Type Systems

John Skaller

October 8, 2020

Contents

Part I

Subtyping Kernel

Chapter 1

Motivation

Felix provides a domain specific sublanguage (DSSL) for binding Objective C APIs. Consider the following simple ObjC class:

```
@interface SmallClass: NSObject {}
- (int)get1977;
@end

@implementation SmallClass
- (instancetype)init {
        self = [super init];
        return self;
}
- (int)get1977 { return 1977; }
@end
```

To lift this code verbatim into Felix we have to bypass the parser by creating text inclusions. These inclusions are emitted verbatim inside one or more of the compiler emitted C++ files.

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

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
""";
```

We can now write a binding to lift the API into Felix:

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

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

fun get1977 : small_class_instance_t -> int = "[$1 get1977]";

var small_class_instance = make_small_class_instance();
var result = get1977 small_class_instance;
println$ "Felix ran objc to get " + result.str;
```

However this is the hard way! here's the easy way, using the objC DSSL:

```
  objc-bind
    @interface small_class
    +(instancetype) alloc;
    -(instancetype) init;
    -(int) get1977;
    -(int) getsum: (int);
    @end
  ;
  println$ "NESTED " + (small_class'alloc.init.getsum' 44).str;
```

ObjC has a construction which specifies a constraint that an object respond to at least the nominated set of messages. Here is an example:

```
objc-bind
  @protocol hasDescription
   -(NSString)description;
  @end
;
```

and another:

```
objc-bind
  @protocol Cpy
  -(instancetype)cpy;
  -(instancetype)cpywithmsg:(NSString);
  @end
;
```

Here is a class interface that uses these protocols:

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

I must explain that what you see above is Felix code. In particular, the protocols above have nothing to do with any Objective C protocols.

Sure, they look objective C'ish. But actually Felix makes a monomorphic nominal primitive type for each protocol and does not associate it with any methods! Instead, it defines a set of methods overloaded on the type of the first argument, which is always present, namely a pointer to an object instance.

```
type hasDescription = "void*";

fun description: hasDescription -> NSString = "[$1 $2]";
```

and

```
type Cpy = "void*";

fun cpy: Cpy -> Cpy = "[$1 cpy]";
fun cpywithmsg (obj:Cpy) (msg:NSString) =>
  cexpr "[cpy: $1 msg: $2]" (obj,msg) endcexpr
;
```

This means it is possible to now write functions which accept any object conforming to a protocol, for example given a pointer to a `SmallClass` instance the following function will work correctly:

```
fun require_description(x:hasDescription) => (x.description).str;
println$ sc.require_description;
```

Now, suppose we have a protocol A, and another B, and we want one that has all the methods of both:

```
objc-bind @protocol C<A,B> @end;
```

The way Felix handles protocols .. and superclasses .. is universally done with an opaque type and one more more subtyping coercions. Since ObjC objects are universally machine pointers, C `void*` is a suitable representation. Emitted subtyping coercions look like this:

```
  supertype C : A = "$1"; // A < C, A -> C
  supertype C : B = "$1"; // B < C, B -> C
```

1.1 Intersection Types Required

And now the real crux of it. We want an *anonymous* type constructor, and that is the intersection operator:

```
  A & B
```

Intersection is symmetric and associative, and an empty intersection is isomorphic to the universal type. There is a normal form for intersections, namely a list of types to intersect, none of which are, or contain, directly or indirectly, any intersection. The intersection is void if any component is void.

1.2 Subtyping Judgements

1.2.1 Primitives

Subtyping judgements between primitives can be made easily by inspecting the graph of coercions, and finding, or failing to find, a path between two types.

1.2.2 Intersections

With the introduction of intersection types, we need two extra rules:

1. a type X is a subtype of A & B if is a subtype of both A and B, and,
2. A & B is a subtype of Y if both A and B are subtypes of Y.

With our power set the rule reads:

1. a type X is a subset of $A \cap B$ if is a subset of both A and B, and,
2. $A \cap B$ is a subset of Y if both A and B are subset of Y.

1.2.3 Pair Judgement

It is worth unravelling the rule for making this judgement:

```
    A & B < C & D
```

There are two ways:

1. Apply rule 1 then rule 2 twice:
   (a) $A\&B < C\&D$
   (b) $A\&B < C$ and $A\&B < D$
   (c) $A < C$ and $B < C$ and $A < D$ and $B < D$
2. Apply rule 2 then rule 1 twice
   (a) $A\&B < C\&D$
   (b) $A < C\&D$ and $B < C\&D$
   (c) $A < C$ and $A < D$ and $B < D$ and $B < D$

Generalising to subtyping n intersections shows the solution can always be obtained with a quadratic number of primitive subtyping judgements.

1.2.4 Unification

The unification engine can implement both rules also since the types are monomorphic the unification makes no contribution to the most general unifier the algorithm must return.

1.2.5 Application Binding

When an overload is successful an application term must be bound, at which time a mismatch between the function parameter type and the argument type will be discovered. Since unification succeeded or we would not reach this point, the path corresponding to the subtyping judgment traces out at least one sequence of coercions which could be applied, and the binder inserts a type coercion.

Note that these coercion terms are purely type coercions supported by an existential proof a composite coercion can be found at a later time. We also defer the examination of the question: what happens if there is more than one path tracing out distinct composite coercions?

1.2.6 Overloading

This leaves the machinery missing a description of how the right methods are found. ObjC methods are C functions so they exist in global scope in ObjC, so Felix puts them there too. This means they can always be found, provided they're actually defined of course.

But now, overloading works by finding all the methods with the same name and using subtyping judgements to find all candidates. If the argument type is a subtype of the parameter type the function is a candidate.

The algorithm now attempts to find a best fit as follows: the first candidate is added to the set of best fits.

1. Now for each subsequent candidate, if its parameter type is a proper subtype of the parameter of any best fit function, that function is thrown out of the best fit set, and finally the current candidate is added.
2. If the candidate is a proper supertype of any function in the best fit set, the candidate is ignored.
3. Otherwise the candidate is added to the best fit set because it is either incomparable with all of them, or equivalent to at least one of them.

When we run out of candidates if the best fit set has only one element, that element is the best fit. Otherwise if there are at least two elements, there is at least two good fits neither of which is better than the other.

1.3 Why is this fabulous?

Now the *key point* in this design is that the whole of the method selection system is done by compile time static type analysis, so roughly if it compiles it must run without any possibility of a missing method or nasty null pointer arising from the method binding. Obviously if any objective C methods are called they have to be correctly bound to maintain the guarrantee.

1.4 Ok what's the catch?

But there is a caveat, as always.

The machinery will not work in the general case if the user defined set of coercions are not constrained.

It is vital to understand that because subtyping coercions implicit:

they are perfomed silently and so *we cannot allow unexpected behaviour to arise* because *there is no locally visible culprit to pin the blame on.*

The rest of this paper therefore is primarily concerned with establishing, in the abstract, what the constraints on supertype definitions should be so as to obey the *principle of least surprise.*

Chapter 2

Kernel Notions

2.1 Set Inclusions

Let U be some finite set and let C be the power set of U, the set of all subsets of U. Then C is a category with an arrow $A \to B$ if $A \subseteq B$. We can provide two maps $C \times C \to C$ called intersection and union with the usual binary operators $\cap$ and $\cup$ thereby constructing a lattice. These maps can be shown to be bifunctors.

An concretised version of this kernel abstraction replaces the subtyping judgment by set inclusion arrows with a witness function which provides an actual embedding.

2.2 Abstract Subtyping Kernel

An abstract subtyping kernel is the interface of an implementation. It has the same basic structure as the set inclusion lattice. However the concretised version, which uses actual functions, requires an strong constraint: these functions must be monic.

2.3 Generating Graph

We will specify the subtyping kernel as the category generated by a finite graph we present. In Felix notation we will write for example:

```
    type int = "int";
    type long = "long";
    type A= "A";
    type B= "B";
```

to specify the vertices of the graph as primitive monomorphic types. The code on the RHS is the representation in C++.

The edges of the graph are specified like:

```
    supertype long : int = "(long)$1";
    supertype B : A = "(B)$1";
```

which specifies the second type is a subtype of the first and provides an implementation called a coercion. The two categories generated by this graph consist of the subtyping judgement category and the more concrete coercion category which consists of all compositions of coercions.

It is important to note that no constraint is imposed on the set of coercions provided so that it is possible for the graph to be cyclic. In this case two primitives may be judged equivalent and can be interconverted using any path without loss of information: the objects are isomorphic and the coercion paths are isomorphisms.

There is a cruicial constraint on the representations and coercions functions: the functions must be monic. This causes a serious problem as we shall soon see but the requirement is essential to ensure coherence. We have already met one of the reasons: to be able to interconvert between any set of equivalent types the coercions must be isomorphisms, and this is ensured by the fact that all coercions are monic.

Saunders MacLean says an arrow $m : a \to b$ is monic in a category C if when for any two parallel arrows $f, g : d \to a$, we have that $f \odot m = g \odot m$ implies $f = g$. Note that we are using reverse composition in OO style, so that above the first function is executed first, and the second applied to its result.

Intersection and union types are provided with binary operators like:

```
    a & b \| c
```

we note both these operators are symmetric and associative. The type 0 or `void` is an unit of the union and zero of the intersection, whilst the universal type is called `any` and is the unit of the intersection and zero of the union. An empty intersection is universal, whilst an empty union void.

In order to ensure coherence, no representation can be specified for intersections and unions, instead it is generated.

For the subtyping judgement category the judgements follow automatically from the set inclusion model: an intersection of two types is a subtype of each of them, any two types are subtype of their union.

However we need to be able to construct the coercions.