Felix Objective C DSSL

John Skaller

October 23, 2020

1 Requirements

Felix is a high level programming language with a user defined grammar which generates C++ and has excellent facilities for binding to other programming languages. In particular Felix has a concept called a Domain Specific Sub-Language or DSSL, in which a combination of extensions to the grammar, Felix libraries, possible additions to the Felix run time library which is written in C++, and occasionally small modifications to the compiler, are used to extend the Felix language to provide specialised capabilities. Unlike conventional Domain Specific Languages or DSLs, DSSL code can be embedded in ordinary Felix code and can embed Felix code as well. This means the full capabilities of Felix are typically available within the DSSL.

A requirement has been raised to provide a binding to Apple APIs to facilitate development of MacOS and and iOS applications. These APIs are defined variously in three languages: C, Apple flavoured Objective C, and Swift.

Objective C is a pure extension of C, and the Apple system compiler is clang, which can compile Objective C as well as an extension that supports embedding Objective C in C++, the latter being called Objective C++.

This report documents the Objective C DSSL.

2 Primitive Types

Bindings are provided for a few basic Objective C types used in the Foundation framework. These are types originally provided by Apple for the NextStep environment.

These types are provided in the Felix class `Foundation` which is located in the library file `share/lib/std/apple/Foundation` with original source in `src/packages/apple.fdoc`.

The special type `SEL` is a special type used for method selectors. Felix equips it with a `str` function which extracts the name of the selector as a Felix string (which is in turn a binding to C++ string).

We provide a binding to Objective C most basic type `id`.

Bindings for the basic NextStep primitives:

1. NSObject is the base for most other types
2. NSString is a UCS2 encoded Unicode string
3. NSNumber is the basic number type

are provided. There is also a binding to `NSArray`.

Each of these types is defined in Felix as a pointer to the corresponding type in Objective C like this:

```
type NSString = "NSString*";
```

where the LHS is the Felix type and the RHS is the Objective C type.

3 Basic Subtyping

In Objective C most types are class types with subtyping relations. These are bound in Felix with `supertype` specifications, for example:

```
supertype NSObject : NSString = "$1";
```

which says NSString is a subtype of NSObject, and provides a type coercion which is the identity function. This allows the Felix subtyping system to respect Objective C subtyping, but leaves the actual coercions up to Objective C to implement. Class types in Objective C must be represented as pointers, that is, most types in Objective C are boxed.

4 Literals

Felix provides three literals with the same syntax used in Objective C. These are for NSString, NSNumber, and NSArray. The literals have Felix bindings of these types. The lexical binding is provided as an extension to the Felix grammar which has the capability of specifying user defined literals. Note there is a caveat, since each of these literals starts with the `@` character which is already used in Felix in the first column of a line, so these literals cannot be used in that location.

4.1 NSString

Literals are the same as in Objective C, for example:

```
    @"hello world"
```

4.2 NSNumber

Literals are the same as in Objective C, for example:

```
    @"123.4"
```

4.3 NSArray

The `NSArray` literal is a constructor which creates an Objective C `NSArray` object, however the array elements are Felix expressions. The client is responsible to ensure these expressions generated values of appropriate Objective C types. For example:

```
    @[@"hello",@123.4]
```

5 Code Literals

Felix has existing machinery to inject C++ code into its output. the features described below are standard Felix and are not part of the Objective C DSSL. A brief description is provided because the machinery is the primary way to binding to Objective C as well as to C and C++. It works because Objective C is a pure extension of C.

5.1 Floating Insertions

5.1.1 Header

A specification such as

```
header small_class_interface = c"""
@interface SmallClass: NSObject { }
- (int)get1977;
@end
""";
```

can be used to tag Objective C code and mark it for possible insertion into the C++ header file Felix generates.

5.1.2 Body

A specification such as:

```
body small_class_implementation = c"""
@implementation SmallClass
- (instancetype)init {
        self = [super init];
        return self;
}
- (int)get1977 {
        return 1977;
}
- (int)getsum: (int)toadd {
  return 1977 + toadd;
}

@end
""";
```

can be used to tag Objective C code and mark it for possible insertion into the primary C++ implementation file.

These constructions are called floating insertions because the quoted code floats to a fixed place in the generated C++ files.

There are two kinds of insertions: simple strings and templates.

5.1.3 String insertions

String insertions require the the quotation to use the `c` qualifier and the body of the quotation is inserted literally.

5.1.4 Template Insertions

Template insertions do not use the `c` qualifier, and allow type substitutions to be made at various points including those marked with the `@` character.

If an insertion template is used instead of a string insertion, then `@@` must be written to insert a single `@`.

Note also in string insertions in `*.flx* files an \verb`@+ in column 1 will work, but the same code in an `*.fdoc` file will not. (It's somewhat unfortunate that Felix and Objective C both use the `@` character as a leadin for special processing).

5.1.5 Dependencies

These specifications do not in themselves lead to injection of the relevant code. Instead, the insertion is conditional on the generation of a dependency. A dependency is created in two steps. First, a requires clause is attached to some other construction, most usually a type specification as illustrated here:

```
type small_class_instance_t = "void*"  requires
  small_class_interface,
  small_class_implementation
;
```

The effect is that if the type `small_class_instance_t` is actually used, then the required floating insertions will be injected in the output.

The second step of course is to actually use that type: if the type isn't used, no injection occurs, therefore the generated C++ code will only include code that is actually needed, avoiding clashes, ensuring the output is more readable, and optimising C++ compile times.

5.2 Bindings

Bindings define Felix entities in terms of C++. Types, functions, procedures, and constants can be bound. The `small_class_instance_t` binding shown above is an example of a type binding. Here is a function binding:

```
fun make_small_class_instance:
  1 -> small_class_instance_t
=
  "[[SmallClass alloc] init]"
;
```

This gives the name and type of the function in Felix but defines it in C++, in this case, extended with Objective C.

5.3 Direct Insertions

Felix also provides a way to insert code immediately where specified. There are two constructions, statement and expression insertions.

5.4 Statement insertions

The simplest direct insertion is for statements, for example:

```
cstmt 'printf("Hello World\n");';
```

which can be used to insert the quoted code directly in place, the code must be zero, one, or more C++ statements.

5.4.1 Expression insertions

Slightly more complex, expressions, which are typically parametric:

```
var result2 =
  cexpr[int] "[$1 get1977]" small_class_instance endcexpr
;
```

The type of a `cexpr` must be specified, in this case `int`. The quoted code here is a template because the `c` prefix was not used, the `$1` marks the insertion point for the first component of the argument tuple. The argument here is `small_class_instance` which is translated to a C++ expression and the resulting code then replaces the `$1`. This is the only reliable way to refer to a Felix entity from inserted C++ code.

6 Interface Bindings

Felix provides a special syntax for binding Objective C class interfaces. Here is an example:

```
objc-bind
 @interface SmallClass : Super <hasDescription, Cpy>
  {
     int x;
     y: int;
  }
   +(SmallClass)alloc;
   -(instancetype)init;
   -(int)get1977;
   -(int)getsum:(int);
   @property int z;
   @property (readonly) q:int;
 @end
 requires
   small_class_interface,
   small_class_implementation,
   package "foundation",
   package "objc"
;
```

The `objc-bind` statement allows binding class interfaces and protocols. The example above shows the following supported features.

6.1 Class

The name of the class being interfaced is specified. The effect is to generate a primitive type like this:

```
type SmallClass = "SmallClass*";
```

6.2 SuperClass

A super class may be specified, if omitted it defaults to `NSObject`. The effect is to generate a coercion like this:

```
supertype Super : SmallClass = "$1";
```

6.3 Protocol Conformance

The protocols a class conforms to can be listed in angle brackets with separating comma. The effect is is to generate a supertypes coercions:

```
supertype hasDescription: SmallClass = "$1";
supertype Cpy: SmallClass = "$1";
```

6.4 Instance variables

Instance variables, or *ivars* can be specified as a sequence of instance variable declarations inside curly braces. Two forms are allowed, Felix form and C form.

C form specifies the type and then the variable name, whilst Felix form specifies the variable name with the type following a colon.

The C form will always work if the type name is a single identifier, and may work in other cases. The Felix form handles all types.

Note that the types specified are Felix types with Felix syntax.

The effect is to generate a projection function for each instance variable which are defined by Objective C bindings. For example:

```
fun x: SmallClass -> &int = "&($1->x)";
fun y: SmallClass -> &int = "&($1->y)";
```

This is similar to code generated for a `cstruct` construction except that the argument is already a pointer underneath. This allows access and modification with standard Felix syntax, for example:

```
sc . x <- sc -> y; // assign y to x
```

6.5 Class Methods

Class method specifications follow Objective C syntax, for example:

```
+(SmallClass)alloc;
```

Note that the types specified in parentheses are Felix types, not Objective C types.

Such methods can be invoked by concatenating the class name with the method tag names, using the single quote mark ' as a separator or terminator.

6.6 Instance Methods

Instance method specifications follow Objective C syntax, for example:

```
-(instancetype)init;
-(int)get1977;
-(int)getsum:(int);
```

Such methods are invoked using the concatenation of the method tag names with single quote mark ' after each tag with a subsequent parameter.

```
var sc : SmallClass = #SmallClass'alloc.init;
println$ "Get: " + (sc.get1977) . str ;
println$ "Add: " + (sc.getsum' 42) . str ;
```

If an instance method has no parmeter other than the object, a function is generated like this:

```
fun get1933: SmallClass -> int = "[$1 get1977]";
```

However, is there are arguments, a higher order function is generated whose first argument is the object and subsequent argument is a tuple consisting of all the non-object arguments. For example:

```
fun getsum' (o: SmallClass) (a:int) =
  cexpr [int] "[$1 getsum:$2]" (o,a) endcexpr
;
```

The effect of the currying allows a closure over the object to be formed, for example:

```
var cls = sc.getsum'; // form closure
println$ cls.42;      // apply closure
```

However the primary reason for using this formulation is that all instance methods have a uniform representation, taking a single argument, of the object. This turns out be essential.

Note that varargs are not currently supported for methods due to the lack of a motivation in the form of a test case.

6.7 Properties

Felix provides Objective C like property syntax with the effect of generating getter method and, if the `readonly` attribute is not specified a setter method. The getter method name is the property name, the setter is named `set` followed by the property name.

Property syntax allows both the C form and the Felix form of variable and type specification. Note again the types are always Felix types.

6.8 Covariance

Objective C has a special syntax to specify a covariant return type: the return type is named `instancetype`. If a covariant return is specified, it stands for the type of the object to which the method is applied.

In particular, a method of a superclass or protocol with a covariant return returns the static type of the object to which it is applied rather than the type of the class in which it is defined.

This is implemented in the Felix compiler by inspecting and type of the object argument and replacing the `instancetype` specification with it.

7 Protocols

A protocol specification is a cut down verion of a class interface and generates exactly the same code. Here is an example:

```
objc-bind
  @protocol hasDescription
   -(NSString)description;
  @end
  requires package "foundation", package "objc"
;
```

Note: due to a bug it is currently necessary to specify a requirement for package "objc" for all objc-bind statements.

8 Method Overloading

Within an interface or protocol names of methods in Objective C must be unique. Because of this, and the uniform representation of instance methods, overloading on the object type alone is sufficient to invoke the correct method.

This is true even if the method is specified in the superclass or a protocol due to subtyping. It is important to understand Felix does not construct any kind of container for a class interface binding: there is no record of the methods of the class other than the subset of entries in the symbol table with the object type as domain.

Felix method selection is rigidly statically typed, it is not possible to call a method on an inappropriate object. Therefore it should be possible for clang compiler to optimise all dynamic method lookup away for all methods specified in interface or protocol bindings.

9 Subtyping considerations

Felix has general polymorphic subtyping for general type constuctors and uses a database of user specified coercions for nominally type primitives. Subtyping is transitive and coercions are automatically composed with one of the shortest paths in the graph of coercions being arbitrarily chosen by the compiler.

10 Intersection Types

The Felix compiler was modified to support intersection types. The support is intended to be complete for monomorphic primitives with subtyping coercions specified, directly or indirectly, by supertype definitions. If `A` and `B` are types then their intersection is specifed by

```
A & B
```

10.1 Subtyping rules

The subtyping rules for an intersection are as follows: if X is a subtype of both A and B, it is a subtype of their intersection, and, the intersection of A and B is a subtype of both A and B.1

Intersection is symmetric and associative, and has the type `void` as a zero. The empty intersection is the universal type `any`. Note this is *not* the universal type `id` of Objective C.

Intersections can easily be understood by considering types as sets, then an intersection of types is just the intersection of sets.

However this requires very careful understanding of the meaning of a protocol: a protocol specification places a *lower bound* on the methods of an object, so the set of objects conforming to a protocol consists of all object with *at least* the specified methods. Therefore every object with at least those methods is of the protocol type.

This means than an intersection restricts the set of such objects to be those which have all the methods specified by both intersectands: this is *more methods*, but *less conforming objects*.

10.2 Utility

Intersections types are *mandatory* for writing generic functions. Here is an example:

```
fun dc (x: hasDescription & Cpy) => x.cpy.description.str;
```

Notice that this function will work for any object `x` obeying both the protocols specified in the intersection, which ensures it has both the `cpy` method and the `description` method.

It is worth noting

Felix can do Objective C better than Objective C

since the latter does not have structural intersection types.

You may wonder how this works. The detail is as follows: the `cpy` method takes an object parameter of type `Cpy`. Since `Cpy & hasDescription` is a subtype of `Cpy`, there is a coercion to `Cpy` which allows an object of the intersection type to be used where merely `Cpy` is required, as in the case of the `cpy` method.

Similarly, there is a coercion of the intersection to `hasDescription`, so the `description` method can be called with the object after applying the coercion.

The coercions for intersections cannot be specified by the user, and they are not generated by the compiler using the `supertype` construction either: in fact the coercion is taken to be the identity function and is therefore never generated! In fact, the coercion is not the identity, but the application is left up to the Objective C compiler, since it already handles this.

Therefore in some sense, intersection in Objective C bindings in Felix come for free. The only work to be done is ensure type correctness by applying the subtyping rules as a type check. The type coercion applied is erased.

Note: the full set of subtyping judgements are not yet implemented. As before, a use case is required as a test to motivate completion.