Objects in Unicon

Sumant U. Tambe and Clinton Jeffery

Unicon Technical Report
December 10, 2003

Abstract

Objects in Unicon were supported using “special identifiers” in the equivalent procedural Icon program. This situation has been changed and Unicon runtime system now supports objects internally truly as objects. This technical report gives details of changes made in the system for this purpose.

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003

1. Introduction

Objects in Unicon are implemented using procedural Icon. The Unicon compiler translates an object-oriented Unicon program into an equivalent procedural Icon program. Icon does not support objects. Therefore, Unicon objects are represented as an Icon record with "special" identifiers in the record. There were a few undesirable things with these "special" identifiers. First, these identifiers used to consume 16 bytes of space per object in a typical Unicon installation. Second, these identifiers were user accessible. Third, the Unicon runtime system was tricked to implement the objects and it had little support for object's own internal representation.

This technical report documents the changes made in the Unicon system to solve the problems associated with the special identifiers.

2. Old structure of objects

An object-oriented Unicon program is translated into procedural Icon program by Unicon compiler. For every class in the source Unicon program, it generates 3 identifiers in the global namespace of the procedural Icon program.

- Record *__state (* replaced by classname)
- Record *__methods (* replaced by classname)
- And *__oprec global variable to hold the pointer to methods vector. (* replaced by classname)

For example:

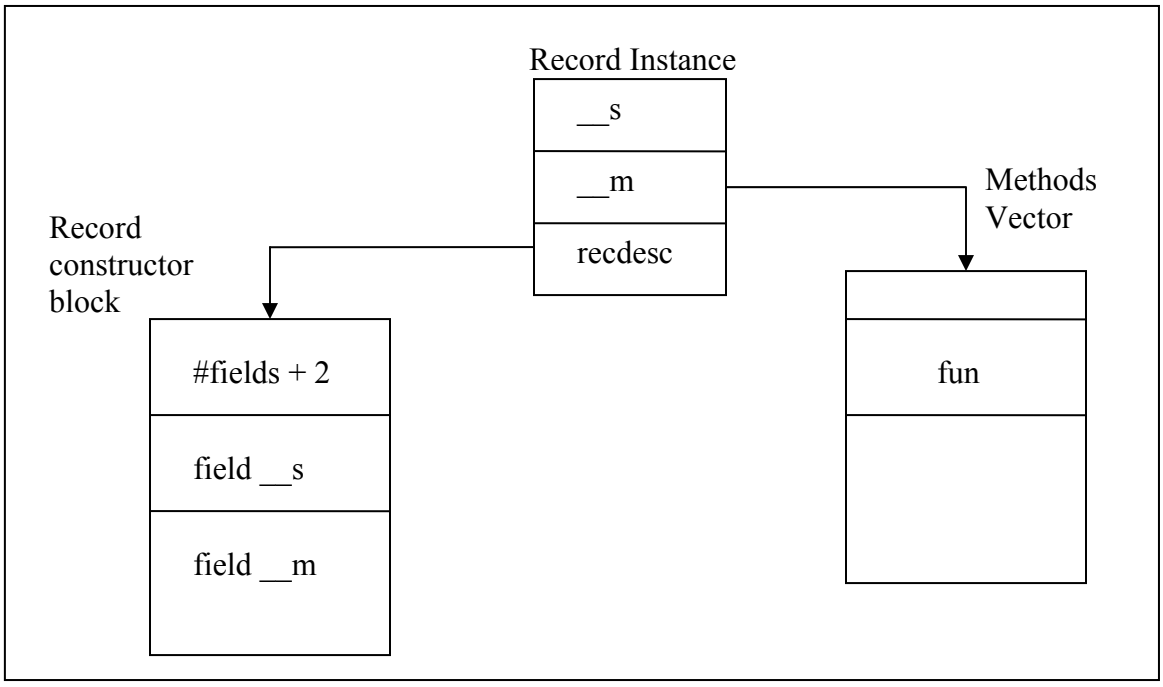
Unicon program with a class	Old equivalent procedural Icon program

class Myclass(val)
method fun()
write(val)
end
end
 	
procedure main()
c := Myclass(100)
c.fun()
end
	procedure Myclass_fun(self)
	write(self.val);
	end
	record Myclass__state(__s,__m,val)
	record Myclass__methods(fun)
	global Myclass__oprec
	procedure Myclass(val)
	local self,clone
	initial {
	if /Myclass__oprec then Myclassinitialize()
	}
	self := Myclass__state(&null,Myclass__oprec,val)
	self.__s := self
	return self
	end

	procedure Myclassinitialize()
	  initial
	Myclass__oprec:=Myclass__methods(Myclass_fun)
	end
	
	procedure main();
	  c := Myclass(100);
	  c.fun();
	end

As shown in the procedural Icon program, the Myclass__state record has 2 additional identifiers, __s and __m. The purpose of __s is to indicate that it is an object and not a normal Icon record. __m is used to hold the pointer to methods vector. These 2 special identifiers are replicated in every object of class Myclass.

Icon translator creates record descriptor for every record in the Icon program. Therefore, Myclass__state also has its corresponding record constructor block. There is a pointer from record instance to its corresponding record constructor block. This structure is represented diagrammatically below.



The diagram shown above is not a comprehensive layout of how Unicon object is represented in memory. It shows only the relevant part of the structure and fields are not in order. The record instance shows 2 additional fields in the record which are “special”.

As stated earlier, __s is just to indicate special nature of the record. (Hey, I am an object!) Record constructor block has descriptors of names of fields and integer showing number of fields and much more relevant information. Methods vector hold descriptors which ultimately point to the methods in icode.

Whenever an object's method is called, __m is used to locate the methods vector and method is invoked.

2. New structure of objects

Unicon compiler still generates 3 identifiers in the global namespace of the procedural Icon program. *__state, *__methods and *__oprec (* replaced by classname) But there is a change in the number of fields in *__state.

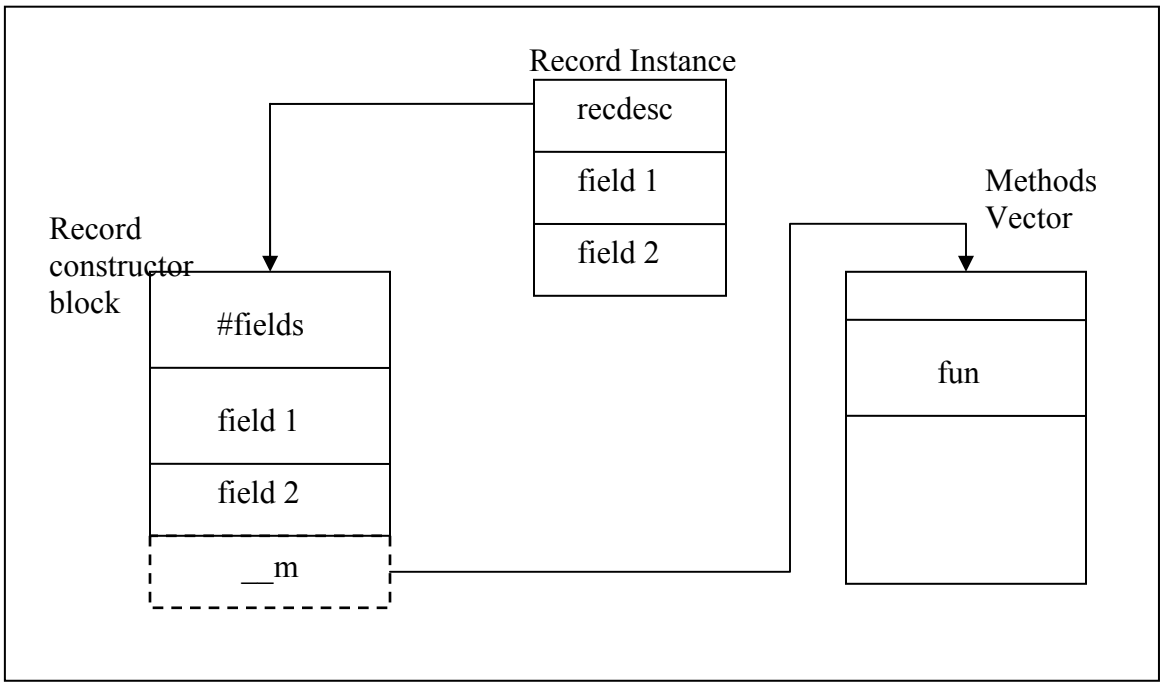
For example:

Unicon program with a class	**New equivalent procedural Icon program**

class Myclass(val)
method fun()
write(val)
end
end
 	
procedure main()
c := Myclass(100)
c.fun()
end
	procedure Myclass_m(self)
	write(self.val);
	end
	record Myclass__state(val,__m)
	record Myclass__methods(m)
	global Myclass__oprec
	procedure Myclass(val)
	local self,clone
	initial {
	if /Myclass__oprec then
	Myclassinitialize()
	}
	self :=
	Myclass__state(val,Myclass__oprec)
	return self
	end
	
	procedure Myclassinitialize()
	initial Myclass__oprec :=
	Myclass__methods(Myclass_m)
	end
	
	procedure main();
	local c;
	c := Myclass(10)
	c.m();
	end

As shown in the new procedural Icon program, the Myclass__state record has only one additional identifier: __m. __s has been completely removed from the generated procedural Icon program.

Although we can see __m in the generated procedural Icon program, it is not taken into account when icode file is loaded. The Icon translator and loader in the runtime system are modified to eliminate the __m from the record instance. This modification necessitates changes in internal representation of objects in Unicon icode file and runtime system. This new structure is represented diagrammatically below.



Again, the diagram shown above is not a comprehensive layout of new Unicon object representation in memory. It shows only the relevant part of the structure and fields are not in order. The record instance shows neither __s and __m. As usual, record constructor block has descriptors of names of fields and integer showing number of fields and other relevant information. The __m field is moved from record instance to the end of field list in record constructor block. It is shown dotted because although it is there in icode file and in memory, it is not counted in #fields of the record. The detailed description of how all the things are put together is given in next section. Methods vector hold descriptors which ultimately point to the methods in icode.

Whenever an object's method is called, recdesc field in every object is used to locate its corresponding record constructor block and __m located in record constructor block is used to locate the methods vector and method is invoked.

3. Modifications in detail

5 files in the Unicon implementation were modified to reflect the structural changes in code.

- lcode.c
- imain.r
- imisc.r
- invoke.r
- rmisc.r

Changes in structure:

The icode file generated by icont translator contains block for every record in the program. The block of an object is a special case. Although __s and __m are both removed from the object instance __m is still present at the end of record constructor block to hold the pointer to methods vector. The ndynam variable is used to indicate it as an object. When ndynam is -3, icont and iconx both understand that it is an object.

Thus record constructor has ndynam equal to -3 and __m descriptor at the end of it. __m is not counted in the number of fields. So number of fields is decremented before icode file is actually written. Block size is unchanged.

Additions in source files:

lcode.c file was modified to reflect the changes in the structure.

When loader reads icode file, it pays special attention to value -3 in the ndynam variable. It identifies that it is an object and therefore has __m at the end of all fields although #fields is decremented to avoid allocating memory for __m.

As described in earlier section, for object method invocation, __m in record constructor block is used. It is very necessary to initialize it before any method call takes place. This initialization is done before icode main() begins its execution. *initialize method of every class is called before main() which in fact allocates memory for methods vector for that class and initializes it. The pointer to methods vector is copied in *__oprec global variable. This can be seen in the generated Icon program. The loader then picks up the value from *__oprec and puts it in the __m field. Code for this was added in imain.r.

initilize method creates an instance of methods vector record for the class. In the original implementation of Unicon, methods vector instance is created when first instance of that class is created. New implementation creates methods vectors for all the classes in the program before main() begins its execution. This is achieved by calling initialize() methods. Op_Noop, Op_Invoke, and Op_Quit icode instructions are put in the instruction buffer and generator frame pointers(gfp) and expression frame pointers(efp) are pushed

on stack. When instruction buffer and stack is setup, descriptor of method to be called is pushed on stack. interp() is called just like it is called for execution of main(). initilize() method assignes the newly created method vector pointer to <classname>__oprec global variable. This new value is also copied into __m field of the record constructor block in later part of the code.

imisc.r, invoke.r and rmisc.r have comparatively few additions. It primarily involves comparing ndynam variable with -3 and taking appropriate actions for the object.

4. A note for Unicon users

An interesting feature has also been introduced in the language due to the changes in the internal representation. It is guaranteed that any code inside *initialize method of procedural Icon program shall be called before main() begins its execution. This feature can be utilized to grab "class level" resources prior to execution of main.

Another side effect is that, __m is no more a valid field and is considered reserved from now on. Any attempt to access this field will result in run-time error at least for now. Later on, Unicon compiler will be modified to report a semantic error if it encounters __m used as a field member.

5. Consequences

The most important benefit we get from this new structure is saving of memory space. Two descriptors (16 bytes) memory is saved per instance of class. Saving in memory space in this way often leads to less frequent calls to garbage collector and thereby improving efficiency of program.

Unicon runtime system now supports objects in its own right. Objects are no more "simulated" in procedural Icon where Icon runtime system knows nothing about objects.

Additional feature in Unicon language which allows programmer to call some part of the code even before execution of main begins.

6. Future work

Unicon compiler needs to be modified to report an error when programmer tries to use __m as a field in any class. Secondly, __s seems to be still lingering around in the context of inheritance where derived class needs to call super class's overridden method. Eliminating __s in inheritance context will be a good semester project.

7. References

"The Implementation of the Icon Programming Language" by Ralph E. Griswold and Madge T. Griswold.