IX Language Specification

Computer Science 4120
Cornell University

Version of December 21, 2009

Thanks to the hard work and brilliant engineering of the compilers team, Iota$_9$ has been gaining market share. In response to customer demands for object-oriented features, a new object-oriented version of the language, called IX, has been been partly designed. Your task is to extend your Iota$_9$ implementation with the new object-oriented features, and to add some language extension of your own design.

The IX is backward compatible with Iota$_9$, so this language description focuses on the differences. The original Iota$_9$ spec is still available.

1 Class definitions

An IX program may contain class definitions in addition to function definitions. 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:

```
class Point {
    x, y: int;

    move(dx:int, dy:int) = {
        x = x + dx;
        y = y + dy;
    }
    coords() : (int, int) = {
        return (x,y);
    }
    add(p: Point) : Point = {
        return createPoint(x + p.x, y + p.y);
    }
    clone() : Point = {
        return createPoint(x, y);
    }
}

createPoint(x:int, y:int): Point = {
    return initPoint(new Point, x, y);
}

initPoint(p: Point, x0:int, y0:int): Point = {
    p.x = x0;
    p.y = y0;
    return p;
}
```

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.

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). To be used from a different module, the class must be declared in an interface that includes all the methods of the class[1]. Instance variables are always private. 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
}

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

// Initialize a point to contain (x,y).
// Requires: p is uninitialized.
initPoint(p: Point, x: int, y:int)
```

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 {
    Color c;
    color() : Color = {
        return c;
    }
}

initColoredPoint(p: ColoredPoint, x0:int, y0:int, col:Color): ColoredPoint = {
    c = col;
    dummy:Point = initPoint(p, x0, y0);
    return p;
}
```

There are no class variables or class methods in IX, because ordinary functions as in Iota$_9$, and the newly introduced global variables, can be used instead.

3 Modules and interfaces

To distinguish between Iota$_9$ programs and IX programs, the extension `.ix` is used for module definitions. The extension `.ixi` 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.ixi`.

[1]The lack of private methods makes it possible to lay out dispatch tables at compile time

If a module that defines a class `C` references an interface that declares class `C`, the class definition must match the interface declaration. Therefore, it must implement all of the methods that the interface declares, and no additional methods. The order of the methods (in the object's dispatch table) is as defined in the interface, not as defined in the module. In typical usage, a module defined in `my_module.ix` would contain a statement "`use my_module;`", causing the compiler to read the interface `my_module.ixi` and to check the definitions against the interface. However, an interface is not required; even if there is an interface, classes need not be declared in that interface (these would correspond to private classes in Java).

4 Subtyping and conformance

The subtyping relationship of Iota$_9$ 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`), and class types are not related to array types.

Classes must conform to their declared superclasses: if they override methods, the new method signature must match the signature in the superclass. They may not declare methods or fields whose names shadow those in superclasses.

5 Initialization

For type safety, all storage is initialized before possible use in IX. Local variables, array elements, and object fields are initialized with a default value determined by their types[2]. The language implementation may choose to elide this initialization if the storage is always overwritten with another value before use.

5.1 Null

The special value `null` is a member of all class types and also a member of all array types. It is the default initialization value for all variables with class or array type, including global variables and array elements.

It is a run-time error to perform any operation on `null` that is specific to objects or arrays. The value `null` must be implemented as a pointer to memory location 0. A reference to this memory location will cause a page fault that will halt this program. This is an adequate implementation of the run-time checking for null values.

5.2 Other defaults

The default initialization values for `int` and `bool` are 0 and `false`, respectively.

6 New operators

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

The equality comparison == can be used on object types, where it serves as a pointer comparison, much like in Java. It is a static error to compare an object and an array in this fashion. However, any value with class or array type can be compared to the special value `null`.

[2]The memory returned by `_I_alloc_i` is automatically initialized to zero, as are global variables in section `.bss`

7 Non-object-oriented extensions

7.1 Global variables

The lack of global variables in Iota$_9$ has been remedied in IX. A module can contain global variable declarations, such as:

```
center, 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 values. The length may be given either as an integer literal or as the name of a global variable.

Global variables can be declared in interfaces, e.g.:

```
// The point (0,0)
zero: Point
tenpoints: Point[]
```

7.2 Multidimensional arrays

Multidimensional arrays may be defined as global or local variables. Consider this code:

```
matrix: Point[2][3];
```

This acts similarly to the Java statement:

```
Point[][] matrix = new Point[2][3];
```

That is, it creates an array of length 2 whose elements are `Point` arrays of length 3. Therefore the expression `matrix(1)(2)` will be a legal array element. It may tempting to read the declaration as `matrix: (Point[2])[3]`, but this is incorrect.

As in Java, lengths need not be provided for all dimensions. The following declaration only creates the top-level array; its two elements of type `Point[]` are initialized to `null`.

```
twoarrays: Point[2][]; // similar to Java: new Point[2][]
```

In a multidimensional array definition, all dimensions that specify lengths must occur to the left of all dimensions that do not have lengths. This semantics may seem counterintuitive but it matches Java.

7.3 Assigning to tuple elements

In Iota$_9$, tuples could be used as arrays in an assignment statement, with the semantics that a new array was created and discarded. This was useless and confusing to programmers. In IX, such an assignment now imperatively updates the tuple itself. Bounds checking is still performed on the index being assigned to. For example, the following code updates the second component of the tuple `words` to contain the string `‘‘Thanks’’`, and the third component of `numbers` is changed from `30` to `3`:

```
    words: (int[], int[], int[]) = ("Please", "Thanks", "You're Welcome");
    words(1) = "Thank you";
    numbers: (int, int, int, int) = (1,2,30,4);
    numbers(2) = 3;
```

This feature enables small fixed-length arrays to be placed inside objects without the overhead of an indirection.

To make type-checking convenient, we introduce a new judgment $\Gamma \vdash e : \tau\ \text{lvalue}$, which means that e is an expression that can be assigned values of type τ (an *lvalue*). With this rule, we can rewrite the (ASSIGN) rule in a more generic form that encompasses the old rule and also the old (ARRASSIGN) rule:

$$\frac{\Gamma \vdash e_1 : \tau\ \text{lvalue} \quad \Gamma \vdash_u e_2 : \tau}{\Gamma \vdash e_1 = e_2 : \Gamma}\ (\textsc{Assign})$$

The lvalues are variables, array elements, and tuple elements:

$$\frac{\Gamma(x) = \texttt{var}\ \tau}{\Gamma \vdash x : \tau\ \text{lvalue}}\ (\textsc{VarLValue}) \qquad \frac{\Gamma \vdash e_1 : \tau\texttt{[]} \quad \Gamma \vdash_u e_2 : \texttt{int}}{\Gamma \vdash e_1\ e_2 : \tau\ \text{lvalue}}\ (\textsc{ArrLValue})$$

$$\frac{\Gamma \vdash e_1 : (\tau, \ldots, \tau)\ \text{lvalue} \quad \Gamma \vdash_u e_2 : \texttt{int}}{\Gamma \vdash e_1\ e_2 : \tau\ \text{lvalue}}\ (\textsc{TupLValue})$$

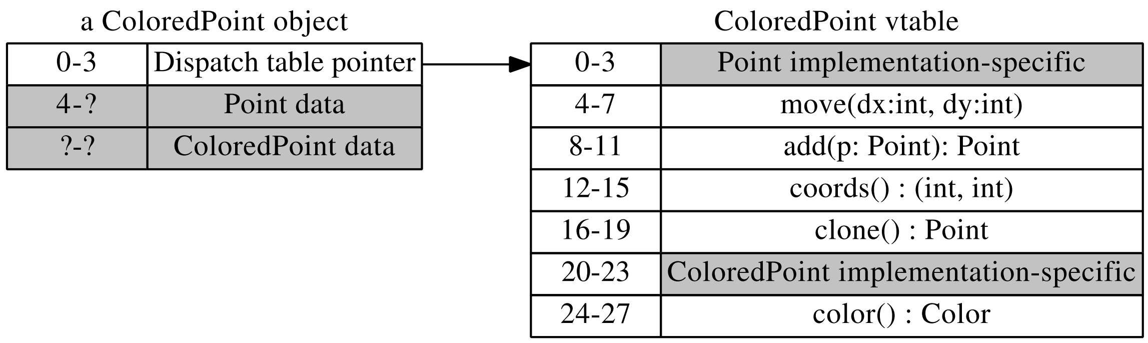
8 ABI

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

8.1 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 topmost class in 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

0-3	Dispatch table pointer
4-?	Point data
?-?	ColoredPoint data

ColoredPoint vtable

0-3	Point implementation-specific
4-7	move(dx:int, dy:int)
8-11	add(p: Point): Point
12-15	coords() : (int, int)
16-19	clone() : Point
20-23	ColoredPoint implementation-specific
24-27	color() : Color

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 tuples. Object references are passed to and returned to from functions, and stored in tuples and arrays, 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 tables.

8.2 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`.

8.3 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 vtable into its own.

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

The size variables are to be initially set to 0[3] to denote that the size information and the vtable 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 vtable information. When initializing its own vtable, 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 vtable. An implementation is expected to avoid computing object sizes and vtables 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.

9 Changes to original document

- Dec. 15: Field initialization expressions do not have to be supported.
- Dec. 9: Rules clarified on checking conformance between interfaces and modules, and between classes and superclasses.
- Dec. 8: Typing rules for assignment updated (Section 7.3)
- Dec. 7: Mangling specification for global variables.
- Dec. 7: Clarification: multidimensional array definitions are supported and work similarly to Java array creation.
- Dec. 6: Clarification: global `int` and `bool` variables can have constant initialization expressions
- Dec. 4: More fixes to code examples.
- Dec. 3: Another tuple assignment example, and a typo fix.
- Dec. 2: Various small fixes to code examples.

[3] An implementation may set them to the correct value if it is able to compute both the size and the vtable fully at compile time