The Coral Language Specification

Kateřina Nikola Lisová

May 30, 2014

Contents

Preface

Coral is a Ruby-like programming language which enhances advanced object-oriented programming with elements of functional programming. Everything is an object, in this sense it's a pure object-oriented language. Object blueprints are described by classes. Classes can be composed in multiple ways – classic inheritance and/or mixin composition, along with prototype-oriented inheritance.

Coral is also a functional language in the sense that every function is also an object, and generally, everything is a value. Therefore, function definitions can be nested and higher-order functions are supported out-of-the-box. Coral also has a limited support for pattern matching, which can emulate the algebraic types used in other functional languages.

Coral has been developed since 2012 in a home environment out of pure enthusiasm for programming and out of a desire for a truly versatile language. This document is a work in progress and will stay that way forever. It acts as a reference for the language definition and some core library classes.

Some of the languages that had major influence on the development of Coral, including syntax and behavior patterns, are Ruby, Ada, Scala, Java, C#, F# and Clojure. Coral tries to inherit their good parts and put them together in its own way.

The vast majority of Coral's syntax is inspired by *Ruby*. Coral uses keyword program parentheses in Ruby fashion. There is **`class`** …**`end`**, **`def`** …**`end`**, **`do`** …**`end`**, **`loop`** …**`end`**. Ruby itself is inspired by other languages, so this relation is transitive and Coral is inspired by those languages as well (for example, Ada).

Coral is inspired by *Ada* in the way that user identifiers are formatted: `Some_Constant_Name` and — unlike in Ada, but quite similar to it — `some_method_name`. Also, some control structures are inspired by Ada, such as loops, named loops, return expressions and record types. Pretty much like in Ada, Coral's control structures can be usually ended the same way: **`class`** …**`end class`** etc.

Scala influenced the type system in Coral. Syntax for existential types comes almost directly from it. However, Coral is a rather dynamically typed language, so the type checks are made eventually in runtime (but some limited type checks can be made during compile time as well). Moreover, the structure of this mere specification is inspired by Scala's specification.

From *F#*, Coral borrows some functional syntax (like function composition) and F# also inspired the feature of Units of Measure.

Clojure inspired Coral in the way functions can get their names. Coral realizes that turning function names into sentences does not always work, so it is possible to use dashes, plus signs and slashes inside of function names. Therefore, `call/cc` is a legit function identifier. Indeed, binary operators are required to be properly surrounded by whitespace or other non-identifier characters.

Chapter 1

Lexical Syntax

Coral programs are written using the Unicode character set; Unicode supplementary characters are supported as well. Coral programs are preferably encoded with the UTF-8 character encoding. While every Unicode character is supported, usage of Unicode escapes is encouraged, since fonts that IDEs might use may not support the full Unicode character set.

Grammar of lexical tokens is given in the following sections. These tokens are then used as terminal symbols of the semantical grammar.

1.1 Identifiers

Syntax:

```
simple_id   ::= lower [id_rest]
variable_id ::= simple_id | '_'
ivar_id     ::= '@' simple_id
cvar_id     ::= '@@' simple_id
function_id ::= simple_id [id_rest_fun]
constant_id ::= upper [id_rest_con]
id_rest     ::= {letter | digit | '_'}
id_rest_con ::= id_rest [id_rest_mid]
id_rest_fun ::= id_rest [id_rest_mid] ['?' | '!' | '=']
id_rest_mid ::= id_rest {('/' | '+' | '-') id_rest}
```

There are three kinds of identifiers.

First, *variable identifiers*, which are simply a lower-case letter followed by arbitrary sequence of letters (any-case), digits and underscores, or just one underscore (which has special meaning). Additionally, *instance variable identifiers* are just prepended with a "`@`" sign and *class instance variable identifiers* are just prepended with "`@@`".

Second, *function identifiers*, which are the most complicated ones. They can start as a variable identifier, then optionally followed by one of "`/`", "`+`" and "`-`", and then optionally ended with "`?`", "`!`" or "`=`". Furthermore, function identifiers ending with "`=`" are never used at call site with this last character, but without it and as a target of an assignment expression (they are naming simple setters).

And third, *constant identifiers*, which are just like function identifiers, but starting with an upper-case letter, never just an underscore and never ending with "`?`", "`!`" or "`=`".

Coral programs are parsed greedily, so that a longest match rule applies. Letters from the syntax may be any Unicode letters, but English alphabet letters are recommended, along with English names.

1.2 Keywords

A set of identifiers is reserved for language features instead of for user identifiers. However, unlike in most other languages, keywords are not being recognized inside of paths, except for a few specific cases.

The following names are the reserved words.

alias	**annotation**	**as**	**begin**	**bitfield**
break	**case**	**cast**	**catch**	**class**
clone	**constant**	**constructor**	**declare**	**def**
destructor	**do**	**else**	**elsif**	**end**
ensure	**enum**	**for**	**for-some**	**function**
goto	**if**	**implements**	**in**	**include**
interface	**is**	**let**	**loop**	**match**
memoize	**message**	**method**	**mixin**	**module**
native	**next**	**nil**	**no**	**of**
opaque	**operator**	**out**	**prepend**	**property**
protocol	**raise**	**range**	**record**	**redo**
refine	**rescue**	**retry**	**return**	**self**
skip	**struct**	**super**	**template**	**test**
then	**this**	**throw**	**transparent**	**type**
undef	**unless**	**until**	**union**	**unit-of-measure**
use	**val**	**var**	**void**	**yes**
when	**while**	**with**	**yield**	

Not every reserved word is a keyword in every context, this behavior will be further explained. For example, the bitfield reserved word is only recognized as a keyword inside an enumeration definition context, in a specific place. Every reserved word may be used as a function identifier, with a little workaround when used with an implicit receiver.

1.3 Newline Characters

Syntax:

```
semi ::= nl {nl} | ';'
```

Coral is a line-oriented language, in which statements are expressions and may be terminated by newlines, as well as by semi-colon operator. A newline in a Coral source file is treated as the special separator token `nl` if the following criterion is satisfied:

1. The token immediately preceding the newline can terminate an expression.

Since Coral may be interpreted in a REPL[1] fashion, there are no other suitable criteria. Such a token that can terminate an expression is, for instance, not a binary operator or a message sending operator, which both require further

[1]Read-Eval-Print Loop

tokens to create an expression. Keywords that expect any following tokens also can not terminate expressions. Coral interpreters and compilers do not look-ahead beyond newlines.

If the token immediately preceding the newline can not terminate an expression and is followed by more than one newline, Coral still sees that as only a one significant newline, to prevent any confusion.

Keywords that can terminate an expression are: **`break`**, **`end`**, **`opaque`**, **`native`**, **`next`**, **`nil`**, **`no`**, **`redo`**, **`retry`**, **`return`**, **`self`**, **`skip`**, **`super`**, **`this`**, **`transparent`**, **`void`**, **`yes`**, **`yield`**.

1.4 Operators

A set of identifiers is reserved for language features, some of which may be overridden by user space implementations. Operators have language-defined precedence rules that are supposed to usually comply to user expectations (principle of least surprise), and another desired precedence may be obtained by putting expressions with operators inside of parenthesis pairs.

The following character sequences are the operators recognized by Coral.

```
:=     +=     -=     *=     **=    /=     %=     ||=    &&=    ^^=
|=     &=     |=     &=     ^=     ~=     <<     >>     <<<    >>>
<<=    >>=    <<<=   >>>=   ;      =      !=     ==     !==    ===
!===   =~     !~     <>     <      >      <=     >=     <=>    +
-      *      **     /      div    %      mod    ||     or     &&
and    !      not    ^^     xor    |      &      ^      ~      ..
...    ,      ->     <-     ~>     <~     =>     ::     :      <:
:>     <<|    |>>    <|     |>     (      )      [      ]      {
}      .
```

Some of these operators have multiple meanings, usually up to two. Some are binary, some are unary, none is ternary.

Binary (infix) operators have to be separated by whitespace or non-letter characters on both sides, unary operators on left side – the right side is what they are bound to.

Unary operators are: `+`, `-`, `&`, `not`, `!` and `~`. The first three of these are binary as well. The `;` operator is used to separate expressions (see Newline Characters). Parentheses are postcircumfix operators. Coral has no postfix operators.

Coral allows for custom user-defined operators, but those have the lowest precedence and need to be parenthesized in order to express any precedence. Such custom operators can't be made of letter characters.

1.5 Literals

There are literals for numbers (including integer, floating point and complex), characters, booleans, strings, symbols, regular expressions and collections (including tuples, lists, dictionaries and bags).

Syntax:

```
literal ::= integer_literal
     | floating_point_literal
     | complex_literal
     | character_literal
     | string_literal
     | symbol_literal
     | regular_expression_literal
     | collection_literal
     | 'nil'
```

1.5.1 Integer Literals

Syntax:

```
integer_literal     ::= ['+' | '-'] (decimal_numeral
     | hexadecimal_numeral
     | octal_numeral
     | binary_numeral)
decimal_numeral     ::= '0' | non_zero_digit {['_'] digit}
hexadecimal_numeral ::= '0x' | hex_digit {['_'] hex_digit}
digit               ::= '0' | non_zero_digit
non_zero_digit      ::= '1' | … | '9'
hex_digit           ::= '1' | … | '9' | 'a' | … | 'f'
octal_numeral       ::= '0' oct_digit {'_' oct_digit}
oct_digit           ::= '0' | … | '7'
binary_numeral      ::= '0b' bin_digit {['_'] bin_digit}
bin_digit           ::= '0' | '1'
```

Integers are usually of type `Number`, which is a class cluster of all classes that can represent numbers. Unlike Java, Coral supports both signed and unsigned integers directly. Usually integer literals that are obviously unsigned integers are automatically represented internally by a class that stores the integer unsigned, like `Integer_64_Unsigned`. Math operations on numbers are handled internally in such a way that the user does't need to worry about the actual types of the numbers — when an integer overflow would occur, the result is stored in a larger container type.

Underscores used in integer literals have no special meaning, other than to improve readability of larger literals, i.e., to separate thousands.

Integral members of the `Number` class cluster include the following container types.

1. `Integer_8` (-2^7 to $2^7 - 1$), alias `Byte`
2. `Integer_8_Unsigned` (0 to 2^8), alias `Byte_Unsigned`
3. `Integer_16` (-2^{15} to $2^{15} - 1$), alias `Short`
4. `Integer_16_Unsigned` (0 to 2^{16}), alias `Short_Unsigned`
5. `Integer_32` (-2^{31} to $2^{31} - 1$)
6. `Integer_32_Unsigned` (0 to 2^{32})
7. `Integer_64` (-2^{63} to $2^{63} - 1$), alias `Long`
8. `Integer_64_Unsigned` (0 to 2^{64}), alias `Long_Unsigned`
9. `Integer_128` (-2^{127} to $2^{127} - 1$), alias `Double_Long`
10. `Integer_128_Unsigned` (0 to 2^{128}), alias `Double_Long_Unsigned`
11. `Decimal` ($-\infty$ to ∞)
12. `Decimal_Unsigned` (0 to ∞)

The special `Decimal` & `Decimal_Unsigned` container types are also for storing arbitrary precision floating point numbers. All the container types are constants defined in the `Number` class and can be imported into scope if needed.

Moreover, a helper type `Number::Unsigned` exists, which can be used for type casting in cases where an originally signed number needs to be treated as unsigned.

Weak conformance applies to the inner members of `Number` class.

For use with range types, `Number::Integer` and `Number::Integer_Unsigned` exist, to allow constraining of the range types to integral numbers.

1.5.2 Floating Point Literals

Syntax:

```
float_literal ::= ['+' | '-'] non_zero_digit
        {['_'] digit} '.' digit {['_'] digit}
        [exponent_part] [float_type]
    | ['+' | '-'] digit {['_'] digit} exponent_part [float_type]
    | ['+' | '-'] digit {['_'] digit} [exponent_part] [float_type]
    | ['+' | '-'] '0x' hex_digit
        {['_'] hex_digit} '.' hex_digit {['_'] hex_digit}
        [float_type]
    | ['+' | '-'] '0b' bin_digit
        {['_'] bin_digit} '.' bin_digit {['_'] bin_digit}
        [float_type]
exponent_part ::= 'e' ['+' | '-'] digit {['_'] digit}
float_type    ::= 'f' | 'd' | 'q'
```

Floating point literals are of type `Number` as well as integral literals, and have fewer container types. Compiler infers the precision automatically, unless the `float_type` part is present.

1. `Float_32` (IEEE 754 32-bit precision), alias `Float`.
2. `Float_64` (IEEE 754 64-bit precision), alias `Double`.
3. `Float_128` (IEEE 754 128-bit precision).
4. `Decimal` ($-\infty$ to ∞).
5. `Decimal_Unsigned` (0 to ∞).

Letters in the exponent type and float type literals have to be lower-case in Coral sources, but functions that parse floating point numbers do support them being upper-case for compatibility.

1.5.3 Imaginary Number Literals

Syntax:

```
imaginary_literal ::= real_number_literal 'i'
complex_literal ::= real_number_literal ('+' | '-') imaginary_literal
    | imaginary_literal ('+' | '-') real_number_literal
real_number_literal ::= integer_literal | float_literal
number_literal ::= real_number_literal
    | imaginary_literal
    | complex_literal
```

1.5.4 Units of Measure

Coral has an addition to number handling, called *units of measure*. Number instances can be annotated with a unit of measure to ensure correctness of arithmetic operations.

Syntax:

```
annotated_number ::= number_literal '[<' units_of_measure_expr '>]'
```

1.5.5 Character Literals

Syntax:

```
character_literal ::= '%'' (character | unicode_escape) '''
```

1.5.6 Boolean Literals

Syntax:

```
boolean_literal ::= 'yes' | 'no'
```

Both literals are members of type `Boolean`. The **`no`** literal has also a special behavior when being compared to **`nil`**: **`no`** equals to **`nil`**, while not actually being **`nil`**. Identity equality is indeed different. The implication is that both **`nil`** and **`no`** are false conditions in **`if`**-expressions.

1.5.7 String Literals

Syntax:

```
string_literal               ::= simple_string_literal
                               | interpolable_string_literal
simple_string_literal        ::= ''' {string_element} '''
string_element               ::= printable_char | char_escape_seq
interpolable_string_literal ::= '"' {int_string_element} '"'
int_string_element           ::= string_element | interpolated_expr
interpolated_expr            ::= '#{' expr '}'
```

String literals are members of the type `String`. Single quotes in simple string literals have to be escaped (\') and double quotes in interpolable string literals have to be escaped (\"). Interpolated expression can be preceded only by an even number of escape characters (backslashes, \), so that the # does't get escaped. This is a special *requirement* for any Coral compiler.

1.5.8 Symbol Literals

Syntax:

```
symbol_literal       ::= simple_symbol | quoted_symbol
simple_symbol        ::= ':' simple_id
quoted_symbol        ::= simple_quoted_symbol | interpolable_symbol
simple_quoted_symbol ::= ':'' {string_element} '''
interpolable_symbol  ::= ':"' {int_string_element} '"'
```

Symbol literals are members of the type `Symbol`. They differ from String Literals in the way runtime handles them: while there may be multiple instances of the same string, there is always up to one instance of the same symbol. Unlike in Ruby, they do get released from memory when no code references to them anymore, so their object id (sometimes) varies with time. Coral does not require their ids to be constant in time.

1.5.9 Type Parameters

Syntax:

```
type_param ::= '$' (simple_id | constant_id)
```

Type parameters are not members of any type, rather they stand-in for a real type, like a variable which only holds types.

1.5.10 Regular Expression Literals

Syntax:

```
regexp_literal ::= '%/' regexp_content_int '/' [regexp_flags]
    | '%r/' regexp_content_int '/' [regexp_flags]
    | '%r#' regexp_content '#' [regexp_flags]
    | '%r~' regexp_content_int '~' [regexp_flags]
regexp_content_int ::= regexp_element_int {regexp_element_int}
regexp_element_int ::= string_element | int_string_element
regexp_content ::= string_element {string_element}
```

Regular expression literals are members of the type `Regular_Expression` with alias of `Regexp`.

1.5.11 Collection Literals

Collection literals are paired syntax tokens and as such, they are a kind of parentheses in Coral sources.

Syntax:

```
collection_literal ::= tuple_literal
    | list_literal
    | dictionary_literal
    | bag_literal
tuple_literal ::= '(' exprs ')'
list_literal ::= '%' collection_flags '[' exprs ']'
dictionary_literal ::= '%' collection_flags '{' dict_exprs '}'
bag_literal ::= '%' collection_flags '(' exprs ')'
exprs ::= expr {',' expr}
dict_exprs ::= dict_expr {',' dict_expr}
dict_expr ::= expr '=>' expr
    | simple_id ':' expr
collection_flags ::= printable_char {printable_char}
```

Tuple literals are members of the `Tuple` type family. List literals are members of the `List` type, usually `Array_List` with alias of `Array`. Dictionary literals are members of the `Dictionary` type with alias of `Map`, usually `Hash_Dictionary` with alias of `Hash_Map`. Bag literals are members of the `Bag` type, usually `Hash_Bag` or `Hash_Set`. Collection flags may change the actual class of the literal, along with some other properties, described in the following text.

List literal collection flags:

1. Flag `i` = **i**mmutable, makes the list frozen.
2. Flag `l` = **l**inked, makes the list a member of `Linked_List`.
3. Flag `w` = **w**ords, the following expressions are treated as words, converted to strings for each word separated by whitespace.

Dictionary literals collection flags:

1. Flag `i` = **i**mmutable, makes the dictionary frozen.
2. Flag `l` = **l**inked, makes the dictionary a member of `Linked_Hash_Dictionary` (also has alias `Linked_Hash_Map`).
3. Flag `m` = **m**ulti-map, the dictionary items are then either the items themselves, if there is only one for a particular key, or a set of items, if there is more than one item for a particular key. The dictionary is then a member of `Multi_Hash_Dictionary` (alias `Multi_Hash_Map`) or `Linked_Multi_Hash_Dictionary` (alias `Linked_Multi_Hash_Map`).

Bag literal collection flags:

1. Flag `i` = **i**mmutable, makes the bag frozen.
2. Flag `s` = **s**et, the collection is a set instead of a bag (a specific bag, such that for each item, its tally is always 0 or 1, thus each item is in the collection up to once).
3. Flag `l` = **l**inked, makes the collection linked, so either a member of `Linked_Hash_Bag` in case of a regular bag, or `Linked_Hash_Set` in case of a set.

Linked collections have a predictable iteration order in case of bags and dictionaries, or are simply stored differently in case of lists.

1.6 Whitespace & Comments

Tokens may be separated by whitespace characters and/or comments. Comments come in two forms:

A single-line comment is a sequence of characters that starts with `//` and extends to the end of the line.

A multi-line comment is a sequence of characters between `/*` and `*/`. Multi-line comments may be nested.

Documentation comments are multi-line comments that start with `/*!`.

Chapter 2

Identifiers, Names & Scopes

Names in Coral identify various types, values, methods and constants, which are the *entities*. Names are introduced by local definitions and declarations, inheritance, use clauses or module clauses, which are the *bindings*.

Bindings of different kinds have a different precedence defined on them:

1. Definitions and declarations that are local have the highest precedence.
2. Explicit **`use`** clauses (imports) have the next highest precedence.[1]
3. Inherited definitions and declarations have the next highest precedence.
4. Definitions and declarations made available by module clause have the next highest precedence.
5. Definitions and declarations that are not in the same compilation unit (a different script or a different module) have the next highest precedence.
6. Definitions and declarations that are not bound have the lowest precedence. This happens when the binding simply can't be found anywhere, and probably will result in a name error (if not resolved dynamically), while being inferred to be of type `Object`.

There is only one root name space, in which a single fully-qualified binding designates always up to one entity.

Every binding has a *scope* in which the bound entity can be referenced using a simple name (unqualified). Scopes are nested, inner scopes inherit the same

[1]Explicit imports have such high precedence in order to allow binding of different names than those that would be otherwise inherited.

bindings, unless shadowed. A binding in an inner scope *shadows* bindings of lower precedence in the same scope (and nested scopes) as well as bindings of the same or lower precedence in outer scopes. Shadowing is a partial order, and bindings can become ambiguous – fully qualified names can be used to resolve binding conflicts. This restriction is checked in limited scope during compilation[2] and fully in runtime.

If at any point of the program execution a binding would change (ie., by introducing a new type in a superclass that is closer in the inheritance tree to the actual class than the previous binding), and such a change would be incompatible with the previous binding, then a warning[3] will be issued by the runtime. Also, if a new binding would be ambiguous[4], then it is an error.

As shadowing is only a partial order, in a situation like

```
var x := 1
use p::x
x
```

neither binding of x shadows the other. Consequently, the reference to x on the third line above is ambiguous and the compiler will happily refuse to proceed.

A reference to an unqualified identifier x is bound by a unique binding, which

1. defines an entity with name x in the same scope as the identifier x, and
2. shadows all other bindings that define entities with name x in that name scope.

It is syntactically not an error if no such binding exists, thanks to the dynamic features of the language (unbound references are implicitly bound to the same scope and are resolved by dynamic method callbacks). The same applies to fully qualified bindings that don't resolve into any entity. However, it is an error if a binding is ambiguous or fails to get resolved dynamically.

If x is bound by explicit **use** import clause, then the simple name x is consided to be equivalent to the fully-qualified name to which x is mapped by the import clause. If x is bound by a definition or declaration, then x refers to the entity introduced by that binding, thus the type of x is the type of the referenced entity.

[2]This is due to the hybrid typing system in Coral, to make use of all the available information as soon as possible.

[3]TBD – shouldn't that be an error?

[4]Coral runtime actually checks for bindings until the binding-candidate would not be able to shadow the already found binding-candidates and caches the result.

Chapter 3

Types

When we say *type* in the context of Coral, we are talking about a blueprint of an entity, while the type itself is an entity. Every type in Coral is backed by a *class*, which is an instance of the type `Class`.

We distinguish a few different properties of types in Coral. There are first-order types and type constructors, which take type parameters and yield new types. A subset of first-order types called *value types* represents set of first-class values. Value types are either *concrete* or *abstract*.

Concrete value types can be either a *class type* (ie. referenced with a type designator, referencing a class or maybe a mixin), or a *compound type* representing an intersection of types, possibly with a refinement that further constrains the types of its members. Both class types and compound types may be bound to a constant, but only class types referencing a concrete class can be blueprints of values – *objects*. Compound types can only constrain bindings to a subset of other types.

Non-value types capture properties of identifiers that are not values. For instance, a type constructor does not directly specify a type of values, but a type constructor, when applied to the correct type arguments, yields a first-order type, which may be a value type. Non-value types are expressed indirectly in Coral. In example, a method type is described by writing down a method signature, which is not a real type itself, but it creates a corresponding method type.

3.1 About Coral's Type System

There are two main streams of typing systems out there – statically typed and dynamically typed. Static typing in a language usually means that the language is compiled into an executable with a definite set of types and every operation is type checked. Dynamic typing means that these checks are deferred until needed, in runtime.

Let's talk about Java. Java uses static typing – but, in a very limited and unfriendly way, you may use class loaders and a lot of type casts to dynamically load a new class. And then possibly endure a lot of pain using it.

Let's talk about Ruby. Ruby uses dynamic typing – but, using types blindly can possibly lead to some confusion. Ruby is amazing though, because you can write programs with it really fast and enjoy the process at the same time. But when it comes to type safety, you need to be careful.

And now, move on to Coral. Coral uses hybrid typing. In its core, it uses dynamic typing all the way. But, it allows to opt-in for some limited static typing[1]. Unlike in Ruby, you can overload methods (not just override!). You can constrain variables, constants, properties, arguments and return types to particular types. But you don't have to. Types in Coral were heavily inspired by Scala's type system, but modified for this dynamic environment that Coral provides. Unlike in Ruby, you can have pure interfaces (called protocols[2]), or interfaces with default method implementations (similar to Java 8). Unlike in Java, you can have mixins, union types and much more. Unlike in Java, you may easily modify classes, even from other modules (*pimp my library!*). You may even easily add more classes if needed, and possibly shadow existing ones. In face of static typing in Coral, *no type* specified is saying that the value is of any type.

While Coral is so dynamic, it also needs to maintain stability and performance. Therefore, it "caches" its bindings and tracks versions of each type[3]. If a *cached binding* would change, it is ok – as long as the new binding would conform to the old one. Practically, the code that executes first initiates the binding – first to come, first to bind. Bindings are also cached, so that the Coral interpreter does not need to traverse types all the time – it only does so if the needed binding does not exist (initial state), or if the cached version does not match the actual version of the bound type. This mechanism is also used for caching methods, not only types. Moreover, this mechanism ensures that type projections (§3.3.2) are valid at any time of execution, even if their binding changes.

Types in Coral are represented by objects that are members of the `Class` type.

[1]This feature is expected to be gradually improved and un-limited.

[2]Interfaces in Coral are used to extract the *public interface* of classes in modules, so that only a small amount of code may be distributed along with the module to allow binding to it.

[3]Versions are simply integers that are incremented with each significant change to the type and distributed among its subtypes.

3.2 Paths

Syntax:

```
Path             ::= Stable_Id
                   | 'this'
                   | [constant_id '#'] 'self'
Stable_Id        ::= constant_id
                   | ['::'] Path '::' constant_id
                   | [constant_id '#'] 'super' [Class_Qualifier]
                     '::' constant_id
Class_Qualifier  ::= '[' constant_id ']'
```

Paths are not types themselves, but they can be a part of named types and in that function form a role in Coral's type system.

A path is one of the following: [4]

- The empty path ϵ (which can not be written explicitly in user programs).
- **`this`**, which references the directly enclosing class.
- C**`#self`**, where C references a class or a mixin. The path **`self`** is taken as a shorthand for C**`#self`**, where C is the name of the class directly enclosing the reference.
- p`::`x, where p is a path and x is a member of p. Additionally, p allows modules to appear instead of references to classes or mixins, but no module reference can follow a class or a mixin reference: `{module_ref '::'} {(class_ref|mixin_ref) '::'}` ….
- C**`#super::`**x or C**`#super`**`[`M`]::`x, where C references a class or a mixin and x references a member of the superclass or designated parent class M of C. The prefix **`super`** is taken as a shorthand for C**`#super`**, where C is the name of the class directly enclosing the reference, and **`super`**`[`M`]` as a shorthand for C**`#super`**`[`M`]`, where C is yet again the name of the class directly enclosing the reference.

3.3 Value Types

Every value in Coral has a type which is of one of the following forms.

[4]This section might need a review of what a path is, since we claim that the referenced entity is a member, yet the syntax only mentions `constant_id`.

3.3.1 Value & Singleton Type

Syntax:

```
Simple_Type ::= Path '#' 'type'
Simple_Type ::= Path '#' 'singleton-type'
```

A singleton type is of the form p#`singleton-type` and a special type that denotes the set of values consisting of **`nil`** and the value denoted by p. A value type, on the other hand, is a special type that denotes the set of values consisting of **`nil`** and every value that conforms to the type of value denoted by p.[5]

3.3.2 Type Projection

Syntax:

```
Simple_Type ::= Simple_Type '#' constant_id
```

A type projection T#x references type member named x of type T. [6]

3.3.3 Type Designators

Syntax:

```
Simple_Type ::= Stable_Id
```

A type designator refers to a named value type. It can be simple or qualified. All such type designators are shorthands for type projections.

Specifically, the unqualified type name t where t is bound in some class, object or module C is taken as a shorthand for C#**`self`**#**`type`**#t. If t is not bound in a class, object or module, then t is taken as a shorthand for ϵ#**`type`**#t.

A qualified type designator has the form p`::`t, where p is a path (§3.2) and t is a type name. Such a type designator is equivalent to the type projection p#**`type`**#t.

[5]This is useful when using a value as prototype of new values.

[6]Type projection operator # is a language construct and can't be overridden by user programs. There is a similarity between this construct and the `::` scope operator. The difference is, type projection operator is expected to be rarely needed, but it does provide a type projection and can refer in a stable way to a type of anything. Scope operator, on the other hand, does not care about types, it merely resolves a member of a particular expression at runtime.

3.3.4 Parameterized Types

Syntax:

```
Simple_Type ::= Simple_Type Type_Args
Type_Args   ::= '[' Types ']'
Types       ::= Type {',' Type}
```

A parameterized type $T[T_1, \ldots, T_n]$ consists of a type designator T and type parameters $T_1, \ldots, T_n$, where $n \geq 1$. T must refer to a type constructor which takes exactly n type parameters $a_1, \ldots, a_n$.

Say the type parameters have lower bounds $L_1, \ldots, L_n$ and upper bounds $U_1, \ldots, U_n$. The parameterized type is well-formed if each actual type parameter *conforms to its bounds*, so that $L_i <: \sigma a_i <: U_i$, where σ is the substitution $[a_1 := T_1, \ldots, a_n := T_n]$. Also, U_i must never be a subtype of L_i, since no other type ever would be able to fulfil the bounds (U_i and L_i may be the exact same type though, but in that case the type parameter would be invariant and the whole point of having a parameterized type would be useless).

Example 3.3.1 Given the generic type definitions:

```
class Tree_Map[$A <: Comparable[$A], $B] ... end
class List[$A] ... end
class I; implements Comparable[I]; ... end

class F[$M[_], $X] ... end
class S[$K <: String] ... end
class G[$M[$Z <: Comparable[$I]], $I] ... end
```

the following parameterized types are well-formed:

```
Tree_Map[I, String]
List[I]
List[List[Boolean]]

F[List, Number]
G[S, String]
```

3.3.5 Tuple Types

3.3.6 Annotated Types

3.3.7 Compound Types

3.3.8 Function Types

3.3.9 Existential Types

3.4 Non-Value Types

3.4.1 Method Types

3.4.2 Polymorphic Method Types

3.4.3 Type Constructors

3.5 Relations Between Types

3.5.1 Type Equivalence

3.5.2 Conformance

Chapter 4

Basic Declarations & Definitions

4.1 Variable Declarations & Definitions

4.2 Property Declarations & Definitions

4.3 Instance Variable Definitions

4.4 Type Declarations & Aliases

4.5 Type Parameters

4.6 Variance of Type Parameters

4.7 Function Declarations & Definitions

4.7.1 Positional Parameters

4.7.2 Optional Parameters

4.7.3 Repeated Parameters

4.7.4 Named Parameters

4.7.5 Procedures

4.7.6 Method Return Type Inference

4.8 Use Clauses

Chapter 5

Classes & Objects

5.1 Class Definitions

5.1.1 Class Linearization

5.1.2 Constructor & Destructor Definitions

5.1.3 Class Block

5.1.4 Class Members

5.1.5 Overriding

5.1.6 Inheritance Closure

5.1.7 Modifiers

5.2 Mixins

5.3 Unions

5.4 Enums

5.5 Compound Types

5.6 Range Types

5.7 Units of Measure

5.8 Record Types

Chapter 6

Expressions

6.1 Expression Typing

6.2 Literals

6.3 The Nil Value

6.4 Designators

6.5 Self, This & Super

6.6 Function Applications

6.6.1 Named and Optional Arguments

6.6.2 Input & Output Arguments

6.6.3 Function Compositions & Pipelines

6.7 Method Values

6.8 Type Applications

6.9 Tuples

6.10 Instance Creation Expressions

6.11 Blocks

Chapter 7

Implicit Parameters & Views

Chapter 8

Pattern Matching

8.1 Patterns

8.1.1 Variable Patterns

8.1.2 Typed Patterns

8.1.3 Literal Patterns

8.1.4 Constructor Patterns

8.1.5 Tuple Patterns

8.1.6 Extractor Patterns

8.1.7 Pattern Alternatives

8.1.8 Regular Expression Patterns

8.2 Type Patterns

8.3 Pattern Matching Expressions

8.4 Pattern Matching Anonymous Functions

Chapter 9

Top-Level Definitions

9.1 Compilation Units

9.2 Modules

9.3 Module Objects

9.4 Module References

9.5 Top-Level Classes

9.6 Programs

Chapter 10

Annotations

Chapter 11

Naming Guidelines

Chapter 12

The Coral Standard Library

12.1 Root Classes

12.1.1 The Object Class

12.1.2 The Nothing Class

12.2 Value Classes

12.3 Standard Reference Classes

Chapter A

Coral Syntax Summary