The Emerald Programming Language[1]

Report

Norman C. Hutchinson, Rajendra K. Raj,
Andrew P. Black, Henry M. Levy, and Eric Jul[2]

Revised Version of the Technical Report
October 1991

Department of Computer Science
University of British Columbia
Vancouver BC, Canada V6T 1Z2

Welcome added by
Olav Johan Ekblom & Gaute Svanes Lunde
University of Oslo
November 2020

Abstract:

The programming language Emerald was designed and developed to demonstrate that the object-based style of programming can be incorporated both elegantly and efficiently in a distributed programming environment. At the same time, Emerald is a modern programming language providing excellent features for abstractions and polymorphism. Primarily a language for distributed environments, Emerald includes features for dealing with the location of objects, and extends exception handling mechanisms to recovering from partial failures of distributed systems. This report presents a concise description of the Emerald programming language.

[1]This work was supported in part by the Natural Science and Engineering Research Council of Canada under Grants No. [?], by the National Science Foundation under Grants No. MCS-8004111 and DCR-8420945, by Københavns Universitet (the University of Copenhagen), Denmark under Grant J.nr. 574-2,2, by Digital Equipment Corporation External Research Grants, and by an IBM Graduate Fellowship.

[2]Authors' current addresses: Norman Hutchinson, Department of Computer Science, University of British Columbia, Vancouver, BC, Canada V6T 1Z4. Andrew Black, Digital Equipment Corporation, 550 King Street, Littleton, MA 01460. Eric Jul, DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark.

Contents

1 Introduction

The primary goal of Emerald [**?**, **?**, **?**, **?**] is to simplify distributed programming through language support while providing acceptable performance and flexibility both in local and distributed environments. Emerald also demonstrates that the object-based model of programming can be incorporated both elegantly and efficiently in distributed systems. Emerald draws heavily upon the experience gained from Smalltalk [**?**], the Argus Language and System [**?**] and, in particular, the Eden system [**?**, **?**] and the Eden Programming Language (EPL) [**?**].

Featuring an object-oriented style of programming, Emerald presents a unified semantic view of objects appropriate for private, local, data-only objects as well as shared, remote, concurrently-executing objects. The nature of objects in Emerald is similar to that in Smalltalk [**?**], i.e., all data items are objects with a uniform semantic model for operations on them, but Emerald does not have any notion of *class*. Emerald was explicitly designed to support data abstraction: all typing of objects is at an abstract level and does not depend on the implementation chosen. Abstract typing aids in the dynamic construction of distributed programs by allowing any object in a large, possibly distributed, program to be replaced by any other type-consistent object. Type consistency or *conformity* is an important aspect of Emerald, and is discussed below. Another advantage of treating types as first class objects is that it makes polymorphism inherent in Emerald.

Recognizing *location* as an important attribute of an object in distributed programs, Emerald gives the programmer access to the location of objects through primitives that permit the inspection and selection of location. Alternatively, when desired, the location details may be left to the reasonably-chosen system-defaults. However, this recognition of the importance of location for distributed programming has its drawbacks, viz., the semantics of Emerald are complicated both because location is apparent and because systems may be partially unavailable.

This report defines the Emerald programming language. The Emerald approach to programming is discussed in [**?**], where several examples of Emerald programs may be found.

2 Notation and Vocabulary

This report uses a slight variation of the commonly-used *Extended Backus Naur Form* (EBNF) to express the syntax of Emerald. Terminal symbols in Emerald (i.e. symbols in its vocabulary) are shown in the syntax descriptions in typewriter font as `,` or `"`, or in bold font for reserved words like **loop**. Non-terminal symbols are denoted by italicized English words that intuitively illustrate the meaning of the syntactic constructs. In EBNF, alternatives are indicated by |:

A | B

means choosing either A or B; optional elements are shown using square brackets [ ]:

[A]

means either zero or one A; and (possibly empty) sequences by braces { }:

{ A }

means zero or more repetitions of A.

Emerald is case insensitive—the case of input letters is significant only in character literals and string literals.

2.1 Literals

Literal objects in Emerald are divided into the following categories:

Numeric

numericLiteral	::=	`0x` { *hexdigit* }
	|	`0` { *octdigit* }
	|	*digit* { *digit* } `.` { *digit* }
	|	*digit* { *digit* }
digit	::=	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
octdigit	::=	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`
hexdigit	::=	*digit* | `a` | `b` | `c` | `d` | `e` | `f`

Numeric literals without a decimal point (`.`) denote objects of the predefined type *Integer*; those with decimal points denote objects of the predefined type *Real*. Literals beginning with `0x` are interpreted in hexadecimal; literals beginning with `0` are interpreted in octal, For example, 12, 014, and 0xc are *Integer* literals representing the decimal number 12, and 2.1 and 215.45 are *Real* literals.

Booleans

booleanLiteral ::= **true** | **false**

The reserved words **true** and **false** refer to the two objects of builtin type *Boolean*.

Nil

nilLiteral ::= **nil**

The reserved word **nil** refers to the distinguished nil object, whose type is *None*.

Characters and Strings

characterLiteral	::=	' *ccharacter* '
ccharacter	::=	*AnyCharacterExceptBackSlash*
	|	*scharacter*
stringLiteral	::=	" { *scharacter* } "
scharacter	::=	*AnyCharacterExceptDoubleQuoteOrBackSlash*
	|	\ *anyCharacterExceptUpArrow*
	|	\^*anyCharacter*
	|	*oneTwoOrThreeOctalDigits*

A character literal denotes an object of builtin type *Character* and consists of a single character written within single quotes. The character \ permits the introduction of escape sequences for the entry of special characters. \\ generates a single \ character, \^C where C is any character generates a control

character in an implementation-defined manner[3]. Standard escape sequences as in C (\n, \t, etc.) are permitted, including one, two, or three octal digits following a \ which represents a character by giving its numerical (octal) equivalent. \ followed by any other character stands for that character.

Examples of characters are `'A'`, `'r'`, `'\^C'`, `'\\'`, `'\^?'`, `'\^J'`, `'\n'` and `'\012'`; the last three examples equivalently denote the newline character.

A string literal denotes an object of the builtin type *String* and consists of a possibly empty sequences of characters enclosed in double quotes, using the same escape conventions as character literals. Examples of strings are `"Emerald City"`, `"The \"Evergreen\" State"`, and `""`.

Vectors

vectorLiteral ::= `{` *expression* { `,` *expression* } [`:` *typeExpression*] `}`

A vector literal is a sequence of expressions enclosed in curly braces, representing immutable (read-only) vectors. The type of the expression is *ImmutableVector.of[t]*, where t is either:

- the type expression (if present), otherwise
- the syntactic type of the elements, if they are all the same, otherwise
- *Any*

Examples of vector literals are `{1, 3, 5}` (with type *ImmutableVector.of[Integer]*), `{1, 3, 5 : Any}` with type *ImmutableVector.of[Any]* , and `{ 1, 'a', true}` (with type *ImmutableVector.of[Any]*).

Objects

Emerald objects are created using object literals (also known as object constructors) or one of the syntactic extensions which translate to an object constructor, which include class constructors, record constructors and enumeration constructors. These are discussed in Section **??**.

Types

Emerald types are created using type constructors, which are described in Section **??**.

2.2 Identifiers

An Emerald identifier is a non-empty sequence of letters, digits and the underscore character "_", beginning with a letter or the underscore character. Identifiers are case-insensitive and significant up to 64 characters in length. Identifiers are used as reserved words, constant names, variable names, operation names (cf. Section **??**), parameter names, and local names of objects.

2.3 Reserved Identifiers

Reserved identifiers are identifiers that have been reserved for special use and may not be used otherwise as identifiers. Reserved identifiers are further subdivided into keywords and literals.

[3]In ASCII implementations, \^C generates the ascii character formed by turning off the upper 2 bits in the character code for C. Thus, \^J is the newline character, and \^@ is the null character. The exception is the delete character, (octal 177) which is generated by the sequence \^?.

Literals

The reserved literal identifiers are:

false	**nil**	**self**	**true**

Keywords

Emerald keywords are used to delimit language constructs; for example, the keywords **loop** and **end loop** are used to enclose a loop body.

The reserved keywords are:

all	**and**	**as**	**assert**
at	**attached**	**awaiting**	**begin**
builtin	**by**	**checkpoint**	**class**
codeof	**confirm**	**const**	**else**
elseif	**end**	**enumeration**	**exit**
export	**external**	**failure**	**field**
fix	**for**	**forall**	**from**
function	**if**	**immutable**	**initially**
isfixed	**islocal**	**locate**	**loop**
monitor	**move**	**nameof**	**new**
object	**op**	**operation**	**or**
primitive	**process**	**record**	**recovery**
refix	**restrict**	**return**	**returnandfail**
signal	**syntactictypeof**	**suchthat**	**then**
to	**typeobject**	**typeof**	**unavailable**
unfix	**var**	**view**	**visit**
wait	**welcomable**	**welcome**	**when**
where	**while**		

2.4 Operators

operatorCharacter	::=	`!`	|	`#`	|	`&`	|	`*`
	|	`+`	|	`-`	|	`/`	|	`<`
	|	`=`	|	`>`	|	`?`	|	`@`
	|	`^`	|	`|`	|	`~`		
operator	::=	*operatorCharacter* { *operatorCharacter* }						

An operator is a non-empty sequence of operator characters. Operators are used as punctuation and as operation names.

Reserved Operators

Reserved operators are operators that have been reserved for special use and may not be used otherwise as operators or operation names. Reserved operators are further subdivided into expression operators and punctuation.

Expression operators

Expression operators are used within expressions. The reserved expression operators are[4]:

```
<>      ==      !==      |>
```

These operators are described in Section **??**.

Punctuation operators

Punctuation operators are used to delimit language constructs. The reserved punctuation operators are:

```
<-      ->
```

2.5 Separators

Separators are sequences consisting of only spaces, tabs, and newlines; they are used to separate consecutive language tokens. Consecutive identifiers, operators and/or numeric literals must be separated by at least one separator.

2.6 Comments

Comments in Emerald are line-oriented. A comment starts on any line with the comment delimiter, `%`, and terminates at the end of the same line. The comment delimiter is ignored within string and character literals. A comment is lexically equivalent to a separator.

3 Declarations

Every identifier used in Emerald must be declared. There are two (general purpose) declarative forms: one for constants and one for variables.

declaration	::=	*variableDeclaration*
	|	*constantDeclaration*
constantDeclaration	::=	[**attached**] **const** *identifier* [*: type*] *initializer*
variableDeclaration	::=	[**attached**] **var** *identifierList : type* [*initializer*]
identifierList	::=	*identifier* { **,** *identifier* }
initializer	::=	← *expression*

A constant declaration introduces an identifier that refers to a single object throughout its lifetime. If the optional type clause is present, the type of the identifier is the value of the *type* expression, and the type of the *initializer* expression must conform to the type of the identifier. If the type clause is not present, the type of the identifier is the syntactic type of the *initializer* (cf. Section **??**). While a constant identifier always refers to the same object; the object's state may change if it is mutable.

[4]The symbols typeset as ←, →, <>, and ▷ are typed using `<-`, `->`, `*>`, and `*>` respectively. The potential ambiguity caused by using the same symbol for conforms and matches is resolved by context.

A variable declaration introduces a new variable for each element of the *identifierList*. The type of each identifier is the value of the *type* expression. When an initializer clause is present the type of the *initializer* must conform to the type of the identifiers, and each variable in the list is assigned the value of the *initializer* expression. When not explicitly initialized, variables initially name the object **nil**. A variable identifier may have its value changed by assignment (cf. Section **??**).

The optional **attached** permits the programmer to provide one-way attachment between objects; the relocation of an object additionally relocates all co-located objects attached to it (cf. Section **??**).

A declaration is an executable construct, and the initialization of constants and variables is performed each time the declaration is executed.

3.1 Scope

An identifier name is visible throughout the scope in which it is declared, not just textually after that declaration. The following constructs open new scopes for identifiers, and identifiers are imported implicitly into nested scopes where they are not redefined:

- if, elseif, and else clauses (cf. Section **??**)
- loop statement bodies (cf. Section **??**)
- blocks, unavailable, and failure handlers (cf. Section **??**)
- operation definitions and signatures (cf. Section **??**)
- typeobject constructors (cf. Section **??**)
- object constructor, process, initially-block and recovery-block definitions (cf. Section **??**)

Since object constructors and typeobject constructors create new objects that are independent of their enclosing referencing environment (closures), identifiers imported into these constructs are treated specially. When the type or object constructor is executed, all imported identifiers are made constant. Throughout the lifetime of the created type or object, these identifiers will have the values that they had when the object constructor was executed. Consider the following example:

```
for i : Integer ← 0 while i < 10 by i ←i + 1
    o ← object trivial
        export operation getI → [r : Integer]
            r ← i
        end getI
    end trivial
end for
```

This loop creates ten identical objects, except that the value of the identifier i is different for each object. Once the first object (whose $i = 0$) is created, changes to the loop control variable i are not visible to it, as the i that it sees was made constant when that object was created.

While identifiers are in scope for the entire block in which they are declared (even textually before their declaration), they are normally given values as their declarations are executed. For example, the following operation definition is correct as far as scope is concerned, but the final value of x will be **nil** while the final value of of y will be 6.

```
operation nonintuitive
    const x ← y
    const y ← 6
end nonintuitive
```

There are two exceptions to this rule:

1. In a compilation unit, all exported identifiers get their values simultaneously. This permits the construction of mutually recursive object structures.
2. When typechecking an invocation of a polymorphic operation, the values of all identifiers introduced in whee and for all clauses get their values simultaneously. This permits the construction of the mutually recursive type objects that are often necessary when creating polymorphic operations.

4 Expressions

Expressions are Emerald constructs that denote objects.

4.1 Literals and Identifier Expressions

expression	::=	*literal*
	|	*constantIdentifier*
	|	*variableIdentifier*

A literal expression directly denotes an object. A constant identifier names the object it was initialized to while a variable identifier names the object most recently bound to it.

4.2 Operator Expressions

Before examining Emerald expressions that involve operators, we define the precedence of the operators used. In Table **??**, the operators are ordered by increasing precedence. Operators of the same precedence level are evaluated from left to right.

4.3 Reserved Operators

The reserved operators have meanings defined by the language, and may not be redefined.

expression	::=	*expression* { *reservedop expression* }
reservedop	::=	== | ! == | o> | **or** | **and**

== evaluates to **true** if the two expressions denote the same object; ! == is its opposite. The o> operator compares its two operands (each of type *type*) for conformity, i.e., it evaluates to **true** if the left operand conforms to the right operand (cf. Section **??**).

The operator **and** is a *conditional and* and evaluates as follows: if the left operand evaluates to **false**, the result is **false**; otherwise, the result is the value of the right expression. The operator **or** is a *conditional or* and evaluates as follows: if the left operand evaluates to **true**, the result is **true**; otherwise, the result is the value of the right expression.

Precedence Level	Operator	Operation
1	**view-as**	Narrow or widen object's type
	restrict-to	Permanently narrow or widen object's type
2	|	Logical or
	or	Logical conditional (short-circuit) or
3	&	Logical and
	and	Logical conditional (short-circuit) and
4	!	Logical negation
5	==, !==	Object identity and distinction
	<>	Type conformity
	=, ! =, <, <=, >=, >	Relational operators
6	+,−	Additive operators
7	*, /	Multiplicative operators
	#	Modulus
	User-defined	
8	−, ~	Arithmetic negation
	isfixed	Checks if object is fixed at node
	islocal	Checks if object is on the local node
	locate	Finds a possible location of the operand
	awaiting	Processes waiting on condition
	codeof	Concrete type (implementation) of an object
	nameof	Name of an object
	typeof	Type of an object
	syntactictypeof	Compile time type of an expression
	welcome	Obtains a reference to an incoming object conforming to the operand

Table 1: Precedence of Emerald Operators

expression ::= **locate** *expression*
| **isfixed** *expression*
| **islocal** *expression*
| **awaiting** *expression*
| **codeof** *expression*
| **nameof** *expression*
| **typeof** *expression*
| **syntactictypeof** *expression*
| **welcome** *expression*

isfixed *e* evaluates to **true** if *e* is currently fixed at a site, otherwise it evaluates to **false**. **islocal** *e* evaluates to true if *e* is currently on the local node, and **false** otherwise. **locate** *e* evaluates to an object (of type *Node*) that gives a location of the operand object during the execution of this expression. This is explained in detail in Section **??**.

The **awaiting** operator takes as its operand an expression of type *Condition* and returns **true** if at least one process is suspended on the operand condition, and **false** otherwise (cf. Section **??**).

codeof, **nameof**, and **typeof** return the concrete type object (a *ConcreteType*), the name (a *String*), or the type (a *Signature*) of any object (including **nil**). **syntactictypeof** an expression returns the compile time type of an expression (cf. Section **??**). See Appendix **??** for a description of these builtin types.

welcome blocks until a welcomable object that conforms to the abstract type evaulated from the following expression is moved onto the local node, and returns a reference to that object. If the foreign object arrives from a discovered node, the node graph of the local node is merged with the node graph of the object's origin.

4.4 Invocations

expression ::= *invocation*

Any invocation which returns exactly one object may be used as an expression. Invocations are discussed in Section **??**.

4.5 Other Operators

All other operators are translated into object invocations. Each occurrence of a unary operator is translated into an invocation of the operand with the invocation name being the name of the operator and with no arguments. Each occurrence of a binary operator is translated into an invocation of the left operand with the invocation name being the name of the operator and with a single argument which is the right operand. For example: $!e$ is translated as $e.!$, and $a + b$ is translated as $a.+[b]$.

4.6 Field selection

Emerald supports syntactic sugar to facilitate accessing the data components of objects. Conventionally, a visible data component named *f* will have an operation named *getF* which returns the value of the component and an operation *setF* which modifies the value of the component. Two syntactic forms support this convention by making the invocation of get and set operations more convenient.

fieldSelection ::= *expression* `$` *identifier*

In an expression context (as an r-value) *a$f* is translated as *a.getF* while in an assignment context (as an l-value) *a$f* ← c is translated as a.setF[c]. The get operation always takes no arguments and returns one result, while the set operation always takes one argument and returns no results.

4.7 Subscripts

By convention, subscriptable values like *Vectors* and *Arrays* will have an operation *getElement* to retrieve values and an operation *setElement* to store values. This convention is supported by the subscripting expression syntax:

subscript ::= *expression* `[` *expression* { `,` *expression* } `]`

The variable number of expressions inside the square brackets may be used to support subscriptable values of higher dimension, or to select ranges of values. Such interpretation is up to the implementor of the subscripted object.

In an expression context (as an r-value) *a*[*b, c, d*] is translated as *a.getElement*[*b, c, d*], while in an assignment context (as an l-value) a[b, c, d] ← e is translated as a.setElement[b, c, d, e].

4.8 Type widening and narrowing

expression ::= **view** *expression* **as** *typeExpression*
| **restrict** *expression* **to** *typeExpression*

The **view** expression permits an object to be regarded as being of a different type, subject to the restriction that no object expression be viewed as a type it does not conform to. In other words, this expression permits the user to narrow or widen the type of an object. The restrict expression is similar, except that it is later impossible to widen the type of the resulting object reference to a type wider than that denoted by *typeExpression*.

5 Statements

5.1 Assignment statement

assignment ::= *identifierList* ← *expressionList*
| [*identifierList* ←] *procedureInvocation*

In the first case, the expression list is evaluated to yield a number of objects. In the latter case, the procedure invocation is performed, resulting in a number of objects (possibly 0). In both cases, the resulting objects are positionally bound to the variables on the left side of the assignment operator. The number of variables on the left side and the number of resulting objects on the right must be equal and must positionally conform in type (see Section **??**).

5.2 Selection

ifStatement ::= **if** *expression* **then**
declarationsAndStatements
{ **elseif** *expression* **then**
declarationsAndStatements }
[**else**
declarationsAndStatements]
end if

The expressions following the **if** and optional **elseif** keywords (which must be of type *Boolean*) are evaluated in textual order until one evaluates to **true** or they are exhausted. In the former case, the statements following the next **then** keyword are executed, and in the latter case, the statements following the **else** keyword (when present) are executed.

5.3 Iteration

5.3.1 Loop statement

loopStatement ::= **loop**
declarationsAndStatements
end loop

The statements bracketed by **loop** and **end loop** are executed repeatedly until an exit statement at the same level of nesting is executed.

5.3.2 Exit statement

exitStatement ::= **exit** [**when** *expression*]

This statement terminates the execution of the textually inner-most enclosing loop; this statement is invalid if there is no such loop. The simple **exit** provides an unconditional exit from the loop; the optional **when** clause permits a conditional exit if the evaluated expression, which must be of type *Boolean*, evaluates to **true**.

5.3.3 For statement

Emerald has two forms of the for statement. These are conveniences whose semantics are defined in terms of their translations as given below.

forStatement ::= **for** (*initial* : *condition* : *step*)
declarationsAndStatements
end for

This is equivalent to:

```
begin
   initial
   loop
      exit when !condition
      begin
         declarationsAndStatements
      end
      step
   end loop
end
```

forStatement ::= **for** *identifier* **:** *typeExpression initialization* **while** *condition* **by** *step*
declarationsAndStatements
end for

This is equivalent to

```
begin
   var identifier : typeExpression initialization
   loop
      exit when !condition
      begin
         declarationsAndStatements
      end
      step
   end loop
end
```

5.4 Compound statement

compoundStatement ::= **begin**
blockBody
end

blockBody ::= *declarationsAndStatements*
[*unavailableHandler*]
[*failureHandler*]

The compound statement permits several statements to be grouped together as one composite statement. In addition, it permits suitable recovery code to be attached in the form of handlers dealing with object unavailability and failures (cf. Sections **??** and **??**).

5.5 Assertions

assertStatement ::= **assert** *expression*

The expression, whose type must be *Boolean*, is evaluated. If the result is **false**, a failure occurs (as explained in Section **??**). If the result is **true**, the statement has no further effect.

5.6 Concurrency

Concurrency features are described in detail in Section **??** and are briefly outlined here. Each object may have an optional process associated with it; this process is created after the termination of the object's **initially** section and it executes until it reaches the end of its block. Any object which is defined as **monitor** guarantees mutual exclusion in the execution of all of its operations. Objects of system-implemented type *Condition* may be used for synchronization within monitored objects; the semantics of condition waits and signals follows that proposed by Hoare. Note that a condition object used in a wait or signal statement or an awaiting expression must be used only inside the monitored object by which it was created.

5.6.1 Wait statement

waitStatement ::= **wait** *expression*

The wait statement must be executed inside a monitored object, and the type of the *expression* must be *Condition*. The process executing the wait is suspended on the condition object, and the monitor lock is passed on to the next process waiting to enter the monitor; if no process is waiting to enter, the monitor lock is released.

5.6.2 Signal statement

signalStatement ::= **signal** *expression*

The type of the *expression* must be *Condition*. If the condition object has one or more processes suspended on it, one of these processes will be resumed, the monitor lock will be passed to it, and the signalling process will be placed at the head of the monitor entry queue. If the condition object does not have any processes suspended on it, the signal statement has no effect.

5.7 Location-related Statements

Mobility is an important feature of Emerald ([**?**, **?**]) and is supported via several language constructs. The statements that permit the programmer to specify and change the location of objects are discussed below.

5.7.1 Fix statement

fixStatement ::= **fix** $expression_1$ **at** $expression_2$

The object named by $expression_1$ is moved to the location of the object named by $expression_2$, and forced to remain there; the *unfix* and *refix* statements described below permit the movement of previously fixed objects. Attempts to move or fix previously fixed objects result in failures (cf. Section **??**).

5.7.2 Unfix statement

unfixStatement ::= **unfix** *expression*

The object denoted by the *expression* is made free to move. It is not an error to unfix an object not currently fixed at any location.

5.7.3 Refix statement

refixStatement ::= **refix** *expression*$_1$ **at** *expression*$_2$

This statement unfixes the object named by *expression*$_1$ and fixes it at some (presumably different) location; the refix is performed atomically.

5.7.4 Move statement

moveStatement ::= **move** *expression*$_1$ **to** *expression*$_2$

The object denoted by *expression*$_1$ is moved to the current location of the object denoted by *expression*$_2$. The statement fails if the object denoted by *expression*$_1$ is fixed. The **move** primitive is actually a hint, i.e., the implementation is not required to perform the move suggested. On the other hand, the primitives **fix** and **refix** have stronger semantics, and when they succeed, the object must stay at the specified destination until explicitly unfixed or refixed.

5.8 The Checkpoint statement

checkpointStatement ::= **checkpoint**

The checkpoint statement permits an object to store its state on permanent storage. On node failure and subsequent recovery, the object uses this stored state and continues from that state, first performing any programmer-specified recovery action.

5.9 The Return Statement

returnStatement ::= **return**

This statement is used to terminate the execution of an operation and return to the invoking object. It may also be used to prematurely terminate an initially, process, or recovery section.

5.10 The ReturnAndFail statement

returnAndFailStatement ::= **returnandfail**

The return and fail statement is analogous to the return statement, but in addition, it permits the invoked object to report a failure to the invoking object. The return happens first so the state of the invoked object is not affected by the failure (cf. Section **??**).

5.11 The Primitive Statement

primitiveStatement ::= **primitive** [**self**] [**var**] *primitiveImplementation*
[*identifierList*] ←[*identifierList*]
primitiveImplementation ::= { *stringLiteral* | *integerLiteral* }

This statement is used to implement lower-level calls to the underlying operating system and to implement certain operations on builtin-types. Primitive statements are used only in the implementation of the builtin Emerald types and should not be used by the Emerald programmer directly.

The optional **self** means that the receiving object ought to be pushed onto the stack before the argument variables, otherwise it is not pushed. The optional **var** means that all the things pushed onto the stack and all the results are assumed to be variables represented as 2 words: a data pointer and a concrete type pointer. Otherwise all arguments and results are assumed to just be data (1 word). The strings and integers in the primitiveImplementation are placed in the instruction stream as a sequence of bytes. Strings are looked up as either names of bytecodes in ".../lib/bcdef" or system defined operations in ".../lib/jsdef" or ".../lib/ccdef". The list of identifiers on the right provide the arguments for the primitive, while the list of identifiers on the left get the results of the primitive. The following code is generated:

- If self is present, the push the receiving object onto the stack
- Push the value of each of the argument identifiers onto the stack, left to right
- Emit the literals in the primitive implementation into the instruction stream
- Pop results off the stack storing them in the result identifiers, left to right

There is no check that the implementation of the primitive actually expects the number of arguments or returns the number of results that the argument and result identifier lists mention.

6 Operations

Emerald objects communicate with one another only through the invocation of operations. This section describes the definition and invocation of operations.

6.1 Defining operations

operationSignature ::= *operationKind operationName* [*parameterList*]
["→" *resultList*] { *clause* }
operation ::= [**export**] *operationSignature*
blockBody
end *operationName*
operationKind ::= **op** | **operation** | **function**
parameterList ::= [*parameter* { , *parameter* }]
resultList ::= *parameterList*
parameter ::= [**attached**] [*identifier :*] *type*
clause ::= *whereClause*
| *forallClause*
| *suchthatClause*

Emerald provides two kinds of operations: procedural and functional. Procedural operations are heralded by the keyword **operation** or **op**, while the keyword **function** indicates a functional operation. In declaring a functional operation, the programmer asserts that the operation is side-effect free, i.e., the abstract state of the system is not modified by the execution of the operation[5]. Note that the burden is on the programmer; the Emerald system may perform optimizations on function invocations that are incorrect if the operation has side effects.

The operation signature (cf. Section **??**) includes the operation name and the number, names and abstract types of its arguments and results. An object may implement multiple operations with the same name, provided that the number of arguments that they accept is different. Where clauses serve to introduce new names for types whose scope is the entire operation signature (and body if present). For all clauses declare new type identifiers. Such that clauses impose constraints on the formal parameters. These clauses are primarily useful for the implementation of polymorphic types and are defined in Section **??**.

6.2 Parameter Passing

The Emerald language uses call-by-object-reference semantics for all invocations, local or remote. That is, a reference to the argument object is passed to the called procedure. There are no restrictions on the types of objects that can be passed.

6.3 Making Invocations

procedureInvocation	::=	*expression* **.** *operationName* [*argumentList*]
operationName	::=	*identifier* | *operator*
argumentList	::=	**[** *argument* { **,** *argument* } **]**
argument	::=	[**move** | **visit**] *expression*

An invocation specifies the target object, the operation to be invoked, and any arguments. When an invocation returns results, they are assigned to variables using an assignment (see Section **??**).

Because Emerald objects are mobile, it may be possible to optimize an invocation by avoiding many remote references by moving argument objects to the site of a remote invocation. The keywords **move** and **visit** suggest that the expression be physically moved to the same node as the invoked object; **visit** further suggests that the expression be moved back when the invocation returns. These two parameter passing modes are called *call-by-move* and *call-by-visit* respectively. Neither mode affects the location-independent semantics of the invoked operation.

Executing an operation invocation involves:

- evaluating the invocation target expression,
- evaluating the argument objects and then positionally assigning them to the formal parameters of the operation,
- executing the body of the operation in the context of the target object of the invocation, and
- returning the final values of any output parameters of the invocation.

[5]Note that Emerald does not rule out the possibility of the operation having concrete side-effects (sometimes termed *beneficial* or *benevolent* side-effects).

7 Types

A type is defined as a collection of operation signatures, where each operation signature includes the operation name, and the names and types of its arguments and results. Types, being objects, are first-class citizens in Emerald. Each type object exports a function without arguments called *getSignature* that returns an object of the predefined **Signature** type. In other words, any object that conforms to the following type:

immutable typeobject *type*
  function *getSignature* → [*Signature*]
end *type*

is a type. Note that each object with type *signature* has a getSignature operation that returns self, thus Signatures are Types.

7.1 Type Constructors

Signatures are created using type constructors. A type constructor has the following structure:

typeConstructor ::= [**immutable**] **typeobject** *typeIdentifier*
{ *operationSignature* }
end *typeIdentifier*

Operation signatures have been defined in Section **??**, however in type constructors, the identifiers in parameter declarations may be omitted. An immutable type implies that its objects are abstractly immutable, i.e. its objects cannot change their state over time. For example, the predefined type **Integer** is immutable because its objects represent integer values which cannot change; for instance, the integer 3 cannot be changed to the integer 4.

7.2 Syntactictypeof and typeof

The syntactictypeof operator evaluates to the statically determined type of an expression. That is, the type of the expression as it can be determined by the compiler. This operation is always evaluated at compile time. In contrast the **typeof** operator returns the most accurate type of the expression at run time, which can be a wider type than the syntactic type, if either implicit or explicit narrowing has occurred. This most accurate type is also called the *best fitting* type of the expression. See section **??**. It is always the case that for any expression e, **typeof** e $\circ\!\!>$**syntactictypeof** e.

7.3 Conformity

Conformity is the basic relationship between types. A type S conforms to a type T (written $S \circ\!\!> T$) if:

1. S is immutable if T is immutable.
2. For each operation o_T in T, there exists an operation o_S in S with the same name and number of arguments, and
3. o_T and o_S have the same number of results, and
4. The types of the results of o_S operations conform to the types of the results of o_T, and

5. The types of the arguments of o_T conform to the types of the arguments of o_S (i.e., arguments must conform in the opposite direction).

If either S or T is recursive (the definition of at least one of its operations uses its own name), then the previous checks must be performed under an assumption that $S \circ\!\!> T$.

This simple description of conformity suffices for all invocations that do not involve parametric polymorphism. Discussion of polymorphic operations and the extensions to the type checking rules required to type check them is deferred until Section **??**.

Some types in the system are exceptions to the standard rules for conformity. For ensuring correctness, types such as *Boolean*, *Condition*, *Node*, *Signature* and *Time* must be implemented only by the system. For performance enhancement, the types *Character*, *Integer*, *Real*, and *String* are also restricted to be implemented only by the system.

7.4 Polymorphism

Inclusion polymorphism, where one type is a subtype of another and can be used in any context in which the supertype is expected, is fundamental to Emerald. Its type system is defined so that the subtyping relation (conformity) is as large as possible while still guaranteeing safety: an operation will never be performed on an object that doesn't implement that operation.

Emerald also supports parametric polymorphism, where types are either explicitly or implicitly passed as arguments to invocations. The cardinal rule regarding types is that it must be possible for the compiler at compile time to determine the value of every expression that is used in a position where a type is required. Such positions include variable, constant, and parameter declarations, and the second argument to view and restrict expressions.

Since types are objects, no special type parameterization form is necessary in Emerald; types are passed to operations in the same way as are other objects. The operation signature definition syntax is supplemented with three clauses that permit:

- the introduction of dependent types whose values depend on the value of type arguments (the where clause)
- the declaration of type variables that may take on the value of the types of other arguments (the for all clause)
- the specification of constraints on the values of type variables (the such that clause).

7.4.1 For all clause

forallClause ::= **forall** *identifier*

The for all clause defines the *identifier* as a type variable without constraint. If the identifier is not defined elsewhere in the operation signature, the for all clause also serves to define it. The type variable will take as its value the type value positionally assigned to it during invocation. The primary purpose of the for all clause is to capture the type of some other argument to an invocation, as in the following example of the polymorphic identity function which returns its argument and in which the type of the result is the same as the type of the argument:

```
function identity[a : t] → [b : t]
    forall t
    b ← a
end identity
```

The for all clause can also be used to introduce a type variable whose value is further constrained by a such that clause.

7.4.2 Where clause

whereClause ::= **where** *identifier* ← *typeExpression*

A where clause is semantically equivalent to a constant declaration. It defines the *identifier* to have the value of the given *typeExpression*. The type expression is evaluated during type checking of invocations after type values have been bound to the argument identifiers defined in the operation signature and any type variables have also been bound to their values. This permits the construction of dependent types whose values depend on the values of type arguments to the invocation, as demonstrated in the following example:

```
function makeVector[arg : argType] → [res : resType]
    forall argType
    where resType ← Vector.of[argType]
    res ← resType.create[10]
    for i : Integer ← 0 while i < 10 by i ← i + 1
        res[i] ← arg
    end for
end makeVector
```

7.4.3 Such that clause

suchthatClause ::= **suchthat** *identifier* ▷ *typeLiteral*

The such that clause allows the possible values that may be taken on by a type variable to be constrained. Any value bound to the given *identifier* (which must be a type variable) must match the *typeExpression*. This clause allows the programmer to require that an argument type have a particular collection of operations, as demonstrated in the following example:

```
function inOrder[a : t, b : t, c : t] → [r : Boolean]
    forall t
    suchthat t ◇> typeobject comparable
        function <=[comparable, comparable] → [Boolean]
    end comparable
    r ← a <= b and b <= c
end inOrder
```

7.5 Typechecking operation definitions

In the presence of parametric polymorphism, some of the identifiers defined in the signature of an operation definition will be type variables, which will take on different values for each invocation of the operation. In type checking the body of the operation, we can assume only that the value of each type variable will match its constraint. Therefore we type check the body of the operation with all type variables bound to their constraints. Once the operation body has been shown to be type correct under this assumption, it

will be type correct for every invocation since the actual type bound to each type variable must match the constraint on that type variable.

7.6 Typechecking invocations

In the presence of parametric polymorphism, the simple rules for typechecking invocations (that the type of each argument expression must conform to the type of its corresponding formal parameter) is insufficient. Therefore, when typechecking an invocation involving type variables, the following steps are performed:

1. Bind each type variable to its corresponding type value from the actual arguments. These may be the arguments themselves, or in the case of type variables introduced by for all clauses, the types of the arguments.
2. Create the dependent type introduced in each where clauses in the operation signature. All of these objects are created simultaneously so that recursive type structures can be successfully created.
3. Check that the value bound to each type variable matches the constraint on that type variable.
4. Check that the type of each argument conforms to the type of its corresponding formal parameter.
5. Determine the result types of the invocation by using the current values of any type variables or dependent type identifiers.

7.7 Conformity revisited

Parametric polymorphism complicates the conformity rules as well, since it introduces type variables and constraints on them. The previous rules for conformity still must be satisfied, but in addition:

0 If either or both of S and T is a bound type variable, then consider the values bound to them rather than the type variable when checking the other rules.

5 If T is an unbound type variable then S must be also, and the constraint on S must match the constraint on T.

7.8 Matches

The matches relation ($\triangleright$) between types is very similar to the conformity relation ($\diamond\!\!>$). In fact, if the types being considered are not recursive then the two relations are equivalent. When checking the conformity of two recursive types (S and T) we must first assume that $S \diamond\!\!> T$. When checking whether S matches T we assume instead that S and T are equivalent types — that $S \diamond\!\!> T$ and that $T \diamond\!\!> S$ — because if the types do match then they will be bound together (since matches only comes into play in the presence of type variables).

```
const Set ← immutable object Set
   export function of[eType : type] → [result : NewSetType]
      suchthat
         eType ▷
            immutable typeobject eType
               function =[eType] → [Boolean]
            end eType
      where
         NewSetType ←
            immutable typeobject NewSetType
               operation empty → [NewSet]
               operation singleton[eType] → [NewSet]
               operation create[sequenceOfeType] → [NewSet]
            end NewSetType
      where
         sequenceOfeType ←
            immutable typeobject sequenceOfeType
               function lowerbound → [Integer]
               function upperbound → [Integer]
               function getElement[Integer] → [eType]
            end sequenceOfeType
      where
         NewSet ←
            immutable typeobject NewSet
               function contains[eType] → [Boolean]
               function +[NewSet] → [NewSet]
               function *[NewSet] → [NewSet]
               function −[NewSet] → [NewSet]
               function cardinality → [Integer]
            end NewSet
      result ←
         object SetCreator
            export operation create[v : sequenceOfeType]→ [result : NewSet]
               result ←
                  object NewSet
                     const repType ← Vector.of[eType]
                     var rep : repType

                     % The implementation of the operations and functions.
               end NewSet
            end create
            export operation empty → [r : NewSet]
               r ← self.create[nil]
            end new
            export operation singleton[e : eType] → [r : NewSet]
               r ← self.create[e]
            end singleton
         end SetCreator
   end of
end Set
```

Figure 1: A Polymorphic Set Object

7.9 Polymorphism Example

To demonstrate the polymorphism present in Emerald, a polymorphic *Set* object is presented in Figure **??**. *Set* has an operation *of* that takes a type as an argument and returns an object that can be used as the abstract type of, as well as a creator of, sets of things conforming to the original argument to the operation *of*. The element type for a set (the type passed to the *of* function) must be immutable and must implement an = operation that returns a **Boolean** object. With this *Set* definition, we can define creators for sets of integers and strings as:

```
const IntSet ← Set.of[Integer]
const StringSet ← Set.of[String]
```

and we can create singleton sets of integers and strings as:

```
const i ← IntSet.singleton[6]
const s ← StringSet.singleton["abc"]
```

8 Objects

Emerald provides a single general purpose object constructor which creates all objects, as well as a number of syntactic shorthands for commonly occurring usage patterns.

8.1 Object Constructors

An object constructor defines the complete representation, operations, and active behaviour of a single object. Objects are created when an object constructor is executed. In other words, object constructors are expressions. The form of a constructor shown below demonstrates its generality, i.e., all Emerald objects may be defined using this feature.

```
objectConstructor ::= [ immutable ] [ welcomable ] [ monitor ] object identifier
                          { declaration }
                          { operation | initially | process | recovery }
                      end identifier
process           ::= process
                          blockBody
                      end process
initially         ::= initially
                          blockBody
                      end initially
recovery          ::= recovery
                          blockBody
                      endrecovery
```

Each object in Emerald owes its existence to either an implicit or explicit execution of an object constructor. The object constructor provides the necessary information about the object's implementation, i.e.,

- Representation declarations for data and processes that are contained in the object.

- A collection of operation bodies containing both the signature as well as the implementation of each operation that the object is capable of executing.

The type of an object constructor expression is determined by including in the type the signature of every exported operation in the constructor. This defines the *best fitting type*, and may be retrieved during execution by the **typeof** expression.

8.1.1 Initialization

When an object constructor is executed, the newly created object is initialized by performing the following steps in order:

1. Initialize any implicitly created constants holding the values of imported identifiers.
2. Initialize all variables and constants declared in the constructor in textual order.
3. Execute the declarations and statements in the initially section, if present.

Any attempt to invoke any operation on the object is deferred until its initialization is complete. The only exception to his rule is that invocations of the object by itself during initialization are allowed.

Once initialization is complete, the process defined by the object constructor is started and then the execution of the object constructor expression terminates. The value of the expression is a reference to the newly created object.

8.1.2 Recovery

When an Emerald node on which objects have previously checkpointed recovers, all checkpointed object have their state restored as of the time of the most recently completed checkpoint, and then the declarations and statements in the recovery section of the object constructor that caused the creation of the object are executed. During recovery, invocations on the object are deferred as during initialization. When the execution of the recovery block is complete, the process defined by the object constructor is started anew.

8.1.3 Object creators

An object whose primary purpose is the creation of other object is termed an object creator. No additional language mechanisms are needed to program object creators; one simply nests one object constructor inside another, but see also Section **??**.

8.2 Objects as Types

Type constructors are the basic method for constructing abstract types in Emerald (cf. Section **??**). Since typing in Emerald is based entirely on the signatures of operations, any object which conforms to the type

```
immutable typeobject type
    function getSignature → [Signature]
end type
```

is a type. Thus objects which serve other useful purposes can also be used as types. Object creators in particular can take advantage of this to allow a single object to serve as both a creator and a type.

8.3 Classes

Emerald does not have a notion of *class*. That is, it is not possible to distinguish a class object from some other object. Or, stated differently again, Emerald does not have a type *class* which class objects conform to but other objects do not. However, Emerald does have a syntactic construct called a class that provides the functionality normally expected of classes.

class	::=	[**immutable**] [**welcomable**] [**monitor**] **class** *identifier* [(*baseClass*)] [*parameterList*] { *classoperation* } { *declaration* } { *operation* | *initially* | *process* | *recovery* } **end** *identifier*
baseClass	::=	*identifier*
classoperation	::=	**class** *operation*

Classes are expanded syntactically into two nested object constructors. The outer object (the class, factory, or creator object) is immutable and declares a single constant, a *Signature* which represents the type of the instances and whose name is derived from the name of the type with the string "type" appended. This signature object contains the signature of each operation that is exported from the inner object constructor (which defines the instances), and is the *best fitting* type of those instances. The class exports operations *getSignature* and *create* in addition to the class operations defined by the programmer. The *getSignature* operation returns the signature constant described above. The *parameterList* specifies the parameter list to the *create* operation; the body of the create operation is a single assignment statement which returns the result of executing the inner object constructor. The inner object constructor is given the name of the class prefixed with the string "a" (or "an" as appropriate). The rest of the components of the class construct become the body of the inner object constructor and thereby define the class's instances.

This is most easily understood through examples. Suppose we write the following declaration:

```
const Complex ← immutable class Complex[r : Real, i : Real]
    class export operation fromReal[a : Real] → [e : Complex]
        e ← self.create[a, 0.0]
    end fromReal
    export function +[other : Complex] → [e : Complex]
        e ← Complex.create[other.getReal + r, other.getImag + i]
    end +
    export function getReal → [e : Real]
        e ← r
    end getReal
    export function getImag → [e : Real]
        e ← i
    end getImag
end Complex
```

This is rearranged into the following:

```
const Complex ← immutable object Complex
    const ComplexType ← immutable typeobject ComplexType
        function +[ComplexType] → [ComplexType]
        function getReal → [ComplexType]
        function getImag → [ComplexType]
    end ComplexType
    export function getSignature → [r : Signature]
        r ← ComplexType
    end getSignature
    export operation fromReal[a : Real] → [e : Complex]
        e ← self.create[a, 0.0]
    end fromReal
    export operation create[r : Real, i : Real] → [e : Complex]
        e ← immutable object aComplex
            export function +[other : Complex] → [e : Complex]
                e ← Complex.create[other.getReal + r, other.getImag + i]
            end +
            export function getReal → [e : Real]
                e ← r
            end getReal
            export function getImag → [e : Real]
                e ← i
            end getImag
        end aComplex
    end create
end Complex
```

Inheritance

Emerald supports *single inheritance*. That is, every class may have at most one superclass from which it inherits. There are three kinds of components that may be inherited from the parent class:

- class operations
- instance declarations (constants and variables)
- instance operations including any initially, recovery, and process
- parameters to the class.

The parameters to the subclass are concatenated to the end of the list of parameters to the superclass in order to form the final parameter list for the subclass.

Inheritance of the other three kinds of components is performed by considering in turn each component of the superclass, and searching for an identically named component of the same kind in the subclass. If the subclass contains the component then the superclass component is ignored, otherwise the superclass component is added to the subclass.

There is no support for changing the visibility of a component (exporting in the subclass an operation that is private in the superclass or making private in the subclass an operation that is exported in the superclass), nor is there support for deleting a component. Because subclass components completely replace those from the superclass, changing the visibility of a component may be accomplished by copying the component from the superclass manually and changing the visibility of the copy. A data component may be effectively deleted

by declaring an identically named constant in the subclass whose value is sufficiently simple that it need not be stored (an excellent choice of value is 0).

Because any component of the superclass may be redefined in the subclass, there is no guarantee that either the type of the subclass will conform to the type of the superclass, or that the type of instances of the subclass will conform to the type of instances of the superclass. The objects resulting from inheritance do not retain at run time any record of their inheritance relationships. That is, there is no operation that can be performed on a class object to retrieve its superclass. This is a side effect of the fact that there is no type *class*. Such operations can easily be implemented, if desired, as in the following example:

```
const parent ← class parent
end parent

const child ← class child (parent)
   class export function getSuperClass → [r : Any]
      r ← parent
   end getSuperClass
end child
```

8.4 Enumerations

enum ::= **enumeration** *identifier*
enumIdentifier { **,** *enumIdentifier* }
end *identifier*

An enumeration is a class that represents an ordered collection of identifiers. The operations on the class consist of:

- a function getSignature, which returns a *Signature* describing the type of the enumeration instances
- for each enumeration identifier a, a creation operation named a that returns an object representing that element of the enumeration
- *first* and *last* that return the first and last elements of the enumeration, respectively
- an operation named *create* that takes an integer argument n and returns the nth element of the enumeration, with the numbering starting at 0.

Each instance of the class implements the following operations:

- the comparison functions $<$, $<=$, $=$, $!=$, $>=$, $>$
- functions *succ* which returns the successor object (or fails if invoked on the last element of the enumeration) and *pred* which returns the predecessor object (or fails if invoked on the first element of the enumeration)
- a function *ord* which returns the position of the element in the enumeration ordering, starting at 0
- a function *asString* which returns the name of the element as a *String*.

All instance of enumeration classes are immutable. To be concrete, consider the declaration:

```
const colors ← enumeration colors red, blue, green end colors
```

The class object *colors* will have type:

```
immutable typeobject ColorCreatorType
    function getSignature → [Signature]
    operation create[Integer] → [ColorType]
    operation first → [ColorType]
    operation last → [ColorType]
    operation red → [ColorType]
    operation green → [ColorType]
    operation blue → [ColorType]
end ColorCreatorType
```

and each element of the enumeration will have type:

```
immutable typeobject ColorType
    function <[ColorType] → [Boolean]
    function <=[ColorType] → [Boolean]
    function =[ColorType] → [Boolean]
    function !=[ColorType] → [Boolean]
    function >=[ColorType] → [Boolean]
    function >[ColorType] → [Boolean]
    function succ → [ColorType]
    function pred → [ColorType]
    function ord → [Integer]
    function asString → [String]
end ColorType
```

8.5 Fields

field ::= [**attached**] **const field** *identifier* **:** *type initializer*
| [**attached**] **field** *identifier* **:** *type* [*initializer*]

It is often convenient to declare an externally accessible data element of an object. A field declaration does exactly this. Field declarations can only occur within the declaration part of an object constructor. Constant field declarations expand to a constant declaration and an operation to get the value of the constant. Variable fields expand to a variable declaration and operations to both get and set the value of the variable. The expansion of the constant field:

```
attached const field f : t ← init
```

is

```
attached const f : t ← init
export function getF → [x : t]
    x ← f
end getF
```

And the expansion of the variable field:

```
attached field f : t ← init
```

is

```
attached var f : t ← init
export operation setF[x : t]
    f ← x
end setF
export function getF → [x : t]
    x ← f
end getF
```

8.6 Records

record ::= [**immutable**] **record** *identifier*
recordfield { *recordfield* }
end *identifier*
recordfield ::= [**attached**] [**var**] *fieldIdentifier : type*

A record is a class that contains a field for each element of the record, and a *create* function that takes as its parameters initial values for each field. If the record is defined as immutable, then the class will be immutable as well, and each field will be a constant field. For example, the declaration of the mutable record:

record *aRecord*
a : *Integer*
c : *String*
end *aRecord*

expands to a class:

class *aRecord*[*xa* : *Integer, xc* : *String*]
field a : *Integer* ← *xa*
field c : *String* ← *xc*
end *aRecord*

Given the above declaration, the following code declares and initializes a record variable.

var *a* : *aRecord*

a ← *aRecord.create*[*34, "A string"*]

8.7 Predefined objects

Emerald implements a number of pre-defined objects; these objects are outlined in Table **??** and specified in greater detail in Appendix **??**.

9 Location and Reliability

Emerald was developed primarily to facilitate the construction of distributed application programs. To be resilient to machine crashes, these programs should be capable of detecting and recovering from such crashes. They should also be able to control the location of component objects so that the available nodes in the system are optimally exploited. This section discusses the Emerald location-related constructs.

There are two Emerald concepts that concern location. These correspond to two desires that motivate application programmers to deal with location. As stated previously, invocation in Emerald is location independent. This means that the location of an object need not be determined in order to invoke it. There are however two considerations that we expect to motivate application programmers to concern themselves with location: performance and reliability/availability.

Performance

Since remote invocation will necessarily be at least an order of magnitude more expensive than local invocation, the placement of Emerald objects may seriously affect their performance. In order to provide the programmer with control over the placement of objects the move statement (see Section **??**) is provided. In addition, the call-by-move implementation strategy for arguments to invocations (see Section **??**) allows further optimizations.

Predefined Type	*Type Description*
Any	*Has no operations.*
Array	*A polymorphic, flexible array.*
AOpVector	*A sequence of operation signatures in a Signature.*
AOpVectorE	*A single operation signature in a Signature.*
AParamList	*A list of parameters in an operation Signature.*
BitChunk	*A container of bits supporting bit-level operations.*
Boolean	*Logical values with literals* **true** *and* **false**.
Character	*Individual characters with operations such as <, >, =, ord, etc.*
ConcreteType	*A container for the executable code of an object.*
Condition	*Condition variables satisfying Hoare monitor semantics.*
COpVector	*A sequence of operation definitions in a ConcreteType.*
COpVectorE	*A single operation definition in a ConcreteType.*
Directory	*An object defining the type of primitive name server directories.*
Handler	*An object defining the type of objects capable of receiving Node state change updates from the run time system.*
InterpreterState	*An internal object capturing the state of the execution of a process.*
ImmutableVector	*Read-only vector.*
ImmutableVectorOfAny	*Read-only vector of Any.*
ImmutableVectorOfInt	*Read-only vector of Integers.*
InStream	*Input streams.*
Integer	*Signed integers.*
Node	*Objects representing machines.*
NodeList	*Immutable vectors of node descriptions.*
NodeListElement	*Immutable node descriptions.*
None	*The type of* **nil**.
OutStream	*Output streams.*
Real	*Approximations of real numbers.*
RISA	*Readable Indexed Sequence of Any.*
RISC	*Readable Indexed Sequence of Character.*
Signature	*Primitive abstract type*
String	*Character strings.*
Time	*Times and dates*
Type	*The type of all types.*
Vector	*Fixed sized polymorphic vectors.*
VectorOfChar	*Vector.of[Character].*
VectorOfInt	*Vector.of[Integer].*

Table 2: Built-in Types

Reliability and Availability

Since an object may be moved at arbitrary times by any other object with a reference to it, a more permanent binding between objects and locations is often required. In particular, in order to implement an available replicated service, it is necessary to place the replicas on differing machines and not allow them to move. This allows the programmer to guarantee that a single machine failure will not cause more than one of his replicas to become unavailable.

In order to provide for this requirement, the fix and unfix statements (see Sections **??** and **??**) may be used. An object, once fixed at a particular location, may not be moved from there. Any attempt to do so will fail (see subsection **??**).

9.1 Unavailable objects

Due to machine crashes or communication network failures, objects may be temporarily or permanently unavailable. Emerald provides unavailable handlers to allow programmers to detect such situations and attempt recovery.

unavailableHandler ::= **unavailable** [[*identifier*]]
blockBody
end unavailable

An object is regarded as being unavailable when it cannot be located at any available node following suitable system action [**?**]. When an attempt is made to invoke an object which is unavailable, the appropriate hander is located in a manner similar to that for failures (see Section **??**), the unavailable object is bound to the identifier declared in that unavailable handler (if present), and the body of the handler is executed. The type of the identifier in the handler definition is *Any*.

While invoking an object which is unavailable results in an unavailable exception, the typeof, codeof, and nameof expressions will correctly identify even unavailable objects.

9.2 Failures

Failures can result from a number of causes; these include attempting to invoke a **nil** reference, assertion failures, divide-by-zero and subscript-range errors.

failureHandler ::= **failure**
blockBody
end failure

After a failure is detected, the following action is taken.

1. The appropriate failure handler to execute is found. This handler is the handler attached to the smallest block containing the statement. Note that failures are considered a "superclass" of unavailables, and so when an unavailable exception has been raised, each block that may have a handler is first searched for an unavailable handler, and then a failure handler, and only if no handler can be found is the next larger enclosing block searched. Similarly when propagating an unavailable exception up the call stack, each block is first searched for an unavailable handler and then for a failure handler.

2. If the block body of a monitored operation, the initially section, or the recovery section of an object fails, then the object is said to have failed. Any subsequent invocation attempted on the object will fail, and any invocations that have started but have not yet completed also fail.

3. An unhandled failure in the block body of an operation is propagated by causing the corresponding invocation to fail.

4. An unhandled failure in an initially section implies that the object creation has failed; this is propagated by causing the statement containing the object constructor expression to fail.

5. An unhandled failure in the block body of a recovery section cannot be propagated because its execution did not result from an invocation.

6. An unhandled failure in a process block body cannot be propagated. The process is terminated, but the object itself does not fail.

7. When an object fails, no attempt is made to immediately track down and fail all processes (including the one contained in the object) that have threads of control that have passed through the object. When these threads of control return to the body of any operation inside the object, they will then fail.

x ./reserved.tex, 1374 bytes, 3 tape blocks

A Reserved Words in Emerald

The reserved words in Emerald are listed out below under the four categories mentioned in Section **??**. It should be borne in mind that Emerald is an active research language and is constantly being modified so there may be additional reserved words not mentioned below.

A.1 Keywords

and	**as**	**assert**	**at**	**attached**
awaiting	**begin**	**checkpoint**	**const**	**else**
elseif	**end**	**enumeration**	**exit**	**export**
failure	**fix**	**from**	**function**	**if**
immutable	**import**	**initially**	**isfixed**	**locate**
loop	**monitor**	**move**	**object**	**on**
op	**operation**	**or**	**primitive**	**process**
record	**recovery**	**refix**	**return**	**returnandfail**
signal	**then**	**to**	**type**	**unavailable**
unfix	**union**	**var**	**view**	**visit**
wait	**welcomable**	**welcome**	**when**	**where**

A.2 Literals

false	**nil**	**self**	**true**

B Built-in Objects

This appendix defines the built-in objects. These descriptions will mention that an object is a type rather than explicitly refering to the object's immutability and its getSignature operation.

B.1 Any

Any is the type that requires no operations; every Emerald object has type Any.

B.2 Array

Arrays implement expandable indexable storage. The of operation on Array takes a Type, and returns an array creator. As arrays can expand and shrink, common data types such as Stacks and Queues can be implemented using Arrays: Stacks use addUpper and removeUpper, while Queues use addUpper and removeLower.

The object Array is immutable and has the following interface:

function *of*[*T* : *type*] → [*aNewArrayCreatorType*] **forall** *T*

The object resulting from *Array.of*[*T*] (with type aNewArrayCreatorType) is a creator as well as a type. It is immutable and has the following interface:

operation *empty* → [*aNewArrayType*]
Return a new empty *Array*.
operation *literal*[*Sequence.of*[*T*]] → [*aNewArrayType*]
Return a new *Array* initialized with all the elements from the given sequence.
operation *create*[*size* : *Integer*] → [*aNewArrayType*]
Return a new *Array* with size elements all initialized to **nil**.

Objects with type *Array.of*[*T*] have the following interface, named aNewArrayType:

function *getElement*[*index* : *Integer*] → [*T*]
Get the element indexed by *index*, failing if *index* is out of range.
operation *setElement*[*index* :*Integer, value* :*T*]
Set the element indexed by *index* to *value*, failing if *index* is out of range.
function *upperbound* → [*Integer*]
Return the highest valid index.
function *lowerbound* → [*Integer*]
Return the lowest valid index.
function *getElement*[*lb* : *Integer, length* : *Integer*] → [*aNewArrayType*]
Return a new *Array*, a, with lower bound *lb*, and length *length*, such that for lb ≤ i ≤ lb + length - 1: self[i] == a[i]. Fail if lb or lb + length - 1 is out of range.
function *getSlice*[*lb* :*Integer, length* :*Integer*] → [*aNewArrayType*]
Same as getElement.
operation *setElement*[*lb* : *Integer, length* : *Integer, a* : *RIS*]
Set the elements indexed starting at *lb* for *length* elements, so that for each such i in that range: self[i] == a[i]. Fail if lb or lb + length - 1 is out of range.

operation *setSlice*[*lb* : *Integer*, *length* : *Integer*, *a* : *Sequence.of*[*T*]]
Same as setElement.
operation *slideTo*[*newlb* : *Integer*]
Change the valid indices for self so that the new lowerbound is *newlb*.
operation *addUpper*[*value* : *T*]
Extend the set of valid indices, changing ub to ub + 1, and setting the element indexed by the new ub to be *value*.
operation *removeUpper* → [*T*]
Return the element indexed by ub, after contracting the set of valid indices to lb ≤ i ≤ ub − 1.
operation *addLower*[*value* : *T*]
Extend the set of valid indices, changing lb to lb − 1, and setting the element indexed by the new lb to be *value*.
operation *removeLower* → [*T*]
Return the element indexed by lb, after contracting the set of valid indices to lb + 1 ≤ i ≤ ub.
function *empty* → [*Boolean*]
Return **true** if lb == ub + 1.
operation *catenate*[*a* : *RIS*] → [*r* : *aNewArrayType*]
Create a new array like self, and then add (using addUpper) each element in *a* to that new array.

B.3 BitChunk

BitChunks allow the manipulation of arbitrarily sized sequences of bits. There are operations to set or retrieve collections of bits at arbitrary bit positions with lengths up to 32 bits. BitChunk is a type with the following interface:

export operation *create*[*n* : *Integer*] → [*Bitchunk*]
Create a bitChunk large enough to hold *n* bytes of information.

An object whose type is BitChunk has the following interface:

function *getSigned*[*off*: *Integer*, *len*: *Integer*] → [*Integer*]
Return the bits at offset *off* for length *len* as a signed *Integer* (treat the highest order bit as a sign bit).
function *getUnsigned*[*off* : *Integer*, *len* : *Integer*] → [*Integer*]
Return the bits at offset *off* for length *len* as an unsigned *Integer*
function *getElement*[*Integer*, *Integer*] → [*Integer*]
Equivalent to getUnsigned.
operation *setSigned*[*off*:*Integer*, *len*:*Integer*, *val*:*Integer*]
Set the bits at offset *off* for length *len* to the low order bits of *val*.
operation *setUnsigned*[*Integer*, *Integer*, *Integer*]
Equivalent to setSigned.
operation *setElement*[*Integer*, *Integer*, *Integer*]
Equivalent to setSigned.
operation *ntoh*[*off* : *Integer*, *len* : *Integer*]
Convert the bits at offset *off* with length *len* from network to host byte order. Len must be either 16 or 32.

B.4 Boolean

In addition to the operations on Booleans listed here, Booleans are involved in the evaluation of the conditional and (**and**) and conditional or (**or**) expressions. The immutable object Boolean has the following interface:

function *create*[*ord* : *Integer*] → [*Boolean*]
    If ord is 0 then return **false**, otherwise return **true**.

Objects whose type is *Boolean* are immutable with the following interface: Comparisons are based on the ordinal values as given by ord

function > [*Boolean*] → [*Boolean*]
function >= [*Boolean*] → [*Boolean*]
function < [*Boolean*] → [*Boolean*]
function <= [*Boolean*] → [*Boolean*]
function = [*Boolean*] → [*Boolean*]
function != [*Boolean*] → [*Boolean*]
    Comparison functions.
function & [*Boolean*] → [*Boolean*]
    Logical and.
function | [*Boolean*] → [*Boolean*]
    Logical or.
function ! → [*Boolean*]
    Logical negation.
function *ord* → [*Integer*]
    Return 0 when invoked on **false**, 1 when invoked on **true**.
function *asString* → [*String*]
    Return either "true" or "false".
function *hash* → [*Integer*]
    return self.ord

B.5 Character

The immutable object Character has the following interface:

function *Literal*[*o* : *Integer*] → [*Character*]
    Return the *Character* whose ordinal is given by o.

Objects whose type is Character are immutable and have the following interface:

function > [*Character*] → [*Boolean*]
function >= [*Character*] → [*Boolean*]
function < [*Character*] → [*Boolean*]
function <= [*Character*] → [*Boolean*]
function = [*Character*] → [*Boolean*]
function != [*Character*] → [*Boolean*]
    Comparison functions.

function *asString* → [*String*]
Return a single character string "c" when invoked on a character 'c'.
function *ord* → [*Integer*]
Return the character's ordinal number.
function *hash* → [*Integer*]
Return self.ord.
function *isalpha* → [*r* : *Boolean*]
Return **true** if the character appears in the alphabet.
function *isupper* → [*r* : *Boolean*]
Return **true** if the character is upper case.
function *islower* → [*r* : *Boolean*]
Return **true** if the character is lower case.
function *isdigit* → [*r* : *Boolean*]
Return **true** if the character is a decimal digit.
function *isxdigit* → [*r* : *Boolean*]
Return **true** if the character is a hexadecimal digit.
function *isalnum* → [*r* : *Boolean*]
Return **true** if the character is alphanumeric.
function *isspace* → [*r* : *Boolean*]
Return **true** if the character is white space (blank, tab, newline, etc).
function *ispunct* → [*r* : *Boolean*]
Return **true** if the character is a punctuation mark.
function *isprint* → [*r* : *Boolean*]
Return **true** if the character is printable.
function *isgraph* → [*r* : *Boolean*]
Return **true** if the character is visible when printed.
function *iscntrl* → [*r* : *Boolean*]
Return **true** if the character is a control character.
function *toupper* → [*r* : *Character*]
If the character is a lower case alphabetic, return the upper case letter corresponding to it, otherwise return the character itself.
function *tolower* → [*r* : *Character*]
If the character is an upper case alphabetic, return the lower case letter corresponding to it, otherwise return the character itself.

B.6 ConcreteType

A ConcreteType captures the implementation of some other object. It contains all the information required to create and operate on the collection of objects created from a single object constructor. ConcreteTypes can only be usefully created by the compiler. The object ConcreteType is a type and has the following interface:

operation *create*[
instanceSize : *Integer*, instanceTagMask : *Integer*, ops : COpVector, name : *String*, filename : *String*, template : *String*] → [ConcreteType]

Objects whose type is ConcreteType are immutable and have the following interface. All of these operations return information describing all of the objects that could ever be created using this object constructor.

function *getInstanceSize* → [*Integer*]
Return the size in bytes of the data area of the object.
function *getInstanceTagMask* → [*Integer*]
Return a collection of flag bits that include whether the objects are immutable and how they are represented by the interpreter.
function *getOps* → [*COpVector*]
Return a list of the operations defined for the objects.
function *getName* → [*String*]
Return the name of the object constructor.
function *getFileName* → [*String*]
Return the file name of the Emerald source file that contained the typeobject from which I was created.
function *getTemplate* → [*String*]
Return the template that describes the data area of the objects. The format of the template string is private.
function *getLiterals* → [*ImmutableVectorOfInt*]
Return a list of literals that are referenced by the code for operations on the object. The format of this structure is private.

COpVector is ImmutableVector.of[COpVectorE].

A COpVectorE describes an operation. COpVectorE is a type with the following interface:

operation *create*[
id : *Integer*, nArgs : *Integer*, nRess : *Integer*, name : *String*, template : *String*, code : *String*] → [n : COpVectorEType]

Objects whose type is COpVectorE are immutable and have the following interface:

function *getID* → [*Integer*]
Return the internal id of the operation.
function *getNArgs* → [*Integer*]
Return the number of arguments to the operation.
function *getNRess* → [*Integer*]
Return the number of results returned by the operation.
function *getName* → [*String*]
Return the name of the operation.
function *getTemplate* → [*String*]
Return the template that describes the activation record of the operation. The format of the template string is private.
function *getCode* → [*String*]
Return the code for the operation, as a string.

The internal representation of the state of the interpreter. InterpreterState is a type with the following interface:

operation *create*[
pc : *Integer*, sp : *Integer*, fp : *Integer*, sb : *Integer*, o : *Any*, e : *Any*] → [InterpreterState]

Objects whose type is InterpreterState have the following interface:

function *getPC* → [*Integer*]
operation *setPC*[*Integer*]
function *getSP* → [*Integer*]
operation *setSP*[*Integer*]
function *getFP* → [*Integer*]
operation *setFP*[*Integer*]
function *getO* → [*Any*]
operation *setO*[*Any*]
function *getSB* → [*Integer*]
operation *setSB*[*Integer*]
function *getE* → [*Any*]
operation *setE*[*Any*]

PC is the program counter, SP is the stack pointer, FP is the frame pointer, and SB is the base of the stack. O is the current object, and E is the environment of the current process. Each Emerald process has a per-process environment which can be used at the programmer's discretion.

B.7 Condition

A condition object may only be used within the monitor within which it was created. The object Condition is immutable, and has the following interface:

operation *create* → [*Condition*]

Objects whose type is Condition have no operations; wait, signal, and awaiting are language primitives.

B.8 InStream

InStream objects provide the ability to read files. The InStream object is immutable and has the following interface:

operation *fromUnix*[*fn* : *String*, *mode* : *String*] → [*InStream*]
Return a new input stream attached to the given operating system file. Mode is as in fopen in C, and must begin with 'r'. The operation fails if the file does not exist or cannot be opened.
operation *create*[*file* : *Integer*] → [*InStream*]
Return a new input stream attached to the given file descriptor.

All of the input operations on InStream objects fail if the operation cannot be performed. In particular, they fail upon reaching end-of-file or if the stream has been closed. Objects whose type is InStream have the following interface:

operation *getChar* → [*Character*]
Return the next character from the input source. Fail if the stream is closed or eos has been reached.
operation *unGetChar*[*c* : *Character*]
Push the character on the the front of the input stream. There is no requirement that the character be one previously read from the stream. Only the amount of available memory limits the amount of data that can be pushed back on a stream.

operation *getString* → [*String*]
Return one line of input, including the terminating newline character, if any.
function *eos* → [*Boolean*]
Return **true** if the end of the input stream has been reached. Note that this predicate will block for terminal input on a terminal stream if necessary in order to determine whether the end of the stream has been reached.
operation *close*
Close the stream.
function *isAtty* → [*Boolean*]
Return **true** if the underlying Unix file is a tty.
operation *fillVector*[*VectorOfChar*] → [*Integer*]
Reads up to one line from the input, placing the characters in the provided vector. Returns the number of characters read. The vector will contain the newline character that caused reading to terminate unless no newline character is read before the vector is full.
operation *rawRead*[*VectorOfChar*] → [*Integer*]
Reads from the input placing the characters in the provided vector. Returns the number of characters read.

B.9 Integer

Conversion between integers and reals is accomplished by the asReal operation on Integers, and the asInteger operation on Reals. The immutable object Integer has the following interface:

function *literal*[*rep* : *String*] → [*Integer*]
Return the integer parsed from the front of the string *rep*. The C function strtol is used to parse the integer.

Objects whose type is Integer are immutable, and have the following interface. For those operations that manipulate bits, bit numbering starts at 0, which represents the high order bit of the integer.

function + [*Integer*] → [*Integer*]
function − [*Integer*] → [*Integer*]
function ∗ [*Integer*] → [*Integer*]
function / [*Integer*] → [*Integer*]
function # [*Integer*] → [*Integer*]
Arithmetic functions. # represents modulus.
function > [*Integer*] → [*Boolean*]
function >= [*Integer*] → [*Boolean*]
function < [*Integer*] → [*Boolean*]
function <= [*Integer*] → [*Boolean*]
function = [*Integer*] → [*Boolean*]
function != [*Integer*] → [*Boolean*]
Comparison functions.
function ~ → [*Integer*]
Negation.
function − → [*Integer*]
Negation, identical to ~.

function *asString* → [*String*]
Return a *String* representing the value of the *Integer* with no leading zeros in decimal.
function *hash* → [*Integer*]
Return self.
function *abs* → [*Integer*]
Return the absolute value of self.
function *asReal* → [*Real*]
Return a *Real* value representing the same value as self.
function & [*other* : *Integer*] → [*Integer*]
Return the bitwise and of self and *other*.
function | [*other* : *Integer*] → [*Integer*]
Return the bitwise or of self and *other*.
function *setBit*[*o* : *Integer*, *v* : *Boolean*] → [*r* : *Integer*]
Return an *Integer* like self except in the *o*'th bit position, where it has the bit *v*.
function *getBit*[*o* : *Integer*] → [*r* : *Boolean*]
Return the bit in *o*'th bit position.
function *setBits*[*o* : *Integer*, *l* : *Integer*, *v* : *Integer*] → [*r* : *Integer*]
Return an *Integer* like self except in the *o* through $o + l - 1$'th positions, where the bits represent the low order bits of the *Integer* v.
function *getBits*[*o* : *Integer*, *l* : *Integer*] → [*r* : *Integer*]
Return the bits in the *o* through $o + l - 1$'th bit positions, without sign extension.

B.10 Node, NodeList, NodeListElement, Directory, Handler

The nodeEventHandler entries allow appropriate operations to be invoked when the node detects changes in the network topology. The operations that query network topology use the auxiliary types NodeList and NodeListElement which are described below. The object Node is a type with the following interface:

operation *getStdin* → [*InStream*]
Return an InStream representing the current *Node*'s standard input.
operation *getStdout* → [*OutStream*]
Return an OutStream representing the current *Node*'s standard output.

Objects with type Node are mutable, are fixed at their initial locations, and have the following interface:

operation *getActiveNodes* → [*NodeList*]
Return a list containing information about all nodes in the Emerald environment that are known to be functioning.
operation *getAllNodes* → [*NodeList*]
Return a list containing information about all nodes in the Emerald environment, whether they are functioning or not.
operation *getNodeInformation* → [*NodeListElement*]
Return information about the node. This is not currently implemented.
operation *getTimeOfDay* → [*Time*]
Return an object of type *Time* representing the current wall clock time.
operation *delay*[*howlong* : *Time*]
Put the current process to sleep until the amount of time specified by *howlong* has passed.

operation *waitUntil*[*untilwhen* : *Time*]
Put the current process to sleep until the absolute time specified by *untilwhen* has arrived.
operation *getLoadAverage* → [*Real*]
Return a *Real* number representing the load on the machine on which the target node is executing. This is not currently implemented.
operation *setNodeEventHandler*[*h* : *HandlerType*]
Register *h* as an additional handler to receive notification when nodes come up or do down.
operation *removeNodeEventHandler*[*h* : *HandlerType*]
Unregister h as a node event handler.
operation *setDiscoveredNodeEventHandler*[*h* : *HandlerType*]
Register *h* as an additional handler to receive notification when nearby nodes are discovered.
operation *getStdin* → [*InStream*]
Return an InStream representing the target *Node*'s standard input.
operation *getStdout* → [*OutStream*]
Return an OutStream representing the target *Node*'s standard output.
operation *mergeWith* [*ip* : *String*, *port* : *Integer*]
Merge the *Node*'s node graph with the node graph of the *Node* specified by *ip* and *port*.
function *getLNN* → [*Integer*]
Return the current node's Logical Node Number. This value is not interesting.
function *getName* → [*String*]
Return a string containing the name of the current host. If the current host name cannot be determined (as on DOS) return a string containing as much information as is available.
function *getRootDirectory* → [*Directory*]
Return the root directory of the Emerald universe. Each cooperationg collection of Emerald nodes has a single root directory which is the root of the name service.

A NodeListElement is an immutable record with 4 fields that provide information about a node:

function *getTheNode* → [*Node*]
function *getUp* → [*Boolean*]
function *getIncarnationTime* → [*Time*]
function *getLNN* → [*Integer*]

NodeList is *ImmutableVector.of*[*NodeListElement*].

Directory is the type of name service directories. Directory is a type with interface:

operation *create* → [*r* : *Directory*]
Return a new directory.

Objects whose type is Directory have the following interface:

operation *insert*[*name* : *String*, *value* : *Any*]
Insert the object *value* in the directory under the name *name*.
function *lookup*[*name* : *String*] → [*Any*]
Return the object in the directory stored under the name *name*. If there is no object stored under that name, return **nil**.
operation *delete*[*String*]
Delete the object in the directory stored under the name *name*. It is not an error to delete a nonexistent entry.

function *list* → [*ImmutableVectorOfString*]
Return a list of all of the names defined in the directory.

Handler represents the type of objects that can register to be informed when Nodes come up and go down. Such objects must conform to the type Handler, which is:

typeobject *HandlerType*
operation *nodeUp*[*Node, Time*]
operation *nodeDown*[*Node, Time*]
end *HandlerType*

B.11 None

None is the type that supports all operations, and is therefore implemented only by the **nil** object. It is defined to complete the lattice structure of Emerald types; None represents the top element of the type lattice.

B.12 OutStream

OutStream objects provide the ability to write files. The OutStream object is a type with the following interface:

operation *toUnix*[*fn* : *String, mode* : *String*] → [*OutStream*]
Return a new output stream attached to the given operating system file. Mode is as in fopen in C, and must begin with 'w' or 'a'. The operation fails if the file cannot be opened in the given mode.
operation *create*[*file* : *Integer*] → [*OutStream*]
Return a new output stream attached to the given file descriptor.

Objects with type OutStream have the following interface. All of the output operations on OutStream objects fail if the requested operation cannot be performed.

operation *putChar*[*c* : *Character*]
Append the character *c* on the stream.
operation *putInt*[*n* : *Integer, width* : *Integer*]
Append the decimal string representation of the *Integer n* on the stream, right justified in a field *width* wide. If the value n cannot be accurately represented in width characters, then the width specification will be treated as infinity.
operation *writeInt*[*n* : *Integer, size* : *Integer*]
Write the binary representation of the *Integer n* in *size* bytes, in network byte order. Size must be 1, 2, or 4.
operation *putReal*[*x* : *Real*]
Append the decimal string representation of the *Real x* on the stream. A representation of the value of x will be chosen so as to not lose accuracy.
operation *putString*[*s* : *String*]
Append the *String s* on the stream.
operation *flush*
Flush any buffered output.
operation *close*
Close the stream, flushing any buffered output and preventing any further output.

B.13 Real

The Real type is implemented as a 32-bit floating-point number. The object Real is a type with the following interface:

function *literal*[*s* : *String*] → [*Real*]
Return a Real number parsed from the string *s*. The C language function atof is used to parse the string.

Objects with type Real are immutable and have the following interface:

function + [*Real*] → [*Real*]
function − [*Real*] → [*Real*]
function ∗ [*Real*] → [*Real*]
function / [*Real*] → [*Real*]
Arithmetic functions.
function ^ [*Real*] → [*Real*]
Exponentiation. *a*^*b* returns *a* raised to the exponent *b*.
function > [*Real*] → [*Boolean*]
function >= [*Real*] → [*Boolean*]
function < [*Real*] → [*Boolean*]
function <= [*Real*] → [*Boolean*]
function = [*Real*] → [*Boolean*]
function != [*Real*] → [*Boolean*]
Comparison functions.
function ~ → [*Real*]
Return the negation of self.
function − → [*Real*]
Return the negation of self, same as ~.
function *asString* → [*String*]
Return a string representing my value. The C language function sprintf's %g specification is used.
function *asInteger* → [*Integer*]
Return an integer as close as possible to, but lower than my value.

B.14 Sequence, SequenceOfAny, SequenceOfString

Sequence represents indexable sequences. The object Sequence is immutable and has interface:

function *of*[*T* : *type*] → [*aNewSequenceType* : *Type*] **forall** *T*

The object resulting from *Sequence.of*[*T*] is a *Signature* defined as follows:

typeobject *aNewSequence*
function *lowerbound* → [*Integer*]
function *upperbound* → [*Integer*]
function *getElement*[*Integer*] → [*T*]
end *aNewSequence*

SequenceOfAny is Sequence of Any. Vector.of[Any], ImmutableVector.of[Any], and Array.of[Any] all conform to SequenceOfAny.

SequenceOfCharacter is Sequence of Character. Vector.of[Character], ImmutableVector.of[Character], and Array.of[Character] all conform to SequenceOfCharacter.

B.15 Signature, AOpVector, AOpVectorE, AParamList

Signature is the type of the object constructed by the compiler as the result of a typeobject constructor. Signature has no interesting operations.

Objects whose type is Signature are types, and have the following interface, all of which operations return information about the typeobject constructor that was used in the creation of the signature.

function *getIsImmutable* → [*Boolean*]
    Return **true** if the typeobject is immutable.
function *getIsTypeVariable* → [*Boolean*]
    Return **true** if the typeobject is a type variable which resulted from a for all or such that clause.
function *getOps* → [*AOpVector*]
    Return a list of the operations in the typeobject.
function *getName* → [*String*]
    Return the name of the typeobject.
function *getFileName* → [*String*]
    Return the file name of the Emerald source file that contained the typeobject from which I was created.

AParamList is ImmutableVector.of[Signature].

AOpVector is ImmutableVector.of[AOpVectorE].

An AOpVectorE describes an operation in a Signature. AOpVectorE is a type with the following interface:

operation *create*[
    id : *Integer*, NArgs : *Integer*, NRess : *Integer*, isFunction : *Boolean*, name : *String*, arguments : AParamList, results : AParamList] → [AOpVectorE]

Objects whose type is AOpVectorE are immutable and have the following interface:

function *getID* → [*Integer*]
    Return the internal id of the operation.
function *getNArgs* → [*Integer*]
    Return the number of arguments to the operation.
function *getNRess* → [*Integer*]
    Return the number of results returned by the operation.
function *getIsFunction* → [*Boolean*]
    Return **true** if the operation is a function.
function *getName* → [*String*]
    Return the name of the operation.
function *getArguments* → [*AParamList*]
    Return a list of the argument types.
function *getResults* → [*AParamList*]
    Return a list of the result types.

B.16 String

The object String is a type and has the following interface:

operation *Literal*[*rep* : *Sequence.of*[*Character*], *offset* : *Integer*, *len* : *Integer*] → [*String*]
Return a *String* with the characters from the given sequence of characters.
operation *FLiteral*[*rep* : *VectorOfChar*, *offset* : *Integer*, *len* : *Integer*] → [*String*]
Same as literal, but with a more restrictive argument type (it must be a *Vector*), which makes the implementation more efficient.

Objects whose type is String are immutable and have the following interface:

function > [*String*] → [*Boolean*]
function >= [*String*] → [*Boolean*]
function < [*String*] → [*Boolean*]
function <= [*String*] → [*Boolean*]
function = [*String*] → [*Boolean*]
function != [*String*] → [*Boolean*]
Comparison functions.
function *getElement*[*index* : *Integer*] → [*Character*]
Return the character at position *index*, numbering from 0.
function *getSlice*[*lb* : *Integer*, *length* : *Integer*] → [*String*]
Return a substring of the string, starting at position *lb*, for *length* characters.
function *getElement*[*lb* : *Integer*, *length* : *Integer*] → [*String*]
Same as getSlice, but can use the convenient subscript syntax.
function *length* → [*Integer*]
Return the length of the string.
function || [*more* : *String*] → [*String*]
Return the result of concatenating the string *more* at the end of self.
function *asString* → [*String*]
Return self.
function *lowerbound* → [*Integer*]
Return 0.
function *upperbound* → [*Integer*]
Return self.length − 1
function *hash* → [*Integer*]
Return a characteristic *Integer*. The hash function is chosen to make the probability of two distinct strings having the same hash value quite small.
function *index*[*ch* : *Character*] → [*r* : *Integer*]
Return the first position in the string at which the character *ch* appears. If ch does not appear, then return **nil**.
function *rindex*[*ch* : *Character*] → [*r* : *Integer*]
Return the last position in the string at which the character *ch* appears. If ch does not appear, then return **nil**.
function *span*[*s* : *String*] → [*r* : *Integer*]
Return the first position in the string at which a character not in the string *s* appears. If all characters are in s, then return the length of the string.

function *cspan*[*s* : *String*] → [*r* : *Integer*]
Return the first position in the string at which a character in the string *s* appears. If no character in s appears in the string, then return the length of the string.
function *str*[*s* : *String*] → [*r* : *Integer*]
Return the first position in the string at which the string *s* appears as a substring. If s is not a substring of the string, then return **nil**.
operation *token*[*sep* : *String*] → [*token* : *String, rest*: *String*]
Parse a token from the front of the string. Leading characters that appear in the string *sep* are skipped, the first sequence of characters that do not appear in the string sep are returned in the result variable *token*. If any additional characters appear in the string after the token, then return the rest of the string after the token in the result variable *rest*, otherwise return **nil** in rest. If there is no token in the string (all characters are in the string sep), then return **nil** as both token and rest.

B.17 Time

Times represent times and dates. They are stored as a number of seconds (since Jan 1, 1970 when interpreted as dates) and a number of microseconds. They can be used as either dates or times, and the standard arithmetic operations are defined on them (where they make sense). The object Time is a type and has the following interface:

operation *create*[*seconds* : *Integer, microseconds* : *Integer*] → [*Time*]
Create a new time object with the given values for its seconds and microseconds fields.

Objects with type Time are immutable and have the following interface:

function + [*Time*] → [*Time*]
function − [*Time*] → [*Time*]
function ∗ [*n* : *Integer*] → [*Time*]
function / [*Integer*] → [*Time*]
Arithmetic functions.
function > [*Time*] → [*Boolean*]
function >= [*Time*] → [*Boolean*]
function < [*Time*] → [*Boolean*]
function <= [*Time*] → [*Boolean*]
function = [*Time*] → [*Boolean*]
function != [*Time*] → [*Boolean*]
Comparison functions.
function *getSeconds* → [*Integer*]
Return the seconds component.
function *getMicroSeconds* → [*Integer*]
Return the microseconds component.
function *asString* → [*String*]
Return a string representing the number of seconds and microseconds formatted in decimal, as in 6:002342.
function *asDate* → [*String*]
Return a string representing the value of the time as a date. The string has the form: Sat Aug 17 20:19:50 PDT 1996.

B.18 Type

The object *Type* represents the type of all types. It has no other useful operations.

B.19 Vector, VectorOfInt, VectorOfChar,

Vector is the name of an object with a polymorphic operation of that creates mutable vectors. Vectors provide the most primitive mechanism for acquiring indexable storage. The builtin object Array is implemented entirely in Emerald, using the facilities offered by Vector. The object Vector is immutable and has the following interface:

function *of*[*T* : *Type*] → [*aNewVectorCreatorType*]**forall** T

The creator object resulting from *Vector.of*[*T*] is a type with the following interface, named aNewVectorCreatorType:

operation *create*[*length* : *Integer*] → [*aNewVectorType*]
Create a new vector of length *length*, all of whose elements are **nil**.

Objects with type *Vector.of*[*T*] have the following interface, called aNewVectorType:

function *getElement*[*index* : *Integer*] → [*T*]
Return the element at *index*.
operation *setElement*[*index* : *Integer*, *value* : *T*]
Set the element at *index* to *value*.
function *lowerbound* → [*Integer*]
Return 0.
function *upperbound* → [*Integer*]
Return the largest valid index.
function *getSlice*[*lb* : *Integer*, *length*: *Integer*] → [*aNewVectorType*]
Return a new object with type aNewVectorType containing the *length* elements starting at position *lb*.
function *getElement*[*lb* : *Integer*, *length*: *Integer*] → [*aNewVectorType*]
The same as getSlice, but is able to use the subscript notation.

VectorOfInt is Vector.of[Integer].

VectorOfChar is Vector.of[Character].

B.20 ImmutableVector, ImmutableVectorOfAny, ImmutableVectorOfInt, ImmutableVectorOfString

ImmutableVector is the name of an object that creates immutable vectors. The object ImmutableVector is immutable and has the following interface:

function *of*[*T* : *Type*] → [*aNewVectorCreatorType*]**forall** T

The object resulting from *ImmutableVector.of*[*T*] is a type and a creator with the following interface, named aNewVectorCreatorType:

operation *create*[*length* : *Integer*] → [*aNewVectorType*]
Create a new immutable vector of length *length*, all of whose elements are **nil**. This is not terribly useful.
operation *literal*[*other* : *Sequence.of*[*T*]] → [*aNewVectorType*]
Create a new immutable vector, initialized with all of the elements of *other*.
operation *literal*[*other* : *Sequence.of*[*T*], *length* : *Integer*] → [*aNewVectorType*]
Create a new immutable vector of length *length*, initialized with the first length elements of *other*.
operation *literal*[*other* : *Sequence.of*[*T*], *start* : *Integer*, *length* : *Integer*] → [*aNewVectorType*]
Create a new immutable vector of length *length*, initialized with the length elements of *other*, starting at index *start*.

Objects with type *ImmutableVector.of*[*T*] have the following interface, called aNewVectorType:

function *getElement*[*index* : *Integer*] → [*T*]
Return the element at *index*.
function *lowerbound* → [*Integer*]
Return 0.
function *upperbound* → [*Integer*]
Return the largest valid index.
operation *catenate* [*other* : *aNewVectorType*] → [*aNewVectorType*]
Return a new Immutable vector containing all of the elements of self followed by all of the elements of other.
function *getSlice*[*lb* : *Integer*, *length*: *Integer*] → [*aNewVectorType*]
Return a new object with type aNewVectorType containing the *length* elements starting a position *lb*.
function *getElement*[*lb* : *Integer*, *length*: *Integer*] → [*aNewVectorType*]
The same as getSlice, but is able to use the subscript notation.

ImmutableVectorOfAny is ImmutableVector.ofAny.

ImmutableVectorOfInt is ImmutableVector.ofInteger.

ImmutableVectorOfString is ImmutableVector.ofString.

C Compiling Emerald Programs

This appendix presents a sample Emerald program, and shows how Emerald programs are compiled and executed. Examples of the Emerald style of programming can be found in [**?**].

The latest implementation of Emerald uses an interpreter to increase portability. The compiler is written in Emerald, and generates executable files containing byte codes which are then interpreted. The two commands of interest are:

emc
: The emerald compiler is interactive. The primary commands of interest to casual Emerald users are:

  load filename
  : Define the environment in which this compilation is to be executed. Environments are stored in Unix files in the working directory. Exported identifiers are made known in the environment in order to facilitate separate compilation. A compilation environment consists of two files: *filename* and *filename*.idb. Both will be created as needed if they do not already exist.

  filename
  : Compile the given file (whose name should end in .m). The executable is written to a file with the extension .x.

emx
: The emerald interpreter. Typically invoked as:

  emx filename.x

  Emx supports a large number of flags, the most useful of which are:

  -i
  : Invoke the integrated Emerald debugger when errors occur.

  -Tcall
  : Generate a call trace giving the calls and returns of all operations and functions.

  -v
  : Generate summary statistics on program termination.

The environment mechanism permits the sharing of object definitions between compilations. Objects (actually, identifiers) exported from one compilation unit that are to be used in another must be exported to the environment in order to be visible to later compilation units. An Emerald compilation unit is defined as follows:

compilation	::=	{ environmentExport | constantDeclaration }
environmentExport	::=	**export** identifier { **,** identifier }

For example, if an object *Directory* is being defined in compilation, and needs to be used by other objects (being compiled later), a statement:

export *Directory*

is needed. Other source code that uses the identifier *Directory* must be compiled in the context of this export.

```
% Define an Integer stack creation object
export Stack
const Stack ←
   immutable object Stack
      const StackType ←
         typeobject StackType
            operation Push[n: Integer]
            operation Pop → [n: Integer]
            function Empty → [result : Boolean]
         end StackType
      export function getSignature → [r : Signature]
         r ← StackType
      end getSignature
      export operation create → [result : StackType]
         result ←
            object aStack
               const store ← Array.of[Integer].empty
               export operation Push[n: Integer]
                  store.addUpper[n]
               end Push
               export operation Pop → [n: Integer]
                  n ← store.removeUpper
               end Pop
               export function Empty → [result : Boolean]
                  result ← store.empty
               end Empty
            end aStack
      end create
end Stack
```

Figure 2: The file `stack.m`

C.1 A Sample Program

This simple example involves an integer stack creator (*Stack*), which is compiled first, and a test object (*Tester*), which is compiled later and makes use of the *Stack*. The two programs are entered into separate Unix files.

The file `stack.m`

This Emerald program file defines the stack-creator (see Figure **??**); once created, this object accepts *create* invocations and returns a new stack-like object (conforming to *Stacktype*) on each such invocation. See [**?**, **?**] for a better understanding of this program. Note that the name Stack is exported to the compilation environment using the export directive.

```
% This program first creates a stack named myStack by invoking Stack.create.
% It pushes 4 integers into myStack, and then pops and prints them.
const Tester ← object Tester
   process
      const myStack: Stack ← Stack.create
      for i : Integer ← 0 while i < 4 by i ← i + 1
         stdout.PutString["Pushing " || i.asString || " on my stack.\n"]
         myStack.Push[i]
      end for
      stdout.PutString["Printing in reverse order.\n"]
      loop
         var x: Integer
         exit when myStack.Empty
         x ← myStack.Pop
         stdout.PutString["Popped " || x.asString || " from my stack.\n"]
      end loop
      stdout.close
      stdin.close
   end process
end Tester
```

Figure 3: The file `tester.m`

The file `tester.m`

This file (see Figure **??**) contains a simple object that can be used to test the *Stack* object defined in Figure **??**. This object invokes the stack-creator object *Stack* to create the new stack named *myStack*; the rest of the program is fairly straight-forward. The name *Stack* in unbound in this example, therefore this program must be compiled in a compilation environment that includes the name *Stack*. Also note the predefined identifier *stdin* and *stdout* which name the (already opened) standard input and output streams respectively.

C.2 Compiling the Program

Compiling the program requires the following steps. We assume that the source files are named stack.m and stacktest.m. The following steps provide input to the compiler at its "Command:" prompt, after it has been started using the Unix command "emc".

1. Direct the compiler to load the stack environment: "load stackenv".
2. Compile the stack creator: "stack.m".
3. Compile the stack tester: "stacktest.m".
4. Terminate the compiler "quit".

C.3 Executing the Program

The previous steps have generated two emerald executable files, stack.x and stacktest.x. These must be executed together:

emx stack.x stacktest.x

On executing this command, we get the following output:

```
Pushing 0 on my stack.
Pushing 1 on my stack.
Pushing 2 on my stack.
Pushing 3 on my stack.
Printing in reverse order.
Popped 3 from my stack.
Popped 2 from my stack.
Popped 1 from my stack.
Popped 0 from my stack.
```

D Debugging Emerald Programs

This appendix presents the interface to the Emerald debugger, edb. Edb is built into the interpreter and is enabled by invoking the interpreter with the -i option, as either:

    emx -i broken.x

or

    broken.x -i

D.1 Breakpoints

The debugger currently does not support breakpoints, but it does allow one to continue past a failed assertion. Therefore the user may insert breakpoints by:

1. Inserting an assertion into the source program. For a conditional breakpoint, assert the negation of the condition that you want to cause the program to stop. For an unconditional breakpoint, assert false.
2. Recompile the program. Note that if you are compiling a large system, using "unset perfile" will cause all generated code to be added to a single file, named CP (for checkpoint), and later added code will overwrite previous code.
3. Run the code. When the breakpoint happens, you can poke around, and continue using the continue command.

D.2 Commands

The debugger supports the following commands:

quit

q
: Terminate both the debugger and the interpreter.

where
: Provide a brief stack backtrace of the current process.

dump
: Provide a complete stack backtrace of the current process. This backtrace includes instance variable of the target of each invocation as well as parameters, results, and local variables in each invocation.

processes
: List all the processes in the current program. For each, the location of the bottom (most recent) invocation is given.

process n
: Switch the debugger's attention to the stack of process number n. The available processes are given using the "processes" command.

continue

cont

c

Continue execution of the target program. This will likely cause tremendous problems if the cause of the program's stop was anything other than a failed assertion. Segmentation faults can be expected for continuing through other failures.

print e

p e

Display the value of the expression e. Expressions take the form of identifier{.identifier}. Spaces are not permitted in expressions. The current call stack is searched for the first identifier, additional identifiers are taken to be instance variables with that object. The debugger prints as much information as it can determine about the named object. To look at the current object (the one that was invoked at the current stack level) use the name "self".

up

Move up the call stack (towards older activations) one level.

down

Move down the call stack (towards younger activations) one level.

look

info

Print as much information as is available about the current activation.