Conference Slides

April 15, 2013

Lambda: Beyond The Basics

Luc Duponcheel (ImagineJ)

Function

Function

Cbn

```
// Cbn<Z> zName = ...;
// Z zValue = zName._();

// in what follows we name cbn parameters like cbv ones

public interface Cbn<Z> {
 Z _();
}
```

Block

```
// Block block = ...;
// block._();

public interface Block {
 void _();
}
```

Function

```
// Z z = ...;
// Function<Z, Y> z2y = ...;
// Y y = z2y._(z);

public interface Function<Z, Y> {
 Y _(Z z);
}
```

FunctionStatics

```
public class FunctionStatics {
 public static <Z>
 Z identity(Z z) {
  return z;
 }
}
```

BiFunction

```
// Z z = ...;
// Y y = ...;
// BiFunction<Z, Y, X> zny2x = ...;
// X x = zny2x._(z, y);

public interface BiFunction<Z, Y, X> {
 X _(Z z, Y y);
}
```

Ref

```
public class Ref<Z> {
 private Z value;

 public Z deref() {
  return value;
 }

 public void assign(Z value) {
  this.value = value;
 }
}
```

while (CbnApp)

```
private static Function<Block, Block> _while(
 Cbn<Boolean> cond
) {
 return
  block ->
   () -> {
    if (cond._()) {
     block._();
     _while(cond)._(block)._();
    }
    ;
   };
}
```

example (CbnApp)

```
// before 0
// after 10

public static void main(String[] args) {
 Ref<Integer> integerRef = new Ref<>();
 integerRef.assign(0);
 System.out.println("before " + integerRef.deref());
 _while(() ->
  (integerRef.deref() < 10))._(() -> {
  integerRef.assign(integerRef.deref() + 1);
 })._();
 System.out.println("after " + integerRef.deref());
}
```

Option

Option

Option

```
public interface Option<Z> {
```

Some

```
final public class Some<Z>
 implements Option<Z> {

 final Z value;
```

None

```
final public class None<Z>
 implements Option<Z> {
```

fold (Option)

```
// abstract method
// follows the structure of Option

public <Y>
Y fold(
 final Function<Z, Y> z2y,
 final Cbn<Y> y
);
```

fold (Some)

```
// uses z2y

@Override
public <Y>
Y fold(
 final Function<Z, Y> z2y,
 final Cbn<Y> y
) {
 return
  z2y._(value);
}
```

fold (None)

```
// uses y

@Override
public <Y>
Y fold(
 final Function<Z, Y> z2y,
 final Cbn<Y> y
) {
 return
  y._();
}
```

some (OptionStatics)

```
// specific factory method
// not part of generic DSL

public static <Z>
Option<Z> some(
 final Z value
) {
 return
  new Some<>(value);
}
```

none (OptionStatics)

```
// specific factory method
// not part of generic DSL

public static <Z>
Option<Z> none(
) {
 return
  new None<Z>();
}
```

identity (Option)

```
default public Option<Z> identity() {
 return
  fold(
   OptionStatics::some,
   OptionStatics::none
  );
}
```

length (Option)

```
default public int length() {
 return
  fold(
   z -> 1,
   () -> 0
  );
}
```

bindM (Option)

```
// binding is a fundamental
// (functional) programming concept

default public <Y>
Option<Y> bindM(
 final Function<Z, Option<Y>> z2oy
) {
 return
  fold(
   z2oy,
   OptionStatics::none
  );
}
```

one (OptionStatics)

```
// multiplicative method

public static <Z>
Option<Z> one(
 final Z z
) {
 return
  some(z);
}
```

bindF (Option)

```
// typically used at the end of a binding chain

default public <Y>
Option<Y> bindF(
 final Function<Z, Y> z2y
) {
 return
  bindM(z ->
   one(z2y._(z))
  );
}
```

bindA (Option)

```
// multiplicative method
// note: z is only used within the scope of bindF

default public <Y>
Option<Y> bindA(
 final Option<Function<Z, Y>> o_z2y
) {
 return
  bindM(z ->
   o_z2y.bindF(z2y ->
    z2y._(z)
   )
  );
}
```

join (OptionStatics)

```
public static <Z>
Function<Option<Option<Z>>, Option<Z>> join(
) {
 return
  ooz ->
   ooz.bindM(oz ->
    oz
   );
}
```

choice (Option)

```
default public Option<Z> choice(
 final Function<Z, Boolean> z2b,
 final Function<Z, Option<Z>> t_z2oz,
 final Function<Z, Option<Z>> f_z2oz
) {
 return
  bindM(z ->
   z2b._(z)
    ? t_z2oz._(z)
    : f_z2oz._(z)
  );
}
```

example01 (OptionApp)

```
// example01 = one(abc)

Option<String> example01 =
 one("a").bindM(a ->
  one("b").bindM(b ->
   one("c").bindM(c ->
    one(a + b + c)
   )
  )
 );
```

example02 (OptionApp)

```
// example02 = one(abc)

Option<String> example02 =
 one("a").bindM(a ->
  one("b").bindM(b ->
   one("c").bindF(c ->
    a + b + c
   )
  )
 );
```

example03 (OptionApp)

```
// example03 = one(abc)

Option<String> example03 =
 one("a").bindA(
  one("b").bindA(
   one("c").bindF(c -> b -> a ->
    a + b + c
   )
  )
 );
```

example06 (OptionApp)

```
// example06 = one(c)

Option<String> example06 =
 one("a").choice(a ->
  a.equals("b"),
  a -> one("b"),
  a -> one("c")
 );
```

zero (OptionStatics)

```
// additive method

public static <Z>
Option<Z> zero(
) {
 return
  none();
}
```

example04 (OptionApp)

```
// example04 = zero

Option<String> example04 =
 one("a").bindM(a ->
  zero().bindM(z ->
   one("c").bindF(c ->
    a + z + c
   )
  )
 );
```

example05 (OptionApp)

```
// example05 = zero

Option<String> example05 =
 one("a").bindA(
  zero().bindA(
   one("c").bindF(c -> z -> a ->
    a + z + c
   )
  )
 );
```

filter (Option)

```
default public Option<Z> filter(
 final Function<Z, Boolean> z2b
) {
 return
  choice(
   z2b,
   z -> one(z),
   z -> zero()
  );
}
```

example07 (OptionApp)

```
// example07 = zero

Option<String> example07 =
 one("a").filter(a ->
  a.equals("b")
 );
```

plus (Option)

```
// additive method

default public Option<Z> plus(
 final Cbn<Option<Z>> _2oz
) {
 return
  fold(
   OptionStatics::some,
   _2oz
  );
}
```

plus (OptionStatics)

```
// static version of plus

public static <Z>
Option<Z> plus(
 final Option<Z> oz1,
 final Cbn<Option<Z>> oz2
) {
 return
  oz1.plus(oz2);
}
```

someOptions (OptionApp)

```
private static final
Option<String> oneA_plus_oneB =
 one("a").plus(() -> one("b"));
private static final
Option<String> oneC_plus_oneD =
 one("c").plus(() -> one("d"));
```

someOptionsUsingZero (OptionApp)

```
private static final
Option<String> oneA_plus_zero =
 one("a").plus(() -> zero());
// note: sometimes we have to help the type inferencer
private static final
Option<String> zero_plus_oneA =
 OptionStatics.<String>zero().plus(() -> one("a"));
```

example08 (OptionApp)

```
// example08 = one(a)

Option<String> example08 =
 oneA_plus_oneB;
```

example09 (OptionApp)

```
// example09 = one(a)

Option<String> example09 =
 oneA_plus_zero;
```

example10 (OptionApp)

```
// example10 = one(a)

Option<String> example10 =
 zero_plus_oneA;
```

example11 (OptionApp)

```
// example11 = one(ac)

Option<String> example11 =
 oneA_plus_oneB.bindM(apb ->
  oneC_plus_oneD.bindF(cpd ->
   apb + cpd
  )
 );
```

example12 (OptionApp)

```
// example12 = one(a)

Option<String> example12 =
 oneA_plus_oneB.identity();
```

example13 (OptionApp)

```
// example13 = 1

int example13 =
 oneA_plus_oneB.length();
```

Stream

Stream

Stream

```
public interface Stream<Z> {
```

More

```
final public class More<Z>
 implements Stream<Z> {

 final Z current;
 final Cbn<Stream<Z>> next;
```

Done

```
final public class Done<Z>
 implements Stream<Z> {
```

fold (Stream)

```
// abstract method
// follows the recursive structure of Stream

public <Y>
Y fold(
 final BiFunction<Z, Cbn<Y>, Y> zny2y,
 final Cbn<Y> y
);
```

fold (More)

```
@Override
public <Y>
Y fold(
 final BiFunction<Z, Cbn<Y>, Y> zny2y,
 final Cbn<Y> y
) {
 return
  zny2y._(
   current,
   () -> next._().fold(zny2y, y)
  );
}
```

fold (Done)

```
@Override
public <Y>
Y fold(
 final BiFunction<Z, Cbn<Y>, Y> zny2y,
 final Cbn<Y> y
) {
 return
  y._();
}
```

more (StreamStatics)

```
public static <Z>
Stream<Z> more(
 final Z current,
 final Cbn<Stream<Z>> next
) {
 return
  new More<>(current, next);
}
```

done (StreamStatics)

```
public static <Z>
Stream<Z> done(
) {
 return
  new Done<>();
}
```

identity (Stream)

```
default public Stream<Z> identity() {
 return
  fold(
   StreamStatics::more,
   StreamStatics::done
  );
}
```

length (Stream)

```
default public Integer length() {
 return
  fold(
   (z, l) -> 1 + l._(),
   () -> 0
  );
}
```

zero (StreamStatics)

```
public static <Z>
Stream<Z> zero(
) {
 return
  done();
}
```

plus (Stream)

```
default public Stream<Z> plus(
 final Cbn<Stream<Z>> sz
) {
 return
  fold(
   StreamStatics::more,
   sz
  );
}
```

bindM (Stream)

```
default public <Y>
Stream<Y> bindM(
 final Function<Z, Stream<Y>> z2sy
) {
 return
  fold(
   (z, sy) -> z2sy._(z).plus(sy),
   StreamStatics::zero
  );
}
```

one (StreamStatics)

```
public static <Z>
Stream<Z> one(
 final Z z
) {
 return
  more(z, StreamStatics::done);
}
```

example01 (StreamApp)

```
// example01 = one(abc) : zero

Stream<String> example01 =
 one("a").bindM(a ->
  one("b").bindM(b ->
   one("c").bindM(c ->
    one(a + b + c)
   )
  )
 );
```

example02 (StreamApp)

```
// example02 = one(abc) : zero

Stream<String> example02 =
 one("a").bindM(a ->
  one("b").bindM(b ->
   one("c").bindF(c ->
    a + b + c
   )
  )
 );
```

example03 (StreamApp)

```
// example03 = one(abc) : zero

Stream<String> example03 =
 one("a").bindA(
  one("b").bindA(
   one("c").bindF(c -> b -> a ->
    a + b + c
   )
  )
 );
```

example06 (StreamApp)

```
// example06 = one(c) : zero

Stream<String> example06 =
 one("a").choice(a ->
  a.equals("b"),
  a -> one("b"),
  a -> one("c")
 );
```

example04 (StreamApp)

```
// example04 = zero

Stream<String> example04 =
 one("a").bindM(a ->
  zero().bindM(z ->
   one("c").bindF(c ->
    a + z + c
   )
  )
 );
```

example05 (StreamApp)

```
// example05 = zero

Stream<String> example05 =
 one("a").bindA(
  zero().bindA(
   one("c").bindF(c -> z -> a ->
    a + z + c
   )
  )
 );
```

example07 (StreamApp)

```
// example07 = zero

Stream<String> example07 =
 one("a").filter(a ->
  a.equals("b")
 );
```

example08 (StreamApp)

```
// example08 = one(a) : one(b) : zero

Stream<String> example08 =
 oneA_plus_oneB;
```

example09 (StreamApp)

```
// example09 = one(a) : zero

Stream<String> example09 =
 oneA_plus_zero;
```

example10 (StreamApp)

```
// example10 = one(a) : zero

Stream<String> example10 =
 zero_plus_oneA;
```

example11 (StreamApp)

```
// example11 = one(ac) : one(ad) : one(bc) : one(bd) : zero

Stream<String> example11 =
 oneA_plus_oneB.bindM(apb ->
  oneC_plus_oneD.bindF(cpd ->
   apb + cpd
  )
 );
```

example12 (StreamApp)

```
// example12 = example12 = one(a) : one(b) : zero

Stream<String> example12 =
 oneA_plus_oneB.identity();
```

example13 (StreamApp)

```
// example13 = 2

int example13 =
 oneA_plus_oneB.length();
```

take (Stream)

```
// note: a stream can be infinite

public Stream<Z> take(
 int n
);
```

take (More)

```
@Override
public Stream<Z> take(
 int n
) {
 return
  (n > 0)
   ? more(
   current,
   () -> next._().take(n - 1)
  )
   : done();
}
```

take (Done)

```
@Override
public Stream<Z> take(
 int n
) {
 return
  done();
}
```

infinitelyMany (StreamApp)

```
private static <Z>
Stream<Z> infinitelyMany(
 Z z
) {
 return
  one(z).plus(
   () -> infinitelyMany(z)
  );
}
```

example14 (StreamApp)

```
// example14 = one(a) : one(a) : one(a) : one(a) : zero

Stream<String> example14 =
 infinitelyMany("a").take(4);
```

fibonacciNumbersFrom (StreamApp)

```
private static Stream<Integer> fibonacciNumbersFrom(
 Integer fib0,
 Integer fib1
) {
 return
  one(fib0).plus(
   () -> fibonacciNumbersFrom(fib1, fib0 + fib1)
  );
}
```

example15 (StreamApp)

```
// example15 = one(1) : one(2) : one(3) : one(5) : zero

Stream<Integer> example15 =
 fibonacciNumbersFrom(1, 2).take(4);
```

lift (StreamStatics)

```
// cbn version of one

public static <Z>
Cbn<Stream<Z>> lift(
 final Cbn<Z> z
) {
 return
   () -> one(z._());
}
```

liftF (StreamStatics)

```
// static version of bindF

public static <Z, Y>
Function<Stream<Z>, Stream<Y>> liftF(
 final Function<Z, Y> z2y
) {
 return
  sz ->
   sz.bindF(z2y);
}
```

liftA (StreamStatics)

```
// static version of bindA

public static <Z, Y>
Function<Stream<Z>, Stream<Y>> liftA(
 final Stream<Function<Z, Y>> s_z2y
) {
 return
  sz -> sz.bindA(s_z2y);
}
```

liftBF (StreamStatics)

```
public static <Z, Y, X>
BiFunction<Stream<Z>, Cbn<Stream<Y>>, Stream<X>> liftBF(
 final BiFunction<Z, Cbn<Y>, X> zny2x
) {
 return
  (sz, sy) ->
   sz.bindA(
    sy._().bindF(y -> z ->
     zny2x._(z, () -> y)
    )
   );
}
```

foreachDeclaration (Stream)

```
default public <Y, X, MY, MX>
Function<Function<Z, MY>, MX> foreach(
 final Cbn<X> x,
 final BiFunction<Y, Cbn<X>, X> ynx2x,
 final Function<Cbn<X>, Cbn<MX>> lift,
 final Function<
  BiFunction<Y, Cbn<X>, X>,
  BiFunction<MY, Cbn<MX>, MX>
  > liftBF
)
```

foreachDefinition (Stream)

```
{
 return
  z2my ->
   fold(
    (z, mx) ->
     liftBF._(ynx2x)._(z2my._(z), mx),
    lift._(x)
   );
}
```

sequenceOpDeclaration (OptionStatics)

```
public static <Z>
Function<Stream<Option<Z>>, Option<Stream<Z>>> sequenceOp(
 final Cbn<Stream<Z>> sz,
 final BiFunction<
  Stream<Z>, Cbn<Stream<Z>>,
  Stream<Z>
  > sznsz2sz
)
```

sequenceOpDefinition (OptionStatics)

```
{
 return
  soz ->
   soz.foreach(
    sz,
    sznsz2sz,
    OptionStatics::lift,
    OptionStatics::liftBF
   )._(
    liftF(StreamStatics::one)
   );
}
```

sequenceAccDeclaration (OptionStatics)

```
public static <Z>
Function<Stream<Option<Z>>, Option<Stream<Z>>> sequenceAcc(
 final Cbn<Stream<Z>> sz,
 final BiFunction<Z, Cbn<Stream<Z>>, Stream<Z>> znsz2sz
)
```

sequenceAccDefinition (OptionStatics)

```
{
 return
  soz ->
   soz.foreach(
    sz,
    znsz2sz,
    OptionStatics::lift,
    OptionStatics::liftBF
   )._(
    liftF(FunctionStatics::identity)
   );
}
```

someStreamsOfOptions (OptionApp)

```
private static final
Stream<Option<String>> oneOA_plus_oneOB =
 StreamStatics.one(one("a")).plus(() ->
  StreamStatics.one(one("b"))
 );
private static final
Stream<Option<String>> oneZ_plus_oneOB =
 StreamStatics.<Option<String>>one(zero()).plus(() ->
  StreamStatics.one(one("b"))
 );
private static final
Stream<Option<String>> oneOA_plus_oneZ =
 StreamStatics.one(one("a")).plus(() ->
  StreamStatics.one(zero())
 );
```

example14 (OptionApp)

```
// example14 = one(one(a) : one(b) : zero)

Option<Stream<String>> example14 =
 OptionStatics.<String>sequenceOp(
  StreamStatics::zero,
  StreamStatics::plus
 )._(
  oneOA_plus_oneOB
 );
```

example15 (OptionApp)

```
// example15 = zero

Option<Stream<String>> example15 =
 OptionStatics.<String>sequenceAcc(
  StreamStatics::done,
  StreamStatics::more
 )._(
  oneZ_plus_oneOB
 );
```

example16 (OptionApp)

```
// example16 = zero

Option<Stream<String>> example16 =
 OptionStatics.<String>sequenceAcc(
  StreamStatics::done,
  StreamStatics::more
 )._(
  oneOA_plus_oneZ
 );
```

example17 (OptionApp)

```
// example17 = one(one(A)) : zero
Stream<Option<String>> example17 =
 oneA_plus_oneB.<
  Option<String>, Option<String>,
  Stream<Option<String>>, Stream<Option<String>>
  >foreach(
  OptionStatics::zero, FunctionStatics::identity,
  StreamStatics::lift, StreamStatics::liftF
 )._(s ->
  StreamStatics.one(
   one(s.toUpperCase())
  )
 );
```

foreachDeclaration (Option)

```
default public <Y, X, MY, MX>
Function<Function<Z, MY>, MX> foreach(
 final Cbn<X> x,
 final Function<Y, X> y2x,
 final Function<Cbn<X>, Cbn<MX>> lift,
 final Function<Function<Y, X>, Function<MY, MX>> liftF
)
```

foreachDefinition (Option)

```
{
 return
  z2my ->
   fold(
    z -> liftF._(y2x)._(z2my._(z)),
    lift._(x)
   );
}
```

sequenceFunDeclaration (StreamStatics)

```
public static <Z>
Function<Option<Stream<Z>>, Stream<Option<Z>>> sequenceFun(
 final Cbn<Option<Z>> oz,
 final Function<Option<Z>, Option<Z>> oz2oz
)
```

sequenceFunDefinition (StreamStatics)

```
{
 return
  osz ->
   osz.foreach(
    oz,
    oz2oz,
    StreamStatics::lift,
    StreamStatics::liftF
   )._(
    liftF(OptionStatics::one)
   );
}
```

example16 (StreamApp)

```
// example16 = one(one(a)) : zero

Stream<Option<String>> example16 =
 StreamStatics.<String>sequenceFun(
  OptionStatics::zero,
  FunctionStatics::identity
 )._(
  oneSA_plus_oneSB
 );
```

example17 (StreamApp)

```
// example17 = zero

Stream<Option<String>> example17 =
 StreamStatics.<String>sequenceFun(
  OptionStatics::zero,
  FunctionStatics::identity
 )._(
  oneZ_plus_oneSB
 );
```

example18 (StreamApp)

```
// example18 = one(one(a)) : zero

Stream<Option<String>> example18 =
 StreamStatics.<String>sequenceFun(
  OptionStatics::zero,
  FunctionStatics::identity
 )._(
  oneSA_plus_oneZ
 );
```

example19 (StreamApp)

```
// example19 = one(one(A) : one(B) : zero)
Option<Stream<String>> example19 =
 oneA_plus_oneB.<
  Stream<String>, Stream<String>,
  Option<Stream<String>>, Option<Stream<String>>
  >foreach(
  StreamStatics::zero, StreamStatics::plus,
  OptionStatics::lift, OptionStatics::liftBF
 )._(s ->
  OptionStatics.one(
   one(s.toUpperCase())
  )
 );
```

Tuple

Tuple

Tuple

```
// Tuple<Z, Y> zny = ...;
// Z z = zny._1;
// Y y = zny._2

public final class Tuple<Z, Y> {
 public final
 Z _1;
 public final
 Y _2;
```

TupleStatics

```
public final class TupleStatics {
 public static <Z, Y>
 Tuple<Z, Y> tuple(
  final Z z,
  final Y y
 ) {
  return
   new Tuple<>(z, y);
 }
}
```

Unit

Unit

Unit

```
public final class Unit {
 @Override
 public String toString() {
  return "unit";
 }
}
```

UnitStatics

```
public final class UnitStatics {
 public final static
 Unit unit =
  new Unit();
}
```

State

State

State

```
// note: functional interface

public interface State<S, Z> {

 public Function<S, Tuple<Z, S>> open();
```

bindM (State)

```
// note: function literal

default public <Y>
State<S, Y> bindM(
 final Function<Z, State<S, Y>> z2sy
) {
 return
  () ->
   s -> {
    Tuple<Z, S> zns = open()._(s);
    return
     z2sy._(zns._1).open()._(zns._2);
   };
}
```

one (StateStatics)

```
public static <S, Z>
State<S, Z> one(
 final Z z
) {
 return
  () ->
   s ->
    tuple(z, s);
}
```

get (StateStatics)

```
public static <S>
State<S, S> get(
) {
 return
  () ->
   s ->
    tuple(s, s);
}
```

set (StateStatics)

```
static <S>
State<S, Unit> set(
 final S newS
) {
 return
  () ->
   oldS ->
    tuple(unit, newS);
}
```

exec (StateStatics)

```
public static <S>
State<S, Unit> exec(
 final Function<S, S> s2s
) {
 return
  StateStatics.<S>get().bindM(s ->
   set(s2s._(s))
  );
}
```

Status (StateApp)

```
public enum Status {
 FREE, BUSY
}
```

Action (StateApp)

```
public enum Action {
 COIN, CANDY
}
```

Machine (StateApp)

```
public class Machine {
 Status status;
 int candies;
 int coins;
```

actions (StateApp)

```
private static final Stream<Action> coin =
 one(COIN);
private static final Stream<Action> candy =
 one(CANDY);
private static final Stream<Action> actions =
 coin.plus(
  () -> candy
 ).plus(
  () -> coin
 ).plus(
  () -> candy
 );
```

stateMachinePartOne (StateApp)

```
private static State<Machine, Unit> stateMachine(
 final Stream<Action> actions
) {
 return
  actions.<
   Unit, Unit,
   State<Machine, Unit>, State<Machine, Unit>
   >foreach(
   () -> unit, (u, v) -> unit,
   StateStatics::lift, StateStatics::liftBF
  )
```

stateMachinePartTwo (StateApp)

```
    ._(a ->
     exec(
      m ->
       m.candies == 0 ||
        a == COIN && m.status == BUSY ||
        a == CANDY && m.status == FREE
        ? m
        : a == COIN && m.status == FREE
        ? new Machine(BUSY, m.candies, m.coins + 1)
        : a == CANDY && m.status == BUSY
        ? new Machine(FREE, m.candies - 1, m.coins)
        : null // this should never happen
     )
    );
}
```

stateMachineTest (StateApp)

```
System.out.println(
 stateMachine(actions).open()._(
  new Machine(FREE, 10, 0)
 )._2
);
```

Concurrent

Concurrent

SimpleFuture

```
public interface SimpleFuture<Z>
 extends Future<Z> {
 public Z compute(long total);

 default public Z get() {
  return
   compute(Long.MAX_VALUE);
 }

 default public Z get(long total, TimeUnit timeUnit) {
  return
   compute(MILLISECONDS.convert(total, timeUnit));
 }
```

notImplemented

```
default public boolean isDone() {
 throw
  new IllegalStateException("not implemented");
}

default public boolean isCancelled() {
 throw
  new IllegalStateException("not implemented");
}

default public boolean cancel(boolean interruptable) {
 throw
  new IllegalStateException("not implemented");
}
```

Future

```
public interface Future<Z> {

 public Function<ExecutorService, SimpleFuture<Z>> open();
```

bindMDeclaration (Future)

```
default public <Y>
Future<Y> bindM(
 final Function<Z, Future<Y>> z2fy
)
```

bindMDefinition (Future)

```
{
 return
  () -> es -> totalTime -> {
   try {
    final long start = currentTimeMillis();
    final Z z = open()._(es).get(totalTime, MILLISECONDS);
    final long stop = currentTimeMillis();
    final long remainingTime = totalTime - (stop - start);
    final Future<Y> fy = z2fy._(z);
    return
     fy.open()._(es).get(remainingTime, MILLISECONDS);
   } catch (Exception e) {
    throw new IllegalStateException("not implemented");
   }
  };
}
```

one (FutureStatics)

```
public static <Z>
Future<Z> one(
 final Z z
) {
 return
  () ->
   es ->
    timeout ->
     z;
}
```

bindADeclaration (Future)

```
// note: has other default definition

default public <Y>
Future<Y> bindA(
 final Future<Function<Z, Y>> f_z2y
)
```

bindADefinitionPartOne (Future)

```
{
 return
  () -> es -> {
   final SimpleFuture<Z> sfz =
    open()._(es);
   final SimpleFuture<Function<Z, Y>> sf_z2y =
    f_z2y.open()._(es);
```

bindADefinitionPartTwo (Future)

```
    return
     totalTime -> {
      try {
       final long start = currentTimeMillis();
       final Z z = sfz.get(totalTime, MILLISECONDS);
       final long stop = currentTimeMillis();
       final long remainingTime = totalTime - (stop - start);
       final Function<Z, Y> z2y =
        sf_z2y.get(remainingTime, MILLISECONDS);
       return z2y._(z);
      } catch (Exception e) {
       throw new IllegalStateException("not implemented");
      }
     };
   };
}
```

bindF (Future)

```
// note: has other default definition

default public <Y>
Future<Y> bindF(
 final Function<Z, Y> z2y
) {
 return
  bindA(
   one(z2y)
  );
}
```

forkDeclaration (FutureStatics)

```
public static <Z>
Future<Z> fork(
 final Cbn<Future<Z>> fz
)
```

forkDefinitionPartOne (FutureStatics)

```
{
 return
  () -> es -> {
   final CountDownLatch latch = new CountDownLatch(1);
   final Ref<Z> rz = new Ref<Z>();
   es.submit(
    () -> {
     try {
      rz.assign(fz._().open()._(es).get());
      latch.countDown();
     } catch (Exception e) {
      throw new IllegalStateException("not implemented");
     }
    }
   );
```

forkDefinitionPartTwo (FutureStatics)

```
    return
     timeout -> {
      try {
       latch.await(timeout, MILLISECONDS);
       return rz.deref();
      } catch (Exception e) {
       throw new IllegalStateException("not implemented");
      }
     };
   };
}
```

async (FutureStatics)

```
public static <Z>
Future<Z> async(
 final Cbn<Z> z
) {
 return
  fork(
   () -> one(z._())
  );
}
```

sleep (FutureApp)

```
private static void sleep(long time) {
 try {
  Thread.sleep(time);
 } catch (Exception e) {
  throw new IllegalStateException("not implemented");
 }
}
```

mkExecutorService (FutureApp)

```
private static final ExecutorService executorService =
 Executors.newScheduledThreadPool(10);
```

fourTimesZero (FutureApp)

```
private final static
Stream<Integer> fourTimesZero =
 one(0).plus(
  () -> one(0)
 ).plus(
  () -> one(0)
 ).plus(
  () -> one(0)
 );
```

future (FutureApp)

```
private static Future<Integer> future(int i) {
 return
  async(() -> {
   sleep(5000);
   return 10;
  });
}
```

foreachTest (FutureApp)

```
// one(10) : one(10) : one(10) : one(10) : zero

System.out.println(
 fourTimesZero.<
  Integer, Stream<Integer>,
  Future<Integer>, Future<Stream<Integer>>
  >foreach(
  StreamStatics::done, StreamStatics::more,
  FutureStatics::lift, FutureStatics::liftBF
 )._(
  i -> future(10)
 ).open()._(executorService).get()
);
```

fourTimesFuture (FutureApp)

```
private static final
Stream<Future<Integer>> fourTimesFutureTen =
 one(future(10)).plus(
  () -> one(future(10))
 ).plus(
  () -> one(future(10))
 ).plus(
  () -> one(future(10))
 );
```

sequenceTest (FutureApp)

```
// one(10) : one(10) : one(10) : one(10) : zero

System.out.println(
 FutureStatics.<Integer>sequenceAcc(
  StreamStatics::done, StreamStatics::more
 )._(
  fourTimesFutureTen
 ).open()._(executorService).get()
);
```