The Fortress Language Specification

May 14, 2012

Contents

Chapter 1

Introduction

Chapter 2

Overview of Fortress

Chapter 3

Getting Started

Chapter 4

Programs

Chapter 5

Types

This chapter should be revised according to Victor's tuple story (his email titled "Tuple story" on 12/07/07).

No dimensions and units.

The Fortress type system supports a mixture of nominal and structural types. Specifically, Fortress provides nominal trait types and some constructs that combine types structurally. There are also a few special types. Most trait types are defined by trait declarations in a program (often in libraries), but a few are built-in.

Victor: This doesn't cover the special types, dimension types, or object expression types. But I think it's okay for this intro, as long as we don't assert that these are all the types.

Every expression has a *static type*, and every value has an *ilk* (also known as a *dynamic or runtime type*). Fortress programs are checked before they are executed to ensure that if an expression e evaluates to a value v, the runtime type of v is a subtype of the static type of e. Sometimes we abuse terminology by saying that an expression has the runtime type of the value it evaluates to (see Section **??** for a discussion about evaluation of expressions).

Some types may be parameterized by *static parameters*; we call these types *generic types*. See Chapter **??** for a discussion of static parameters. A static parameter may be instantiated with a type or a value depending on whether it is a type parameter.

Victor: I don't know if the follow really belongs here. It isn't complete in any case, because of union/intersection types.

Two types are identical if and only if they are the same kind and their names and static arguments (if any) are identical. Types are related by several relationships as described in Section 5.2.

Syntactically, the positions in which a type may legally appear (i.e., *type contexts*) is determined by the nonterminal *Type* in the Fortress grammar, defined in Appendix **??**.

5.1 Kinds of Types

Fortress supports the following kinds of types:

- trait types,
- object expression types,
- tuple types,
- arrow types,
- function types, and
- three special types: Any, $\mathrm{BottomType}$ and $()$.

Object expression types, function types and $\mathrm{BottomType}$ are not first-class types: they cannot be written in a program. However, some values have object expression types and function types and `throw` and `exit` expressions have the type $\mathrm{BottomType}$.

Collectively, trait types and object expression types are *defined* types; the trait and object declarations and object expressions are the types' *definitions*.

Victor: I don't know if you like this terminology for trait types and object expression types. Putting this all together allowed some repetition to be eliminated, and emphasizes the similarity, which I think you wanted to retain. I still want to separate object expression types from object trait types. I think the difference here is not as evident as it might be if we handled generics properly: it is the type environment of object expression types that make them a pain, and I'm not dealing with that for the most part.

5.2 Relationships between Types

We define two relations on types: *subtype* and *exclusion*.

add coercion in full language

The subtype relation is a partial order on types that defines the type hierarchy; that is, it is reflexive, transitive and antisymmetric. Any is the top of the type hierarchy: all types are subtypes of Any. $\mathrm{BottomType}$ is the bottom of the type hierarchy: all types are supertypes of $\mathrm{BottomType}$. For types T and U, we write $T \preceq U$ when T is a subtype of U, and $T \prec U$ when $T \preceq U$ and $T \neq U$; in the latter case, we say T is a *strict* subtype of U. We also say that T is a *supertype* of U if U is a subtype of T. Thus, in the execution of a valid Fortress program, an expression's runtime type is always a subtype of its static type. We say that a value is *an instance of* its runtime type and of every supertype of its runtime type; immediate subtypes of Any comprises of tuple types, arrow types, $()$, and Object.

The exclusion relation is a symmetric relation between two types whose intersection is $\mathrm{BottomType}$. Because $\mathrm{BottomType}$ is uninhabited (i.e., no value has type $\mathrm{BottomType}$), types that exclude each other are disjoint: no value can have a type that is a subtype of two types that exclude each other.

The converse is technically not true: we can define two traits with method declarations such that no trait or object can extend them both, so no value can have a type that is a subtype of both trait types, but the trait types do not exclude each other. For example: trait A f(self, y: Object) = 1; end trait B f(x: Object, self) = 2; end trait C f() = 3; end No pair of the three trait types defined above can be extended but none of them exclude any of the others.

Note that $\mathrm{BottomType}$ excludes every type including itself (because the intersection of $\mathrm{BottomType}$ and any type is $\mathrm{BottomType}$), and also that no type other than $\mathrm{BottomType}$ excludes Any (because the intersection of any type T and Any is T). $\mathrm{BottomType}$ is the only type that excludes itself. Also note that if two types exclude each other then any subtypes of these types also exclude each other.

Fortress also allows *coercion* between types (see Chapter **??**). A coercion from T to U is defined in the declaration of U. We write $T \rightarrow U$ if U defines a coercion from T. We say that T *can be coerced to* U, and write $T \rightsquigarrow U$, if U defines a coercion from T or any supertype of T: $T \rightsquigarrow U \iff \exists T' : T \preceq T' \wedge T' \rightarrow U$.

The Fortress type hierarchy is acyclic with respect to both subtyping and coercion relations except for the following:

- The trait Any is a single root of the type hierarchy and it forms a cycle as described in Chapter **??**.
- There exists a bidirectional coercion between two tuple types if and only if they have the same sorted form.

These relations are defined more precisely in the following sections describing each kind of type in more detail. Specifically, the relations are the smallest ones that satisfy all the properties given in those sections (and this one).

5.3 Trait Types

Syntax:

Type ::= *TraitType*

A *trait type* is a named type defined by a trait or object declaration. It is an *object trait type* if it is defined by an object declaration.

Victor: The following text really belongs to traits, not to types. Traits are declared by trait and object declarations (described in Chapter 10 and Chapter 11). A trait has a *trait type* of the same name.

A trait type is a subtype of every type that appears in the `extends` clause of its definition. In addition, every trait type is a subtype of Any.

A trait declaration may also include an `excludes` clause, in which case the defined trait type excludes every type that appears in that clause. Every trait type also excludes every arrow type and every tuple type, and the special types $()$ and $\mathrm{BottomType}$.

5.3.1 Object Trait Types

An object trait type is a trait type defined by an object declaration. Object declarations do not include `excludes` clauses, but an object trait type cannot be extended. Thus, an object trait type excludes any type that is not its supertype.

5.4 Object Expression Types

An object expression type is defined by an object expression (described in Section **??**) and the program location of the object expression. Every evaluation of a given object expression has the same object expression type. Two object expressions at different program locations have different object expression types.

Like trait types, an object expression type is a subtype of all trait types that appear in the `extends` clause of its definition, as well as the type Any.

Like object trait types, an object expression type excludes any type that is not its supertype.

5.5 Tuple Types

Syntax:

TupleType ::= (*Type*, *TypeList*)
TypeList ::= *Type*(, *Type*)*

Victor: The following two sentences don't belong in this section: A tuple expression is an ordered sequence of expressions separated by commas and enclosed in parentheses. See Section **??** for a discussion of tuple expressions.

A tuple type consists of a parenthesized, comma-separated list of two or more types.

Every tuple type is a subtype of Any. No other type encompasses all tuple types. Tuple types cannot be extended by trait types. Tuple types are covariant: a tuple type X is a subtype of tuple type Y if and only if they have the same number of element types and each element type of X is a subtype of the corresponding element type of Y.

A tuple type excludes any non-tuple type other than Any. A tuple type excludes every tuple type that does not have the same number of element types. Also, tuple types that have the same number of element types exclude each other if any pair of corresponding element types exclude each other.

Intersection of nonexclusive tuple types are defined elementwise; the intersection of nonexclusive tuple type X and Y is a tuple type with exactly corresponding elements, where the type in each element type is the intersection of the types in the corresponding element types of X and Y. Note that intersection of any exclusive types is $\mathrm{BottomType}$ as described in Section 5.8.

5.6 Arrow Types

Arrow types should include `io`-ness.

Arrow types with contracts
Function contracts consist of three parts: a `requires` part, an `ensures` part, and an `invariant` part. All three parts are evaluated in the scope of the function body extended with a special variable *arg*, bound to an immutable array of all function arguments. (This array is useful for describing contracts in the presence of higher-order functions; see Section 5.6).
Is the special variable *arg* reserved? What if one of the function's parameters is named arg?
At runtime, when a function is bound to a variable or parameter f whose type includes a contract, the contract is evaluated when the function is called through a reference to f. (Note that this contract is distinct from the contract attached to the function bound to f, which is also evaluated upon a call to f). This contract is evaluated in the enclosing scope of the arrow type, extended with any keyword argument names provided in the arrow type, and the special variable arg bound to an immutable array containing the arguments provided at the call site (in the order provided at the call site). If the `requires` clauses stipulated in the type of f are not satisfied, a $\mathrm{CallerViolation}$ exception is thrown. Otherwise, if the `requires` clauses attached to the function bound to f is not satisfied, a $\mathrm{ContractBinding}$ exception is thrown. If any other part of the contract of the function bound to f is not satisfied, a $\mathrm{CalleeViolation}$ exception is thrown. Otherwise, if any part of the contract stipulated in the type of f is not satisfied, a $\mathrm{ContractBinding}$ exception is thrown.

Syntax:

Type	::=	*ArgType* $\rightarrow$ *Type Throws*?
ArgType	::=	((*Type*,)* *Type*...)
	|	*TupleType*

The static type of an expression that evaluates to a function value is an *arrow type* (or possibly an intersection of arrow types). An arrow type has three constituent parts: a *parameter type*, a *return type*, and a set of *exception types*. (Multiple parameters or return values are handled by having tuple types for the parameter type or return type respectively.)

Victor: I'd rather say "an exception type", where that type can be a union type, but I'm avoiding mentioning union types for now.

Syntactically, an arrow type consists of the parameter type followed by the token $\rightarrow$, followed by the return type, and optionally a `throws` clause that specifies the set of exception types.

Victor: What about io? Ignoring that for now.

If there is no `throws` clause, the set of exception types is empty.

The parameter type of an arrow type may end with a "varargs entry", $T\ldots$. The resulting parameter type is not a first-class type; rather, it is the union of all the types that result from replacing the varargs entry with zero or more entries of the type T. For example, $(T\ldots)$ is the union of $()$, T, (T,T), (T,T,T), and so on.

We say that an arrow type is *applicable* to a type A if A is a subtype of the parameter type of the arrow type. (If the parameter type has a varargs entry, then A must be a subtype of one of the types in the union defined by the parameter type.)

Every arrow type is a subtype of Any. No other type encompasses all arrow types. A type parameter can be instantiated with an arrow type. Arrow types cannot be extended by trait types. Arrow types are covariant in their return type and exception types and contravariant in their parameter type; that is, arrow type "$A \rightarrow B$ `throws` C" is a subtype of arrow type "$D \rightarrow E$ `throws` F" if and only if:

- D is a subtype of A,
- B is a subtype of E, and
- for all X in C, there exists Y in F such that X is a subtype of Y.

For arrow types S and T, we say that S is *more specific than* T if the parameter type of S is a subtype of the parameter type of T. We also say that S is *more restricted than* T if the return type of S is a subtype of the return type of T and for every X in the set of exception types of S, there exists Y in the set of exception types of T such that X is a subtype of Y. Thus, S is a subtype of T if and only if T is more specific than S and S is more restricted than T.

An arrow type excludes any non-arrow type other than Any and the function types that are its subtypes (see Section 5.7). However, arrow types do not exclude other arrow types because of overloading as described in Chapter **??**.

Here are some examples:

5.7 Function Types

Function types are the runtime types of function values. We distinguish them from arrow types to handle overloaded functions. A function type consists of a finite set of arrow types. However, not every set of arrow types is a well-formed function type. Rather, a function type F is *well-formed* if, for every pair (S_1, S_2) of distinct arrow types in F, the following properties hold:

- the parameter types of S_1 and S_2 are not the same,
- if S_1 is more specific than S_2 then S_1 is more restricted than S_2, and
- if the intersection of the parameter types of S_1 and S_2 is not $\mathrm{BottomType}$ (i.e., the parameter types of S_1 and S_2 do not exclude each other) then F has some constituent arrow type that is more specific than both S_1 and S_2 (recall that the more specific relation is reflexive, so the required constituent type may be S_1 or S_2).

Henceforth, we consider only well-formed function types. The overloading rules ensure that all function values have well-formed function types.

We say that a function type F is *applicable* to a type A if any of its constituent arrow types is applicable to A. We extend this terminology to values of the corresponding types. That is, for example, we say that a function of type F is applicable to a value of type A if F is applicable to A. Note that if a well-formed function type F is applicable to a type A, then among all constituent arrow types of F that are applicable to A (and there must be at least one), one is more specific than all the others. We say that this constituent type is the *most specific* type of F applicable to A.

A function type is a subtype of each of its constituent arrow types, and also of Any. Like object trait types and object expression types, a function type excludes every type that is not its supertype.

5.8 Special Types

Fortress provides three special types: Any, $\mathrm{BottomType}$ and $()$. The type Any is the top of the type hierarchy: it is a supertype of every type. The only type it excludes is $\mathrm{BottomType}$.

The type $()$ is the type of the value $()$. Its only supertype (other than itself) is Any, and it excludes every other type.

Fortress provides a special *bottom type*, $\mathrm{BottomType}$, which is an uninhabited type. No value in Fortress has $\mathrm{BottomType}$; `throw` and `exit` expressions have $\mathrm{BottomType}$. $\mathrm{BottomType}$ is a subtype of every type and it excludes every type (including itself). As mentioned above, $\mathrm{BottomType}$ is not a first-class type: programmers must not write $\mathrm{BottomType}$.

Victor's comments The complex numbers have precision like the integers, but are they intended to be only integral complex numbers? That seems odd to me. Also, why say "imaginary and complex", since we don't provide a separate type for just the imaginary numbers? Guy's going to provide APIs for complex numbers.

5.9 Types in the Fortress Standard Libraries

The Fortress standard libraries define simple standard types for literals such as $\mathrm{BooleanLiteral}[\![b]\!]$, $()$ (pronounced "void"), $\mathrm{Character}$, String, and $\mathrm{Numeral}[\![n, m, r, v]\!]$ for appropriate values of b, n, m, r, and v (See Section 12.1

for a discussion of Fortress literals). Moreover, there are several simple standard numeric types. These types are mutually exclusive; no value has more than one of them. Values of these types are immutable.

The numeric types share the common supertype Number. Fortress includes types for arbitrary-precision integers (of type $\mathbb{Z}$), their unsigned equivalents (of type $\mathbb{N}$), rational numbers (of type $\mathbb{Q}$), real numbers (of type $\mathbb{R}$), complex numbers (of type $\mathbb{C}$), 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 (described below), intervals (of type $\mathrm{Interval}[\![X]\!]$, abbreviated as $\langle\!|X|\!\rangle$, 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 $\mathrm{Polar}[\![X]\!]$ where X can be instantiated with any real number type).

The Fortress standard libraries also define other simple standard types such as Any, Object, $\mathrm{Exception}$, $\mathrm{Boolean}$, and $\mathrm{BooleanInterval}$ as well as low-level binary data types such as $\mathrm{LinearSequence}$, $\mathrm{HeapSequence}$, and $\mathrm{BinaryWord}$. See Parts **??** and **??** for discussions of the Fortress standard libraries.

5.10 Intersection and Union Types

For every finite set of types, there is a type denoting a unique *intersection* of those types. The intersection of a set of types S is a subtype of every type $T \in S$ and of the intersection of every subset of S. There is also a type denoting a unique *union* of those types. The union of a set of types S is a supertype of every type $T \in S$ and of the union of every subset of S. Neither intersection types nor union types are first-class types; they are used solely for type inference (as described in Chapter 17) and they cannot be expressed directly in programs.

The intersection of a set of types S is equal to a named type U when any subtype of every type $T \in S$ and of the intersection of every subset of S is a subtype of U. Similarly, the union of a set of types S is equal to a named type U when any supertype of every type $T \in S$ and of the union of every subset of S is a supertype of U. For example:

```
trait S comprises {U, V} end
trait T comprises {V, W} end
trait U extends S excludes W end
trait V extends {S, T} end
trait W extends T end
```

because of the `comprises` clauses of S and T and the `excludes` clause of U, any subtype of both S and T must be a subtype of V. Thus, $V = S \cap T$.

Intersection types (denoted by $\cap$) possess the following properties:

- Commutativity: $T \cap U = U \cap T$.
- Associativity: $S \cap (T \cap U) = (S \cap T) \cap U$.
- Subsumption: If $S \preceq T$ then $S \cap T = S$.
- Preservation of shared subtypes: If $T \preceq S$ and $T \preceq U$ then $T \preceq S \cap U$.
- Preservation of supertype: If $S \preceq T$ then $\forall U.\ S \cap U \preceq T$.
- Distribution over union types: $S \cap (T \cup U) = (S \cap T) \cup (S \cap U)$.

Union types (denoted by $\cup$) possess the following properties:

- Commutativity: $T \cup U = U \cup T$.
- Associativity: $S \cup (T \cup U) = (S \cup T) \cup U$.
- Subsumption: If $S \preceq T$ then $S \cup T = T$.
- Preservation of shared supertypes: If $S \preceq T$ and $U \preceq T$ then $S \cup U \preceq T$.
- Preservation of subtype: If $T \preceq S$ then $\forall U.\ T \preceq S \cup U$.
- Distribution over intersection types: $S \cup (T \cap U) = (S \cup T) \cap (S \cup U)$.

5.11 Type Aliases

Syntax:

TypeAlias ::= `type` *Id StaticParams*? = *Type*

Fortress allows names to serve as aliases for more complex type instantiations. A *type alias* begins with `type` followed by the name of the alias type, followed by optional static parameters, followed by =, followed by the type it stands for. Parameterized type aliases are allowed but recursively defined type aliases are not. Here are some examples:

`type` IntList = List⟦ℤ64⟧
`type` BinOp = Float × Float → Float
`type` SimpleFloat⟦`nat` e, `nat` s⟧ = DetailedFloat⟦Unity, e, s, *false*, *false*, *false*, *false*, *true*⟧

All uses of type aliases are expanded before type checking. Type aliases do not define new types nor nominal equivalence relations among types.

Chapter 6

Lexical Structure

6.1 Reserved Words

The following tokens are *reserved words*:

`BIG`	`FORALL`	`SI_unit`	`absorbs`	`abstract`	`also`	`api`
`asif`	`at`	`atomic`	`bool`	`case`	`catch`	`coerce`
`coerces`	`component`	`comprises`	`default`	`dim`	`do`	`elif`
`else`	`end`	`ensures`	`except`	`excludes`	`exit`	`export`
`extends`	`finally`	`fn`	`for`	`forbid`	`from`	`getter`
`hidden`	`if`	`import`	`int`	`invariant`	`io`	`juxtaposition`
`label`	`most`	`nat`	`native`	`object`	`of`	`opr`
`or`	`outcome`	`override`	`private`	`property`	`provided`	`requires`
`self`	`settable`	`setter`	`spawn`	`syntax`	`test`	`then`
`throw`	`throws`	`trait`	`try`	`tryatomic`	`type`	`typecase`
`typed`	`unit`	`value`	`var`	`where`	`while`	`widens`
`with`	`wrapped`					

Victor: I don't think 'or" should be reserved. It is only so for its occurrence in 'widens or coerces" but we can recognize it specially in that context, which is never ambiguous because 'widens" and 'coerces" are reserved.

The following operators on units are also reserved words:

cubed cubic in inverse per square squared

To avoid confusion, Fortress reserves the following tokens:

`goto` `idiom` `public` `pure` `reciprocal` `static`

They do not have any special meanings but they cannot be used as identifiers.

Victor: Some other words we might want to reserve: subtype, subtypes, is, coercion, function, exception, match

Chapter 7

Names and Declarations

Chapter 8

Variables

Chapter 9

Functions

Chapter 10

Traits

`export` Executable

$run() = println($"`Hello, world!`")

File	::=	CompilationUnit
	|	Imports$^?$ Exports Decls$^?$
	|	Imports$^?$ AbsDecls
	|	Imports AbsDecls$^?$
CompilationUnit	::=	Component
	|	Api

Chapter 11

Objects

Chapter 12

Expressions

12.1 Literals

12.2 Identifier References

12.3 Dotted Field Accesses

12.4 Function Calls

12.5 Operator Applications

12.6 Tuple Expressions

12.7 Aggregate Expressions

12.8 Function Expressions

12.9 Blocks

A block is a series of one or more declarations and expressions that are to be evaluated sequentially. A block is not in itself syntactically an expression, but appears as part of a `do` expression Section 12.10, `label` expression Section 12.11, `while` expression Section 12.12, `for` expression Section 12.14, `if` expression Section 12.18, `case` expresion Section 12.19, `typecase` expression Section 12.20, or `try` expression Section 12.23. For purposes of type-checking and evaluation, it may be regarded as an expression.

Block	::=	BlockElement^{+}
BlockElement	::=	LocalVarFnDecl
	|	Expr (, GeneratorList)$^{?}$

It is a static error if the last BlockElement in a Block is an expression of the form "$a = b$" that cannot be regarded as a local declaration. It is a static error if any block element other than the last one is an expression whose type is not $()$. If the last block element of a block is a declaration, then the type of the block is $()$; if the last block element of a block is an expression, then the type of the block is the type of that expression.

Every block introduces a new scope. For the scoping behavior of local declarations within a block, see Section **??** and Section **??**.

Not shown in the grammar is the fact that adjacent block elements may be separated by a semicolon ';' if desired. This is necessary if the block elements appear in the same line of source code. If a newline occurs between two adjacent block elements, then no semicolon is necessary.

A block is evaluated by evaluating its block elements sequentially, in order from left to right; evaluation of each block element must complete before evaluation of the next can begin. If evaluation of any block element completes abruptly for some reason, then evaluation of the block completes abruptly for the same reason, and no further block elements are evaluated. If the evaluations of all block elements complete normally, then the value of the block is the value of the last block element.

If it is desired to have, as a non-final block element, an expression e whose type is not $()$, one may instead write the local variable declaration $\_ = e$, thus signifying explicitly that the value of e is to be discarded (by binding an anonymous variable to the value).

If it is desired to have, as a final block element, an equality test expression of the form $a = b$, simply enclose the expression in parentheses.

Here is an example of a `do` expression that contains a single block having six block elements:

```
do
  f(w: ℤ32) = w + 1      ⊛ Local function declaration
  y = x + 1              ⊛ Local variable declaration (immutable)
  println("y is" y)      ⊛ Expression (has a useful side effect)
  var z: ℝ64 = 0         ⊛ Local variable declaration (mutable)
  z += f(y)              ⊛ Compound assignment
  |z|                    ⊛ Expression (the value of the block)
end
```

12.10 Do and Do-Also Expressions

In the simplest case, the keywords `do` and `end` surround a single block to be executed. In the more general case, a `do` expression can contain multiple blocks to be executed independently (perhaps but not necessarily, concurrently). Each block may be governed by its own `atomic` modifier.

```
DoExpr    ::=  ( DoFront also )* DoFront end
DoFront   ::=  ( at Expr )? atomic? do Block?
```

If a `do` expression contains a single Block, then the type of the `do` expresssion is the type of the block. If a `do` expression contains two or more blocks (separated by `also`), then the type of the `do` expression is $()$, and it is a static error if any of the blocks has a type that is not $()$.

If the `do` keyword preceding any block of a multi-block `do` expression is preceded by an `atomic` modifier, it is as if (a) the following block were enclosed by a new pair of `do` and `end` keywords to form a single-block `do` expression, and (b) the `atomic` modifier appeared instead before that single-block `do` expression, thus forming an `atomic` expression. Thus, allowing `atomic` to appear as part of a multi-block `do` expression is merely a syntactic convenience that allows programs to be slightly shorter and perhaps read more naturally. For example:

```
atomic do
  x += 1
  y += 2
also atomic do
  b += 1
  y += 3
end
```

is simply an abbreviation of

```
do
  atomic do
    x += 1
    y += 2
  end
also do
  atomic do
    b += 1
    y += 3
  end
end
```

A `do` expression with a single block is evaluated by evaluating the block. If the block completes abruptly for some reason, then the `do` expression completes abruptly for the same reason. Otherwise, the value of the `do` expression is the value of the block.

A `do` expression with multiple blocks block is evaluated by evaluating all the blocks independently (perhaps, but not necessarily, concurrently). If more blocks complete abruptly for some reason, then the `do` expression completes abruptly for one of those reasons (evaluation of other blocks may or may not be completed, but evaluation of the `do` expression is not complete until the evaluations of all blocks have been terminated). Otherwise, evaluation of the `do` expression completes only after evaluation of all blocks is complete, and the value of the `do` expression is $()$.

12.11 Label and Exit

12.12 While Loops

The `while` statement evaluates a test and a `do` expression, sequentially and repeatedly, until the test fails. The test may be either a Boolean expression or a generator binding.

WhileExpr	::=	`while` Generator DoExpr
Generator	::=	GeneratorBinding
	|	Expr
GeneratorBinding	::=	Id $\leftarrow$ Expr
	|	(Id , IdList) $\leftarrow$ Expr

If the Generator is an expression, it is a static error if the type of the expression does not conform to $\mathrm{Boolean}$. If the Generator is a generator binding, it is a static error if the type of the expression in the generator binding does not conform to $\mathrm{Condition}[\![T]\!]$ for some T. The type of a `while` expression is $()$.

A `while` expression with an expression is evaluated by first evaluating Expr. If this evaluation completes abruptly for some reason, evaluation of the `while` expression completes abruptly for the same reason. Otherwise, evaluation continues by making a choice based on the resulting value v. If v is false, no further action is taken; evaluation of the `while` expression completes normally with value $()$. But if v is true, then the `do` expression is evaluated. If this evaluation completes abruptly for some reason, evaluation of the `while` expression completes abruptly for the same reason; otherwise, the entire `while` expression is evaluated again (beginning by re-evaluating Expr). For example:

```
do
  ⊛ Print the numbers 0 through 9 on separate lines.
  k: ℤ := 0
  while (k < 10) do
    println(k)
```

```
    k += 1
  end
end
```

A `while` expression with a generator binding is evaluated by first evaluating the Expr in the generator binding. If this evaluation completes abruptly for some reason, evaluation of the `while` expression completes abruptly for the same reason. Otherwise, evaluation continues by making a choice based on the resulting value v. If v does not contain a value, no further action is taken; evaluation of the `while` expression completes normally with value $()$. If v contains a value w, then the pattern in the generator binding is matched to that value w and then the `do` expression is evaluated, and variables bound by the pattern are visible within the `do` expression. If this evaluation completes abruptly for some reason, evaluation of the `while` expression completes abruptly for the same reason; otherwise, the entire `while` expression is evaluated again (beginning by re-evaluating Expr). For example:

```
object Chain(item: ℤ, link: Maybe⟦Chain⟧) end
printValues(x: Maybe⟦Chain⟧): () = do
    cursor: Maybe⟦Chain⟧ = x
    while (next ← cursor) do
      println(next.item)
      cursor := next.link
    end
  end
end
run() = printValues(Just Chain(1, Just Chain(2, Just Chain(3, Just Chain(4, None)))))
```

12.13 Generators

12.14 For Expressions

12.15 Ranges

12.16 Reduction Expresions

12.17 Comprehensions

12.18 If Expressions

The `if` statement allows conditional evaluation of a block or conditional evaluation of at most one of a set of blocks. The choice is made by sequentially executing one or more tests and choosing the first statement for which a test succeeds. Each test may be either a Boolean expression or a generator binding.

IfExpr	::=	`if` Generator `then` Block ElifClause* ElseClause? `end`
	|	[() `if` Generator `then` Block ElifClause* ElseClause `end`? []]
Generator	::=	GeneratorBinding
	|	Expr
GeneratorBinding	::=	Id ← Expr
	|	(Id , IdList) ← Expr
ElifClause	::=	`elif` Expr `then` Block
ElseClause	::=	`else` Block

An `if` expression consists of `if` followed by a Generator clause (discussed in Section 12.13), followed by `then`, a Block, a possibly empty sequence of `elif` clauses (each consisting of `elif` followed by a Generator clause, `then`, and a Block), an optional `else` clause (consisting of `else` followed by a Block), and finally `end`. The reserved word `end` may be elided if the `if` expression is immediately enclosed by parentheses; in such a case, the `else` clause is required, not optional.

Each Block is a series of one or more block elements (declarations and expressions). See Section **??** for a description of the various syntactic and semantic properties of blocks.

For each Generator, if it is an expression, it is a static error if the type of the expression does not conform to $\mathrm{Boolean}$; if it is a generator binding, it is a static error if the type of the expression in the generator binding does not conform to $\mathrm{Condition}[\![T]\!]$ for some T. If the `if` expression has no `else` clause, it is a static error if the type of the block after the first `then`, or in any `elif` clause, is not $()$. The type of an `if` expression is the union of the types of all its blocks.

An `if` expression whose first Generator is an expression is evaluated by first evaluating Expr. If this evaluation completes abruptly for some reason, evaluation of the `if` expression completes abruptly for the same reason; otherwise, evaluation continues by making a choice based on the resulting value v. If v is true, then the first block is evaluated. If this evaluation completes abruptly for some reason, evaluation of the `if` expression completes abruptly for the same reason; otherwise, the value of the `if` expression is value of this block. If v is false, then `elif` clauses are considered (see below).

An `if` expression whose first Generator is a generator binding is evaluated by first evaluating the Expr in the generator binding. If this evaluation completes abruptly for some reason, evaluation of the `if` expression completes abruptly for the same reason; otherwise, evaluation continues by making a choice based on the resulting value v. If v contains a value w, then the pattern in the generator binding is matched to that value w and then the first block is evaluated and variables bound by the pattern are visible within the block. If this evaluation completes abruptly for some reason, evaluation of the `if` expression completes abruptly for the same reason; otherwise, the value of the `if` expression is value of this block. If v does not contain a value, then `elif` clauses are considered (see below).

If evaluation of an `if` expression must consider `elif` clauses, they are examined sequentially, working from left to right, treating each **??** and each block in exactly the same manner as the first generator and first block of the `if` expression. As soon as a generator is found that produces the value true or a condition value v that contains another value w, the corresponding block is evaluated, and its value becomes the value of the `if` expression; but if evaluation of any Generator or block completes abruptly for some reason, evaluation of the `if` expression completes abruptly for the same reason. If consideration of `elif` clauses does not result in evaluating an block (possibly because no `elif` clauses are present), then any `else` clause is considered.

If evaluation of an `if` expression must consider any `else` clause, there are two cases. If an `else` clause is present, then its block is evaluated, and its value becomes the value of the `if` expression; but if evaluation of the block completes abruptly for some reason, evaluation of the `if` expression completes abruptly for the same reason. If no `else` clause is present, then evaluation of the `if` expression completes normally with value $()$.

Examples:

```
if x > 0 then println x end
```

```
if x > 0 then
  println(x "is positive")
else
  println(x "is nonpositive")
end
```

```
println(x "is" (if v > 0 then "positive" else "nonpositive"))
```

```
z = if x < 0 then 0
    elif x ∈ {1,2,3} then 3
    elif x ∈ {4,5,6} then 6
    else 9 end
```

12.19 Case Expressions

12.20 Typecase Expressions

12.21 Atomic Expressions

12.22 Throw Expressions

12.23 Try Expressions

12.24 Type Ascription

12.25 Asif Expressions

Chapter 13

Exceptions

Chapter 14

Operators

Chapter 15

Overloading and Dispatch

Keyword and varargs parameters are not yet supported.

Fortress allows functions and methods (collectively called *procedures*) to be *overloaded*. That is, there may be multiple declarations for the same procedure name visible in a single scope (which may include inherited method declarations), and several of them may be applicable to any particular procedure call. This raises the question of which definition will actually be executed for any given procedure invocation. The simple answer is that the *dynamically most specific applicable visible* definition is chosen. That is, one first considers the set of all definitions that are *dynamically visible*; then, among those in the set that are *applicable* to the given argument values, the *most specific* one is chosen.

At compile time, typechecking performs a related series of tests: the type of a procedure call is the return type of the *statically most specific applicable visible* declaration. That is, one first considers the set of all definitions that are *statically visible*; then, among those in the set that are *applicable* to the types of the given arguments, the *most specific* one is chosen.

`trait` T `extends` $\{A, B\}$ `end`

In this chapter, we describe how to determine which declarations are *visible* to a particular procedure call (boths statically and dynamically); how to further determine which of these are *applicable* to that procedure call (boths statically and dynamically); and how the most specific one is chosen. We also introduce some rules for writing procedure declarations that ensure the soundness of the type system, as well as the existence and uniqueness of a most specific applicable declaration (at compile time) or definition (at run time). The result is that if a program successfully compiles, then procedure calls always succeed and are never ambiguous.

Section 15.1 introduces some terminology and notation. In Section 15.3, we show how to determine which declarations are applicable to a *named procedure call* (a function call described in Section 12.4 or a naked method invocation described in Section **??**) when all declarations have only ordinary parameters (without varargs or keyword parameters). We discuss how to handle dotted method calls (described in Section **??**) in Section 15.4, and declarations with varargs and keyword parameters in Section 15.5. Determining which declaration is applied, if several are applicable, is discussed in Section 15.6.

15.1 Principles of Overloading

Fortress allows multiple procedure declarations of the same name to be declared in a single scope. However, recall from Chapter **??** the following shadowing rules:

- dotted method declarations shadow top-level function declarations with the same name, and
- dotted method declarations provided by a trait or object declaration or object expression shadow functional method declarations with the same name that are provided by a different trait or object declaration or object expression.

Also, note that a trait or object declaration or object expression must not have a functional method declaration and a dotted method declaration with the same name, either directly or by inheritance. Therefore, top-level functions can overload with other top-level functions and functional methods, dotted methods with other dotted methods, and functional methods with other functional methods and top-level functions. It is a static error if any top-level function declaration is more specific than any functional method declaration. If a top-level function declaration is overloaded with a functional method declaration, the top-level function declaration must not be more specific than the functional method declaration.

Operator declarations with the same name but different fixity are not a valid overloading; they are unambiguous declarations. An operator method declaration whose name is one of the operator parameters (described in Section **??**) of its enclosing trait or object may be overloaded with other operator declarations in the same component. Therefore, such an operator method declaration must satisfy the overloading rules (described in Chapter **??**) with every operator declaration in the same component.

This restriction will be relaxed.

Recall from Chapter 5 that we write $T \preceq U$ when T is a subtype of U, and $T \prec U$ when $T \preceq U$ and $U \not\preceq T$.

15.2 Visibility to Named Procedure Calls

15.3 Applicability to Named Procedure Calls

In this section, we show how to determine which declarations are applicable to a named procedure call when all declarations have only ordinary parameters (i.e., neither varargs nor keyword parameters).

For the purpose of defining applicability, a named procedure call can be characterized by the name of the procedure and its argument type. Recall that a procedure has a single parameter, which may be a tuple (a dotted method has a receiver as well). We abuse notation by using *static call* $f(A)$ to refer to a named procedure call with name f and whose argument has static type A, and *dynamic call* $f(X)$ to refer to a named procedure call with We use *call* $f(C)$ to refer to a named procedure call with name f and whose argument, when evaluated, has dynamic type C. (Note that if the type system is sound—and we certainly hope that it is!—then $X \preceq A$ for all well-typed calls to f.) We use the term *call* $f(C)$ to refer to static and dynamic calls collectively. We assume throughout this chapter that all static variables in procedure calls have been instantiated or inferred.

We also use *function declaration* $f(P) : U$ to refer to a function declaration with function name f, parameter type P, and return type U.

For method declarations, we must take into account the self parameter, as follows:

A *dotted method declaration* $P_0.f(P) : U$ is a dotted method declaration with name f, where P_0 is the trait or object type in which the declaration appears, P is the parameter type, and U is the return type. (Note that despite the suggestive notation, a dotted method declaration does not explicitly list its self parameter.)

A *functional method declaration* $f(P) : U$ *with self parameter at* i is a functional method declaration with name f, with the parameter `self` in the ith position of the parameter type P, and return type U. Note that the static type of the self parameter is the trait or object trait type in which the declaration $f(P) : U$ occurs. In the following, we will use P_i to refer to the ith element of P.

We elide the return type of a declaration, writing $f(P)$ and $P_0.f(P)$, when the return type is not relevant to the discussion. Note that static parameters may appear in the types P_0, P, and U.

A declaration $f(P)$ is *applicable* to a call $f(C)$ if the call is in the scope of the declaration and $C \preceq P$. (See Chapter **??** for the definition of scope.) If the parameter type P includes static parameters, they are inferred as described in Chapter 17 before checking the applicability of the declaration to the call.

Note that a named procedure call $f(C)$ may invoke a dotted method declaration if the declaration is provided by the trait or object enclosing the call. To account for this, let C_0 be the trait or object declaration immediately enclosing the call. Then we consider a named procedure call $f(C)$ as $C_0.f(C)$ if C_0 provides dotted method declarations applicable to $f(C)$, and use the rule for applicability to dotted method calls (described in Section 15.4) to determine which declarations are applicable to $C_0.f(C)$.

15.4 Applicability to Dotted Method Calls

Dotted method applications can be characterized similarly to named procedure applications, except that, analogously to dotted method declarations, we use C_0 to denote the dynamic type of the receiver object, and, as for named procedure calls, C to denote the dynamic type of the argument of a dotted method call. We write $C_0.f(C)$ to refer to the call.

A dotted method declaration $P_0.f(P)$ is *applicable* to a dotted method call $C_0.f(C)$ if $C_0 \preceq P_0$ and $C \preceq P$. If the types P_0 and P include static parameters, they are inferred as described in Chapter 17 before checking the applicability of the declaration to the call.

15.5 Applicability for Procedures with Varargs and Keyword Parameters

The basic idea for handling varargs and keyword parameters is that we can think of a procedure declaration that has such parameters as though it were (possibly infinitely) many declarations, one for each set of arguments it may be called with. In other words, we expand these declarations so that there exists a declaration for each number of arguments that can be passed to it.

A declaration with a varargs parameter corresponds to an infinite number of declarations, one for every number of arguments that may be passed to the varargs parameter. In practice, we can bound that number by the maximum number of arguments that the procedure is called with anywhere in the program (in other words, a given program will contain only a finite number of calls with different numbers of arguments). The expansion described here is a conceptual one to simplify the description of the semantics; we do not expect a real implementation to actually expand these declarations at compile time. For example, the following declaration:

$f(x : \mathbb{Z}, y : \mathbb{Z}, z : \mathbb{Z} \ldots) : \mathbb{Z}$

would be expanded into:

$f(x : \mathbb{Z}, y : \mathbb{Z}) : \mathbb{Z}$
$f(x : \mathbb{Z}, y : \mathbb{Z}, z_1 : \mathbb{Z}) : \mathbb{Z}$
$f(x : \mathbb{Z}, y : \mathbb{Z}, z_1 : \mathbb{Z}, z_2 : \mathbb{Z}) : \mathbb{Z}$
$f(x : \mathbb{Z}, y : \mathbb{Z}, z_1 : \mathbb{Z}, z_2 : \mathbb{Z}, z_3 : \mathbb{Z}) : \mathbb{Z}$
$\ldots$

A declaration with a varargs parameter is applicable to a call if any one of the expanded declarations is applicable.

15.6 Overloading Resolution

Victor: Does this paragraph, other than the last sentence, really belong in this section?

To evaluate a given procedure call, it is necessary to determine which procedure declaration to dispatch to. To do so, we consider the declarations that are applicable to that call at run time. If there is exactly one such declaration, then the call dispatches to that declaration. If there is no such declaration, then the call is *undefined*, which is a static error. (However, see Section **??** for how coercion may add to the set of applicable declarations.) If multiple declarations are applicable to the call at run time, then we choose an arbitrary declaration among the declarations such that no other applicable declaration is more specific than them.

We use the subtype relation to compare parameter types to determine a more specific declaration. Formally, a declaration $f(P)$ is *more specific* than a declaration $f(Q)$ if $P \prec Q$. Similarly, a declaration $P_0.f(P)$ is more specific than a declaration $Q_0.f(Q)$ if $P_0 \prec Q_0$ and $P \prec Q$. (See Section **??** for how coercion changes the definition of "more specific".) Restrictions on the definition of overloaded procedures (see Chapter **??**) guarantee that among all applicable declarations, one is more specific than all the others. If the declarations include static parameters, they are inferred as described in Chapter 17 before comparing their parameter types to determine which declaration is more specific.

Chapter 16

Coercion

Chapter 17

Type Inference

Chapter 18

Components and APIs

Appendix A

Simplified Grammar for Application Programmers and Library Writers

A.1 Components and API

File	::=	CompilationUnit
	|	Imports$^?$ Exports Decls$^?$
	|	Imports$^?$ AbsDecls
	|	Imports AbsDecls$^?$
CompilationUnit	::=	Component
	|	Api
Component	::=	`component` DottedId Imports$^?$ Exports Decls$^?$ `end`
Api	::=	`api` DottedId Imports$^?$ AbsDecls$^?$ `end`
Imports	::=	Import$^+$
Import	::=	`import` ImportFrom
	|	`import` AliasedDottedIds
ImportFrom	::=	$*$ (`except` Names)$^?$ *from* DottedId
	|	AliasedNames *from* DottedId
Names	::=	Name
	|	{ NameList }
NameList	::=	Name (, Name)*
AliasedNames	::=	AliasedName
	|	{ AliasedNameList }
AliasedName	::=	Id (*as* DottedId)$^?$
	|	`opr` Op (*as* Op)$^?$
	|	`opr` LeftEncloser RightEncloser (*as* LeftEncloser RightEncloser)$^?$
AliasedNameList	::=	AliasedName (, AliasedName)*
AliasedDottedIds	::=	AliasedDottedId
	|	{ AliasedDottedIdList }
AliasedDottedId	::=	DottedId (*as* DottedId)$^?$
AliasedDottedIdList	::=	AliasedDottedId (, AliasedDottedId)*
Exports	::=	Export$^+$
Export	::=	`export` DottedIds
DottedIds	::=	DottedId

| | { DottedIdList }
DottedIdList ::= DottedId (, DottedId)*

A.2 Top-level Declarations

Decls ::= Decl+
Decl ::= TraitDecl
| ObjectDecl
| VarDecl
| FnDecl
AbsDecls ::= AbsDecl+
AbsDecl ::= AbsTraitDecl
| AbsObjectDecl
| AbsVarDecl
| AbsFnDecl

A.3 Trait Declaration

TraitDecl ::= TraitHeader GoInATrait? `end`
TraitHeader ::= TraitMods? `trait` Id StaticParams? Extends? TraitClauses?
TraitClauses ::= TraitClause+
TraitClause ::= Excludes
| Comprises
GoInATrait ::= GoFrontInATrait GoBackInATrait?
| GoBackInATrait
GoFrontInATrait ::= GoesFrontInATrait+
GoesFrontInATrait ::= AbsFldDecl
| GetterSetterDecl
GoBackInATrait ::= GoesBackInATrait+
GoesBackInATrait ::= MdDecl
AbsTraitDecl ::= TraitHeader AbsGoInATrait? `end`
AbsGoInATrait ::= AbsGoFrontInATrait AbsGoBackInATrait?
| AbsGoBackInATrait
AbsGoFrontInATrait ::= AbsGoesFrontInATrait+
AbsGoesFrontInATrait ::= ApiFldDecl
| AbsGetterSetterDecl
AbsGoBackInATrait ::= AbsGoesBackInATrait+
AbsGoesBackInATrait ::= AbsMdDecl
| AbsCoercion

A.4 Object declaration

ObjectDecl ::= ObjectHeader GoInAnObject? `end`
ObjectHeader ::= ObjectMods? `object` Id StaticParams? ObjectValParam? Extends? FnClauses
ObjectValParam ::= (ObjectParams?)

```
ObjectParams          ::= ( ObjectParam , )* ObjectKeyword ( , ObjectKeyword )*
                        | ObjectParam ( , ObjectParam )*
ObjectKeyword         ::= ObjectParam = Expr
ObjectParam           ::= FldMods? Param
                        | transient Param
GoInAnObject          ::= GoFrontInAnObject GoBackInAnObject?
                        | GoBackInAnObject
GoFrontInAnObject     ::= GoesFrontInAnObject+
GoesFrontInAnObject   ::= FldDecl
                        | GetterSetterDef
GoBackInAnObject      ::= GoesBackInAnObject+
GoesBackInAnObject    ::= MdDef
AbsObjectDecl         ::= ObjectHeader AbsGoInAnObject? end
AbsGoInAnObject       ::= AbsGoFrontInAnObject AbsGoBackInAnObject?
                        | AbsGoBackInAnObject
AbsGoFrontInAnObject  ::= AbsGoesFrontInAnObject+
AbsGoesFrontInAnObject::= ApiFldDecl
                        | AbsGetterSetterDecl
AbsGoBackInAnObject   ::= AbsGoesBackInAnObject+
AbsGoesBackInAnObject ::= AbsMdDecl
                        | AbsCoercion
```

A.5 Variable Declaration

```
VarDecl               ::= VarWTypes InitVal
                        | VarWoTypes = Expr
                        | VarWoTypes : TypeRef ... InitVal
                        | VarWoTypes : SimpleTupleType InitVal
VarWTypes             ::= VarWType
                        | ( VarWType ( , VarWType )+ )
VarWType              ::= VarMods? Id IsType
VarWoTypes            ::= VarWoType
                        | ( VarWoType ( , VarWoType )+ )
VarWoType             ::= VarMods? Id
InitVal               ::= ( = | := ) Expr
AbsVarDecl            ::= VarWTypes
                        | VarWoTypes : TypeRef ...
                        | VarWoTypes : SimpleTupleType
```

A.6 Function Declaration

```
FnDecl                ::= FnDef
                        | AbsFnDecl
FnDef                 ::= FnMods? FnHeaderFront FnHeaderClause = Expr
AbsFnDecl             ::= FnMods? FnHeaderFront FnHeaderClause
                        | Name : ArrowType
```

FnHeaderFront	::=	Id StaticParams$^?$ ValParam
	|	OpHeaderFront

A.7 Headers

Extends	::=	`extends` TraitTypes
Excludes	::=	`excludes` TraitTypes
Comprises	::=	`comprises` ComprisingTypes
TraitTypes	::=	TraitType
	|	{ TraitTypeList }
TraitTypeList	::=	TraitType (, TraitType)*
ComprisingTypes	::=	TraitType
	|	{ ComprisingTypeList }
ComprisingTypeList	::=	. . .
	|	TraitType (, TraitType)* (, . . .)$^?$
FnHeaderClause	::=	IsType$^?$ FnClauses
FnClauses	::=	Throws$^?$
Throws	::=	`throws` MayTraitTypes
MayTraitTypes	::=	{ }
	|	TraitTypes
CoercionClauses	::=	Throws$^?$
UniversalMod	::=	`private`
TraitMod	::=	`value`
	|	UniversalMod
TraitMods	::=	TraitMod$^+$
ObjectMods	::=	TraitMods
FnMod	::=	LocalFnMod
	|	UniversalMod
FnMods	::=	FnMod$^+$
VarMod	::=	`var`
	|	UniversalMod
VarMods	::=	VarMod$^+$
AbsFldMod	::=	`hidden` | `settable` | UniversalMod
AbsFldMods	::=	AbsFldMod$^+$
FldMod	::=	`var`
	|	AbsFldMod
FldMods	::=	FldMod$^+$
ApiFldMod	::=	`hidden` | `settable` | UniversalMod
ApiFldMods	::=	ApiFldMod$^+$
LocalFnMod	::=	`atomic`
LocalFnMods	::=	LocalFnMod$^+$
StaticParams	::=	⟦ StaticParamList ⟧
StaticParamList	::=	StaticParam (, StaticParam)*
StaticParam	::=	Id Extends$^?$
	|	`nat` Id
	|	`int` Id

| | `bool` Id
| | `opr` Op
| | *ident* Id

A.8 Parameters

ValParam ::= BindId
| (Params?)
Params ::= (Param ,)* Keyword (, Keyword)*
| (Param ,)*
| Param (, Param)*
Keyword ::= Param = Expr
PlainParam ::= BindId IsType?
| TypeRef
Param ::= PlainParam
OpHeaderFront ::= `opr` StaticParams? (LeftEncloser | Encloser) Params (RightEncloser | Encloser) (:=
| `opr` StaticParams? ValParam Op
| `opr` (Op | Encloser) StaticParams? ValParam
SubscriptAssignParam ::= Param

A.9 Method Declaration

MdDecl ::= MdDef
| AbsMdDecl
MdDef ::= FnMods? MdHeaderFront FnHeaderClause = Expr
| Coercion
AbsMdDecl ::= `abstract` ? FnMods? MdHeaderFront FnHeaderClause
MdHeaderFront ::= (Receiver .)? Id StaticParams? MdValParam
| OpHeaderFront
Receiver ::= Id
| `self`
GetterSetterDecl ::= GetterSetterDef
| AbsGetterSetterDecl
GetterSetterDef ::= FnMods? GetterSetterMod MdHeaderFront FnHeaderClause = Expr
GetterSetterMod ::= `getter` | `setter`
AbsGetterSetterDecl ::= `abstract` ? FnMods? GetterSetterMod MdHeaderFront FnHeaderClause
Coercion ::= *widening*? *coercion* StaticParams? (Id IsType) CoercionClauses = Expr
AbsCoercion ::= *widening*? *coercion* StaticParams? (Id IsType) CoercionClauses

A.10 Method Parameters

MdValParam ::= (MdParams?)
MdParams ::= (MdParam ,)* MdKeyword (, MdKeyword)*
| (MdParam ,)*
| MdParam (, MdParam)*

MdKeyword	::=	MdParam = Expr
MdParam	::=	Param
	|	`self`

A.11 Field Declarations

FldDecl	::=	FldWTypes InitVal
	|	FldWoTypes = Expr
	|	FldWoTypes : TypeRef ... InitVal
	|	FldWoTypes : SimpleTupleType InitVal
FldWTypes	::=	FldWType
	|	(FldWType (, FldWType)$^+$)
FldWType	::=	FldMods$^?$ Id IsType
FldWoTypes	::=	FldWoType
	|	(FldWoType (, FldWoType)$^+$)
FldWoType	::=	FldMods$^?$ Id

A.12 Abstract Filed Declaration

AbsFldDecl	::=	AbsFldWTypes
	|	AbsFldWoTypes : TypeRef ...
	|	AbsFldWoTypes : SimpleTupleType
AbsFldWTypes	::=	AbsFldWType
	|	(AbsFldWType (, AbsFldWType)$^+$)
AbsFldWType	::=	AbsFldMods$^?$ Id IsType
AbsFldWoTypes	::=	AbsFldWoType
	|	(AbsFldWoType (, AbsFldWoType)$^+$)
AbsFldWoType	::=	AbsFldMods$^?$ Id
ApiFldDecl	::=	ApiFldMods$^?$ Id IsType

A.13 Expressions

Expr	::=	AssignLefts AssignOp Expr
	|	OpExpr
	|	DelimitedExpr
	|	FlowExpr
	|	`fn` ValParam IsType$^?$ Throws$^?$ $\Rightarrow$ Expr
	|	Expr as TypeRef
	|	Expr `asif` TypeRef
AssignLefts	::=	[() AssignLeft (, AssignLeft)*)
	|	AssignLeft
AssignLeft	::=	SubscriptExpr
	|	FieldSelection
	|	BindId
SubscriptExpr	::=	Primary [ExprList$^?$]

```
FieldSelection   ::= Primary . Id
OpExpr           ::= EncloserOp OpExpr? EncloserOp?
                   | OpExpr EncloserOp OpExpr?
                   | Primary
EncloserOp       ::= Encloser
                   | Op
Primary          ::= Comprehension
                   | Id ⟦ StaticArgList ⟧
                   | BaseExpr
                   | LeftEncloser ExprList? RightEncloser
                   | Primary [ ExprList? ]
                   | Primary . Id ( ⟦ StaticArgList ⟧ )? TupleExpr
                   | Primary . Id ( ⟦ StaticArgList ⟧ )? ( )
                   | Primary . Id
                   | Primary ^ BaseExpr
                   | Primary ExponentOp
                   | Primary TupleExpr
                   | Primary ( )
                   | Primary Primary
                   | TraitType . coercion ( ⟦ StaticArgList ⟧ )? ( Expr )
FlowExpr         ::= exit Id? ( with Expr )?
                   | Accumulator ( [ GeneratorList ] )? Expr
                   | atomic AtomicBack
                   | tryatomic AtomicBack
                   | throw Expr
AtomicBack       ::= AssignLefts AssignOp Expr
                   | OpExpr
                   | DelimitedExpr
GeneratorList    ::= Generator ( , Generator )*
Generator        ::= GeneratorBinding
                   | Expr
GeneratorBinding ::= Id ← Expr
                   | ( Id , IdList ) ← Expr
```

A.14 Expressions Enclosed by Keywords

```
DelimitedExpr    ::= TupleExpr
                   | ObjectExpr
                   | DoExpr
                   | LabelExpr
                   | WhileExpr
                   | ForExpr
                   | IfExpr
                   | CaseExpr
                   | TypecaseExpr
                   | TryExpr
TupleExpr        ::= ( ( Expr , )* ( Expr ... , )? Binding ( , Binding )* )
                   | NoKeyTuple
NoKeyTuple       ::= ( ( Expr , )* Expr ... )
```

| | ((Expr ,)* Expr)
ObjectExpr ::= `object` Extends? GoInAnObject `end`
DoExpr ::= (DoFront `also`)* DoFront `end`
DoFront ::= (`at` Expr)? `atomic`? `do` Block?
LabelExpr ::= `label` Id Block `end` Id
WhileExpr ::= `while` Generator DoExpr
ForExpr ::= `for` GeneratorList DoFront `end`
IfExpr ::= `if` Generator `then` Block ElifClause* ElseClause? `end`
| [() `if` Generator `then` Block ElifClause* ElseClause `end`? [)]
ElifClause ::= `elif` Expr `then` Block
ElseClause ::= `else` Block
CaseExpr ::= `case` Expr Op? `of` CaseClauses CaseElseClause? `end`
CaseClauses ::= CaseClause+
CaseClause ::= Expr ⇒ Block
CaseElseClause ::= `else` ⇒ Block
TypecaseExpr ::= `typecase` TypecaseBindings `of` TypecaseClauses CaseElseClause? `end`
TypecaseBindings ::= (BindingList)
| Binding
| Id
BindingList ::= Binding (, Binding)*
Binding ::= BindId = Expr
TypecaseClauses ::= TypecaseClause+
TypecaseClause ::= TypecaseTypeRefs ⇒ Block
TypecaseTypeRefs ::= (TypeRefList)
| TypeRef
TryExpr ::= `try` Block Catch? (`finally` Block)? `end`
Catch ::= `catch` Id CatchClauses
CatchClauses ::= CatchClause+
CatchClause ::= TraitType ⇒ Block
Comprehension ::= { Expr | GeneratorList }
| { Entry | GeneratorList }
| < Expr | GeneratorList >
| [ArrayComprehensionClause+]
Entry ::= Expr ↦ Expr
ArrayComprehensionLeft ::= IdOrInt ↦ Expr
| (IdOrInt , IdOrIntList) ↦ Expr
ArrayComprehensionClause ::= ArrayComprehensionLeft
| GeneratorList
IdOrInt ::= Id
| IntLiteral
IdOrIntList ::= IdOrInt (, IdOrInt)*
BaseExpr ::= NoKeyTuple
| Literal
| Id
| `self`
ExprList ::= Expr (, Expr)*

A.15 Local Declarations

Block	::=	BlockElement$^+$
BlockElement	::=	LocalVarFnDecl
	|	Expr (, GeneratorList)$^?$
LocalVarFnDecl	::=	LocalFnDecl$^+$
	|	LocalVarDecl
LocalFnDecl	::=	LocalFnMods$^?$ FnHeaderFront FnHeaderClause = Expr
LocalVarDecl	::=	LocalVarWTypes InitVal
	|	LocalVarWTypes
	|	LocalVarWoTypes = Expr
	|	LocalVarWoTypes : TypeRef ... InitVal$^?$
	|	LocalVarWoTypes : SimpleTupleType InitVal$^?$
LocalVarWTypes	::=	LocalVarWType
	|	(LocalVarWType (, LocalVarWType)$^+$)
LocalVarWType	::=	`var` ? Id IsType
LocalVarWoTypes	::=	LocalVarWoType
	|	(LocalVarWoType (, LocalVarWoType)$^+$)
LocalVarWoType	::=	`var` ? Id

A.16 Literals

Literal	::=	( )
	|	NumericLiteral
	|	CharLiteral
	|	StringLiteral
RectElements	::=	Expr MultiDimCons*
MultiDimCons	::=	RectSeparator Expr
RectSeparator	::=	; +
	|	Whitespace

A.17 Types

IsType	::=	: TypeRef
TypeRef	::=	TraitType
	|	ArrowType
	|	TupleType
	|	(TypeRef ?)
TraitType	::=	DottedId (⟦ StaticArgList ⟧)$^?$
	|	{ TypeRef ↦ TypeRef }
	|	< TypeRef >
	|	TypeRef [ArraySize$^?$]
	|	TypeRef ^ IntLiteral
	|	TypeRef ^ (ExtentRange (× ExtentRange)*)
ArrowType	::=	TypeRef → TypeRef Throws$^?$
TupleType	::=	((TypeRef ,)* (TypeRef ... ,)$^?$ KeywordType (, KeywordType)*)

	|	((TypeRef ,)* TypeRef ...)
	|	SimpleTupleType
KeywordType	::=	Id = TypeRef
SimpleTupleType	::=	(TypeRef , TypeRefList)
TypeRefList	::=	TypeRef (, TypeRef)*
StaticArgList	::=	StaticArg (, StaticArg)*
StaticArg	::=	Op
	|	TypeRef
	|	(StaticArg)
ArraySize	::=	ExtentRange (, ExtentRange)*
ExtentRange	::=	StaticArg? # StaticArg?
	|	StaticArg? : StaticArg?
	|	StaticArg

A.18 Symbols and Operators

AssignOp	::=	:=
	|	Op =
Accumulator	::=	$\sum$ | $\prod$ | `BIG` Op

A.19 Identifiers

IdList	::=	Id (, Id)*
Name	::=	Id
	|	`opr` Op
DottedId	::=	Id (. Id)*
BindId	::=	Id
	|	-

Bibliography

[1] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.