `:=` *Expr*
VarDecl	::=	*VarMod* Id IsType*
IsType	::=	`:` *TypeRef*

```
UniversalMod     ::=  static | test
TraitMod         ::=  private | value | UniversalMod
ObjectMod        ::=  TraitMod
FnMod            ::=  atomic | io | private | pure | UniversalMod
MdMod            ::=  getter | setter | FnMod
VarMod           ::=  unit | var | UniversalMod
FldMod           ::=  hidden | settable | var | wrapped | UniversalMod
FnDef            ::=  FnMod* FnHeader = Expr
FnDecl           ::=  FnMod* FnHeader
FnHeader         ::=  Id [TypeParams] ValParams [IsRetType] FnClauses
                  |   OpHeader
OpHeader         ::=  opr Op [TypeParams] ValParams [IsRetType] FnClauses
                  |   opr [TypeParams] ValParams Op [IsRetType] FnClauses
                  |   opr [TypeParams] LeftEncloser Params RightEncloser [IsRetType] FnClauses
FnClauses        ::=  [Throws] [Where] [Contract]
Throws           ::=  throws MayTraitTypes
MayTraitTypes    ::=  {}
                  |   TraitTypes
TraitTypes       ::=  TraitType
                  |   { TraitTypeList }
TraitTypeList    ::=  TraitType (, TraitType)*
Where            ::=  where { WhereClauseList }
WhereClauseList  ::=  WhereClause (, WhereClause)*
WhereClause      ::=  Id Extends
                  |   TypeAlias
Contract         ::=  [Requires] [Ensures] [Invariant]
Requires         ::=  requires Expr+
Ensures          ::=  ensures (Expr+ [provided Expr])+
Invariant        ::=  invariant Expr+
TypeParams       ::=  ⟦TypeParamList ⟧
TypeParamList    ::=  TypeParam (, TypeParam)*
TypeParam        ::=  Id [Extends]
                  |   dim Id
                  |   nat Id
                  |   opr Op
ValParams        ::=  ([Params])
Params           ::=  SimpleParamList
                  |   (SimpleParam,)* VarArgParam
                  |   (SimpleParam,)* KwdParamList
                  |   (SimpleParam,)* VarArgParam, KwdParamList
SimpleParamList  ::=  (SimpleParam,)* SimpleParam
SimpleParam      ::=  Id [IsType]
VarArgParam      ::=  Id : VarArgType
KwdParamList     ::=  (KwdParam,)* KwdParam
KwdParam         ::=  Id [IsType] = Expr
IsRetType        ::=  : RetType
LeftEncloser     ::=  [ | { | (/ | [/ | {/ | </ | <</ | (\ | [\ | {\ | <\ | <<\ | [* | {*
                  |   Encloser
RightEncloser    ::=  ] | } | /) | /] | /} | /> | />> | \) | \] | \} | \> | \>> | *] | *}
                  |   Encloser
Encloser         ::=  | | / | \ | || | // | \\ | ||| | /// | \\\
```

TraitDef	::=	*TraitHeader* (*MdDef* $\mid$ *MdDecl*)* `end`
TraitDecl	::=	*TraitHeader* *MdDecl** `end`
TraitHeader	::=	*TraitMod** `trait` *Id* [*TypeParams*] [*Extends*] [*Excludes*] [*Bounds*] [*Where*]
Extends	::=	`extends` *TraitTypes*
Excludes	::=	`excludes` *TraitTypes*
Bounds	::=	`bounds` *MayTraitTypes*
MdDef	::=	*MdMod** *FnHeader* = *Expr*
MdDecl	::=	[`abstract`] *MdMod** *FnHeader*
ObjectDef	::=	*ObjectHeader* *FldDef** *MdDef** `end`
ObjectDecl	::=	*ObjectHeader* *FldDecl** *MdDecl** `end`
ObjectHeader	::=	*ObjMod** `object` *Id* [*ObjectParams*] [*Traits*] *FnClauses*
ObjectParams	::=	[*TypeParams*] *ObjParams*
Traits	::=	`traits` *TraitTypes*
ObjParams	::=	`()`
	$\mid$	`(` *SimpleObjParamList* `)`
	$\mid$	`(` (*SimpleObjParam*`,`)* `transient` *VarArgParam* `)`
	$\mid$	`(` (*SimpleObjParam*`,`)* *KwdObjParamList* `)`
	$\mid$	`(` (*SimpleObjParam*`,`)* `transient` *VarArgParam*`,` *KwdParamList* `)`
SimpleObjParamList	::=	(*SimpleObjParam*`,`)* *SimpleObjParam*
SimpleObjParam	::=	*FldMod** *SimpleParam*
	$\mid$	`transient` *SimpleParam*
KwdObjParamList	::=	(*KwdObjParam*`,`)* *KwdObjParam*
KwdObjParam	::=	*FldMod** *KwdParam*
	$\mid$	*FldMod** `transient` *KwdParam*
FldDef	::=	*FldMod** *Id* [*IsType*] = *Expr*
	$\mid$	*FldMod** *Id* *IsType* [`:=` *Expr*]
FldDecl	::=	*FldMod** *Id* *IsType*
TypeRef	::=	*SimpleType*
	$\mid$	*ArgType* $\rightarrow$ *RetType* [*Throws*]
SimpleType	::=	*TraitType*
	$\mid$	`()`
	$\mid$	`(` *TypeRef* `)`
	$\mid$	*DimType*
TraitType	::=	*DottedName*
	$\mid$	*DottedName* $[\![$*TypeArgList* $]\!]$
	$\mid$	$\{$ *TypeRef* $\}$
	$\mid$	`[` *TypeRef* $\mapsto$ *TypeRef* `]`
	$\mid$	$\langle$ *TypeRef* $\rangle$
	$\mid$	*TypeRef* `[` *MatrixSize* `]`
	$\mid$	*TypeRef* `[` [*ArraySize*] `]`
ArgType	::=	*SimpleType*
	$\mid$	`(` *TypeRef*`,` *TypeRefList* `)`
	$\mid$	*WithVarArgType*
	$\mid$	`(` (*TypeRef*`,`)* [*VarArgType*`,`] *KwdArgTypeList* `)`
WithVarArgType	::=	*VarArgType*
	$\mid$	`(`*VarArgType*`)`
	$\mid$	`(` *TypeRefList*`,` *VarArgType* `)`
VarArgType	::=	*TypeRef* `...`
KwdArgTypeList	::=	*KwdArgType* (`,` *KwdArgType*)*
KwdArgType	::=	*Id* *IsType*
TypeRefList	::=	*TypeRef* (`,` *TypeRef*)*
RetType	::=	*TypeRef*
	$\mid$	`(` *TypeRef*`,` *TypeRefList* `)`
	$\mid$	*WithVarArgType*

TypeArgList	::=	*TypeArg* (`,` *TypeArg*)*
TypeArg	::=	*TypeRef*
	$\mid$	*NatTypeArg*
	$\mid$	*OprTypeArg*
OprTypeArg	::=	*Id*
	$\mid$	*Op*
ArraySize	::=	*Extent* (, *Extent*)*
Extent	::=	*NatTypeArg*
	$\mid$	*NatTypeArg*#*NatTypeArg*
MatrixSize	::=	*NatTypeArg* ($\times$ *NatTypeArg*)$^+$
DimType	::=	`Unity`
	$\mid$	*DottedName*
	$\mid$	*DimType DimType*
	$\mid$	*DimType* ˆ *NatTypeArg*
	$\mid$	*DimType* $\cdot$ *DimType*
	$\mid$	*DimType* $\times$ *DimType*
	$\mid$	*DimType* `/` *DimType*
	$\mid$	`1/`*DimType*
	$\mid$	`(`*DimType*`)`
NatTypeArg	::=	*Number*
	$\mid$	*Id*
	$\mid$	*NatTypeArg NatTypeArg*
	$\mid$	*NatTypeArg* `+` *NatTypeArg*
	$\mid$	*NatTypeArg* $\cdot$ *NatTypeArg*
	$\mid$	`(`*NatTypeArg*`)`
Expr	::=	*Id*
	$\mid$	*Let*
	$\mid$	*Flow*
	$\mid$	*Value*
	$\mid$	*Comprehension*
	$\mid$	*Expr* [⟦*TypeArgList*⟧] *Expr**
	$\mid$	*PreOp Expr*
	$\mid$	*Expr Op* [*Expr*]
	$\mid$	*Expr* `.` *Id*
	$\mid$	*Expr AssignOp Expr* [`,` *GeneratorList*]
	$\mid$	*ExternalSyntax*
Flow	::=	*Accumulator* `[`*GeneratorList*`]` *Expr*
	$\mid$	`throw` *Expr*
	$\mid$	`atomic` *Do*
	$\mid$	`tryatomic` *Do*
	$\mid$	`exit` [*Id*] [`with` *Expr*]
	$\mid$	*Do*
	$\mid$	`if` *Expr* `then` *Expr*$^+$ (`elif` *Expr* `then` *Expr*$^+$)* [`else` *Expr*$^+$] `end`
	$\mid$	`try` *Expr*$^+$ [`catch` *Id* (*TraitType* $\Rightarrow$ *Expr*$^+$)$^+$]
		[`forbid` *TraitTypes*] [`finally` *Expr**] `end`
	$\mid$	`case` *Expr* [*Id*] `of` (*Expr* $\Rightarrow$ *Expr*$^+$)$^+$ [`else` $\Rightarrow$ *Expr*$^+$] `end`
	$\mid$	`dispatch` *DispatchTypecase*
	$\mid$	`typecase` *DispatchTypecase*
	$\mid$	`spawn` [*Expr*] *Do*
	$\mid$	`for` *GeneratorList Do*
	$\mid$	`while` *Expr Do*
	$\mid$	`label` *Id Expr** `end` *Id*

Value	::=	*Number* | ∞
	|	*String*
	|	*Empty*
	|	`fn` *ValParams* [*IsRetType*] [*Throws*] ⇒ *Expr*
	|	`object` [*Traits*] *FldDef** *MdDef** `end`
	|	*Aggregate*
	|	*Parenthesized*
Empty	::=	{} | `[]` | ⟨ ⟩ | `()`
Aggregate	::=	{ *ExprList* }
	|	`[` *EntryList* `]`
	|	⟨ *ExprList* ⟩
	|	`(` *ExprList* `)`
	|	`[` *Expr* (*Expr* | `;`)* *Expr* `]`
	|	`[` *ExprList* `]`
ExprList	::=	*Expr* (`,` *Expr*)*
EntryList	::=	*Entry* (`,` *Entry*)*
Entry	::=	*Expr* ↦ *Expr*
Parenthesized	::=	`|` *Expr* `|`
	|	*IntervalL* *Expr*`,`*Expr* *IntervalR*
Comprehension	::=	{ *Expr* `|` *Expr** *GeneratorList* }
	|	`[` (*Binding* `|` *Expr** *GeneratorList*)$^+$ `]`
IntervalL	::=	⦗
IntervalR	::=	⦘
Accumulator	::=	∑ | ∏ | ∀ | ∃ | `BIG` *Op*
AssignOp	::=	`:=` | *Op*`=`
GeneratorList	::=	*Generator* (`,` *Generator*)*
Generator	::=	*IdList* ← *Expr*
Do	::=	`do` *Expr** `end`
DispatchTypecase	::=	*DispatchBindings* `in` (*DispatchTypeRefs* ⇒ *Expr*$^+$)$^+$ [`else` ⇒ *Expr*$^+$] `end`
DispatchBindings	::=	*Id*
	|	*Bindings*
DispatchTypeRefs	::=	*TraitType*
	|	`(`*TraitTypeList*`)`
Bindings	::=	*Binding*
	|	`(` *BindingList* `)`
Binding	::=	*Id* = *Expr*
BindingList	::=	*Binding* (`,` *Binding*)*
Let	::=	*L-Vals* = *Expr*
	|	*L-Vals* [`:=` *Expr*]
	|	*L-ValNoTypes* [*IsType*] = *Expr*
	|	*L-ValNoTypes* [*IsType*] [`:=` *Expr*]
	|	*L-ValNoTypes* `:` *TypeRef** = *Expr*
	|	*L-ValNoTypes* `:` *TypeRef** [`:=` *Expr*]
	|	*Id* *ValParams* [*IsRetType*] [*Throws*] = *Expr*
L-Vals	::=	*L-Val*
	|	`(` *L-Val*`,` *L-ValList* `)`
L-Val	::=	[`var`] *Id* [*IsType*]
	|	*Unpasting*
L-ValList	::=	*L-Val* (`,` *L-Val*)*
L-ValNoTypes	::=	*L-ValNoType*
	|	`(` *L-ValNoType*`,` *L-ValNoTypeList* `)`
L-ValNoType	::=	[`var`] *Id*
L-ValNoTypeList	::=	*L-ValNoType* (`,` *L-ValNoType*)*

Unpasting	::=	`[` *L-Elt* (*Paste L-Elt*)* `]`
L-Elt	::=	*Id* [`[` *L-ArraySize* `]`]
	|	*Unpasting*
L-ArraySize	::=	*L-Extent* ($\times$*L-Extent*)*
L-Extent	::=	*Expr*
	|	*Expr*`:`*Expr*
	|	*Expr*#*Expr*
Paste	::=	(*Whitespace* | `;`)$^+$
ExternalSyntax	::=	`syntax` *Id* [*Id Id*] [*Escape*] = *Expr*
	|	`syntax` *Id* (*ParenthesizedId*)$^+$ [*Escape*] = *Expr*
Escape	::=	`escape` *String*
ParenthesizedId	::=	{ *Id* } | `(` *Id* `)` | `[` *Id* `]`

Appendix A

Fortress Calculi

A.1 A Fortress Basic Core Calculus

In this section, we define a basic core calculus for Fortress. We call this calculus *Basic Core Fortress*. Following the precedent set by prior core calculi such as Featherweight Generic Java [12], we have abided by the restriction that all valid Basic Core Fortress programs are valid Fortress programs.

A.1.1 Syntax

A syntax for Basic Core Fortress is provided in Figure A.1. We use the following notational conventions:

- We use $\triangleleft$ for `extends` and `traits`.
- For brevity, we omit separators such as `,` and `;` from Basic Core Fortress.
- $\overrightarrow{\tau}$ is a shorthand for a (possibly empty) sequence $\tau_1, \cdots, \tau_n$.
- Similarly, we abbreviate a sequence of relations $\alpha_1 \triangleleft N_1, \cdots, \alpha_n \triangleleft N_n$ to $\overrightarrow{\alpha \triangleleft N}$
- We use τ_i to denote the ith element of $\overrightarrow{\tau}$.
- For simplicity, we assume that every name (type variables, field names, and parameters) is different and every trait/object declaration declares unique name.
- We prohibit cycles in type hierarchies.

The syntax of Basic Core Fortress allows only a small subset of the Fortress language to be formalized. Basic Core Fortress includes trait and object definitions, method and field invocations, and `self` expressions. The types of Basic Core Fortress include type variables, instantiated traits, instantiated objects, and the distinguished trait `Object`. Note that we syntactically prohibit extending objects. Among other features, Basic Core Fortress does *not* include top-level variable and function definitions, overloading, `excludes` clauses, `bounds` clauses, `where` clauses, object expressions, and function expressions. Basic Core Fortress will be extended to formalize a larger set of Fortress programs in the future.

$$
\begin{array}{llll}
\alpha, \beta & & & \text{type variables} \\
f & & & \text{method name} \\
x & & & \text{field name} \\
T & & & \text{trait name} \\
O & & & \text{object name} \\
\tau, \tau', \tau'' & ::= & \alpha & \\
 & | & N & \\
 & | & O\llbracket\overrightarrow{\tau}\rrbracket & \\
N, M, L & ::= & T\llbracket\overrightarrow{\tau}\rrbracket & \\
 & | & \mathtt{Object} & \\
e & ::= & x & \\
 & | & \mathtt{self} & \\
 & | & O\llbracket\overrightarrow{\tau}\rrbracket\mathtt{(}\overrightarrow{e}\mathtt{)} & \\
 & | & e\mathtt{.}x & \\
 & | & e\mathtt{.}f\llbracket\overrightarrow{\tau}\rrbracket\mathtt{(}\overrightarrow{e}\mathtt{)} & \\
\mathit{fd} & ::= & f\llbracket\overrightarrow{\alpha \lhd N}\rrbracket\mathtt{(}\overrightarrow{x\mathtt{:}\tau}\mathtt{)}\mathtt{:}\tau = e & \\
\mathit{vd} & ::= & x\mathtt{:}\tau = e & \\
\mathit{td} & ::= & \mathtt{trait}\, T\llbracket\overrightarrow{\alpha \lhd N}\rrbracket \lhd \{\overrightarrow{N}\ \mathtt{Object}\}\ \overrightarrow{\mathit{fd}}\ \mathtt{end} & \\
\mathit{od} & ::= & \mathtt{object}\, O\llbracket\overrightarrow{\alpha \lhd N}\rrbracket\mathtt{(}\overrightarrow{x\mathtt{:}\tau}\mathtt{)} \lhd \{\overrightarrow{N}\ \mathtt{Object}\}\ \overrightarrow{\mathit{vd}}\ \overrightarrow{\mathit{fd}}\ \mathtt{end} & \\
d & ::= & \mathit{td} & \text{trait definition} \\
 & | & \mathit{od} & \text{object definition} \\
p & ::= & \overrightarrow{d}\ e & \text{program}
\end{array}
$$

Figure A.1: Syntax of Basic Core Fortress

A.1.2 Dynamic semantics

A dynamic semantics for Basic Core Fortress is provided in Figure A.2. This semantics has been mechanized via the PLT Redex tool [15]. It therefore follows the style of explicit evaluation contexts and redexes. The Basic Core Fortress dynamic semantics consists of two evaluations rules: one for field access and another for method invocation. For simplicity, we use ‘ _ ’ to denote some parts of the syntax that do not have key roles in a rule.

A.1.3 Static semantics

A static semantics for Basic Core Fortress is provided in Figures A.3, A.4, and A.5. The Basic Core Fortress static semantics is based on the static semantics of Featherweight Generic Java (FGJ) [12]. The major difference is the division of classes into traits and objects. Both trait and object definitions include method definitions but only object definitions include field definitions. With traits, Basic Core Fortress supports multiple inheritance. However, due to the similarity of traits and objects, many of the rules in the Basic Core Fortress dynamic and static semantics combine the two cases. Note that Basic Core Fortress allows parametric polymorphism, subtype polymorphism, and overriding in much the same way that FGJ does.

A.1.4 Type soundness proof

We prove the type soundness of Core Fortress using the standard technique of proving a progress theorem and a subject reduction theorem. The proof follows.

Lemma A.1.1 (Method Bodies). *If* $\text{mtype}_p(f, O\llbracket\overrightarrow{\tau}\rrbracket) = \{\llbracket\overrightarrow{\alpha \lhd N}\rrbracket\ \overrightarrow{\tau'} \to \tau'_0\}$ *then there exists* e *such that*

Evaluation contexts and redexes

$$\begin{array}{lcl}
v & ::= & O[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\texttt{)} \\
E & ::= & \texttt{[]} \\
 & | & O[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\,E\,\overrightarrow{e}\texttt{)} \\
 & | & E\texttt{.}x \\
 & | & E\texttt{.}f[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{e}\texttt{)} \\
 & | & O[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\texttt{)}\texttt{.}f[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\,E\,\overrightarrow{e}\texttt{)} \\
R & ::= & O[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\texttt{)}\texttt{.}x \\
 & | & O[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\texttt{)}\texttt{.}f[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\texttt{)} \\
C & ::= & T \\
 & | & O
\end{array}$$

Evaluation rules: $\boxed{p \vdash E\texttt{[}R\texttt{]} \longrightarrow E\texttt{[}e\texttt{]}}$

[R-FIELD] $$\frac{\texttt{object}\,O[\![\overrightarrow{\alpha \triangleleft \_}]\!]\texttt{(}\overrightarrow{x'\texttt{:}\ \_}\texttt{)}\ \_\ \overrightarrow{x\texttt{:}\ \_ \texttt{=} e}\ \_ \in p}{p \vdash E\texttt{[}O[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\texttt{)}\texttt{.}x_i\texttt{]} \longrightarrow E\texttt{[}[\overrightarrow{\tau/\alpha}][\overrightarrow{v/x'}]e_i\texttt{]}}$$

[R-METHOD] $$\frac{\texttt{object}\,O\ \_\ \texttt{(}\overrightarrow{x'\texttt{:}\ \_}\texttt{)}\ \_ \in p \qquad mbody_p(f[\![\overrightarrow{\tau'}]\!], O[\![\overrightarrow{\tau}]\!]) = \{\texttt{(}\overrightarrow{x}\texttt{)} \rightarrow e\}}{p \vdash E\texttt{[}O[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\texttt{)}\texttt{.}f[\![\overrightarrow{\tau'}]\!]\texttt{(}\overrightarrow{v'}\texttt{)}\texttt{]} \longrightarrow E\texttt{[}[\overrightarrow{v/x'}][O[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\texttt{)}/\texttt{self}][\overrightarrow{v'/x}]e\texttt{]}}$$

Method body lookup: $\boxed{mbody_p(f[\![\overrightarrow{\tau}]\!], \tau) = \{\texttt{(}\overrightarrow{x}\texttt{)} \rightarrow e\}}$

[MB-BOTH] $$\frac{\_\ C[\![\overrightarrow{\alpha \triangleleft \_}]\!]\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f[\![\overrightarrow{\alpha' \triangleleft \_}]\!]\texttt{(}\overrightarrow{x'\texttt{:}\ \_}\texttt{)}\ \_\ \texttt{=}\, e \in \{\overrightarrow{fd}\}}{mbody_p(f[\![\overrightarrow{\tau'}]\!], C[\![\overrightarrow{\tau}]\!]) = \{[\overrightarrow{\tau/\alpha}][\overrightarrow{\tau'/\alpha'}]\texttt{(}\overrightarrow{x'}\texttt{)} \rightarrow e\}}$$

[MB-SUPER] $$\frac{\_\ C[\![\overrightarrow{\alpha \triangleleft \_}]\!]\ \_\ \triangleleft \{\overrightarrow{N}\}\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f \not\in \{\overrightarrow{Fname(fd)}\}}{mbody_p(f[\![\overrightarrow{\tau'}]\!], C[\![\overrightarrow{\tau}]\!]) = \bigcup_{N_i \in \{\overrightarrow{N}\}} mbody_p(f[\![\overrightarrow{\tau'}]\!], [\overrightarrow{\tau/\alpha}]N_i)}$$

[MB-OBJ] $$mbody_p(f[\![\overrightarrow{\tau}]\!], \texttt{Object}) = \emptyset$$

Function/method name lookup: $\boxed{Fname(fd) = f}$

$Fname(f[\![\overrightarrow{\alpha \triangleleft N}]\!]\texttt{(}\overrightarrow{x\texttt{:}\tau}\texttt{)}\texttt{:}\tau \texttt{=} e) = f$

Figure A.2: Dynamic Semantics of Basic Core Fortress

Environments

$$\begin{array}{lll} \Delta & ::= & \overrightarrow{\alpha <: N} \\ \Gamma & ::= & \overrightarrow{x : \tau} \end{array}$$

Program typing: $\boxed{\vdash p : \tau}$

$$[\text{T-PROGRAM}] \quad \frac{p = \overrightarrow{d\,\texttt{;}}\ e \qquad p \vdash \overrightarrow{d} \text{ ok} \qquad p; \emptyset; \emptyset \vdash e : \tau}{\vdash \overrightarrow{d\,\texttt{;}}\ e : \tau}$$

Definition typing: $\boxed{p \vdash d \text{ ok}}$

$$[\text{T-TRAITDEF}] \quad \frac{\begin{array}{c}\Delta = \overrightarrow{\alpha <: N} \qquad p; \Delta \vdash \overrightarrow{N} \text{ ok} \qquad p; \Delta \vdash \overrightarrow{M} \text{ ok} \qquad p; \Delta; \emptyset; T \vdash \overrightarrow{fd} \text{ ok} \\ \forall f \in \{\overrightarrow{Fname(fd)}\}.|owner_p(f, T)| \leq 1\end{array}}{p \vdash \texttt{trait}\ T\llbracket\overrightarrow{\alpha \triangleleft N}\rrbracket \triangleleft \{\overrightarrow{M}\}\ \overrightarrow{fd}\ \texttt{end} \text{ ok}}$$

$$[\text{T-OBJECTDEF}] \quad \frac{\begin{array}{c}\Delta = \overrightarrow{\alpha <: N} \qquad p; \Delta \vdash \overrightarrow{N} \text{ ok} \qquad p; \Delta \vdash \overrightarrow{\tau} \text{ ok} \qquad p; \Delta \vdash \overrightarrow{M} \text{ ok} \\ p; \Delta; \overrightarrow{x : \tau} \vdash \overrightarrow{vd} \text{ ok} \qquad p; \Delta; \overrightarrow{x : \tau}; O \vdash \overrightarrow{fd} \text{ ok} \\ \forall f \in \{\overrightarrow{Fname(fd)}\}.|owner_p(f, O)| \leq 1\end{array}}{p \vdash \texttt{object}\ O\llbracket\overrightarrow{\alpha \triangleleft N}\rrbracket\texttt{(}\overrightarrow{x\texttt{:}\tau}\texttt{)} \triangleleft \{\overrightarrow{M}\}\ \overrightarrow{vd}\ \overrightarrow{fd}\ \texttt{end} \text{ ok}}$$

Field typing: $\boxed{p; \Delta; \Gamma \vdash vd \text{ ok}}$

$$[\text{T-FIELDDEF}] \quad \frac{p; \Delta \vdash \tau \text{ ok} \qquad p; \Delta; \Gamma \vdash e : \tau' \qquad p; \Delta \vdash \tau' <: \tau}{p; \Delta; \Gamma \vdash x \texttt{:}\ \tau \texttt{ = } e \text{ ok}}$$

Method typing: $\boxed{p; \Delta; \Gamma; C \vdash fd \text{ ok}}$

$$[\text{T-METHODDEF}] \quad \frac{\begin{array}{c}\_\ C\llbracket\overrightarrow{\alpha' \triangleleft \_}\rrbracket\ \_\ \triangleleft \{\overrightarrow{M}\}\ \_ \in p \qquad override_p(f, \{\overrightarrow{M}\}, \llbracket\overrightarrow{\alpha \triangleleft N}\rrbracket\ \overrightarrow{\tau} \to \tau_0) \\ \Delta' = \Delta\ \overrightarrow{\alpha <: N} \qquad p; \Delta' \vdash \overrightarrow{N} \text{ ok} \qquad p; \Delta' \vdash \overrightarrow{\tau} \text{ ok} \qquad p; \Delta' \vdash \tau_0 \text{ ok} \\ p; \Delta'; \Gamma\ \texttt{self} : C\llbracket\overrightarrow{\alpha'\texttt{,}}\rrbracket\ \overrightarrow{x : \tau} \vdash e : \tau' \qquad p; \Delta' \vdash \tau' <: \tau_0\end{array}}{p; \Delta; \Gamma; C \vdash f\llbracket\overrightarrow{\alpha \triangleleft N}\rrbracket\texttt{(}\overrightarrow{x\texttt{:}\tau}\texttt{)}\texttt{:}\tau_0 \texttt{ = } e \text{ ok}}$$

Method overriding: $\boxed{override_p(f, \{\overrightarrow{N}\}, \llbracket\overrightarrow{\alpha \triangleleft N}\rrbracket\ \overrightarrow{\tau} \to \tau)}$

$$[\text{OVERRIDE}] \quad \frac{\begin{array}{c}\bigcup_{L_i \in \{\overrightarrow{L}\}} mtype_p(f, L_i) = \{\llbracket\overrightarrow{\beta \triangleleft M}\rrbracket\ \overrightarrow{\tau'} \to \tau_0'\} \\ \text{implies} \\ \overrightarrow{N} = [\overrightarrow{\alpha/\beta}]\overrightarrow{M} \qquad \overrightarrow{\tau} = [\overrightarrow{\alpha/\beta}]\overrightarrow{\tau'} \qquad p; \overrightarrow{\alpha <: N} \vdash \tau_0 <: [\overrightarrow{\alpha/\beta}]\tau_0'\end{array}}{override_p(f, \{\overrightarrow{L}\}, \llbracket\overrightarrow{\alpha \triangleleft N}\rrbracket\ \overrightarrow{\tau} \to \tau_0)}$$

Figure A.3: Static Semantics (I)

Expression typing: $\boxed{p; \Delta; \Gamma \vdash e : \tau}$

[T-VAR] $$p; \Delta; \Gamma \vdash x : \Gamma(x)$$

[T-SELF] $$p; \Delta; \Gamma \vdash \texttt{self} : \Gamma(\texttt{self})$$

[T-OBJECT] $$\frac{\texttt{object}\ O\ \_\ \texttt{(}\ \overrightarrow{\_\texttt{:}\tau'}\ \texttt{)}\ \_ \in p \qquad p; \Delta \vdash O[\![\overrightarrow{\tau}]\!]\ \text{ok} \qquad p; \Delta; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau''} \qquad p; \Delta \vdash \overrightarrow{\tau''}\ <:\ \overrightarrow{\tau'}}{p; \Delta; \Gamma \vdash O[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{e}\texttt{)} : O[\![\overrightarrow{\tau}]\!]}$$

[T-FIELD] $$\frac{p; \Delta; \Gamma \vdash e_0 : \tau_0 \qquad \mathit{bound}_\Delta(\tau_0) = O[\![\overrightarrow{\tau}]\!] \qquad \texttt{object}\ O[\![\overrightarrow{\alpha \triangleleft \_}]\!]\ \_\ \overrightarrow{x\texttt{:}\tau'\texttt{=}e}\ \_ \in p}{p; \Delta; \Gamma \vdash e_0\texttt{.}x_i : [\overrightarrow{\tau/\alpha}]\tau'_i}$$

[T-METHOD] $$\frac{\begin{array}{c} p; \Delta; \Gamma \vdash e_0 : \tau_0 \qquad \mathit{mtype}_p(f, \mathit{bound}_\Delta(\tau_0)) = \{[\![\overrightarrow{\alpha \triangleleft N}]\!]\ \overrightarrow{\tau'} \to \tau'_0\} \\ p; \Delta \vdash \overrightarrow{\tau}\ \text{ok} \qquad p; \Delta \vdash \overrightarrow{\tau}\ <:\ [\overrightarrow{\tau/\alpha}]\overrightarrow{N} \\ p; \Delta; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau''} \qquad p; \Delta \vdash \overrightarrow{\tau''}\ <:\ [\overrightarrow{\tau/\alpha}]\overrightarrow{\tau'} \end{array}}{p; \Delta; \Gamma \vdash e_0\texttt{.}f[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{e}\texttt{)} : [\overrightarrow{\tau/\alpha}]\tau'_0}$$

Subtyping: $\boxed{p; \Delta \vdash \tau\ <:\ \tau}$

[S-REFL] $$p; \Delta \vdash \tau\ <:\ \tau$$

[S-TRANS] $$\frac{p; \Delta \vdash \tau_1\ <:\ \tau_2 \qquad p; \Delta \vdash \tau_2\ <:\ \tau_3}{p; \Delta \vdash \tau_1\ <:\ \tau_3}$$

[S-VAR] $$p; \Delta \vdash \alpha\ <:\ \Delta(\alpha)$$

[S-BOTH] $$\frac{\_\ C[\![\overrightarrow{\alpha \triangleleft \_}]\!]\ \_\ \triangleleft\ \{\overrightarrow{N}\}\ \_ \in p}{p; \Delta \vdash C[\![\overrightarrow{\tau}]\!]\ <:\ [\overrightarrow{\tau/\alpha}]N_i}$$

Well-formed types: $\boxed{p; \Delta \vdash \tau\ \text{ok}}$

[W-OBJ] $$p; \Delta \vdash \texttt{Object}\ \text{ok}$$

[W-VAR] $$\frac{\alpha \in \mathit{dom}(\Delta)}{p; \Delta \vdash \alpha\ \text{ok}}$$

[W-BOTH] $$\frac{\_\ C[\![\overrightarrow{\alpha \triangleleft N}]\!]\ \_ \in p \qquad p; \Delta \vdash \overrightarrow{\tau}\ \text{ok} \qquad p; \Delta \vdash \overrightarrow{\tau}\ <:\ [\overrightarrow{\tau/\alpha}]\overrightarrow{N}}{p; \Delta \vdash C[\![\overrightarrow{\tau}]\!]\ \text{ok}}$$

Figure A.4: Static Semantics (II)

Method type lookup: $\boxed{\mathit{mtype}_p(f,\tau) = \{\llbracket\overrightarrow{\alpha \triangleleft N}\rrbracket\ \overrightarrow{\tau} \rightarrow \tau\}}$

$$[\textsc{Mt-Both}] \qquad \frac{\_\ C\llbracket\overrightarrow{\alpha \triangleleft \_}\rrbracket\ \_\ \overrightarrow{\mathit{fd}}\ \_ \in p \qquad f\llbracket\overrightarrow{\beta \triangleleft M}\rrbracket(\overrightarrow{\_ : \tau'}) : \tau'_0 = e \in \{\overrightarrow{\mathit{fd}}\}}{\mathit{mtype}_p(f, C\llbracket\overrightarrow{\tau}\rrbracket) = \{[\overrightarrow{\tau/\alpha}]\llbracket\overrightarrow{\beta \triangleleft M}\rrbracket\ \overrightarrow{\tau'} \rightarrow \tau'_0\}}$$

$$[\textsc{Mt-Super}] \qquad \frac{\_\ C\llbracket\overrightarrow{\alpha \triangleleft \_}\rrbracket\ \_\ \triangleleft \{\overrightarrow{N}\}\ \_\ \overrightarrow{\mathit{fd}}\ \_ \in p \qquad f \notin \{\overrightarrow{\mathit{Fname}(\mathit{fd})}\}}{\mathit{mtype}_p(f, C\llbracket\overrightarrow{\tau}\rrbracket) = \underset{N_i \in [\overrightarrow{\tau/\alpha}]\{\overrightarrow{N}\}}{\qquad} \mathit{mtype}_p(f, N_i)}$$

$$[\textsc{Mt-Obj}] \qquad \mathit{mtype}_p(f, \texttt{Object}) = \emptyset$$

Owner lookup: $\boxed{\mathit{owner}_p(f,C) = \{C\}}$

$$[\textsc{O-Both}] \qquad \frac{\_\ C\llbracket\overrightarrow{\alpha \triangleleft \_}\rrbracket\ \_\ \overrightarrow{\mathit{fd}}\ \_ \in p \qquad f \in \{\overrightarrow{\mathit{Fname}(\mathit{fd})}\}}{\mathit{owner}_p(f, C) = \{C\}}$$

$$[\textsc{O-Super}] \qquad \frac{\_\ C\llbracket\overrightarrow{\alpha \triangleleft \_}\rrbracket\ \_\ \triangleleft \{\overrightarrow{N}\}\ \_\ \overrightarrow{\mathit{fd}}\ \_ \in p \qquad f \notin \{\overrightarrow{\mathit{Fname}(\mathit{fd})}\}}{\mathit{owner}_p(f, C) = \underset{T\llbracket\overrightarrow{\tau}\rrbracket \in \{\overrightarrow{N}\}}{\qquad} \mathit{owner}_p(f, T)}$$

Bound of type: $\boxed{\mathit{bound}_\Delta(\tau) = \tau}$

$$\begin{array}{lcl} \mathit{bound}_\Delta(\alpha) & = & \Delta(\alpha) \\ \mathit{bound}_\Delta(N) & = & N \\ \mathit{bound}_\Delta(O\llbracket\overrightarrow{\tau}\rrbracket) & = & O\llbracket\overrightarrow{\tau}\rrbracket \end{array}$$

Figure A.5: Static Semantics (III)

$\text{mbody}_p(f[\![\overrightarrow{\tau''}]\!], O[\![\overrightarrow{\tau}]\!]) = \{\texttt{(}\ \overrightarrow{x}\ \texttt{)} \rightarrow e\}$.

Proof. Trivial induction over the derivation of $mtype_p(f, O[\![\overrightarrow{\tau}]\!]) = \{[\![\overrightarrow{\alpha \triangleleft N}]\!]\ \overrightarrow{\tau'} \rightarrow \tau'_0\}$. □

Theorem 1 (Progress). *If $p;\Delta;\Gamma \vdash e : \tau$ then either e is a value or $p \vdash e \longrightarrow e'$.*

Proof. The proof is by case analysis on the current redex in e (in the case that e is not a value).

Case $O[\![\overrightarrow{\tau}]\!]\texttt{(}\ \overrightarrow{v}\ \texttt{)}\texttt{.}x_i$: By the well-typedness of e, we know $p;\Delta;\Gamma \vdash O[\![\overrightarrow{\tau}]\!]\texttt{(}\ \overrightarrow{v}\ \texttt{)}\texttt{.}x_i : \tau''_i$ for some τ''_i. By the typing rule [T-FIELD], we know $\texttt{object}\ O[\![\overrightarrow{\alpha \triangleleft \_}]\!]\ \_\ \overrightarrow{x\texttt{:}\tau'\texttt{=}e}\ \_ \in p$. Therefore the evaluation rule [R-FIELD] can be applied.

Case $O[\![\overrightarrow{\tau}]\!]\texttt{(}\ \overrightarrow{v}\ \texttt{)}\texttt{.}f[\![\overrightarrow{\tau'}]\!]\texttt{(}\ \overrightarrow{v'}\ \texttt{)}$: By the well-typedness of e, we know $p;\Delta;\Gamma \vdash O[\![\overrightarrow{\tau}]\!]\texttt{(}\ \overrightarrow{v}\ \texttt{)}\texttt{.}f[\![\overrightarrow{\tau'}]\!]\texttt{(}\ \overrightarrow{v'}\ \texttt{)} : [\overrightarrow{\tau'/\alpha}]\tau$ for some τ. By the typing rule [T-METHOD], we know $mtype_p(f, O[\![\overrightarrow{\tau}]\!]) = [\![\overrightarrow{\alpha \triangleleft N}]\!]\overrightarrow{\tau''} \rightarrow \tau$ for some $\overrightarrow{N}$, $\overrightarrow{\tau''}$. By Lemma A.1.1, we have $mbody_p(f[\![\overrightarrow{\tau'}]\!], O[\![\overrightarrow{\tau}]\!]) = \texttt{(}\ \overrightarrow{x}\ \texttt{)} \rightarrow e$ for some e. The size of $\overrightarrow{v'}$ and $\overrightarrow{x}$ are equal because both equal the size of $\overrightarrow{\tau''}$ by [T-METHOD]. A similar argument holds for $\overrightarrow{v}$. So the evaluation rule [R-METHOD] can be applied. □

Lemma A.1.2 (Replacement). *If $p;\Delta;\Gamma \vdash E\texttt{[}e\texttt{]} : \tau_0$ and $p;\Delta;\Gamma \vdash e : \tau_1$ and $p;\Delta;\Gamma \vdash e' : \tau'_1$ where $p;\Delta \vdash \tau'_1 <: \tau_1$ then $p;\Delta;\Gamma \vdash E\texttt{[}e'\texttt{]} : \tau'_0$ where $p;\Delta \vdash \tau'_0 <: \tau_0$.*

Proof. This proof is a replacement argument in the typing derivation. □

Lemma A.1.3 (Weakening). *Suppose $p;\Delta\ \overrightarrow{\alpha <: N} \vdash \overrightarrow{N}$ ok and $p;\Delta \vdash \tau_0$ ok.*

1. *If $p;\Delta \vdash \tau <: \tau'$ then $p;\Delta\ \overrightarrow{\alpha <: N} \vdash \tau <: \tau'$.*
2. *If $p;\Delta \vdash \tau$ ok then $p;\Delta\ \overrightarrow{\alpha <: N} \vdash \tau$ ok.*
3. *If $p;\Delta;\Gamma \vdash e : \tau$ then $p;\Delta;\Gamma\ x : \tau_0 \vdash e : \tau$ and $p;\Delta\ \overrightarrow{\alpha <: N};\Gamma \vdash e : \tau$.*

Proof. Each of them is proved by straightforward induction on the derivation of $p;\Delta \vdash \tau <: \tau'$ and $p;\Delta \vdash \tau$ ok and $p;\Delta;\Gamma \vdash e : \tau$. □

Lemma A.1.4 (Type Substitution Preserves Subtyping). *If $p;\Delta_1\ \overrightarrow{\alpha <: N}\ \Delta_2 \vdash \tau <: \tau'$ and $p;\Delta_1 \vdash \overrightarrow{\tau} <: [\overrightarrow{\tau/\alpha}]\overrightarrow{N}$ with $p;\Delta_1 \vdash \overrightarrow{\tau}$ ok and none of $\overrightarrow{\alpha}$ appear in Δ_1 then $p;\Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2 \vdash [\overrightarrow{\tau/\alpha}]\tau <: [\overrightarrow{\tau/\alpha}]\tau'$.*

Proof. By induction on the derivation of $p;\Delta_1\ \overrightarrow{\alpha <: N}\ \Delta_2 \vdash \tau <: \tau'$.

Case [S-REFL]: Trivial.

Case [S-TRANS], [S-BOTH]: Easy.

Case [S-VAR]: $\tau = \alpha \quad \tau' = (\Delta_1\ \overrightarrow{\alpha <: N}\ \Delta_2)(\alpha)$
If $\alpha \in dom\texttt{(}\Delta_1\texttt{)} \cup dom\texttt{(}\Delta_2\texttt{)}$, then the conclusion is immediate. Otherwise, if $\alpha = \alpha_i$, then, by assumption, we have $p;\Delta_1 \vdash \tau_i <: [\overrightarrow{\tau/\alpha}]N_i$. Lastly, Lemma A.1.3 gives us the desired result. □

Lemma A.1.5 (Type Substitution Preserves Well-Formedness). *If* $p; \Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2 \vdash \tau$ *ok and* $p; \Delta_1 \vdash \overrightarrow{\tau}\ <:\ [\overrightarrow{\tau/\alpha}]\overrightarrow{N}$ *with* $p; \Delta_1 \vdash \overrightarrow{\tau}$ *ok and none of* $\overrightarrow{\alpha}$ *appearing in* Δ_1*, then* $p; \Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2 \vdash [\overrightarrow{\tau/\alpha}]\tau$ *ok.*

Proof. By straightforward induction on the derivation of $p; \Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2 \vdash \tau$ ok. ☐

Lemma A.1.6. *Suppose* $p; \Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2 \vdash \tau$ *ok and* $p; \Delta_1 \vdash \overrightarrow{\tau'}\ <:\ [\overrightarrow{\tau'/\alpha}]\overrightarrow{N}$ *with* $p; \Delta_1 \vdash \overrightarrow{\tau'}$ *ok and none of* $\overrightarrow{\alpha}$ *appear in* Δ_1*. Then,* $p; \Delta_1\ [\overrightarrow{\tau'/\alpha}]\Delta_2 \vdash \text{bound}_{\Delta_1\ [\overrightarrow{\tau'/\alpha}]\Delta_2}([\overrightarrow{\tau'/\alpha}]\tau)\ <:\ [\overrightarrow{\tau'/\alpha}]\text{bound}_{\Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2}(\tau)$.

Proof. The case where τ is a nonvariable type is trivial. If $\tau = \alpha \in dom(\Delta_1) \cup dom(\Delta_2)$ then the proof is easy. If $\tau = \alpha_i$ then $bound_{\Delta_1\ [\overrightarrow{\tau'/\alpha}]\Delta_2}([\overrightarrow{\tau'/\alpha}]\tau) = \tau'_i$ and $[\overrightarrow{\tau'/\alpha}]bound_{\Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2}(\tau) = [\overrightarrow{\tau'/\alpha}]N_i$. Lemma A.1.3 finishes the proof. ☐

Lemma A.1.7. *If* p *is well-typed and* $p; \Delta \vdash \tau$ *ok and* $\text{mtype}_p(f, \text{bound}_\Delta(\tau)) = [\![\overrightarrow{\alpha \triangleleft N}]\!]\overrightarrow{\tau'} \rightarrow \tau_0$*, then for some* τ' *such that* $p; \Delta \vdash \tau'\ <:\ \tau$ *and* $p; \Delta \vdash \tau'$ *ok, we have* $\text{mtype}_p(f, \text{bound}_\Delta(\tau')) = [\![\overrightarrow{\alpha \triangleleft N}]\!]\overrightarrow{\tau'} \rightarrow \tau'_0$ *and* $p; \Delta\ \overrightarrow{\alpha\ <:\ N} \vdash \tau'_0\ <:\ \tau_0$.

Proof. By induction on the derivation of $p; \Delta \vdash \tau'\ <:\ \tau$.

Case [S-REFL]: Trivial.

Case [S-VAR]: Trivial because $bound_\Delta(\tau) = bound_\Delta(\tau')$.

Case [S-TRANS]: Easy.

Case [S-BOTH]: $\tau' = C[\![\overrightarrow{\tau}]\!]$ $\quad \tau = [\overrightarrow{\tau/\alpha}]M_i$ $\quad$ where $\_\ C[\![\overrightarrow{\alpha \triangleleft \_}]\!]\ \_\ \triangleleft \{\overrightarrow{M}\}\ \_ \in p$.

Subcase $f \notin \{\overrightarrow{Fname(fd)}\}$: Then $mtype_p(f, C[\![\overrightarrow{\tau}]\!]) = mtype_p(f, [\overrightarrow{\tau/\alpha}]\{\overrightarrow{M}\}) = mtype_p(f, [\overrightarrow{\tau/\alpha}]M_i) = [\![\overrightarrow{\alpha \triangleleft N}]\!]\overrightarrow{\tau'} \rightarrow \tau_0$.

Subcase $f[\![\overrightarrow{\beta \triangleleft L}]\!](\overrightarrow{x : \tau''}) : \tau'_0 = e \in \{\overrightarrow{fd}\}$: By induction on the derivation of $mtype_p(f, \tau)$, we know $mtype_p(f, \tau) = [\overrightarrow{\tau/\alpha}]([\![\overrightarrow{\beta \triangleleft L'}]\!]\ \overrightarrow{\tau'''} \rightarrow \tau''_0)$ where $mtype_p(f, M_i) = [\![\overrightarrow{\beta \triangleleft L'}]\!]\ \overrightarrow{\tau'''} \rightarrow \tau''_0$ and $[\overrightarrow{\tau/\alpha}]\ \tau''_0 = \tau_0$. By [T-METHODDEF] and [OVERRIDE], we have $p; \overrightarrow{\alpha\ <:\ N}\ \overrightarrow{\beta\ <:\ L}, \vdash \tau'_0\ <:\ \tau''_0$. By Lemmas A.1.4 and A.1.3, we have

$$p; \Delta\ \overrightarrow{\beta\ <:\ L}, \vdash [\overrightarrow{\tau/\alpha}]\overrightarrow{\tau'_0}\ <:\ [\overrightarrow{\tau/\alpha}]\tau''_0.$$

Since $mtype_p(f, bound_\Delta(\tau')) = mtype_p(f, \tau') = [\overrightarrow{\tau/\alpha}]([\![\overrightarrow{\beta \triangleleft L}]\!]\ \overrightarrow{\tau''} \rightarrow \tau'_0)$, we are done. ☐

Lemma A.1.8 (Term Substitution Preserves Typing). *If* p *is well-typed and* $p; \Delta; \Gamma\ \overrightarrow{x : \tau} \vdash e : \tau$ *and* $p; \Delta; \Gamma \vdash \overrightarrow{e'} : \overrightarrow{\tau'}$ *and* $p; \Delta \vdash \overrightarrow{\tau'}\ <:\ \overrightarrow{\tau}$*, then* $p; \Delta; \Gamma \vdash [\overrightarrow{e'/x}]e : \tau'$ *for some* τ' *such that* $p; \Delta \vdash \tau'\ <:\ \tau$.

Proof. By induction on the derivation of $p; \Delta; \Gamma\ \overrightarrow{x : \tau} \vdash e : \tau$.

Case [T-SELF]: Trivial.

Case [T-OBJECT]: Easy.

Case [T-VAR] $p;\Delta;\Gamma\ \overrightarrow{x:\tau}\vdash x:\Gamma(x)$:
If $x\in dom(\Gamma)$ then the result is immediate. Otherwise $x=x_i$ and $\tau=\tau_i$. Then $\tau'=\tau_i'$.

Case [T-FIELD] $$\frac{p;\Delta;\Gamma\vdash e_0:\tau_0' \qquad bound_\Delta(\tau_0')=O\llbracket\overrightarrow{\tau'''}\rrbracket \qquad \texttt{object}\,O\llbracket\overrightarrow{\alpha\ \triangleleft\ \_}\rrbracket\ \_\ \overrightarrow{x':\tau'=e;}\ \_\in p}{p;\Delta;\Gamma\ \overrightarrow{x:\tau}\vdash e_0\,\texttt{.}\,x_i':[\overrightarrow{\tau'''/\alpha}]\tau_i'}$$:

By the induction hypothesis we have $p;\Delta;\Gamma\vdash[\overrightarrow{e'/x}]e_0:\tau_0''$ and $p;\Delta\vdash\tau_0''\ <:\ \tau_0'$. By inspection of the definition of *bound*, notice that $\tau_0'=O\llbracket\overrightarrow{\tau'''}\rrbracket$. Then by inspection of the subtyping rules, $\tau_0''=O\llbracket\overrightarrow{\tau'''}\rrbracket$ and the result is immediate.

Case [T-METHOD] $$\frac{\begin{array}{c}p;\Delta;\Gamma\ \overrightarrow{x:\tau}\vdash e_0:\tau_0' \qquad mtype_p(f,bound_\Delta(\tau_0'))=\{\overrightarrow{\tau''}\to\tau_0\}\\ p;\Delta\vdash\overrightarrow{\tau'''}\text{ ok} \qquad p;\Delta\vdash\overrightarrow{\tau'''}\ <:\ [\overrightarrow{\tau'''/\alpha}]\overrightarrow{N}\\ p;\Delta;\Gamma\ \overrightarrow{x:\tau}\vdash\overrightarrow{e}:\overrightarrow{\tau''''} \qquad p;\Delta\vdash\overrightarrow{\tau''''}\ <:\ [\overrightarrow{\tau'''/\alpha}]\overrightarrow{\tau''}\end{array}}{p;\Delta;\Gamma\ \overrightarrow{x:\tau}\vdash e_0\,\texttt{.}\,f\llbracket\overrightarrow{\tau'''}\rrbracket(\overrightarrow{e}):[\overrightarrow{\tau'''/\alpha}]\tau_0}$$:

By the induction hypothesis we have

$$\begin{array}{ll}p;\Delta;\Gamma\vdash[\overrightarrow{e'/x}]e_0:\tau_0'' & \qquad p;\Delta;\Gamma\vdash[\overrightarrow{e'/x}]\overrightarrow{e}:\overrightarrow{\tau'''''}\\ p;\Delta\vdash\tau_0''\ <:\ \tau_0' & \qquad p;\Delta\vdash\overrightarrow{\tau'''''}\ <:\ \overrightarrow{\tau''''}\end{array}$$

By Lemma A.1.7, we have $mtype_p(f,bound_\Delta(\tau_0''))=\llbracket\overrightarrow{\alpha\ \triangleleft\ N}\rrbracket\overrightarrow{\tau''}\to\tau_0'''$ where $p;\Delta\ \overrightarrow{\alpha\ <:\ N}\vdash\tau_0'''\ <:\ \tau_0$. By Lemma A.1.4, we have $p;\Delta\vdash[\overrightarrow{\tau'''/\alpha}]\tau_0'''\ <:\ [\overrightarrow{\tau'''/\alpha}]\tau_0$. By [T-METHOD], we have $p;\Delta;\Gamma\vdash[e'/x](e_0\,\texttt{.}\,f\llbracket\overrightarrow{\tau''',}\rrbracket(\overrightarrow{e})):[\overrightarrow{\tau'''/\alpha}]\tau_0'''$. □

Lemma A.1.9 (Type Substitution Preserves Typing). *If p is well-typed and $p;\Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2;\Gamma\vdash e:\tau$ and $p;\Delta_1\vdash\overrightarrow{\tau}\ <:\ [\overrightarrow{\tau/\alpha}]\overrightarrow{N}$ where $p;\Delta_1\vdash\overrightarrow{\tau}$ ok and none of $\overrightarrow{\alpha}$ appear in Δ_1, then $p;\Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2;[\overrightarrow{\tau/\alpha}]\Gamma\vdash[\overrightarrow{\tau/\alpha}]e:\tau'$ for some τ'such that $p;\Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2\vdash\tau'\ <:\ [\overrightarrow{\tau/\alpha}]\tau$.*

Proof. By induction on the typing derivation, $p;\Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2;\Gamma\vdash e:\tau$ with case analysis on the last rule applied.

Case [T-VAR], [T-SELF]: Trivial.

Case [T-FIELD] $$\frac{\begin{array}{c}p;\Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2;\Gamma\vdash e_0:\tau_0' \qquad bound_{\Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2}(\tau_0')=O\llbracket\overrightarrow{\tau''}\rrbracket\\ \texttt{object}\,O\llbracket\overrightarrow{\beta\ \triangleleft\ M}\rrbracket(\overrightarrow{x':\tau'})\ \triangleleft\ \{\overrightarrow{L}\}\ \overrightarrow{x:\tau'''=e}\ \_\in p\end{array}}{p;\Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2;\Gamma\vdash e_0\,\texttt{.}\,x_i:[\overrightarrow{\tau''/\beta}]\tau_i'''}$$:

By the induction hypothesis, we have $p;\Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2;[\overrightarrow{\tau/\alpha}]\Gamma\vdash[\overrightarrow{\tau/\alpha}]e_0:\tau_0''$ where $p;\Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2\vdash\tau_0''\ <:\ [\overrightarrow{\tau/\alpha}]\tau_0'$. By inspection of the definition of *bound*, notice that $\tau_0'=O\llbracket\overrightarrow{\tau''}\rrbracket$. Then by inspection of the subtyping rules, we know the last rule applied in the derivation of $p;\Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2\vdash\tau_0''\ <:\ [\overrightarrow{\tau/\alpha}]\tau_0'$ is [S-REFL] and $\tau_0''=[\overrightarrow{\tau/\alpha}]O\llbracket\overrightarrow{\tau''}\rrbracket$. The rest of this case is immediate.

Case [T-OBJECT] $$\frac{\begin{array}{c}\texttt{object}\,O\llbracket\overrightarrow{\beta\ \triangleleft\ M}\rrbracket(\overrightarrow{x':\tau'})\ \triangleleft\ \{\overrightarrow{L}\}\ \_\in p\\ p;\Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2\vdash O\llbracket\overrightarrow{\tau}\rrbracket\text{ ok}\\ p;\Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2;\Gamma\vdash\overrightarrow{e}:\overrightarrow{\tau''} \qquad p;\Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2\vdash\overrightarrow{\tau''}\ <:\ \overrightarrow{\tau'}\end{array}}{p;\Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2;\Gamma\vdash O\llbracket\overrightarrow{\tau}\rrbracket(\overrightarrow{e}):O\llbracket\overrightarrow{\tau}\rrbracket}$$:

By Lemma A.1.5, the induction hypothesis, and Lemma A.1.4, we have

$$\begin{array}{l}p;\Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2\vdash[\overrightarrow{\tau/\alpha}]O\llbracket\overrightarrow{\tau}\rrbracket\text{ ok}\\ p;\Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2;[\overrightarrow{\tau/\alpha}]\Gamma\vdash[\overrightarrow{\tau/\alpha}]\overrightarrow{e}:\overrightarrow{\tau'''}\text{ where }p;\Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2\vdash\overrightarrow{\tau'''}\ <:\ [\overrightarrow{\tau/\alpha}]\overrightarrow{\tau''}\\ p;\Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2;\Sigma\vdash[\overrightarrow{\tau/\alpha}]\overrightarrow{\tau''}\ <:\ [\overrightarrow{\tau/\alpha}]\overrightarrow{\tau'}\end{array}$$

Rules [S-TRANS] and [S-BOTH] finish the case.

Case [T-METHOD]

$$\frac{\begin{array}{c} p; \Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2; \Gamma \vdash e_0 : \tau_0' \qquad \mathit{mtype}_p(f, \mathit{bound}_{\Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2}(\tau_0')) = \{\llbracket \overrightarrow{\beta \triangleleft M} \rrbracket \overrightarrow{\tau'} \to \tau_0\} \\ p; \Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2 \vdash \overrightarrow{\tau'''}\ \mathrm{ok} \qquad p; \Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2 \vdash \overrightarrow{\tau'''}\ <:\ [\overrightarrow{\tau'''/\beta}]\overrightarrow{M} \\ p; \Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau''} \qquad p; \Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2 \vdash \overrightarrow{\tau''}\ <:\ [\overrightarrow{\tau'''/\beta}]\overrightarrow{\tau'} \end{array}}{p; \Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2; \Gamma \vdash e_0\,.\,f\llbracket\overrightarrow{\tau'''}\rrbracket(\overrightarrow{e}) : [\overrightarrow{\tau'''/\beta}]\tau_0} :$$

By the induction hypothesis, we have

$$\begin{array}{ll} p; \Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2; [\overrightarrow{\tau/\alpha}]\Gamma \vdash [\overrightarrow{\tau/\alpha}]e_0 : \tau_0'' & p; \Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2; [\overrightarrow{\tau/\alpha}]\Gamma \vdash [\overrightarrow{\tau/\alpha}]\overrightarrow{e} : \overrightarrow{\tau''''} \\ p; \Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2 \vdash \tau_0''\ <:\ [\overrightarrow{\tau/\alpha}]\tau_0' & p; \Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2 \vdash \overrightarrow{\tau''''}\ <:\ [\overrightarrow{\tau/\alpha}]\overrightarrow{\tau''}. \end{array}$$

By Lemma A.1.6, we have $p; \Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2 \vdash \mathit{bound}_{\Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2}(\tau_0'')\ <:\ [\overrightarrow{\tau/\alpha}]\mathit{bound}_{\Delta_1\ \overrightarrow{\alpha\ <:\ N}\ \Delta_2}(\tau_0')$.

By Lemma A.1.7, we have $\mathit{mtype}_p(f, \mathit{bound}_{\Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2}(\tau_0'')) = \llbracket \overrightarrow{\beta \triangleleft [\overrightarrow{\tau/\alpha}]M} \rrbracket\ [\overrightarrow{\tau/\alpha}]\overrightarrow{\tau'} \to \tau_0'''$ where $p; \Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2\ \overrightarrow{\beta\ <:\ [\overrightarrow{\tau/\alpha}]M}, \vdash \tau_0'''\ <:\ [\overrightarrow{\tau/\alpha}]\tau_0$. By Lemma A.1.5, we have $p; \Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2 \vdash [\overrightarrow{\tau/\alpha}]\overrightarrow{\tau'''}$ ok. By Lemma A.1.4, we have

$$\begin{array}{l} p; \Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2 \vdash [\overrightarrow{\tau/\alpha}]\overrightarrow{\tau'''}\ <:\ [\overrightarrow{\tau/\alpha}][\overrightarrow{\tau'''/\beta}]\overrightarrow{M} \\ p; \Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2 \vdash [\overrightarrow{\tau/\alpha}]\overrightarrow{\tau''}\ <:\ [\overrightarrow{\tau/\alpha}][\overrightarrow{\tau'''/\beta}]\overrightarrow{\tau'}. \end{array}$$

By [S-TRANS]and without a loss of generality, we have $p; \Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2 \vdash \overrightarrow{\tau''''} <: [\overrightarrow{\tau'/\alpha}][\overrightarrow{\tau/\beta}]\overrightarrow{\tau'}''\ (= [\overrightarrow{[\overrightarrow{\tau/\alpha}]\tau'''/\beta}][\overrightarrow{\tau/\alpha}]\overrightarrow{\tau'})$. By Lemma A.1.4, we have

$$p; \Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2 \vdash [\overrightarrow{\tau'''/\beta}]\tau_0'''\ <:\ [\overrightarrow{\tau/\alpha}][\overrightarrow{\tau'''/\beta}]\tau_0\ (= [\overrightarrow{[\overrightarrow{\tau/\alpha}]\tau'''/\beta}][\overrightarrow{\tau/\alpha}]\tau_0).$$

Finally, [T-METHOD] gives us $p; \Delta_1\ [\overrightarrow{\tau/\alpha}]\Delta_2; [\overrightarrow{\tau/\alpha}]\Gamma \vdash [\overrightarrow{\tau/\alpha}](e_0\,.\,f\llbracket\overrightarrow{\tau'''},\rrbracket(\overrightarrow{e})) : [\overrightarrow{\tau'''/\beta}]\tau_0'''$. □

Theorem 2 (Subject Reduction). *If p is well-typed and $p; \Delta; \Gamma \vdash e : \tau$ and $p \vdash e \longrightarrow e'$ then $p; \Delta; \Gamma \vdash e' : \tau'$ where $p; \Delta \vdash \tau'\ <:\ \tau$.*

Proof. The proof is by case analysis on the evaluation rule applied.

Case [R-FIELD]: $e = E[O\llbracket\overrightarrow{\tau}\rrbracket(\overrightarrow{v'})\,.\,x_i]$ $\quad e' = E[[\overrightarrow{\tau/\alpha}][\overrightarrow{v'/x'}]e_i]$

By the well-typedness of e, we have $p; \Delta; \Gamma \vdash O\llbracket\overrightarrow{\tau}\rrbracket(\overrightarrow{v'})\,.\,x_i : [\overrightarrow{\tau/\alpha}]\tau_i''$ where
`object` $O\llbracket\overrightarrow{\alpha \triangleleft N}\rrbracket(\overrightarrow{x':\tau'}) \triangleleft \{\overrightarrow{M}\}\ \overrightarrow{x:\tau''=e}\ \overrightarrow{fd}$ `end`$\in p$.

By typing rules [T-OBJECT], [T-OBJECTDEF], [T-FIELDDEF], and [W-BOTH], we have:

$(1a)\quad p;\Delta;\Gamma\vdash\overrightarrow{v'} : \overrightarrow{\tau_v}$ $\qquad(1b)\quad p;\Delta\vdash\overrightarrow{\tau_v}\ <:\ \overrightarrow{\tau'}$

$(2a)\quad p;\overrightarrow{\alpha\ <:\ N};\overrightarrow{x':\tau'}\vdash e_i : \tau_i'''$ $\qquad(2b)\quad p;\overrightarrow{\alpha\ <:\ N}\vdash\tau_i'''\ <:\ \tau_i''$

$\qquad(3b)\quad p;\Delta\vdash\overrightarrow{\tau}\ <:\ [\overrightarrow{\tau/\alpha}]\overrightarrow{N}$

$(4a)\quad p;\Delta;\Gamma\vdash O[\![\overrightarrow{\tau}]\!](\overrightarrow{v'}) : O[\![\overrightarrow{\tau}]\!]$

By Lemmas A.1.3 and A.1.8 applied to $(2a)$, $(1a)$, and $(1b)$ we have:

$(5a)\quad p;\Delta\ \overrightarrow{\alpha\ <:\ N};\Gamma\vdash[\overrightarrow{v'/x'}]e_i : \tau_i''''$ $\qquad(5b)\quad p;\Delta\ \overrightarrow{\alpha\ <:\ N}\vdash\tau_i''''\ <:\ \tau_i'''$

By Lemma A.1.9 applied to $(5a)$ and $(3b)$ we have:

$(6a)\quad p;\Delta;[\overrightarrow{\tau/\alpha}]\Gamma\vdash[\overrightarrow{\tau/\alpha}][\overrightarrow{v'/x'}]e_i : \tau_i'''''$ $\qquad(6b)\quad p;\Delta\vdash\tau_i'''''\ <:\ [\overrightarrow{\tau/\alpha}]\tau_i''''$

By Lemmas A.1.3, A.1.4, and [S-TRANS] we have:

$\qquad(7b)\quad p;\Delta\vdash\tau_i'''''\ <:\ [\overrightarrow{\tau/\alpha}]\tau_i''$

Applying Lemma A.1.2 to judgements $(6a)$ and $(7b)$ we finish the case.

Case [R-METHOD]: $e = E\texttt{[}O[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\texttt{)}\texttt{.}f[\![\overrightarrow{\tau',}]\!]\texttt{(}\overrightarrow{v',}\texttt{)]}$ $\quad e' = E\texttt{[}[\overrightarrow{v/x}][\overrightarrow{v'/x'}][O[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\texttt{)/self}]e''\texttt{]}$

where $\mathit{mbody}_p(f[\![\overrightarrow{\tau'}]\!],O[\![\overrightarrow{\tau}]\!]) = \texttt{(}\overrightarrow{x'}\texttt{)}\to e''$

and $\mathit{mtype}_p(f,O[\![\overrightarrow{\tau}]\!]) = [\![\overrightarrow{\beta\ \triangleleft\ M}]\!]\ \overrightarrow{\tau'''}\to\tau_0$

and $(\texttt{object}\ O[\![\overrightarrow{\alpha\ \triangleleft\ N}]\!]\texttt{(}\overrightarrow{x\texttt{:}\tau}\texttt{)}\ \triangleleft\ \{\overrightarrow{M}\}\ \overrightarrow{\mathit{vd}}\ \overrightarrow{\mathit{fd}}\ \texttt{end})\in p$.

By the well-typedness of e we have $p;\Delta;\Gamma\vdash O[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\texttt{)}\texttt{.}f[\![\overrightarrow{\tau',}]\!]\texttt{(}\overrightarrow{v',}\texttt{)} : [\overrightarrow{\tau'/\beta}][\overrightarrow{\tau/\alpha}]\tau_0$.

By typing rules [T-OBJECT], [T-OBJECTDEF], [T-METHODDEF], [T-METHOD], and [W-BOTH] we have:

(1a) $p;\Delta;\Gamma\vdash\overrightarrow{v} : \overrightarrow{\tau_v}$

(1b) $p;\Delta\vdash\overrightarrow{\tau_v}\ <:\ \overrightarrow{\tau''}$

(2b) $p;\Delta\vdash\overrightarrow{\tau}\ <:\ [\overrightarrow{\tau/\alpha}]\overrightarrow{N}$

(3a) $p;\Delta;\Gamma\vdash\overrightarrow{v'} : \overrightarrow{\tau_v'}$

(3b) $p;\Delta\vdash\overrightarrow{\tau_v'}\ <:\ [\overrightarrow{\tau'/\beta}]\overrightarrow{\tau'''}$

(4b) $p;\Delta\vdash\overrightarrow{\tau'}\ <:\ [\overrightarrow{\tau'/\beta}]\overrightarrow{M}$

(5a) $p;\overrightarrow{\alpha\ <:\ N},\overrightarrow{\beta\ <:\ M},;\overrightarrow{x':\tau'''}\ \texttt{self} : O[\![\overrightarrow{\tau}]\!]\ \overrightarrow{x:\tau''}\vdash e''' : \tau_0'$ $\qquad$ where $e'' = [\overrightarrow{\tau'/\beta}][\overrightarrow{\tau/\alpha}]e'''$

(5b) $p;\overrightarrow{\alpha\ <:\ N},\overrightarrow{\beta\ <:\ M},\vdash\tau_0'\ <:\ \tau_0$

(6a) $\quad p;\Delta;\Gamma\vdash O[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\texttt{)} : O[\![\overrightarrow{\tau}]\!]$

By Lemma A.1.9, applied to $(5a)$ and $(4b)$ we have:

(7a) $p;\Delta\ \overrightarrow{\beta\ <:\ M},;[\overrightarrow{\tau/\alpha}]\Gamma\ \overrightarrow{x':\tau'''}\ \texttt{self} : O[\![\overrightarrow{\tau}]\!]\ \overrightarrow{x:\tau''}\vdash[\overrightarrow{\tau/\alpha}]e''' : \tau_0''$

(7b) $p;\Delta\ \overrightarrow{\beta\ <:\ M},\vdash\tau_0''\ <:\ [\overrightarrow{\tau/\alpha}]\tau_0'$

By Lemma A.1.9, applied to $(7a)$ and $(2b)$ we have:

(8a) $p;\Delta;[\overrightarrow{\tau'/\beta}][\overrightarrow{\tau/\alpha}]\Gamma\ \overrightarrow{x':\tau'''}\ \texttt{self} : O[\![\overrightarrow{\tau}]\!]\ \overrightarrow{x:\tau''}\vdash[\overrightarrow{\tau'/\beta}][\overrightarrow{\tau/\alpha}]e''' : \tau_0'''$

(8b) $p;\Delta\vdash\tau_0'''\ <:\ [\overrightarrow{\tau'/\beta}]\tau_0''$

By Lemmas A.1.3 and A.1.8, applied to $(6a)$ and $(7a)$ we have:

(9a) $p;\Delta;[\overrightarrow{\tau'/\beta}][\overrightarrow{\tau/\alpha}]\Gamma\ \overrightarrow{x':\tau'''}\ \overrightarrow{x:\tau''}\vdash[O[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\texttt{)/self}][\overrightarrow{\tau'/\beta}][\overrightarrow{\tau/\alpha}]e''' : \tau_0''''$

(9b) $p;\Delta\vdash\tau_0''''\ <:\ \tau_0'''$

By Lemma A.1.3 and A.1.8, applied to $(8a)$, $(3a)$ and $(3b)$ we have:
(10a) $p; \Delta; [\overrightarrow{\tau'/\beta}][\overrightarrow{\tau/\alpha}]\Gamma\ \overrightarrow{x : \tau''} \vdash [\overrightarrow{v'/x'}][O[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\texttt{)/self}][\overrightarrow{\tau'/\beta}][\overrightarrow{\tau/\alpha}]e''' : \tau_0'''''$
(10b) $p; \Delta \vdash \tau_0''''' \ <: \ \tau_0''''$

By Lemma A.1.3 and A.1.8, applied to $(9a)$, $(1a)$ and $(1b)$ we have:
(11a) $p; \Delta; [\overrightarrow{\tau'/\beta}][\overrightarrow{\tau/\alpha}]\Gamma \vdash [\overrightarrow{v/x}][\overrightarrow{v'/x'}][O[\![\overrightarrow{\tau}]\!]\texttt{(}\overrightarrow{v}\texttt{)/self}][\overrightarrow{\tau'/\beta}][\overrightarrow{\tau/\alpha}]e''' : \tau_0''''''$
(11b) $p; \Delta \vdash \tau_0'''''' \ <: \ \tau_0'''''$

By Lemmas A.1.3, A.1.4, and [S-TRANS], we have:
(12b) $p; \Delta \vdash \tau_0'''''' \ <: \ [\overrightarrow{\tau'/\beta}][\overrightarrow{\tau/\alpha}]\tau_0$

Applying Lemma A.1.2 to judgements $(10a)$ and $(12b)$ we finish the case.

$\square$

Appendix B

Api `components`

We define a special `components` api that provides handles on components and apis, and operations on them, for use by components themselves (e.g., development environments), allowing components to build and maintain other components, manipulate projects and components as objects, compile projects into components, link components together, deploy components on specific sites over the internet, etc. This api is also used by the `upgradable` and `installable` apis. A component implementing this api is installed along with the core library components on every fortress.

Note that `Components` and `Apis` have no public constructors. They can be constructed only from the factory methods provided by a fortress. The components returned from a `Fortress` are also installed in that `Fortress`. Also note the `Fortress` is a singleton class; it has no public constructor, but there is a single instance provided in a static field.

The operations on a fortress provided in this api take components and apis as arguments directly, rather than names of components and apis as the corresponding shell operations are described. This decision is done for the sake of convenience. Note, however, that a component name may be rebound on a fortress, or even uninstalled, while some process p keeps a reference to a corresponding `Component` object. This possibility is not problematic because the component corresponding to this object may be simply kept by the fortress until the object is freed in p. Also, note that upgrade operations on a compound are purely functional: they produce new compound components as a result. Thus, the structure of a component as viewed through a `Component` object does not became stale in the face of upgrades.

We include a method `getSourceFile` on components that returns the source file the component was compiled from. Source files could be included with simple components during compilation as a compiler option. Doing so would allow development tools such as graphical debuggers to easily display the locations of errors without the possibility that source code would not be synchronized with compiled code, as can happen in conventional programming models where compiled code is stored in unencapsulated object files.

```
api components.1.0

import File from fortress.io.1.0
import {List, Set, Date} from fortress.util.1.0

object Fortress
  getComponent(name:Name):Component
  getApi(name:Name): Api
  preferences(api: Api):List[Component]

  compile(source:File):SimpleComponent
  install(serialized:File):Component
  upgradeAll(name:Name, other:Component):()

  link(name:Name, constituents:List[Component],
       export:{Api}, hide:{Api}):Component
    throws LinkException
end

trait FortressElement
  name():Name
  vendor():String
  owner():Fortress
  timestamp():Date
  getVersion():Version
  uninstall():()
end

object Component
  ( imports :{Api},
    exports :{Api},
    provides:{Api},
    visibles:{Api},
    constituents:{Component} )
  traits FortressElement
  execute(args:String[]):()
  constrain(name:Name, apis:{Api}):Component
  hide     (name:Name, apis:{Api}):Component
  extract(prereqs:{Api}):File
  isValidUpgrade(that:Component):Boolean
  abstract upgrade(name:Name, that:Component):Component
    throws UpgradeException
  getSourceFile():File throws SourceNotAvailableException
end

object SimpleComponent traits Component
  upgrade(name:Name, that:Component):Component
end

object CompoundComponent traits Component
  upgrade(name:Name, other:Component):Component
end

object Api
  ( uses:{Api},
    extract:File )
  traits FortressElement
end

object Name
  toString():String
end

object Version
```

```
  major: Int
  minor: Int
end

object UpgradeException(msg:String) traits Exception end
object LinkException(msg:String) traits Exception end
object SourceNotAvailableException(msg:String) traits Exception end
```

Appendix C

Support for Unicode Input in ASCII

ASCII encoding of Unicode in Fortress programs is supported as follows:

1. Names for all Unicode characters except control characters can be written in all caps. Spaces in a name are written as '_' . Additionally, all Unicode character names are aliased with names in which the following substrings are elided:

   ```
   "LETTER "
   "DIGIT "
   "RADICAL "
   "NUMERAL "
   " OPERATOR"
   ```

2. All ASCII-encoded Unicode characters are converted to Unicode before parsing or scanning.
3. If two Unicode names are separated by an ampersand, the ampersand is removed as the two names are converted to Unicode characters.
4. If a line is continued using a final ampersand, and the continuation line begins with an ampersand, and the first ampersand is immediately preceded by an identifier with no intervening space, and the second ampersand is immediately followed by an identifier with no intervening space, then the two identifiers are logically glued together to make one identifier.

Here is a simple example. The expression:

```
(GREEK_SMALL_LETTER_PHI GREEK_SMALL_LETTER_PSI +
GREEK_SMALL_LETTER_OMEGA GREEK_SMALL_LETTER_LAMBDA)
```

is converted to:

```
(ϕ ψ + ω λ)
```

where there are four identifiers in all. To get two identifiers, one writes

```
(GREEK_SMALL_LETTER_PHI&GREEK_SMALL_LETTER_PSI +
GREEK_SMALL_LETTER_OMEGA&GREEK_SMALL_LETTER_LAMBDA)
```

which is converted to:

(ϕψ + ωλ)

We include shorter names for common characters. In particular, the following tokens are converted as follows:

`BY`	*becomes*	$\times$	`*`	*becomes*	$\cdot$
`->`	*becomes*	$\rightarrow$	`=>`	*becomes*	$\Rightarrow$
`-->`	*becomes*	$\longrightarrow$	`==>`	*becomes*	$\Longrightarrow$
`~>`	*becomes*	$\leadsto$	`:->`	*becomes*	$\mapsto$
`>=`	*becomes*	$\geq$	`<=`	*becomes*	$\leq$
`:=`	*becomes*	$\models$	`:-`	*becomes*	$\vdash$
`CUP`	*becomes*	$\cup$	`CAP`	*becomes*	$\cap$
`BOTTOM`	*becomes*	$\bot$	`TOP`	*becomes*	$\top$
`SUM`	*becomes*	$\sum$	`PRODUCT`	*becomes*	$\prod$
`INTEGRAL`	*becomes*	$\int$	`EMPTYSET`	*becomes*	$\emptyset$
`SUBSET`	*becomes*	$\subset$	`NOTSUBSET`	*becomes*	$\not\subset$
`SUBSETEQ`	*becomes*	$\subseteq$	`NOTSUBSETEQ`	*becomes*	$\not\subseteq$
`EQUALS`	*becomes*	$\cong$	`NOTEQUALS`	*becomes*	$\not\cong$
`EQUIV`	*becomes*	$\equiv$	`/=`	*becomes*	$\neq$
`IN`	*becomes*	$\in$	`NOTIN`	*becomes*	$\notin$

The following tokens and there lowercase counterparts are converted as follows:

`AND`	*becomes*	$\wedge$
`OR`	*becomes*	$\vee$
`NOT`	*becomes*	$\neg$
`XOR`	*becomes*	$\oplus$
`INF`	*becomes*	∞
`SQRT`	*becomes*	$\surd$

As a special feature, twenty-four names, in both uppercase and lowercase form, are converted to Greek letters:

```
alpha beta gamma delta epsilon zeta eta theta iota kappa lambda
mu nu xi omicron pi rho sigma tau upsilon phi chi psi omega
```

become:

$\alpha\ \beta\ \gamma\ \delta\ \epsilon\ \zeta\ \eta\ \theta\ \iota\ \kappa\ \lambda\ \mu\ \nu\ \xi\ o\ \pi\ \rho\ \sigma\ \tau\ \upsilon\ \phi\ \chi\ \psi\ \omega$

C.1 Distributed Pasting

There are many sets of Unicode character names for which all characters in the set share the same prefix. For example, consider the prefixes

```
'GREEK_SMALL_LETTER', 'KATAKANA_LETTER', 'CYRILLIC_CAPITAL_LETTER'
```

etc. Furthermore, characters with the same prefix are often typed together.

To facilitate entering characters with a common prefix, Fortress supports distribution of a Unicode prefix over a sequence of character names. If an identifier-pasting ampersand is followed by a left parenthesis, there is distributed meta-pasting over the characters enclosed in parentheses, followed by Unicode conversion followed by pasting. For example,

```
foo&(a&b&c&d&e)
```

becomes

```
fooa&foob&fooc&food&fooe
```

which becomes

```
fooafoobfoocfoodfooe
```

To keep error messages sane, unbalanced parentheses should detected as errors *before* meta-pasting.

Program text is allowed to include Unicode directly. A Fortress program editor is expected to convert ASCII-encoded Unicode to straight Unicode as the characters are typed.

C.2 String Literals

By default, the preprocessing described above is not performed within string literals. Therefore, the string literal “`alpha`” contains five characters.

To enable the unicode transformations, a backslash (\) must be prepended. Therefore, the string literal “`\alpha`” contains only one character, corresponding to the Unicode character for α. In order to embed a literal backslash character, it must also be escaped with a backslash. Therefore, “`\\beta`” contains five characters, the first of which is `\`.

The preprocessing described below, which is for identifiers and numeric literals, is never performed within string literals.

C.3 Identifiers and Numeric Literals

An alphanumeric token consists of a sequence of letters, digits, and underscore characters; if it begins with a digit, or ends with a radix specifier and contains no other underscores, then it may also contain a period (to serve as a decimal point or other radix point). The interpretation and display appearance of the token depends on which of a number of categories it falls into.

First, a preprocessing step: if the token is the (all-uppercase or all-lowercase) name of a Greek letter, or begins with the name of a Greek letter followed by an underscore or a digit, or ends with the name of a Greek letter that is preceded by an underscore, or contains the name of a Greek letter with an underscore on each side of it, then the name of the Greek letter is replaced by the Greek letter itself. A special ad-hoc rule is that if the identifier begins with "`mu_`" and ends with "`_`", then the first underscore is deleted as the "mu" is converted to the Greek letter lowercase mu.

Examples: alpha OMEGA3 `alpha_hat` `theta_elephant` `OMEGA_` `_XI`

α $\Omega 3$ α`_hat` θ`_elephant` Ω`_` `_`Ξ

Now, if the token begins with a digit, it is a numeric token. It is displayed in roman type, with the optional radix as a subscript.

Examples: 27 `7fff_16` `10101101_TWO` 3.14159265 `3.11037552_8`

27 $\mathrm{7fff}_{16}$ 10101101_{2} 3.14159265 3.11037552_{8}

Note: the elegant way to write Avogadro's number is `6.02 TIMES 10^23`.

Otherwise, if the token contains no underscores, it is an identifier. If it consists of one or more letters and then one or more digits, it is displayed by displaying the letters in italic type and the digits as italic subscripts (to distinguish them from true numeric indexing subscripts indicated by brackets, which would be shown in roman type); otherwise, it is simply displayed in italic type.

Examples: Fred foobar a1 a23 alpha1 l33tsp33k

$Fred$ $foobar$ a_1 a_{23} α_1 $l33tsp33k$

If the token contains one underscore and what follows the underscore is either a decimal integer from 2 to 16 or the English name (in all capital letters) of an integer from 2 to 16, it is a numeric token. The part before the underscore is displayed in roman type and the part after the underscore is displayed as a decimal subscript.

Examples: `deadbeef_SIXTEEN` `dead.beef_16` `37X8E2_12` `3.243f6b_16`

$\mathrm{deadbeef}_{16}$ $\mathrm{dead.beef}_{16}$ $\mathrm{37X8E2}_{12}$ $\mathrm{3.243f6b}_{16}$

Otherwise, it is an identifier, and the rules for display are complicated:

If a token ends with an underscore and has no other underscores, it is displayed without the trailing underscore but in roman type rather than italic. (This is typically used for names of SI dimensional units. Here we see the reason for the ad-hoc rule about the treatment of "mu" at the beginning of an identifier ending with an underscore.)

Examples: `m_` `s_` `km_` `V_` `OMEGA_` `mu_s_` `mu_OMEGA_`

m s km V Ω μs $\mu\Omega$

Otherwise, the token is divided into components by its underscores. If any component is empty except the first, then the entire identifier is displayed in italics, underscores and all. Otherwise, the components are displayed as follows. If the first component is empty (that is, the identifier begins with a leading underscore), then the second component is displayed in boldface and then any remaining components are processed from left to right beginning with the third component (this is typically used for vectors and matrices, and for the square root of minus one); otherwise, if the first component is `script`, then the second component is displayed in a script face and then any remaining components are processed from left to right beginning with the third component; otherwise the first component is displayed in italics and remaining components are processed from left to right beginning with the second component. (However, as the second or first component is displayed according to the previous sentence, if the component consists of a sequence of letters followed by a sequence of digits, then the digits are displayed as a subscript in italics.) The remaining components are then processed according to the following rules:

- If a component is `bar`, then a bar is displayed above what has already been displayed.
- If a component is `vec`, then a right-pointing arrow is displayed above what has already been displayed.
- If a component is `hat`, then a hat is displayed above what has already been displayed.
- If a component is `dot`, then a dot is displayed above what has already been displayed; but if the preceding component was also `dot`, then the new dot is displayed appropriately relative to the previous dot(s).
- If a component is `prime`, then a prime mark is displayed after what has already been displayed as a superscript.
- If a component is `super` and another component follows, then that component is displayed as a superscript in roman type, and enclosed in parentheses if it is all digits.
- If the component is the last component, it is displayed as a subscript, in italics if it is all digits, and otherwise in roman type.
- Otherwise, this component and all succeeding components are displayed in italics, each with a preceding underscore.

Examples: `v_vec` `_v` `_M` `_v1` `a_dot` `p_prime` `p13_prime` `T_max` `foo_bar`

$\vec{v}$ $\mathbf{v}$ $\mathbf{M}$ $\mathbf{v}_1$ $\dot{a}$ p' p'_{13} T_{max} foo_bar

Appendix D

Detailed Rules for Operator Precedence

In each of the character lists below, each line gives the Unicode codepoint, the full Unicode name, an indication of what the character looks like in TEX (if possible), then any alternate names or ASCII renderings for the character.

D.1 Bracket Pairs for Enclosing Operators

Here are the bracket pairs that may be used as enclosing operators. Note that there are two groups of four brackets; within such a group, either left bracket may be paired with either right bracket to form an enclosing operator.

U+005B	LEFT SQUARE BRACKET	[	
U+005D	RIGHT SQUARE BRACKET	]	
U+007B	LEFT CURLY BRACKET	{	
U+007D	RIGHT CURLY BRACKET	}	
U+00AB	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK		
U+00BB	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK		
U+2045	LEFT SQUARE BRACKET WITH QUILL		
U+2046	RIGHT SQUARE BRACKET WITH QUILL		
U+2308	LEFT CEILING	⌈	LC
U+2309	RIGHT CEILING	⌉	RC
U+230A	LEFT FLOOR	⌊	LF
U+230B	RIGHT FLOOR	⌋	RF
U+27E6	MATHEMATICAL LEFT WHITE SQUARE BRACKET	⟦	[|
U+2985	LEFT WHITE PARENTHESIS		(|
U+2986	RIGHT WHITE PARENTHESIS		|)
U+27E7	MATHEMATICAL RIGHT WHITE SQUARE BRACKET	⟧	|]
U+27E8	MATHEMATICAL LEFT ANGLE BRACKET	⟨	<|
U+27E9	MATHEMATICAL RIGHT ANGLE BRACKET	⟩	|>
U+27EA	MATHEMATICAL LEFT DOUBLE ANGLE BRACKET	⟪	<<|
U+27EB	MATHEMATICAL RIGHT DOUBLE ANGLE BRACKET	⟫	|>>
U+2983	LEFT WHITE CURLY BRACKET	⦃	{|
U+2984	RIGHT WHITE CURLY BRACKET	⦄	|}

U+2987	Z NOTATION LEFT IMAGE BRACKET	
U+2988	Z NOTATION RIGHT IMAGE BRACKET	
U+2989	Z NOTATION LEFT BINDING BRACKET	
U+298A	Z NOTATION RIGHT BINDING BRACKET	
U+298B	LEFT SQUARE BRACKET WITH UNDERBAR	
U+298C	RIGHT SQUARE BRACKET WITH UNDERBAR	
U+298D	LEFT SQUARE BRACKET WITH TICK IN TOP CORNER	
U+298E	RIGHT SQUARE BRACKET WITH TICK IN BOTTOM CORNER	
U+298F	LEFT SQUARE BRACKET WITH TICK IN BOTTOM CORNER	
U+2990	RIGHT SQUARE BRACKET WITH TICK IN TOP CORNER	
U+2991	LEFT ANGLE BRACKET WITH DOT	
U+2992	RIGHT ANGLE BRACKET WITH DOT	
U+2993	LEFT ARC LESS-THAN BRACKET	
U+2994	RIGHT ARC GREATER-THAN BRACKET	
U+2995	DOUBLE LEFT ARC GREATER-THAN BRACKET	
U+2996	DOUBLE RIGHT ARC LESS-THAN BRACKET	
U+2997	LEFT BLACK TORTOISE SHELL BRACKET	[*
U+2998	RIGHT BLACK TORTOISE SHELL BRACKET	*]
U+29D8	LEFT WIGGLY FENCE	
U+29D9	RIGHT WIGGLY FENCE	
U+29DA	LEFT DOUBLE WIGGLY FENCE	
U+29DB	RIGHT DOUBLE WIGGLY FENCE	
U+29FC	LEFT-POINTING CURVED ANGLE BRACKET	
U+29FD	RIGHT-POINTING CURVED ANGLE BRACKET	
U+300C	LEFT CORNER BRACKET	「 </
U+300D	RIGHT CORNER BRACKET	」 />
U+300E	LEFT WHITE CORNER BRACKET	<</
U+300F	RIGHT WHITE CORNER BRACKET	/>>
U+3010	LEFT BLACK LENTICULAR BRACKET	{*
U+3011	RIGHT BLACK LENTICULAR BRACKET	*}
U+3018	LEFT WHITE TORTOISE SHELL BRACKET	[/
U+3014	LEFT TORTOISE SHELL BRACKET	(/
U+3015	RIGHT TORTOISE SHELL BRACKET	/)
U+3019	RIGHT WHITE TORTOISE SHELL BRACKET	/]
U+3016	LEFT WHITE LENTICULAR BRACKET	{/
U+3017	RIGHT WHITE LENTICULAR BRACKET	/}

In addition, each of these operators may be paired with itself to form an enclosing operator pair (see Section 2.8):

```
   |     ||     |||     /     //     ///     ˜
```

D.2 Arithmetic Operators

D.2.1 Multiplication and Division

The following four operators have the same precedence and may be mixed. Note that `SOLIDUS` and `DIVISION SLASH` must be used loose for this purpose, because when used tight they form tight fractions; therefore, when multiplication and division operators in this group are to be mixed, they must all be loose.

The following are multiplication operators. Note that `ASTERISK OPERATOR` is always a multiplication operator; `ASTERISK` is treated as a synonym for `ASTERISK OPERATOR` where appropriate, but `ASTERISK` also has other uses, for example in the ASCII bracket encodings `[*` and `*]` and `{*` and `*}`.

U+002A	ASTERISK	*
U+00B7	MIDDLE DOT	· DOT
U+00D7	MULTIPLICATION SIGN	× TIMES BY
U+2217	ASTERISK OPERATOR	∗
U+228D	MULTISET MULTIPLICATION	
U+2297	CIRCLED TIMES	⊗ OTIMES
U+2299	CIRCLED DOT OPERATOR	⊙ ODOT
U+229B	CIRCLED ASTERISK OPERATOR	⊛ CIRCLEDAST
U+22A0	SQUARED TIMES	⊠ BOXTIMES
U+22A1	SQUARED DOT OPERATOR	⊡ BOXDOT
U+22C5	DOT OPERATOR	⋅
U+29C6	SQUARED ASTERISK	BOXAST
U+29D4	TIMES WITH LEFT HALF BLACK	
U+29D5	TIMES WITH RIGHT HALF BLACK	
U+2A2F	VECTOR OR CROSS PRODUCT	× CROSS
U+2A30	MULTIPLICATION SIGN WITH DOT ABOVE	
U+2A31	MULTIPLICATION SIGN WITH UNDERBAR	
U+2A34	MULTIPLICATION SIGN IN LEFT HALF CIRCLE	
U+2A35	MULTIPLICATION SIGN IN RIGHT HALF CIRCLE	
U+2A36	CIRCLED MULTIPLICATION SIGN WITH CIRCUMFLEX ACCENT	
U+2A37	MULTIPLICATION SIGN IN DOUBLE CIRCLE	
U+2A3B	MULTIPLICATION SIGN IN TRIANGLE	TRITIMES

The following are division operators. Note that `DIVISION SLASH` is always a multiplication operator; `SOLIDUS` is treated as a synonym for `DIVISION SLASH` where appropriate, but `SOLIDUS` also has other uses, for example in the ASCII bracket encodings `(/` and `/)` and `[/` and `/]` and `{/` and `/}`.

U+002F	SOLIDUS	/
U+00F7	DIVISION SIGN	÷ DIV
U+2215	DIVISION SLASH	∕
U+2298	CIRCLED DIVISION SLASH	⊘ OSLASH
U+29B8	CIRCLED REVERSE SOLIDUS	
U+29BC	CIRCLED ANTICLOCKWISE-ROTATED DIVISION SIGN	
U+29C4	SQUARED RISING DIAGONAL SLASH	BOXSLASH
U+29F5	REVERSE SOLIDUS OPERATOR	⧵
U+29F8	BIG SOLIDUS	⧸

U+29F9	BIG REVERSE SOLIDUS	⧹
U+2A38	CIRCLED DIVISION SIGN	ODIV
U+2AFD	DOUBLE SOLIDUS OPERATOR	⫽

D.2.2 Addition and Subtraction

The following three operators have the same precedence and may be mixed.

U+002B	PLUS SIGN	+ +
U+002D	HYPHEN-MINUS	− -
U+2212	MINUS SIGN	−

They each have lower precedence than any of the following multiplication and division operators:

U+002A	ASTERISK	*
U+002F	SOLIDUS	/
U+00B7	MIDDLE DOT	· DOT
U+00D7	MULTIPLICATION SIGN	× TIMES
U+00F7	DIVISION SIGN	÷ DIV
U+2215	DIVISION SLASH	∕
U+2217	ASTERISK OPERATOR	∗
U+22C5	DOT OPERATOR	⋅
U+2A2F	VECTOR OR CROSS PRODUCT	⨯ CROSS

The following two operators have the same precedence and may be mixed.

U+2214	DOT PLUS	∔
U+2238	DOT MINUS	∸

The following two operators have the same precedence and may be mixed.

U+2A25	PLUS SIGN WITH DOT BELOW
U+2A2A	MINUS SIGN WITH DOT BELOW

The following two operators have the same precedence and may be mixed.

U+2A39	PLUS SIGN IN TRIANGLE
U+2A3A	MINUS SIGN IN TRIANGLE

They each have lower precedence than this multiplication operator:

U+2A3B	MULTIPLICATION SIGN IN TRIANGLE	TRITIMES

The following two operators have the same precedence and may be mixed.

U+2295	CIRCLED PLUS	⊕ OPLUS
U+2296	CIRCLED MINUS	⊖ OMINUS

They each have lower precedence than any of the following multiplication and division operators:

U+2297	CIRCLED TIMES	⊗ OTIMES
U+2298	CIRCLED DIVISION SLASH	⊘ OSLASH
U+2299	CIRCLED DOT OPERATOR	⊙ ODOT
U+229B	CIRCLED ASTERISK OPERATOR	⊛ CIRCLEDAST
U+2A38	CIRCLED DIVISION SIGN	ODIV

The following two operators have the same precedence and may be mixed.

U+229E	SQUARED PLUS	⊞ BOXPLUS
U+229F	SQUARED MINUS	⊟ BOXMINUS

They each have lower precedence than any of these multiplication or division operators:

U+22A0	SQUARED TIMES	⊠ BOXTIMES
U+22A1	SQUARED DOT OPERATOR	⊡ BOXDOT
U+29C4	SQUARED RISING DIAGONAL SLASH	BOXSLASH
U+29C6	SQUARED ASTERISK	BOXAST

These are other miscellaneous addition and subtraction operators:

U+00B1	PLUS-MINUS SIGN	±
U+2213	MINUS-OR-PLUS SIGN	∓
U+2242	MINUS TILDE	
U+2A22	PLUS SIGN WITH SMALL CIRCLE ABOVE	⨢
U+2A23	PLUS SIGN WITH CIRCUMFLEX ACCENT ABOVE	⨣
U+2A24	PLUS SIGN WITH TILDE ABOVE	⨤
U+2A26	PLUS SIGN WITH TILDE BELOW	⨦
U+2A27	PLUS SIGN WITH SUBSCRIPT TWO	$+_2$
U+2A28	PLUS SIGN WITH BLACK TRIANGLE	
U+2A29	MINUS SIGN WITH COMMA ABOVE	⨩
U+2A2B	MINUS SIGN WITH FALLING DOTS	
U+2A2C	MINUS SIGN WITH RISING DOTS	
U+2A2D	PLUS SIGN IN LEFT HALF CIRCLE	
U+2A2E	PLUS SIGN IN RIGHT HALF CIRCLE	

D.2.3 Miscellaneous Arithmetic Operators

The operators MAX, MIN, REM, MOD, GCD, LCM, and CHOOSE, none of which corresponds to a single Unicode character, are considered to be arithmetic operators, having higher precedence than certain relational operators, as described in a later section.

D.2.4 Set Intersection, Union, and Difference

The following are the set intersection operators:

U+2229	INTERSECTION	∩ CAP INTERSECT
U+22D2	DOUBLE INTERSECTION	⋒ CAPCAP
U+2A40	INTERSECTION WITH DOT	
U+2A43	INTERSECTION WITH OVERBAR	⩃
U+2A44	INTERSECTION WITH LOGICAL AND	
U+2A4B	INTERSECTION BESIDE AND JOINED WITH INTERSECTION	
U+2A4D	CLOSED INTERSECTION WITH SERIFS	
U+2ADB	TRANSVERSAL INTERSECTION	

The following are the set union operators:

U+222A	UNION	∪ CUP UNION
U+228E	MULTISET UNION	⊎ UPLUS
U+22D3	DOUBLE UNION	⋓ CUPCUP
U+2A41	UNION WITH MINUS SIGN	
U+2A42	UNION WITH OVERBAR	⩂
U+2A45	UNION WITH LOGICAL OR	
U+2A4A	UNION BESIDE AND JOINED WITH UNION	
U+2A4C	CLOSED UNION WITH SERIFS	
U+2A50	CLOSED UNION WITH SERIFS AND SMASH PRODUCT	

They each have lower precedence than any of the set intersection operators.

This is a miscellaneous set operator:

U+2216	SET MINUS	∖ SETMINUS

D.2.5 Square Arithmetic Operators

The following are the square intersection operators:

U+2293	SQUARE CAP	⊓ SQCAP
U+2A4E	DOUBLE SQUARE INTERSECTION	SQCAPCAP

The following are the square union operators:

U+2294	SQUARE CUP	⊔ SQCUP
U+2A4F	DOUBLE SQUARE UNION	SQCUPCUP

They each have lower precedence than either of the square intersection operators.

D.2.6 Curly Arithmetic Operators

The following is the curly intersection operator:

U+22CF	CURLY LOGICAL AND	⋏	CURLYAND

The following is the curly union operator:

U+22CE	CURLY LOGICAL OR	⋎	CURLYOR

It has lower precedence than the curly intersection operator.

D.3 Relational Operators

D.3.1 Equivalence and Inequivalence Operators

Every operator listed in this section has lower precedence than any operator listed in Section D.2.

The following are equivalence operators. They may be chained. Moreover, they may be chained with any oher single group of chainable relational operators, as described in later sections.

U+003D	EQUALS SIGN	=	EQ
U+2243	ASYMPTOTICALLY EQUAL TO	≃	SIMEQ
U+2245	APPROXIMATELY EQUAL TO	≅	
U+2246	APPROXIMATELY BUT NOT ACTUALLY EQUAL TO		
U+2247	NEITHER APPROXIMATELY NOR ACTUALLY EQUAL TO	≇	
U+2248	ALMOST EQUAL TO	≈	APPROX
U+224A	ALMOST EQUAL OR EQUAL TO	≊	APPROXEQ
U+224C	ALL EQUAL TO		
U+224D	EQUIVALENT TO	≍	
U+224E	GEOMETRICALLY EQUIVALENT TO	≎	BUMPEQV
U+2251	GEOMETRICALLY EQUAL TO	≑	DOTEQDOT
U+2252	APPROXIMATELY EQUAL TO OR THE IMAGE OF	≒	
U+2253	IMAGE OF OR APPROXIMATELY EQUAL TO	≓	
U+2256	RING IN EQUAL TO	≖	EQRING
U+2257	RING EQUAL TO	≗	RINGEQ
U+225B	STAR EQUALS		
U+225C	DELTA EQUAL TO	≜	EQDEL
U+225D	EQUAL TO BY DEFINITION		EQDEF
U+225F	QUESTIONED EQUAL TO		
U+2261	IDENTICAL TO	≡	EQV
U+2263	STRICTLY EQUIVALENT TO		
U+229C	CIRCLED EQUALS		
U+22CD	REVERSED TILDE EQUALS	⋍	
U+22D5	EQUAL AND PARALLEL TO		
U+29E3	EQUALS SIGN AND SLANTED PARALLEL		
U+29E4	EQUALS SIGN AND SLANTED PARALLEL WITH TILDE ABOVE		

U+29E5	IDENTICAL TO AND SLANTED PARALLEL	
U+2A66	EQUALS SIGN WITH DOT BELOW	
U+2A67	IDENTICAL WITH DOT ABOVE	
U+2A6C	SIMILAR MINUS SIMILAR	
U+2A6E	EQUALS WITH ASTERISK	
U+2A6F	ALMOST EQUAL TO WITH CIRCUMFLEX ACCENT	
U+2A70	APPROXIMATELY EQUAL OR EQUAL TO	
U+2A71	EQUALS SIGN ABOVE PLUS SIGN	
U+2A72	PLUS SIGN ABOVE EQUALS SIGN	
U+2A73	EQUALS SIGN ABOVE TILDE OPERATOR	
U+2A75	TWO CONSECUTIVE EQUALS SIGNS	
U+2A76	THREE CONSECUTIVE EQUALS SIGNS	
U+2A77	EQUALS SIGN WITH TWO DOTS ABOVE AND TWO DOTS BELOW	
U+2A78	EQUIVALENT WITH FOUR DOTS ABOVE	
U+2AAE	EQUALS SIGN WITH BUMPY ABOVE	
U+FE66	SMALL EQUALS SIGN	
U+FF1D	FULLWIDTH EQUALS SIGN	

The following are inequivalence operators. They may not be chained.

U+2244	NOT ASYMPTOTICALLY EQUAL TO	≄ NSIMEQ
U+2249	NOT ALMOST EQUAL TO	≉ NAPPROX
U+2260	NOT EQUAL TO	≠ /= NE
U+2262	NOT IDENTICAL TO	≢ NEQV
U+226D	NOT EQUIVALENT TO	≭

D.3.2 Plain Comparison Operators

Every operator listed in this section has lower precedence than any operator listed in Sections D.2.1, D.2.2, and D.2.3.

The following are less-than operators. They may be mixed and chained with each other and with equivalence operators (see Section D.3.1).

U+003C	LESS-THAN SIGN	< LT
U+2264	LESS-THAN OR EQUAL TO	≤ <= LE
U+2266	LESS-THAN OVER EQUAL TO	≦
U+2268	LESS-THAN BUT NOT EQUAL TO	≨
U+226A	MUCH LESS-THAN	≪ <<
U+2272	LESS-THAN OR EQUIVALENT TO	≲
U+22D6	LESS-THAN WITH DOT	⋖ DOTLT
U+22D8	VERY MUCH LESS-THAN	⋘ <<<
U+22DC	EQUAL TO OR LESS-THAN	
U+22E6	LESS-THAN BUT NOT EQUIVALENT TO	⋦
U+29C0	CIRCLED LESS-THAN	
U+2A79	LESS-THAN WITH CIRCLE INSIDE	
U+2A7B	LESS-THAN WITH QUESTION MARK ABOVE	
U+2A7D	LESS-THAN OR SLANTED EQUAL TO	
U+2A7F	LESS-THAN OR SLANTED EQUAL TO WITH DOT INSIDE	
U+2A81	LESS-THAN OR SLANTED EQUAL TO WITH DOT ABOVE	
U+2A83	LESS-THAN OR SLANTED EQUAL TO WITH DOT ABOVE RIGHT	

U+2A85	LESS-THAN OR APPROXIMATE
U+2A87	LESS-THAN AND SINGLE-LINE NOT EQUAL TO
U+2A89	LESS-THAN AND NOT APPROXIMATE
U+2A8D	LESS-THAN ABOVE SIMILAR OR EQUAL
U+2A95	SLANTED EQUAL TO OR LESS-THAN
U+2A97	SLANTED EQUAL TO OR LESS-THAN WITH DOT INSIDE
U+2A99	DOUBLE-LINE EQUAL TO OR LESS-THAN
U+2A9B	DOUBLE-LINE SLANTED EQUAL TO OR LESS-THAN
U+2A9D	SIMILAR OR LESS-THAN
U+2A9F	SIMILAR ABOVE LESS-THAN ABOVE EQUALS SIGN
U+2AA1	DOUBLE NESTED LESS-THAN
U+2AA3	DOUBLE NESTED LESS-THAN WITH UNDERBAR
U+2AA6	LESS-THAN CLOSED BY CURVE
U+2AA8	LESS-THAN CLOSED BY CURVE ABOVE SLANTED EQUAL
U+2AF7	TRIPLE NESTED LESS-THAN
U+2AF9	DOUBLE-LINE SLANTED LESS-THAN OR EQUAL TO
U+FE64	SMALL LESS-THAN SIGN
U+FF1C	FULLWIDTH LESS-THAN SIGN

The following are greater-than operators. They may be mixed and chained with each other and with equivalence operators (see Section D.3.1).

U+003E	GREATER-THAN SIGN	$>$	GT
U+2265	GREATER-THAN OR EQUAL TO	$\geq$	>= GE
U+2267	GREATER-THAN OVER EQUAL TO	$\geqq$	
U+2269	GREATER-THAN BUT NOT EQUAL TO	$\gneqq$	
U+226B	MUCH GREATER-THAN	$\gg$	>>
U+2273	GREATER-THAN OR EQUIVALENT TO	$\gtrsim$	
U+22D7	GREATER-THAN WITH DOT	$\gtrdot$	DOTGT
U+22D9	VERY MUCH GREATER-THAN	$\ggg$	>>>
U+22DD	EQUAL TO OR GREATER-THAN		
U+22E7	GREATER-THAN BUT NOT EQUIVALENT TO	$\gnsim$	
U+29C1	CIRCLED GREATER-THAN		
U+2A7A	GREATER-THAN WITH CIRCLE INSIDE		
U+2A7C	GREATER-THAN WITH QUESTION MARK ABOVE		
U+2A7E	GREATER-THAN OR SLANTED EQUAL TO		
U+2A80	GREATER-THAN OR SLANTED EQUAL TO WITH DOT INSIDE		
U+2A82	GREATER-THAN OR SLANTED EQUAL TO WITH DOT ABOVE		
U+2A84	GREATER-THAN OR SLANTED EQUAL TO WITH DOT ABOVE LEFT		
U+2A86	GREATER-THAN OR APPROXIMATE		
U+2A88	GREATER-THAN AND SINGLE-LINE NOT EQUAL TO		
U+2A8A	GREATER-THAN AND NOT APPROXIMATE		
U+2A8E	GREATER-THAN ABOVE SIMILAR OR EQUAL		
U+2A96	SLANTED EQUAL TO OR GREATER-THAN		
U+2A98	SLANTED EQUAL TO OR GREATER-THAN WITH DOT INSIDE		
U+2A9A	DOUBLE-LINE EQUAL TO OR GREATER-THAN		
U+2A9C	DOUBLE-LINE SLANTED EQUAL TO OR GREATER-THAN		
U+2A9E	SIMILAR OR GREATER-THAN		
U+2AA0	SIMILAR ABOVE GREATER-THAN ABOVE EQUALS SIGN		
U+2AA2	DOUBLE NESTED GREATER-THAN		
U+2AA7	GREATER-THAN CLOSED BY CURVE		
U+2AA9	GREATER-THAN CLOSED BY CURVE ABOVE SLANTED EQUAL		
U+2AF8	TRIPLE NESTED GREATER-THAN		

U+2AFA	DOUBLE-LINE SLANTED GREATER-THAN OR EQUAL TO		
U+FE65	SMALL GREATER-THAN SIGN		
U+FF1E	FULLWIDTH GREATER-THAN SIGN		

The following are miscellaneous plain comparison operators. They may not be mixed or chained.

U+226E	NOT LESS-THAN	≮	NLT
U+226F	NOT GREATER-THAN	≯	NGT
U+2270	NEITHER LESS-THAN NOR EQUAL TO	≰	NLE
U+2271	NEITHER GREATER-THAN NOR EQUAL TO	≱	NGE
U+2274	NEITHER LESS-THAN NOR EQUIVALENT TO	≴	
U+2275	NEITHER GREATER-THAN NOR EQUIVALENT TO	≵	
U+2276	LESS-THAN OR GREATER-THAN	≶	
U+2277	GREATER-THAN OR LESS-THAN	≷	
U+2278	NEITHER LESS-THAN NOR GREATER-THAN		
U+2279	NEITHER GREATER-THAN NOR LESS-THAN		
U+22DA	LESS-THAN EQUAL TO OR GREATER-THAN	⋚	
U+22DB	GREATER-THAN EQUAL TO OR LESS-THAN	⋛	
U+2A8B	LESS-THAN ABOVE DOUBLE-LINE EQUAL ABOVE GREATER-THAN		
U+2A8C	GREATER-THAN ABOVE DOUBLE-LINE EQUAL ABOVE LESS-THAN		
U+2A8F	LESS-THAN ABOVE SIMILAR ABOVE GREATER-THAN		
U+2A90	GREATER-THAN ABOVE SIMILAR ABOVE LESS-THAN		
U+2A91	LESS-THAN ABOVE GREATER-THAN ABOVE DOUBLE-LINE EQUAL		
U+2A92	GREATER-THAN ABOVE LESS-THAN ABOVE DOUBLE-LINE EQUAL		
U+2A93	LESS-THAN ABOVE SLANTED EQUAL ABOVE GREATER-THAN ABOVE SLANTED EQUAL		
U+2A94	GREATER-THAN ABOVE SLANTED EQUAL ABOVE LESS-THAN ABOVE SLANTED EQUAL		
U+2AA4	GREATER-THAN OVERLAPPING LESS-THAN		
U+2AA5	GREATER-THAN BESIDE LESS-THAN		

The following is not really a comparison operator, but it is convenient to list it here because it also has lower precedence than any operator listed in Sections D.2.1, D.2.2, and D.2.3:

U+003A	COLON	:

D.3.3 Set Comparison Operators

Every operator listed in this section has lower precedence than any operator listed in Section D.2.4.

The following are subset comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section D.3.1).

U+2282	SUBSET OF	⊂	SUBSET
U+2286	SUBSET OF OR EQUAL TO	⊆	SUBSETEQ
U+228A	SUBSET OF WITH NOT EQUAL TO	⊊	SUBSETNEQ
U+22D0	DOUBLE SUBSET	⋐	SUBSUB
U+27C3	OPEN SUBSET		
U+2ABD	SUBSET WITH DOT		
U+2ABF	SUBSET WITH PLUS SIGN BELOW		
U+2AC1	SUBSET WITH MULTIPLICATION SIGN BELOW		

U+2AC3	SUBSET OF OR EQUAL TO WITH DOT ABOVE		
U+2AC5	SUBSET OF ABOVE EQUALS SIGN		
U+2AC7	SUBSET OF ABOVE TILDE OPERATOR		
U+2AC9	SUBSET OF ABOVE ALMOST EQUAL TO		
U+2ACB	SUBSET OF ABOVE NOT EQUAL TO		
U+2ACF	CLOSED SUBSET		
U+2AD1	CLOSED SUBSET OR EQUAL TO		
U+2AD5	SUBSET ABOVE SUBSET		

The following are superset comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section D.3.1).

U+2283	SUPERSET OF	⊃	SUPSET
U+2287	SUPERSET OF OR EQUAL TO	⊇	SUPSETEQ
U+228B	SUPERSET OF WITH NOT EQUAL TO	⊋	SUPSETNEQ
U+22D1	DOUBLE SUPERSET	⋑	SUPSUP
U+27C4	OPEN SUPERSET		
U+2ABE	SUPERSET WITH DOT		
U+2AC0	SUPERSET WITH PLUS SIGN BELOW		
U+2AC2	SUPERSET WITH MULTIPLICATION SIGN BELOW		
U+2AC4	SUPERSET OF OR EQUAL TO WITH DOT ABOVE		
U+2AC6	SUPERSET OF ABOVE EQUALS SIGN		
U+2AC8	SUPERSET OF ABOVE TILDE OPERATOR		
U+2ACA	SUPERSET OF ABOVE ALMOST EQUAL TO		
U+2ACC	SUPERSET OF ABOVE NOT EQUAL TO		
U+2AD0	CLOSED SUPERSET		
U+2AD2	CLOSED SUPERSET OR EQUAL TO		
U+2AD6	SUPERSET ABOVE SUPERSET		

The following are miscellaneous set comparison operators. They may not be mixed or chained.

U+2284	NOT A SUBSET OF	⊄	NSUBSET
U+2285	NOT A SUPERSET OF	⊅	NSUPSET
U+2288	NEITHER A SUBSET OF NOR EQUAL TO	⊈	NSUBSETEQ
U+2289	NEITHER A SUPERSET OF NOR EQUAL TO	⊉	NSUPSETEQ
U+2AD3	SUBSET ABOVE SUPERSET		
U+2AD4	SUPERSET ABOVE SUBSET		
U+2AD7	SUPERSET BESIDE SUBSET		
U+2AD8	SUPERSET BESIDE AND JOINED BY DASH WITH SUBSET		

D.3.4 Square Comparison Operators

Every operator listed in this section has lower precedence than any operator listed in Section D.2.5.

The following are square "image of" comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section D.3.1).

U+228F	SQUARE IMAGE OF	⊏	SQSUBSET
U+2291	SQUARE IMAGE OF OR EQUAL TO	⊑	SQSUBSETEQ
U+22E4	SQUARE IMAGE OF OR NOT EQUAL TO		

The following are square “original of” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section D.3.1).

U+2290	SQUARE ORIGINAL OF	⊐ SQSUPSET
U+2292	SQUARE ORIGINAL OF OR EQUAL TO	⊒ SQSUPSETEQ
U+22E5	SQUARE ORIGINAL OF OR NOT EQUAL TO	

The following are miscellaneous square comparison operators. They may not be mixed or chained.

U+22E2	NOT SQUARE IMAGE OF OR EQUAL TO	⋢
U+22E3	NOT SQUARE ORIGINAL OF OR EQUAL TO	⋣

D.3.5 Curly Comparison Operators

Every operator listed in this section has lower precedence than any operator listed in Section D.2.6.

The following are curly “precedes” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section D.3.1).

U+227A	PRECEDES	≺ PREC
U+227C	PRECEDES OR EQUAL TO	≼ PRECEQ
U+227E	PRECEDES OR EQUIVALENT TO	≾ PRECSIM
U+22B0	PRECEDES UNDER RELATION	
U+22DE	EQUAL TO OR PRECEDES	⋞ EQPREC
U+22E8	PRECEDES BUT NOT EQUIVALENT TO	⋨ PRECNSIM
U+2AAF	PRECEDES ABOVE SINGLE-LINE EQUALS SIGN	
U+2AB1	PRECEDES ABOVE SINGLE-LINE NOT EQUAL TO	
U+2AB3	PRECEDES ABOVE EQUALS SIGN	
U+2AB5	PRECEDES ABOVE NOT EQUAL TO	
U+2AB7	PRECEDES ABOVE ALMOST EQUAL TO	
U+2AB9	PRECEDES ABOVE NOT ALMOST EQUAL TO	
U+2ABB	DOUBLE PRECEDES	

The following are curly “succeeds” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section D.3.1).

U+227B	SUCCEEDS	≻ SUCC
U+227D	SUCCEEDS OR EQUAL TO	≽ SUCCEQ
U+227F	SUCCEEDS OR EQUIVALENT TO	≿ SUCCSIM
U+22B1	SUCCEEDS UNDER RELATION	
U+22DF	EQUAL TO OR SUCCEEDS	⋟ EQSUCC
U+22E9	SUCCEEDS BUT NOT EQUIVALENT TO	⋩ SUCCNSIM
U+2AB0	SUCCEEDS ABOVE SINGLE-LINE EQUALS SIGN	
U+2AB2	SUCCEEDS ABOVE SINGLE-LINE NOT EQUAL TO	
U+2AB4	SUCCEEDS ABOVE EQUALS SIGN	
U+2AB6	SUCCEEDS ABOVE NOT EQUAL TO	
U+2AB8	SUCCEEDS ABOVE ALMOST EQUAL TO	
U+2ABA	SUCCEEDS ABOVE NOT ALMOST EQUAL TO	
U+2ABC	DOUBLE SUCCEEDS	

The following are miscellaneous curly comparison operators. They may not be mixed or chained.

U+2280	DOES NOT PRECEDE	⊀ NPREC
U+2281	DOES NOT SUCCEED	⊁ NSUCC
U+22E0	DOES NOT PRECEDE OR EQUAL	⋠
U+22E1	DOES NOT SUCCEED OR EQUAL	⋡

D.3.6 Triangular Comparison Operators

The following are triangular "subgroup" comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section D.3.1).

U+22B2	NORMAL SUBGROUP OF	⊲
U+22B4	NORMAL SUBGROUP OF OR EQUAL TO	⊴

The following are triangular "contains as subgroup" comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section D.3.1).

U+22B3	CONTAINS AS NORMAL SUBGROUP	⊳
U+22B5	CONTAINS AS NORMAL SUBGROUP OR EQUAL TO	⊵

The following are miscellaneous triangular comparison operators. They may not be mixed or chained.

U+22EA	NOT NORMAL SUBGROUP OF	⋪
U+22EB	DOES NOT CONTAIN AS NORMAL SUBGROUP	⋫
U+22EC	NOT NORMAL SUBGROUP OF OR EQUAL TO	⋬
U+22ED	DOES NOT CONTAIN AS NORMAL SUBGROUP OR EQUAL	⋭

D.3.7 Chickenfoot Comparison Operators

The following are chickenfoot "smaller than" comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section D.3.1).

U+2AAA	SMALLER THAN	⪪ SMALLER
U+2AAC	SMALLER THAN OR EQUAL TO	⪬ SMALLEREQ

The following are chickenfoot "larger than" comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section D.3.1).

U+2AAB	LARGER THAN	⪫ LARGER
U+2AAD	LARGER THAN OR EQUAL TO	⪭ LARGEREQ

D.3.8 Miscellaneous Relational Operators

The following operators are considered to be relational operators, having higher precedence than certain boolean operators, as described in a later section.

U+2208	ELEMENT OF	∈	IN
U+2209	NOT AN ELEMENT OF	∉	NOTIN
U+220A	SMALL ELEMENT OF	∊	
U+220B	CONTAINS AS MEMBER	∋	CONTAINS
U+220C	DOES NOT CONTAIN AS MEMBER	∌	
U+220D	SMALL CONTAINS AS MEMBER	∍	
U+22F2	ELEMENT OF WITH LONG HORIZONTAL STROKE		
U+22F3	ELEMENT OF WITH VERTICAL BAR AT END OF HORIZONTAL STROKE		
U+22F4	SMALL ELEMENT OF WITH VERTICAL BAR AT END OF HORIZONTAL STROKE		
U+22F5	ELEMENT OF WITH DOT ABOVE	⋵	
U+22F6	ELEMENT OF WITH OVERBAR	⋶	
U+22F7	SMALL ELEMENT OF WITH OVERBAR	⋷	
U+22F8	ELEMENT OF WITH UNDERBAR	⋸	
U+22F9	ELEMENT OF WITH TWO HORIZONTAL STROKES		
U+22FA	CONTAINS WITH LONG HORIZONTAL STROKE		
U+22FB	CONTAINS WITH VERTICAL BAR AT END OF HORIZONTAL STROKE		
U+22FC	SMALL CONTAINS WITH VERTICAL BAR AT END OF HORIZONTAL STROKE		
U+22FD	CONTAINS WITH OVERBAR	⋽	
U+22FE	SMALL CONTAINS WITH OVERBAR	⋾	
U+22FF	Z NOTATION BAG MEMBERSHIP		

D.4 Boolean Operators

Every operator listed in this section has lower precedence than any operator listed in Section D.3.

The following are the Boolean conjunction operators:

U+2227	LOGICAL AND	∧	AND
U+27D1	AND WITH DOT		
U+2A51	LOGICAL AND WITH DOT ABOVE	⩑	
U+2A53	DOUBLE LOGICAL AND		
U+2A55	TWO INTERSECTING LOGICAL AND	⩕	
U+2A5A	LOGICAL AND WITH MIDDLE STEM		
U+2A5C	LOGICAL AND WITH HORIZONTAL DASH		
U+2A5E	LOGICAL AND WITH DOUBLE OVERBAR		
U+2A60	LOGICAL AND WITH DOUBLE UNDERBAR		

The following are the Boolean disjunction operators:

U+2228	LOGICAL OR	∨	OR
U+2A52	LOGICAL OR WITH DOT ABOVE	⩒	
U+2A54	DOUBLE LOGICAL OR		
U+2A56	TWO INTERSECTING LOGICAL OR	⩖	
U+2A5B	LOGICAL OR WITH MIDDLE STEM		
U+2A5D	LOGICAL OR WITH HORIZONTAL DASH		
U+2A62	LOGICAL OR WITH DOUBLE OVERBAR		
U+2A63	LOGICAL OR WITH DOUBLE UNDERBAR		

They each have lower precedence than any of the Booolean conjunction operators.

The following are miscellaneous Boolean operators:

U+2192	RIGHTWARDS ARROW	→ -> IMPLIES
U+2194	LEFT RIGHT ARROW	↔ <-> IFF
U+22BB	XOR	⊻
U+22BC	NAND	⊼
U+22BD	NOR⊽	

D.5 Other Operators

Each of the following operators has no defined precedence relationships to any of the other operators listed in this appendix.

U+0021	EXCLAMATION MARK	!
U+0023	NUMBER SIGN	#
U+0024	DOLLAR SIGN	$
U+0025	PERCENT SIGN	%
U+003F	QUESTION MARK	?
U+0040	COMMERCIAL AT	@
U+005E	CIRCUMFLEX	ˆ
U+007C	VERTICAL LINE	|
U+007E	TILDE	˜
U+00A1	INVERTED EXCLAMATION MARK	¡
U+00A2	CENT SIGN	CENTS
U+00A3	POUND SIGN	
U+00A4	CURRENCY SIGN	
U+00A5	YEN SIGN	
U+00A6	BROKEN BAR	
U+00AC	NOT SIGN	¬ NOT
U+00B0	DEGREE SIGN	° DEGREES
U+00BF	INVERTED QUESTION MARK	¿
U+2016	DOUBLE VERTICAL LINE	‖ ||
U+203C	DOUBLE EXCLAMATION MARK	‼ !!
U+2190	LEFTWARDS ARROW	← <-
U+2191	UPWARDS ARROW	↑ UPARROW
U+2193	DOWNWARDS ARROW	↓ DOWNARROW
U+2195	UP DOWN ARROW	↕ UPDOWNARROW
U+2196	NORTH WEST ARROW	↖ NWARROW
U+2197	NORTH EAST ARROW	↗ NEARROW
U+2198	SOUTH EAST ARROW	↘ SEARROW
U+2199	SOUTH WEST ARROW	↙ SWARROW
U+219A	LEFTWARDS ARROW WITH STROKE	↚ -/->
U+219B	RIGHTWARDS ARROW WITH STROKE	↛ <-/-
U+219C	LEFTWARDS WAVE ARROW	
U+219D	RIGHTWARDS WAVE ARROW	↝ LEADSTO
U+219E	LEFTWARDS TWO HEADED ARROW	
U+219F	UPWARDS TWO HEADED ARROW	
U+21A0	RIGHTWARDS TWO HEADED ARROW	
U+21A1	DOWNWARDS TWO HEADED ARROW	
U+21A2	LEFTWARDS ARROW WITH TAIL	

U+21A3	RIGHTWARDS ARROW WITH TAIL		
U+21A4	LEFTWARDS ARROW FROM BAR		
U+21A5	UPWARDS ARROW FROM BAR		
U+21A6	RIGHTWARDS ARROW FROM BAR	↦	MAPSTO |->
U+21A7	DOWNWARDS ARROW FROM BAR		
U+21A8	UP DOWN ARROW WITH BASE		
U+21A9	LEFTWARDS ARROW WITH HOOK		
U+21AA	RIGHTWARDS ARROW WITH HOOK		
U+21AB	LEFTWARDS ARROW WITH LOOP		
U+21AC	RIGHTWARDS ARROW WITH LOOP		
U+21AD	LEFT RIGHT WAVE ARROW		
U+21AE	LEFT RIGHT ARROW WITH STROKE		
U+21AF	DOWNWARDS ZIGZAG ARROW		
U+21B0	UPWARDS ARROW WITH TIP LEFTWARDS		
U+21B1	UPWARDS ARROW WITH TIP RIGHTWARDS		
U+21B2	DOWNWARDS ARROW WITH TIP LEFTWARDS		
U+21B3	DOWNWARDS ARROW WITH TIP RIGHTWARDS		
U+21B4	RIGHTWARDS ARROW WITH CORNER DOWNWARDS		
U+21B5	DOWNWARDS ARROW WITH CORNER LEFTWARDS		
U+21B6	ANTICLOCKWISE TOP SEMICIRCLE ARROW		
U+21B7	CLOCKWISE TOP SEMICIRCLE ARROW		
U+21B8	NORTH WEST ARROW TO LONG BAR		
U+21B9	LEFTWARDS ARROW TO BAR OVER RIGHTWARDS ARROW TO BAR		
U+21BA	ANTICLOCKWISE OPEN CIRCLE ARROW		
U+21BB	CLOCKWISE OPEN CIRCLE ARROW		
U+21BC	LEFTWARDS HARPOON WITH BARB UPWARDS	↼	LEFTHARPOONUP
U+21BD	LEFTWARDS HARPOON WITH BARB DOWNWARDS	↽	LEFTHARPOONDOWN
U+21BE	UPWARDS HARPOON WITH BARB RIGHTWARDS	↾	UPHARPOONRIGHT
U+21BF	UPWARDS HARPOON WITH BARB LEFTWARDS	↿	UPHARPOONLEFT
U+21C0	RIGHTWARDS HARPOON WITH BARB UPWARDS	⇀	RIGHTHARPOONUP
U+21C1	RIGHTWARDS HARPOON WITH BARB DOWNWARDS	⇁	RIGHTHARPOONDOWN
U+21C2	DOWNWARDS HARPOON WITH BARB RIGHTWARDS	⇂	DOWNHARPOONRIGHT
U+21C3	DOWNWARDS HARPOON WITH BARB LEFTWARDS	⇃	DOWNHARPOONLEFT
U+21C4	RIGHTWARDS ARROW OVER LEFTWARDS ARROW	⇄	RIGHTLEFTARROWS
U+21C5	UPWARDS ARROW LEFTWARDS OF DOWNWARDS ARROW		
U+21C6	LEFTWARDS ARROW OVER RIGHTWARDS ARROW	⇆	LEFTRIGHTARROWS
U+21C7	LEFTWARDS PAIRED ARROWS	⇇	LEFTLEFTARROWS
U+21C8	UPWARDS PAIRED ARROWS	⇈	UPUPARROWS
U+21C9	RIGHTWARDS PAIRED ARROWS	⇉	RIGHTRIGHTARROWS
U+21CA	DOWNWARDS PAIRED ARROWS	⇊	DOWNDOWNARROWS
U+21CB	LEFTWARDS HARPOON OVER RIGHTWARDS HARPOON		
U+21CC	RIGHTWARDS HARPOON OVER LEFTWARDS HARPOON	⇌	RIGHTLEFTHARPOONS
U+21CD	LEFTWARDS DOUBLE ARROW WITH STROKE	⇍	
U+21CE	LEFT RIGHT DOUBLE ARROW WITH STROKE	⇎	
U+21CF	RIGHTWARDS DOUBLE ARROW WITH STROKE	⇏	
U+21D0	LEFTWARDS DOUBLE ARROW	⇐	
U+21D1	UPWARDS DOUBLE ARROW	⇑	
U+21D2	RIGHTWARDS DOUBLE ARROW	⇒	=>
U+21D3	DOWNWARDS DOUBLE ARROW	⇓	
U+21D4	LEFT RIGHT DOUBLE ARROW	⇔	<=>
U+21D5	UP DOWN DOUBLE ARROW	⇕	
U+21D6	NORTH WEST DOUBLE ARROW		
U+21D7	NORTH EAST DOUBLE ARROW		

U+21D8	SOUTH EAST DOUBLE ARROW	
U+21D9	SOUTH WEST DOUBLE ARROW	
U+21DA	LEFTWARDS TRIPLE ARROW	⇚
U+21DB	RIGHTWARDS TRIPLE ARROW	⇛
U+21DC	LEFTWARDS SQUIGGLE ARROW	
U+21DD	RIGHTWARDS SQUIGGLE ARROW	⇝
U+21DE	UPWARDS ARROW WITH DOUBLE STROKE	
U+21DF	DOWNWARDS ARROW WITH DOUBLE STROKE	
U+21E0	LEFTWARDS DASHED ARROW	⇠
U+21E1	UPWARDS DASHED ARROW	
U+21E2	RIGHTWARDS DASHED ARROW	⇢
U+21E3	DOWNWARDS DASHED ARROW	
U+21E4	LEFTWARDS ARROW TO BAR	
U+21E5	RIGHTWARDS ARROW TO BAR	
U+21E6	LEFTWARDS WHITE ARROW	
U+21E7	UPWARDS WHITE ARROW	
U+21E8	RIGHTWARDS WHITE ARROW	
U+21E9	DOWNWARDS WHITE ARROW	
U+21EA	UPWARDS WHITE ARROW FROM BAR	
U+21EB	UPWARDS WHITE ARROW ON PEDESTAL	
U+21EC	UPWARDS WHITE ARROW ON PEDESTAL WITH HORIZONTAL BAR	
U+21ED	UPWARDS WHITE ARROW ON PEDESTAL WITH VERTICAL BAR	
U+21EE	UPWARDS WHITE DOUBLE ARROW	
U+21EF	UPWARDS WHITE DOUBLE ARROW ON PEDESTAL	
U+21F0	RIGHTWARDS WHITE ARROW FROM WALL	
U+21F1	NORTH WEST ARROW TO CORNER	
U+21F2	SOUTH EAST ARROW TO CORNER	
U+21F3	UP DOWN WHITE ARROW	
U+21F4	RIGHT ARROW WITH SMALL CIRCLE	
U+21F5	DOWNWARDS ARROW LEFTWARDS OF UPWARDS ARROW	
U+21F6	THREE RIGHTWARDS ARROWS	
U+21F7	LEFTWARDS ARROW WITH VERTICAL STROKE	
U+21F8	RIGHTWARDS ARROW WITH VERTICAL STROKE	
U+21F9	LEFT RIGHT ARROW WITH VERTICAL STROKE	
U+21FA	LEFTWARDS ARROW WITH DOUBLE VERTICAL STROKE	
U+21FB	RIGHTWARDS ARROW WITH DOUBLE VERTICAL STROKE	
U+21FC	LEFT RIGHT ARROW WITH DOUBLE VERTICAL STROKE	
U+21FD	LEFTWARDS OPEN-HEADED ARROW	
U+21FE	RIGHTWARDS OPEN-HEADED ARROW	
U+21FF	LEFT RIGHT OPEN-HEADED ARROW	
U+2201	COMPLEMENT	∁
U+2202	PARTIAL DIFFERENTIAL	∂ DEL
U+2204	THERE DOES NOT EXIST	∄
U+2206	INCREMENT	∆
U+220F	N-ARY PRODUCT	∏ PRODUCT
U+2210	N-ARY COPRODUCT	∐ COPRODUCT
U+2211	N-ARY SUMMATION	∑ SUM
U+2218	RING OPERATOR	∘ CIRC RING COMPOSE
U+2219	BULLET OPERATOR	∙ BULLET
U+221A	SQUARE ROOT	√ SQRT
U+221B	CUBE ROOT	CBRT
U+221C	FOURTH ROOT	FOURTHROOT
U+221D	PROPORTIONAL TO	∝ PROPTO

U+2223	DIVIDES	∣ DIVIDES
U+2224	DOES NOT DIVIDE	∤
U+2225	PARALLEL TO	∥ PARALLEL
U+2226	NOT PARALLEL TO	∦ NPARALLEL
U+222B	INTEGRAL	∫
U+222C	DOUBLE INTEGRAL	
U+222D	TRIPLE INTEGRAL	
U+222E	CONTOUR INTEGRAL	∮
U+222F	SURFACE INTEGRAL	
U+2230	VOLUME INTEGRAL	
U+2231	CLOCKWISE INTEGRAL	
U+2232	CLOCKWISE CONTOUR INTEGRAL	
U+2233	ANTICLOCKWISE CONTOUR INTEGRAL	
U+2234	THEREFORE	∴
U+2235	BECAUSE	∵
U+2236	RATIO	
U+2237	PROPORTION	
U+2239	EXCESS	
U+223A	GEOMETRIC PROPORTION	
U+223B	HOMOTHETIC	
U+223C	TILDE OPERATOR	∼
U+223D	REVERSED TILDE	∽
U+223E	INVERTED LAZY S	
U+223F	SINE WAVE	
U+2240	WREATH PRODUCT	≀ WREATH
U+2241	NOT TILDE	≁
U+224B	TRIPLE TILDE	
U+224F	DIFFERENCE BETWEEN	≏ BUMPEQ
U+2250	APPROACHES THE LIMIT	≐ DOTEQ
U+2254	COLON EQUALS	≔ :=
U+2255	EQUALS COLON	≕ =:
U+2258	CORRESPONDS TO	
U+2259	ESTIMATES	
U+225A	EQUIANGULAR TO	
U+225E	MEASURED BY	
U+226C	BETWEEN	≬
U+228C	MULTISET	
U+229A	CIRCLED RING OPERATOR	⊚ CIRCLEDRING
U+229D	CIRCLED DASH	⊝
U+22A2	RIGHT TACK	⊢ VDASH TURNSTILE
U+22A3	LEFT TACK	⊣ DASHV
U+22A4	DOWN TACK	⊤ TOP
U+22A5	UP TACK	⊥ PERP BOTTOM
U+22A6	ASSERTION	⊦
U+22A7	MODELS	⊧
U+22A8	TRUE	⊨
U+22A9	FORCES	⊩
U+22AA	TRIPLE VERTICAL BAR RIGHT TURNSTILE	⊪
U+22AB	DOUBLE VERTICAL BAR DOUBLE RIGHT TURNSTILE	
U+22AC	DOES NOT PROVE	⊬
U+22AD	NOT TRUE	
U+22AE	DOES NOT FORCE	⊮
U+22AF	NEGATED DOUBLE VERTICAL BAR DOUBLE RIGHT TURNSTILE	⊯

U+22B6	ORIGINAL OF		
U+22B7	IMAGE OF		
U+22B8	MULTIMAP	⊸	
U+22B9	HERMITIAN CONJUGATE MATRIX		
U+22BA	INTERCALATE	⊺	
U+22BE	RIGHT ANGLE WITH ARC		
U+22BF	RIGHT TRIANGLE		
U+22C0	N-ARY LOGICAL AND	⋀	BIGAND ALL
U+22C1	N-ARY LOGICAL OR	⋁	BIGOR ANY
U+22C2	N-ARY INTERSECTION	⋂	BIGCAP BIGINTERSECT
U+22C3	N-ARY UNION	⋃	BIGCUP BIGUNION
U+22C4	DIAMOND OPERATOR	⋄	DIAMOND
U+22C6	STAR OPERATOR	⋆	STAR
U+22C7	DIVISION TIMES	⋇	
U+22C8	BOWTIE	⋈	
U+22C9	LEFT NORMAL FACTOR SEMIDIRECT PRODUCT	⋉	
U+22CA	RIGHT NORMAL FACTOR SEMIDIRECT PRODUCT	⋊	
U+22CB	LEFT SEMIDIRECT PRODUCT	⋋	
U+22CC	RIGHT SEMIDIRECT PRODUCT	⋌	
U+22D4	PITCHFORK	⋔	
U+22EE	VERTICAL ELLIPSIS		
U+22EF	MIDLINE HORIZONTAL ELLIPSIS		
U+22F0	UP RIGHT DIAGONAL ELLIPSIS		
U+22F1	DOWN RIGHT DIAGONAL ELLIPSIS		
U+27C0	THREE DIMENSIONAL ANGLE		
U+27C1	WHITE TRIANGLE CONTAINING SMALL WHITE TRIANGLE		
U+27C2	PERPENDICULAR		PERP
U+27D0	WHITE DIAMOND WITH CENTRED DOT		
U+27D2	ELEMENT OF OPENING UPWARDS		
U+27D3	LOWER RIGHT CORNER WITH DOT		
U+27D4	UPPER LEFT CORNER WITH DOT		
U+27D5	LEFT OUTER JOIN		
U+27D6	RIGHT OUTER JOIN		
U+27D7	FULL OUTER JOIN		
U+27D8	LARGE UP TACK		
U+27D9	LARGE DOWN TACK		
U+27DA	LEFT AND RIGHT DOUBLE TURNSTILE		
U+27DB	LEFT AND RIGHT TACK		
U+27DC	LEFT MULTIMAP		
U+27DD	LONG RIGHT TACK		
U+27DE	LONG LEFT TACK		
U+27DF	UP TACK WITH CIRCLE ABOVE		
U+27E0	LOZENGE DIVIDED BY HORIZONTAL RULE		
U+27E1	WHITE CONCAVE-SIDED DIAMOND		
U+27E2	WHITE CONCAVE-SIDED DIAMOND WITH LEFTWARDS TICK		
U+27E3	WHITE CONCAVE-SIDED DIAMOND WITH RIGHTWARDS TICK		
U+27E4	WHITE SQUARE WITH LEFTWARDS TICK		
U+27E5	WHITE SQUARE WITH RIGHTWARDS TICK		
U+27F0	UPWARDS QUADRUPLE ARROW		
U+27F1	DOWNWARDS QUADRUPLE ARROW		
U+27F2	ANTICLOCKWISE GAPPED CIRCLE ARROW		
U+27F3	CLOCKWISE GAPPED CIRCLE ARROW		
U+27F4	RIGHT ARROW WITH CIRCLED PLUS		

U+27F5	LONG LEFTWARDS ARROW
U+27F6	LONG RIGHTWARDS ARROW
U+27F7	LONG LEFT RIGHT ARROW
U+27F8	LONG LEFTWARDS DOUBLE ARROW
U+27F9	LONG RIGHTWARDS DOUBLE ARROW
U+27FA	LONG LEFT RIGHT DOUBLE ARROW
U+27FB	LONG LEFTWARDS ARROW FROM BAR
U+27FC	LONG RIGHTWARDS ARROW FROM BAR
U+27FD	LONG LEFTWARDS DOUBLE ARROW FROM BAR
U+27FE	LONG RIGHTWARDS DOUBLE ARROW FROM BAR
U+27FF	LONG RIGHTWARDS SQUIGGLE ARROW
U+2900	RIGHTWARDS TWO-HEADED ARROW WITH VERTICAL STROKE
U+2901	RIGHTWARDS TWO-HEADED ARROW WITH DOUBLE VERTICAL STROKE
U+2902	LEFTWARDS DOUBLE ARROW WITH VERTICAL STROKE
U+2903	RIGHTWARDS DOUBLE ARROW WITH VERTICAL STROKE
U+2904	LEFT RIGHT DOUBLE ARROW WITH VERTICAL STROKE
U+2905	RIGHTWARDS TWO-HEADED ARROW FROM BAR
U+2906	LEFTWARDS DOUBLE ARROW FROM BAR
U+2907	RIGHTWARDS DOUBLE ARROW FROM BAR
U+2908	DOWNWARDS ARROW WITH HORIZONTAL STROKE
U+2909	UPWARDS ARROW WITH HORIZONTAL STROKE
U+290A	UPWARDS TRIPLE ARROW
U+290B	DOWNWARDS TRIPLE ARROW
U+290C	LEFTWARDS DOUBLE DASH ARROW
U+290D	RIGHTWARDS DOUBLE DASH ARROW
U+290E	LEFTWARDS TRIPLE DASH ARROW
U+290F	RIGHTWARDS TRIPLE DASH ARROW
U+2910	RIGHTWARDS TWO-HEADED TRIPLE DASH ARROW
U+2911	RIGHTWARDS ARROW WITH DOTTED STEM
U+2912	UPWARDS ARROW TO BAR
U+2913	DOWNWARDS ARROW TO BAR
U+2914	RIGHTWARDS ARROW WITH TAIL WITH VERTICAL STROKE
U+2915	RIGHTWARDS ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE
U+2916	RIGHTWARDS TWO-HEADED ARROW WITH TAIL
U+2917	RIGHTWARDS TWO-HEADED ARROW WITH TAIL WITH VERTICAL STROKE
U+2918	RIGHTWARDS TWO-HEADED ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE
U+2919	LEFTWARDS ARROW-TAIL
U+291A	RIGHTWARDS ARROW-TAIL
U+291B	LEFTWARDS DOUBLE ARROW-TAIL
U+291C	RIGHTWARDS DOUBLE ARROW-TAIL
U+291D	LEFTWARDS ARROW TO BLACK DIAMOND
U+291E	RIGHTWARDS ARROW TO BLACK DIAMOND
U+291F	LEFTWARDS ARROW FROM BAR TO BLACK DIAMOND
U+2920	RIGHTWARDS ARROW FROM BAR TO BLACK DIAMOND
U+2921	NORTH WEST AND SOUTH EAST ARROW
U+2922	NORTH EAST AND SOUTH WEST ARROW
U+2923	NORTH WEST ARROW WITH HOOK
U+2924	NORTH EAST ARROW WITH HOOK
U+2925	SOUTH EAST ARROW WITH HOOK
U+2926	SOUTH WEST ARROW WITH HOOK
U+2927	NORTH WEST ARROW AND NORTH EAST ARROW
U+2928	NORTH EAST ARROW AND SOUTH EAST ARROW
U+2929	SOUTH EAST ARROW AND SOUTH WEST ARROW

U+292A	SOUTH WEST ARROW AND NORTH WEST ARROW
U+292B	RISING DIAGONAL CROSSING FALLING DIAGONAL
U+292C	FALLING DIAGONAL CROSSING RISING DIAGONAL
U+292D	SOUTH EAST ARROW CROSSING NORTH EAST ARROW
U+292E	NORTH EAST ARROW CROSSING SOUTH EAST ARROW
U+292F	FALLING DIAGONAL CROSSING NORTH EAST ARROW
U+2930	RISING DIAGONAL CROSSING SOUTH EAST ARROW
U+2931	NORTH EAST ARROW CROSSING NORTH WEST ARROW
U+2932	NORTH WEST ARROW CROSSING NORTH EAST ARROW
U+2933	WAVE ARROW POINTING DIRECTLY RIGHT
U+2934	ARROW POINTING RIGHTWARDS THEN CURVING UPWARDS
U+2935	ARROW POINTING RIGHTWARDS THEN CURVING DOWNWARDS
U+2936	ARROW POINTING DOWNWARDS THEN CURVING LEFTWARDS
U+2937	ARROW POINTING DOWNWARDS THEN CURVING RIGHTWARDS
U+2938	RIGHT-SIDE ARC CLOCKWISE ARROW
U+2939	LEFT-SIDE ARC ANTICLOCKWISE ARROW
U+293A	TOP ARC ANTICLOCKWISE ARROW
U+293B	BOTTOM ARC ANTICLOCKWISE ARROW
U+293C	TOP ARC CLOCKWISE ARROW WITH MINUS
U+293D	TOP ARC ANTICLOCKWISE ARROW WITH PLUS
U+293E	LOWER RIGHT SEMICIRCULAR CLOCKWISE ARROW
U+293F	LOWER LEFT SEMICIRCULAR ANTICLOCKWISE ARROW
U+2940	ANTICLOCKWISE CLOSED CIRCLE ARROW
U+2941	CLOCKWISE CLOSED CIRCLE ARROW
U+2942	RIGHTWARDS ARROW ABOVE SHORT LEFTWARDS ARROW
U+2943	LEFTWARDS ARROW ABOVE SHORT RIGHTWARDS ARROW
U+2944	SHORT RIGHTWARDS ARROW ABOVE LEFTWARDS ARROW
U+2945	RIGHTWARDS ARROW WITH PLUS BELOW
U+2946	LEFTWARDS ARROW WITH PLUS BELOW
U+2947	RIGHTWARDS ARROW THROUGH X
U+2948	LEFT RIGHT ARROW THROUGH SMALL CIRCLE
U+2949	UPWARDS TWO-HEADED ARROW FROM SMALL CIRCLE
U+294A	LEFT BARB UP RIGHT BARB DOWN HARPOON
U+294B	LEFT BARB DOWN RIGHT BARB UP HARPOON
U+294C	UP BARB RIGHT DOWN BARB LEFT HARPOON
U+294D	UP BARB LEFT DOWN BARB RIGHT HARPOON
U+294E	LEFT BARB UP RIGHT BARB UP HARPOON
U+294F	UP BARB RIGHT DOWN BARB RIGHT HARPOON
U+2950	LEFT BARB DOWN RIGHT BARB DOWN HARPOON
U+2951	UP BARB LEFT DOWN BARB LEFT HARPOON
U+2952	LEFTWARDS HARPOON WITH BARB UP TO BAR
U+2953	RIGHTWARDS HARPOON WITH BARB UP TO BAR
U+2954	UPWARDS HARPOON WITH BARB RIGHT TO BAR
U+2955	DOWNWARDS HARPOON WITH BARB RIGHT TO BAR
U+2956	LEFTWARDS HARPOON WITH BARB DOWN TO BAR
U+2957	RIGHTWARDS HARPOON WITH BARB DOWN TO BAR
U+2958	UPWARDS HARPOON WITH BARB LEFT TO BAR
U+2959	DOWNWARDS HARPOON WITH BARB LEFT TO BAR
U+295A	LEFTWARDS HARPOON WITH BARB UP FROM BAR
U+295B	RIGHTWARDS HARPOON WITH BARB UP FROM BAR
U+295C	UPWARDS HARPOON WITH BARB RIGHT FROM BAR
U+295D	DOWNWARDS HARPOON WITH BARB RIGHT FROM BAR
U+295E	LEFTWARDS HARPOON WITH BARB DOWN FROM BAR

U+295F	RIGHTWARDS HARPOON WITH BARB DOWN FROM BAR
U+2960	UPWARDS HARPOON WITH BARB LEFT FROM BAR
U+2961	DOWNWARDS HARPOON WITH BARB LEFT FROM BAR
U+2962	LEFTWARDS HARPOON WITH BARB UP ABOVE LEFTWARDS HARPOON WITH BARB DOWN
U+2963	UPWARDS HARPOON WITH BARB LEFT BESIDE UPWARDS HARPOON WITH BARB RIGHT
U+2964	RIGHTWARDS HARPOON WITH BARB UP ABOVE RIGHTWARDS HARPOON WITH BARB DOWN
U+2965	DOWNWARDS HARPOON WITH BARB LEFT BESIDE DOWNWARDS HARPOON WITH BARB RIGHT
U+2966	LEFTWARDS HARPOON WITH BARB UP ABOVE RIGHTWARDS HARPOON WITH BARB UP
U+2967	LEFTWARDS HARPOON WITH BARB DOWN ABOVE RIGHTWARDS HARPOON WITH BARB DOWN
U+2968	RIGHTWARDS HARPOON WITH BARB UP ABOVE LEFTWARDS HARPOON WITH BARB UP
U+2969	RIGHTWARDS HARPOON WITH BARB DOWN ABOVE LEFTWARDS HARPOON WITH BARB DOWN
U+296A	LEFTWARDS HARPOON WITH BARB UP ABOVE LONG DASH
U+296B	LEFTWARDS HARPOON WITH BARB DOWN BELOW LONG DASH
U+296C	RIGHTWARDS HARPOON WITH BARB UP ABOVE LONG DASH
U+296D	RIGHTWARDS HARPOON WITH BARB DOWN BELOW LONG DASH
U+296E	UPWARDS HARPOON WITH BARB LEFT BESIDE DOWNWARDS HARPOON WITH BARB RIGHT
U+296F	DOWNWARDS HARPOON WITH BARB LEFT BESIDE UPWARDS HARPOON WITH BARB RIGHT
U+2970	RIGHT DOUBLE ARROW WITH ROUNDED HEAD
U+2971	EQUALS SIGN ABOVE RIGHTWARDS ARROW
U+2972	TILDE OPERATOR ABOVE RIGHTWARDS ARROW
U+2973	LEFTWARDS ARROW ABOVE TILDE OPERATOR
U+2974	RIGHTWARDS ARROW ABOVE TILDE OPERATOR
U+2975	RIGHTWARDS ARROW ABOVE ALMOST EQUAL TO
U+2976	LESS-THAN ABOVE LEFTWARDS ARROW
U+2977	LEFTWARDS ARROW THROUGH LESS-THAN
U+2978	GREATER-THAN ABOVE RIGHTWARDS ARROW
U+2979	SUBSET ABOVE RIGHTWARDS ARROW
U+297A	LEFTWARDS ARROW THROUGH SUBSET
U+297B	SUPERSET ABOVE LEFTWARDS ARROW
U+297C	LEFT FISH TAIL
U+297D	RIGHT FISH TAIL
U+297E	UP FISH TAIL
U+297F	DOWN FISH TAIL
U+2980	TRIPLE VERTICAL BAR DELIMITER
U+2981	Z NOTATION SPOT
U+2982	Z NOTATION TYPE COLON
U+2999	DOTTED FENCE
U+299A	VERTICAL ZIGZAG LINE
U+299B	MEASURED ANGLE OPENING LEFT
U+299C	RIGHT ANGLE VARIANT WITH SQUARE
U+299D	MEASURED RIGHT ANGLE WITH DOT
U+299E	ANGLE WITH S INSIDE
U+299F	ACUTE ANGLE
U+29A0	SPHERICAL ANGLE OPENING LEFT
U+29A1	SPHERICAL ANGLE OPENING UP
U+29A2	TURNED ANGLE
U+29A3	REVERSED ANGLE
U+29A4	ANGLE WITH UNDERBAR
U+29A5	REVERSED ANGLE WITH UNDERBAR
U+29A6	OBLIQUE ANGLE OPENING UP
U+29A7	OBLIQUE ANGLE OPENING DOWN
U+29A8	MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING UP AND RIGHT
U+29A9	MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING UP AND LEFT

U+29AA	MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING DOWN AND RIGHT
U+29AB	MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING DOWN AND LEFT
U+29AC	MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING RIGHT AND UP
U+29AD	MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING LEFT AND UP
U+29AE	MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING RIGHT AND DOWN
U+29AF	MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING LEFT AND DOWN
U+29B0	REVERSED EMPTY SET
U+29B1	EMPTY SET WITH OVERBAR
U+29B2	EMPTY SET WITH SMALL CIRCLE ABOVE
U+29B3	EMPTY SET WITH RIGHT ARROW ABOVE
U+29B4	EMPTY SET WITH LEFT ARROW ABOVE
U+29B5	CIRCLE WITH HORIZONTAL BAR
U+29B6	CIRCLED VERTICAL BAR
U+29B7	CIRCLED PARALLEL
U+29B9	CIRCLED PERPENDICULAR
U+29BA	CIRCLE DIVIDED BY HORIZONTAL BAR AND TOP HALF DIVIDED BY VERTICAL BAR
U+29BB	CIRCLE WITH SUPERIMPOSED X
U+29BD	UP ARROW THROUGH CIRCLE
U+29BE	CIRCLED WHITE BULLET
U+29BF	CIRCLED BULLET
U+29C2	CIRCLE WITH SMALL CIRCLE TO THE RIGHT
U+29C3	CIRCLE WITH TWO HORIZONTAL STROKES TO THE RIGHT
U+29C5	SQUARED FALLING DIAGONAL SLASH
U+29C7	SQUARED SMALL CIRCLE
U+29C8	SQUARED SQUARE
U+29C9	TWO JOINED SQUARES
U+29CA	TRIANGLE WITH DOT ABOVE
U+29CB	TRIANGLE WITH UNDERBAR
U+29CC	S IN TRIANGLE
U+29CD	TRIANGLE WITH SERIFS AT BOTTOM
U+29CE	RIGHT TRIANGLE ABOVE LEFT TRIANGLE
U+29CF	LEFT TRIANGLE BESIDE VERTICAL BAR
U+29D0	VERTICAL BAR BESIDE RIGHT TRIANGLE
U+29D1	BOWTIE WITH LEFT HALF BLACK
U+29D2	BOWTIE WITH RIGHT HALF BLACK
U+29D3	BLACK BOWTIE
U+29D6	WHITE HOURGLASS
U+29D7	BLACK HOURGLASS
U+29DC	INCOMPLETE INFINITY
U+29DD	TIE OVER INFINITY
U+29DE	INFINITY NEGATED WITH VERTICAL BAR
U+29DF	DOUBLE-ENDED MULTIMAP
U+29E0	SQUARE WITH CONTOURED OUTLINE
U+29E1	INCREASES AS
U+29E2	SHUFFLE PRODUCT
U+29E6	GLEICH STARK
U+29E7	THERMODYNAMIC
U+29E8	DOWN-POINTING TRIANGLE WITH LEFT HALF BLACK
U+29E9	DOWN-POINTING TRIANGLE WITH RIGHT HALF BLACK
U+29EA	BLACK DIAMOND WITH DOWN ARROW
U+29EB	BLACK LOZENGE
U+29EC	WHITE CIRCLE WITH DOWN ARROW
U+29ED	BLACK CIRCLE WITH DOWN ARROW

U+29EE	ERROR-BARRED WHITE SQUARE	
U+29EF	ERROR-BARRED BLACK SQUARE	
U+29F0	ERROR-BARRED WHITE DIAMOND	
U+29F1	ERROR-BARRED BLACK DIAMOND	
U+29F2	ERROR-BARRED WHITE CIRCLE	
U+29F3	ERROR-BARRED BLACK CIRCLE	
U+29F4	RULE-DELAYED	
U+29F6	SOLIDUS WITH OVERBAR	
U+29F7	REVERSE SOLIDUS WITH HORIZONTAL STROKE	
U+29FA	DOUBLE PLUS	
U+29FB	TRIPLE PLUS	
U+29FE	TINY	
U+29FF	MINY	
U+2A00	N-ARY CIRCLED DOT OPERATOR	⨀ BIGODOT
U+2A01	N-ARY CIRCLED PLUS OPERATOR	⨁ BIGOPLUS
U+2A02	N-ARY CIRCLED TIMES OPERATOR	⨂ BIGOTIMES
U+2A03	N-ARY UNION OPERATOR WITH DOT	BIGUDOT
U+2A04	N-ARY UNION OPERATOR WITH PLUS	BIGUPLUS
U+2A05	N-ARY SQUARE INTERSECTION OPERATOR	BIGSQCAP
U+2A06	N-ARY SQUARE UNION OPERATOR	BIGSQCUP
U+2A07	TWO LOGICAL AND OPERATOR	
U+2A08	TWO LOGICAL OR OPERATOR	
U+2A09	N-ARY TIMES OPERATOR	BIGTIMES
U+2A0A	MODULO TWO SUM	
U+2A10	CIRCULATION FUNCTION	
U+2A11	ANTICLOCKWISE INTEGRATION	
U+2A12	LINE INTEGRATION WITH RECTANGULAR PATH AROUND POLE	
U+2A13	LINE INTEGRATION WITH SEMICIRCULAR PATH AROUND POLE	
U+2A14	LINE INTEGRATION NOT INCLUDING THE POLE	
U+2A1D	JOIN	⋈ JOIN
U+2A1E	LARGE LEFT TRIANGLE OPERATOR	
U+2A1F	Z NOTATION SCHEMA COMPOSITION	
U+2A20	Z NOTATION SCHEMA PIPING	
U+2A21	Z NOTATION SCHEMA PROJECTION	
U+2A32	SEMIDIRECT PRODUCT WITH BOTTOM CLOSED	
U+2A33	SMASH PRODUCT	
U+2A3C	INTERIOR PRODUCT	
U+2A3D	RIGHTHAND INTERIOR PRODUCT	
U+2A3E	Z NOTATION RELATIONAL COMPOSITION	
U+2A3F	AMALGAMATION OR COPRODUCT	
U+2A57	SLOPING LARGE OR	
U+2A58	SLOPING LARGE AND	
U+2A61	SMALL VEE WITH UNDERBAR	
U+2A64	Z NOTATION DOMAIN ANTIRESTRICTION	
U+2A65	Z NOTATION RANGE ANTIRESTRICTION	
U+2A68	TRIPLE HORIZONTAL BAR WITH DOUBLE VERTICAL STROKE	
U+2A69	TRIPLE HORIZONTAL BAR WITH TRIPLE VERTICAL STROKE	
U+2A6A	TILDE OPERATOR WITH DOT ABOVE	
U+2A6B	TILDE OPERATOR WITH RISING DOTS	
U+2A6D	CONGRUENT WITH DOT ABOVE	
U+2ACD	SQUARE LEFT OPEN BOX OPERATOR	
U+2ACE	SQUARE RIGHT OPEN BOX OPERATOR	
U+2AD9	ELEMENT OF OPENING DOWNWARDS	

U+2ADA	PITCHFORK WITH TEE TOP	
U+2ADC	FORKING	
U+2ADD	NONFORKING	
U+2ADE	SHORT LEFT TACK	
U+2ADF	SHORT DOWN TACK	
U+2AE0	SHORT UP TACK	
U+2AE1	PERPENDICULAR WITH S	
U+2AE2	VERTICAL BAR TRIPLE RIGHT TURNSTILE	
U+2AE3	DOUBLE VERTICAL BAR LEFT TURNSTILE	
U+2AE4	VERTICAL BAR DOUBLE LEFT TURNSTILE	
U+2AE5	DOUBLE VERTICAL BAR DOUBLE LEFT TURNSTILE	
U+2AE6	LONG DASH FROM LEFT MEMBER OF DOUBLE VERTICAL	
U+2AE7	SHORT DOWN TACK WITH OVERBAR	
U+2AE8	SHORT UP TACK WITH UNDERBAR	
U+2AE9	SHORT UP TACK ABOVE SHORT DOWN TACK	
U+2AEA	DOUBLE DOWN TACK	
U+2AEB	DOUBLE UP TACK	
U+2AEC	DOUBLE STROKE NOT SIGN	
U+2AED	REVERSED DOUBLE STROKE NOT SIGN	
U+2AEE	DOES NOT DIVIDE WITH REVERSED NEGATION SLASH	
U+2AEF	VERTICAL LINE WITH CIRCLE ABOVE	
U+2AF0	VERTICAL LINE WITH CIRCLE BELOW	
U+2AF1	DOWN TACK WITH CIRCLE BELOW	
U+2AF2	PARALLEL WITH HORIZONTAL STROKE	
U+2AF3	PARALLEL WITH TILDE OPERATOR	
U+2AF4	TRIPLE VERTICAL BAR BINARY RELATION	⫴
U+2AF5	TRIPLE VERTICAL BAR WITH HORIZONTAL STROKE	
U+2AF6	TRIPLE COLON OPERATOR	
U+2AFB	TRIPLE SOLIDUS BINARY RELATION	
U+2AFC	LARGE TRIPLE VERTICAL BAR OPERATOR	
U+2AFE	WHITE VERTICAL BAR	
U+2AFF	N-ARY WHITE VERTICAL BAR	

Bibliography

[1] O. Agesen, L. Bak, C. Chambers, B.-W. Chang, U. Hlzle, J. Maloney, R. B. Smith, D. Ungar, and M. Wolczko. *The Self Programmer's Reference Manual*. `http://research.sun.com/self/release_4.0/Self-4.0/manuals/Self-4.1-Pgme\%rs-Ref.pdf`, 2000.

[2] E. Allen, V. Luchangco, and S. Tobin-Hochstadt. Encapsulated Upgradable Components, Mar. 2005.

[3] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico.*, pages 356–368, Nov. 1994.

[4] R. D. Blumofe, C. F. Joerg, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 132–141, Montreal, Canada, 17–19 June 1998. ACM, SIGPLAN Notices.

[5] G. Bracha, G. Steele, B. Joy, and J. Gosling. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.

[6] R. Cartwright and G. Steele. Compatible genericity with run-time types for the Java Programming Language. In *OOPSLA*, 1998.

[7] W. Clinger. Macros that work. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 155–162. ACM Press, 1991.

[8] T. U. Consortium. *The Unicode Standard, Version 4.0*. Addison-Wesley, 2003.

[9] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in scheme. *Journal of LISP and Symbolic Computation*, 5(4):295–326, 1992.

[10] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 229–236. ACM Press, September 2001.

[11] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy Threads: Implementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, 37(1), Aug. 1996.

[12] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In L. Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[13] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[14] X. Leroy, D. Doligez, J. Garrigue, D. Rmy, and J. Vouillon. *The Objective Caml System, release 3.08*. `http://caml.inria.fr/distrib/ocaml-3.08/ocaml-3.08-refman.pdf`, 2004.

[15] J. Matthews. Operational semantics for scheme via term rewriting. Technical Report TR-2005-02, University of Chicago, Apr. 2005.

[16] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[17] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[18] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. Technical Report TM-449, MIT/LCS, 1991.

[19] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Mar. 1966.

[20] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. *The Scala Language Specification*. `http://scala.epfl.ch/docu/files/ScalaReference.pdf`, 2004.

[21] OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface Version 2.0*. `http://www.openmp.org/specs/mp-documents/fspec20_bars.pdf`, Nov. 2000.

[22] S. Peyton-Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[23] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior.

[24] Java(TM) 2 Platform Standard Edition 6.0 API Specification. `http://www.java.net/download/doc/api/`.