Lazy Immutable

The Groovy `@Immutable` annotation is quite useful; however, it is an all-or-nothing object, meaning that you create it once and then you can't touch it again (other than copying it with different properties). The `@LazyImmutable` annotation in Vanilla allows for a slighly more flexible configuration of immutable objects.

Consider the case of of an immutable Person object:

```
@Immutable
class Person {
    String firstName
    String middleName
    String lastName
    int age
}
```

With `@Immutable` you have to create the object all at once:

```
def person = new Person('Chris','J','Stehno',42)
```

and then you're stuck with it. You can create a copy of it with one or more different properties using the `copyWith` method, but you need to specify the `copyWith=true` in the annotation itself, then you can do something like:

```
Person otherPerson = person.copyWith(firstName:'Bob', age:50)
```

With more complicated immutables, this "all at once" requirement can be annoying. This is where the `@LazyImmutable` annotation becomes useful. With a similar Person class:

```
@LazyImmutable @Canonical
class Person {
    String firstName
    String middleName
    String lastName
    int age
}
```

using the new annotation, you can create and populate the instance over time:

```
def person = new Person('Chris')
person.middleName = 'J'
person.lastName = 'Stehno'
person.age = 42
```

Notice that the `@LazyImmutable` annotation does not apply any other transforms (as the standard `@Immutable` does). It's a standard Groovy object, but with an added method: the `asImmutable()` method is injected via AST Transformation. This method will take the current state of the object and create an immutable version of the object - this does imply that the properties of lazy immutable objects should follow the same rules as those of the standard immutable so that the conversion is determinate. For our example case:

```
Person immutablePerson = person.asImmutable()
```

The created object is the same immutable object as would have been created by using the `@Immutable` annotation and it is generated as an extension of the class you created so that it's type is still valid. The immutable version of the object also has a useful added method, the `asMutable()` method is used to create a copy of the original mutable object.

```
Person otherMutable = immutablePerson.asMutable()
```

You can swap back and forth between immutability and mutability as needed without loss of functionality or data integrity.