A Sample of Mixins in the mix-et Language

Max Jenkins

April 26th, 2024

Introduction

There are many different notions of inheritance that have large conceptual differences[1]. Functionally, the use of mixins creates very simple code re-use, similar to inheritance. mix-et attempts to provide an environment to use mixins not only for their unique code re-use, but as a generalization of other forms of inheritance via mixin composition.

Overvew of mix-et

mix-et is statically typed, purely functional prgramming language which borrows its syntax from Racket. As the name suggests, mix-et's main feature is its object-oriented design(OOD) constructs, which revolve around mixins. Due to the focus on Mixins, normal OOD features such as basic single inheritance and interfaces have been done away with. Instead, mix-et aims to generalize both these functionalities via mixin composition.
Programs in mix-et are defined as follows:

```
(mixes*)
(classes*)
(p)
```

where `mixes*` and `classes*` are 0 or more mixin and class declarations respectively, and `p` is the program body. At the time of writing this paper, mix-et supports only four primitive types:

- Integers (`N`)
- Strings (`String`)
- Booleans (`B`)
- Self (`Self`, used in method definitions)

Types for variables must be declared by the user using the syntax `(x : T)`, where `x` is a variable, and `T` is the type of `x`. This must be declared not only in lambdas, but in a method's arguments as well. mix-et not only borrows its syntax from Racket, but also borrows many functions from Racket. mix-et uses three functions for its objects:

```
(new c args*)
```

where c is the class name, and args is 0 or more arguments. Similarly, field access "`/`" and method calls "`send`" are defined as follows:

```
(/ e f)

(send e g es*)
```

where `e` is an object, `f` and `g` are field and function names respectively, and `es*` is 0 or more arguments for `g`. All methods defined in classes have a `self` as a required first argument, with the `Self` type. This argument is implicit, and doesn't need to be supplied by the user.

Objects in mix-et respect the functional nature of the language and remain immutable. Something to note thus far is that there are no access modifiers in mix-et, so any field or method can be thought of as public.

Motivation

First, an important question: why do we even want to look at a programming language with mixins? To answer this question, we will look at a description of mixins provided in Bracha's *Mixin-Based Inheritance*.

In this paper, Bracha provides three seemingly unconnected systems of inheritance:

- Smalltalk's inheritance, in which child classes are given priority for method use, and can re-define inherited methods at will,
- Beta's inheritance, in which the parent class's definition of a method must always be used, and child classes may add functionalty, and
- CLOS's inheritance, which supports multiple inheritance, and forces linearization of inhertied classes.

While these methods of inheritance seem unrelated, Bracha shows that through the use of mixins, we can generalize all three of these cases to just one case: mixin composition. So, by using mixins, we can express many different kinds of inheritance, which is where the use of mixins in mix-et comes in.

Mixins and Classes

To begin looking at the implementation of mixins in mix-et, we need to discuss mixins. In general, mixins can be thought of as a generalization of classes in an inheritance chain. Additionally, mixins provide us with a simple way to use multiple inheritance[2].

In practice, a mixin appears to be somewhat like a Java interface or abstract class, as it wouldn't make sense to instantiate a mixin. Instead, we will mix them into other mixins or classes. This way, we can gain the benefits of code re-use seen in inheritance, and lends itself well to function and mixin composition. By using mixins, we get the general code re-use and abstraction that inheritance provides, as well as placing more importance on building compositional code. Currently, mix-et doesn't support any override feature, but this can be easily accomplished with a rename or re-define feature[3].

The syntax for classes and mixins are dual to one another in mix-et, reflecting the generalization of classes to mixins. The definitions are as follows:

```
(class cvar (fields fs*) (mix mixins*) (methods*))

(mixin mvar (fields fs*) (mix mixins*) (methods*))
```

where cvar and mvar are class and mixin names respectively, fs* is 0 or more type-annotated field names, mixins* is the list of 0 or more mixins to be "mixed in" to the class, and methods* is 0 or more method definitions. Method definitions are of the form:

(method (m-name self : T_{self} args*) : T_{return} (body))

where m-name is the name of the method, T_{self} is either Self or the class type, args is 0 or more arguments with type annotations, T_{return} is the methods return type, and body is the method's body.

Class and mixin declarations are rigid in their structure: the `fields` keyword must always appear as the tag for the first list in classes and mixins, even if they don't contain any fields. This applies to the next required declaration, `mix` as well. The list of methods is slightly different, as if no methods are declared, the empty list must be put in place of a list of method declarations to declare that the class/mixin has none.

Classes and mixins use different tags in their definitions to ensure that the user is unable to create an instance of a mixin. Mixins are mixed in linear order, and class instantiation takes arguments in this order as well, favoring class arguments, then mixin ones. So, the program:

```
(mixin mailable
       (fields zip-code : String)
       (mix)
       ((method (mail self : Self name : String to-address : String) : String
       (++ "Mailed " (++ name  (++ " at " to-address))))))
(mixin place
       (fields population : String)
       (mix)
       ())

(class Town
       (fields name : String)
       (mix mailable place)
       ())
(let ((my-town (new Town "foobar-town" "12345" "8")))
     (String->N (/ my-town population)))
```

will instantiate a new object of Town class with name = "foobar town", zip-code = "12345", and population = "8". This program will of course return the number 8. Additionally, this example shows a very simple example of mixin composition used as a generalization of inheritance.

Typechecking

Due to the nature of mixins and the ability to have multiple mixes, there are many possible errors that can arise to make programs that utilize mixins not behave as expected. Thus, a robust type-checker is necessary to weed out nonsensical programs.

Several test cases have been carefully selected to highlight programs that the typechecker is meant to reject. One such example is as follows:

```
(mixin m1
       (fields f1 : N f2 : N)
       (mix)
       ((method (m1 self : m1 N : N a1 : N a1 : N) : N
 (+ 1 2))))
```

This mixin definition shouldn't be allowed, as it has two arguments `a1` in `m1`. So, the typechecker will reject it and print this error: `duplicate-argument: argument a1 has been previously defined in m1`.

On a similar note, the program:

```
(mixin m1
       (fields f1 : N f2 : N)
       (mix)
       ((method (m1 self : Self) : N
 (+ 1 2))))
(class c1
(fields f1 : B)
(mix m1)
())
(+ 5 3)
```

will also be stopped by the typechecker, as the user attempts to mix `m1` into `c1`, but both contain a field named `f1`. So, the typechecker gives the error message `duplicate-fields-check: duplicate field f1 in c1`. Now, for some more interesting examples. The program:

```
(mixin m1
       (fields f1 : N)
       (mix m1)
       ())
(+ 5 3)
```

shouldn't make it through the typechecker, as it recursively tries to mix itself. This example highlights a very particular error raised by the typechecker: `mixin-check: m1 cannot mix itself`. Without checking for this case, mix-et will try to mix `m1` forever, thus it throws an error. What would happen if we try to make a mutual mix, i.e. a circular one?

```
(mixin m1
 (fields f1 : N)
 (mix m3)
 ())
(mixin m2
 (fields f2 : N)
 (mix m1)
 ())
(mixin m3
 (fields)
 (mix m2)
 ())
 (class c1
 (mix m2)
 (fields)
 ())
```

The so-called "circular mixing"(seen in figure 1) test is an obvious problem. This chain of mixins violates our rule that mixins and classes can only mix previously defined mixins. Thus, we get the error: `mixin-check: mixin m3 not yet defined in m1`.

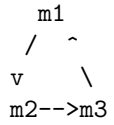

Figure 1: Circular Mixing Example

Consider this program as well:

```
(mixin m1
        (fields f1 : N)
      (mix)
      ())
(mixin m2
        (fields f2 : N)
      (mix m1)
      ())
(mixin m3
        (fields f3 : N)
      (mix m1)
      ())
(class c1
        (fields f4 : N)
        (mix m2 m3)
```

Figure 2: Diamond Mixing Example

```
  ())
(+ 1 2)
```

This "diamond mixing" (illustrated in figure 2 test is in a similar vein to the previous test, but it attempts to mix `m2` and `m3` into `c1`. The problem shows up when we examine the mixes of `m2` and `m3`: they both mix `m1`, so `m1` would be mixed twice in `c1`. Due to the structure of our classes and mixins, the typechecker will first discover the problem when checking the fields that c1 is receiving from its mixins. Therefore, we get the error message: `duplicate-fields-check: duplicate field f1 in c1`.

Conclusion

mix-et provides an OOD framework that highlights the use of mixins as the only form of inheritance. This provides the user with ample opportunity for code re-use as well as a thorough typechecker to provide the user with feedback on many common problems that can arise when using mixins. With these tools, mixins can be studied in depth as a proper inheritance system without the bloat of other object-oriented features.

References

[1] A. Taivaalsari, "On the notion of inheritance," *ACM*, pp. 483–479, 1996.

[2] G. Bracha and W. Cook, "Mixin-based inheritance," *ACM*, 1990.

[3] V. Bono, A. Patel, and V. Shmatikov, "A core calculus of classes and mixins," *ECOOP*, pp. 43–66, 1999.