CS4120/4121/5120/5121—Spring 2023

Rho$_O$ Language Specification

Cornell University

Version of May 20, 2023

In response to customer demands for object-oriented features, a new object-oriented version of the language, called Rho$_O$, has been been partly designed. Your task[1] is to extend your Rho implementation with the new object-oriented features, and to add some language extension of your own design.

Rho$_O$ is backward-compatible with Rho, so this language description focuses on the differences. The original Eta and Rho specs should also be consulted.

0 Changes

- No changes yet.

1 Class definitions

An Rho$_O$ program may contain class definitions in addition to function definitions. Record definitions are understood as simplified class definitions that do not have methods. A class definition may contain instance variable (field) definitions, and method definitions. For example, the following code defines a `Point` class and an associated creator function `createPoint`:

```
1  class Point { // a mutable point
2    x,y: int
3
4    move(dx: int, dy: int) {
5      x = x + dx
6      y = y + dy
7    }
8    coords() : int, int {
9      return x, y
10   }
11   add(p: Point) : Point {
12     return createPoint(x + p.x, y + p.y)
13   }
14   initPoint(x0: int, y0: int): Point {
15       x = x0
16       y = y0
17       return this
18   }
19   clone(): Point { return createPoint(x, y) }
20   equals(p: Point): bool { return this == p }
21 }
22
23 createPoint(x: int, y: int): Point {
24   return new Point.initPoint(x, y)
25 }
```

As in Java, there is a special variable `this` that refers to the method receiver. The instance variables and methods of `this` are automatically in scope within methods. Where visible, instance variables can also be accessed with the usual dot notation, e.g. `p.x` at line 12.

[1]Entirely optional in Spring 2023.

Notice that the fields `x` and `y` can both be declared at once to have the type `int`. The same abbreviated syntax can now be used for local variable declarations, though no initializer expression may be provided in that case.

2 Class declarations

Classes do not have visibility modifiers. All class members, including instance variables, are visible everywhere inside their module (i.e., source file). Instance variables are only visible within the module that defines them and cannot be mentioned in interfaces. For example, we might define an interface file for the `Point` class, hiding the `x` and `y` fields:

```
// A 2D Point with integer coordinates (x,y).
class Point {
    move(dx: int, dy: int)
    add(p: Point): Point
    coords(): int, int
    clone(): Point

    // Initialize this to contain (x,y).
    // Returns: this
    initPoint(x: int, y: int): Point
}

// Create the point (x,y).
createPoint(x: int, y: int): Point
```

A class may inherit from one other class, with an `extends` clause. For example, we might declare a subclass `ColoredPoint` that inherits from `Point`:

```
class Color {
    r,g,b: int
}

class ColoredPoint extends Point {
    col: Color
    color(): Color { return col }

    initColoredPoint(x0: int, y0: int, c: Color): ColoredPoint {
        col = c
        _ = initPoint(x0, y0)
        return this
    }
}
```

There are no class variables or class methods (Java's `static`) in Rho$_O$, because ordinary functions as in Eta, and the newly introduced global variables, can be used instead.

3 Modules and interfaces

The extension `.rh` continues to be used for files containing module definitions and the extension `.ri` is used for interface files. Therefore, the statement `use my_module` appearing in a module causes the compiler to look for the interface in the file `my_module.ri`.

If a module that defines a class `C` references (either explicitly or implicitly) an interface that declares class `C`, the class definition must match the interface declaration. It must implement all of the new methods that the interface declares. It may also add additional methods that are not declared in the interface.

Similarly, if a module references an interface that declares a procedure or function that is defined in the module, the signature of the procedure or function declared in the interface must match the definition in the module.

A module does not need to have an interface. Further, even if there is an interface, not every class, procedure, or function in the module needs to be declared in that interface. Undeclared components are analogous to private classes or private methods in Java because they are not visible outside their own module.

Globals, including classes and functions, are in scope throughout their defining module and throughout any module that uses an interface that declares them.

4 Subtyping and conformance

The subtyping relationship of Eta is extended to classes in a straightforward way. Every class name can be used as a type. A class is a subtype of the class it extends, if any. It is therefore also a subtype of any supertypes of that class. There is no top class in the subtype ordering (no `Object`).

Classes must conform to their declared superclasses. They may override and redeclare methods from the superclass, but if they do, the new method signature must match the signature in the superclass. They may not declare methods whose names shadow those in superclasses.

The subtyping rule on arrays is unchanged—if classes `B` and `C` extend class `A`, you cannot use a value of type `B[]` at type `A[]`, since then you could put a `C` into it and cause a run-time type error.

5 New operators

A new object is created with the syntax `new C`. This can only be done inside the module where `C` is defined—standalone creator operations must be implemented as functions like `createPoint` above. As the `Point` example shows, this operator has higher precedence than "`.`" does.

Inside a module defining the class *C*, the equality comparison == and the inequality comparison != can be used on a value of type *C*. These are implemented as pointer comparisons, much like in Java, though ones that are private to *C*'s module. Such comparisons are legal as long as one of two operands is of type *C*. It is a static error to compare an object with an array or anything else that does not have an object type.

6 Non-OO extensions

6.1 Global variables

A module can contain global variable declarations, with limited support for initialization.

```
center: Point
corner: Point
len: int = 10
tenpoints: Point[len]
```

Integer variables may be initialized to an integer literal, and boolean variables may be initialized to a boolean literal. Global arrays with constant length can be declared, as shown, and are initialized with default element valuesThe length may be given either as an integer literal or as the name of a global variable. Global multidimensional array declarations are also supported.

Global variables cannot be declared in interfaces, so they are private to a given module. Two different global variables in different modules are different global variables even if they happen to be declared with the same name.

6.2 Extended array initialization

Array initializers can now specify the initial values of elements of the array, by assigning to an expression. For example, in the following three lines of code, the first is already supported by Xi but the second two are not:

```
1 a: int[][] = {{1, 0}, {0,1}}
2 b: int[4][3] = 0
3 c: bool[2][] = {true, false, false}
4 d: int[i:5] = i + 1
5 a2: int[i:2][j:2] = (i + j + 1) % 2
6 a3: int[i:2][] = {(i+1)%2, i}
```

The second line initializes `b` to be a 4×3 array whose elements are all zero. The third line initializes `c` to be a 2×3 array whose rows contain `false`, except for the first element.

Array initializers that specify a dimension size may also declare dimension index variables, which are in scope for the rest of the array initializer. For example, the fourth line declares `d` to be an array containing the elements `{1,2,3,4,5}`; the fifth line and sixth lines declare arrays `a2` and `a3` whose contents will be exactly the same as the variable `a`.

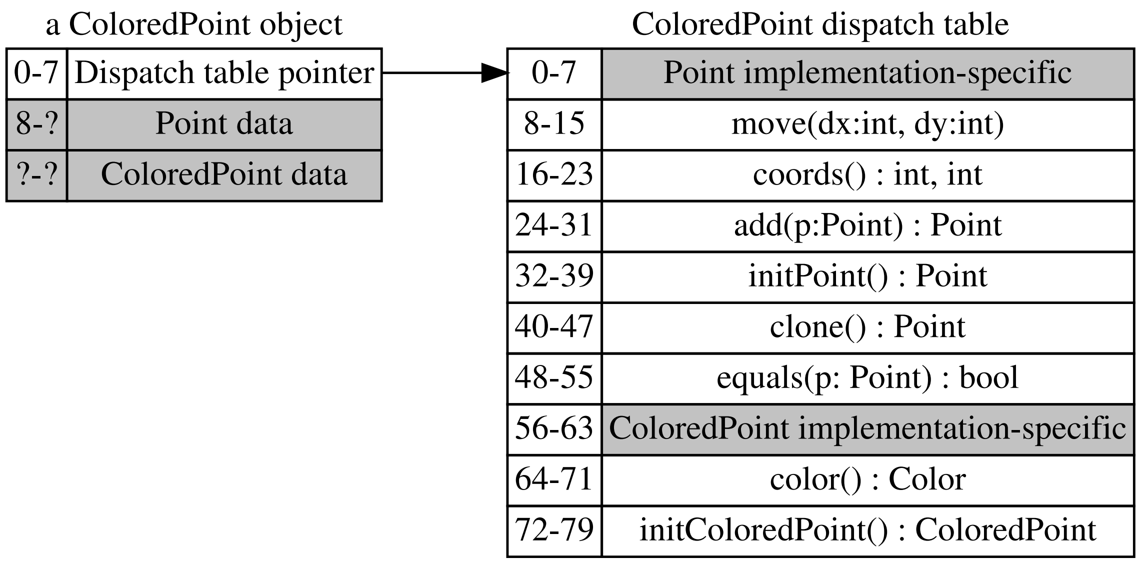
7 ABI

To give implementers flexibility, the ABI for Rho$_O$ specifies as little as is required for different implementations to interoperate, with implementations otherwise free to lay out data as desired.

7.1 Method calls

The first common need is dynamic dispatch of method calls. For this, objects must specify pointers to their classes' dispatch tables at their first memory location, with the tables laid out as follows:

Starting with the class at the top of the inheritance hierarchy, and moving down towards the most concrete class, first allocate a private slot for the use of whatever compiler built that class, then a pointer for each method in the order they are specified in the interface declaration. For example, in the ColoredPoint example above, the layout must look like this:



a ColoredPoint object

0-7	Dispatch table pointer
8-?	Point data
?-?	ColoredPoint data

ColoredPoint dispatch table

0-7	Point implementation-specific
8-15	move(dx:int, dy:int)
16-23	coords() : int, int
24-31	add(p:Point) : Point
32-39	initPoint() : Point
40-47	clone() : Point
48-55	equals(p: Point) : bool
56-63	ColoredPoint implementation-specific
64-71	color() : Color
72-79	initColoredPoint() : ColoredPoint

When a method is invoked, the reference to the receiver object is passed as if it was the first argument, before the actual arguments, but after any hidden argument used to return multiple results. Object references are stored in arrays and are passed to and returned from functions in the same way as other scalar types like `int` and `bool`. Top-level functions that take or return object reference should encode their types into method signatures as follows:

1. `o`
2. A number giving the length of the unescaped type name.
3. The name, with underscores escaped in the usual fashion.

For example, the method `average(a: Point, b: Point): Point` would have its name encoded as `_Iaverage_o5Pointo5Pointo5Point`. The naming of symbols for methods is implementation-specific since they cannot be called from a different compilation unit by name, only via dispatch vectors.

7.2 Fields

Fields of objects are laid out in the order they are declared in the class definition, with each field taking up one 64-bit word.

7.3 Global variables

The symbol names for global variables should be encoded as follows:

1. `_I_g_`
2. The name of the variable, with underscores escaped.
3. `_`
4. Encoding of the variable's type.

For example, a variable `points` of type `QPoint[]` will be encoded as `_I_g_points_ao6QPoint`.

7.4 Initialization

Notice that in order to allocate objects of type `ColoredPoint`, the size of objects of type `Point` must be known, but it may not be available during `ColoredPoint`'s compilation. Similarly, if `ColoredPoint` does not override some methods from `Point`, it needs to copy pointers to them from `Point`'s dispatch vector into its own.

Because of this, object sizes and dispatch vectors are to be computed at application startup time. The size of an object of type `someClass`, including areas for superclasses and the dispatch vector pointer, should be stored in the `_I_size_someClass` variable, while its dispatch vector should be stored under the `_I_vt_someClass` symbol, with underscores in the class name escaped under usual rules.

The size variables are initially set to 0[2] to denote that the size information and the dispatch vector for the given class have not yet been computed. In that case, the function `_I_init_someClass()` is expected to fully compute the size and dispatch vector information. When initializing its own dispatch vector, a subclass must copy pointers for all the methods it does not override, as well as all of its superclass' private class information pointers into the appropriate slots in its dispatch vector. An implementation is expected to avoid computing object sizes and dispatch vector more than once.

You may arrange for initialization functions to be called at startup by placing their addresses into the `.ctors` section of the object file; see `examples/init.s` in the runtime distribution for an example. The order of invocation of these initializers is not specified, however; and therefore any superclasses must have their initialization functions called recursively if necessary.

[2] An implementation may set them to the correct value if it is able to compute both the size and the dispatch vector fully at compile time.