`inout` design

Syntax extension to linear dafny to support in-place mutation of datatype fields through mutable references. Mutable references can only be constructed as method formals; non-argument variables cannot contain mutable references.

Mutable references can refer to `ordinary` fields of `linear` `datatype`s (`ordinary inout` reference) or to `linear` fields of `linear` `datatype`s (`linear inout` reference). There can be a "path" of `linear` fields of `linear` datatypes to reach the `linear` or `ordinary` field which is mutably referenced; no `ordinary` fields are permitted in the path (except at the end). Support for mutation of `ordinary` fields of `ordinary` datatypes is out of scope for this design (and likely unsound in the general case).

Mutating (transitive) fields of `linear` datatypes is safe as there exists only a single reference to the outer `linear` datatype.

example of path of `linear` fields

```
1  linear datatype Loco = Loco(fuel: nat)
2  linear datatype Car = Car(passengers: nat)
3  linear datatype Train = Train(linear loco: Loco, linear car1:
Car, linear car2: Car)
4
5  method Rail(linear train: Train) {
6    → valid path for an `ordinary inout` reference:
train.car2.passengers
7    → valid path for a `linar inout` reference: train.car2
8  }
```

syntax

method definition

A `method` parameter takes a mutable reference if it's marked `inout` (`ordinary inout` reference). It can be optionally marked `linear` (`linear inout` reference). A `linear inout` parameter must be of a `linear` `datatype`.

```
method Method([linear] inout param: Type)
```

example

```
linear datatype Loco = Loco(fuel: nat)
linear datatype Car = Car(passengers: nat)

method LoadPassengers(linear inout self: Car, count: nat)
```

A `method` that's a member of a `linear` datatype can be marked `linear inout` to take the `datatype` as a `linear inout` mutable reference. This is equivalent to taking `this` as a `linear inout` parameter.

```
linear inout Member()
```

example

```
linear datatype Train = Train(linear loco: Loco, linear car1: Car, linear car2: Car)
{
  linear inout method LoadFuel(fuel: nat) {
    // `this` is `linear inout` here
```

`requires` and `ensures`

A `requires` clause can refer to the value of `param` before the method has executed as `param`. An `ensures` clause can refer to the value of `param` before the method has executed as `old(param)` and it can refer to the value of `param` after execution as `param`. Note that, differently from `old` in dynamic frames dafny, because `param` is linear and doesn't contain pointers, `old(param).fieldname` and `old(param.fieldname)` have identical meaning. To simplify translation, we only permit `old(param)` (no complex expressions within `old`) for `inout` method parameters.

example

```
method LoadPassengers(linear inout self: Car, count: nat)
ensures self.passengers == old(self).passengers + count
```

method body

The body of a `method` that takes one or more mutable references as parameters will allow assignment to the `inout` variable. If it's a `linear inout` reference, the variable is treated linearly within the method body (it must be available on `return`). Mutable references can be constructed by passing the `linear inout` variable (or a path with `linear` fields, as described earlier) to a `method` taking an `inout` argument. Assigments and mutations of the `inout`

variable affect the datatype from which the mutable reference was constructed.

call site

A method call can construct a mutable reference by passing (a path of `linear` fields of) a `linear` variable or a `linear inout` argument, prefixed with `[linear] inout`. The variable/argument must be available before the call and remains available after the call.

example #1, `ordinary inout` reference

given the trusted library `method Assign` (see "proposed trusted methods")

```
method {:extern} Assign<V>(inout v: V, newV: V)
ensures v == newV
```

```
method LoadPassengers(linear inout self: Car, count: nat)
ensures self.passengers == old(self).passengers + count
{
  var newCount := self.passengers + count;
  Assign(inout self.passengers, newCount);
}
```

example #2, `linear inout` reference

```
linear var train: Train := ...;
LoadPassengers(linear inout train.car1);
```

(`car1` is a `linear` field of `Train`)

ghost code

Similarly to `linear` variables, `[linear] inout` variables (formals) can be used in ghost expressions and assigned to `ghost` variables.

example

```
method LoadPassengers(linear inout self: Car, count: nat)
ensures self.passengers == old(self).passengers + count
{
  var newCount := self.passengers + count;
  ghost var beforeLoad := self;
  Assign(inout self.passengers, newCount);
  assert beforeLoad == old(self);
  assert beforeLoad.passengers == self.passengers - count;
}
```

trusted methods

These are trusted library methods that complement the new syntax.

`Assign`

```
method {:extern} Assign<V>(inout v: V, newV: V)
ensures v == newV
```

`Assign` enables changing the value of an `ordinary` field of a `linear` datatype without additional syntax. We can later add syntactic sugar to support regular assignment syntax.

`Replace`

```
method {:extern} Replace<V>(linear inout v: V, linear newV: V)
returns (linear replaced: V)
ensures replaced == old(v)
ensures v == newV
```

`Replace` enables updating the value of a `linear` field of a `linear` datatype without additional syntax. We can later add syntactic sugar to support regular assignment syntax.

`Swap`

```
method {:extern} Swap<V>(linear inout a: V, linear inout b: V)
ensures b == old(a)
ensures a == old(b)
```

`Swap` enables swapping the values of two `linear` fields of the same type of `linear` datatypes without additional syntax.

translation (verification conditions) [WIP]

For verification (translation to *Boogie*), the `[linear] inout` parameters are translated to regular dafny as follows:

method definition

```
method Method([linear] inout param: Type)
  returns (ret1: RetType)
  requires Expr(param)
  ensures Expr(param, after(param))
```

translates to:

```
method Method(param: Type)
  returns (param': Type, ret1: RetType)
  requires Expr(param)
  ensures Expr(/* param -> */ param, /* after(param) */ -> param')
```

body

Introduce preamble:

```
{
  var param := param;
```

introduce epilogue:

```
  param' := param;
}
```

call site

For

```
linear var thing;
```

Case #1

```
var ret1 := Method(inout thing);
```

translates to:

```
var ret1;
thing, ret1 := Method(thing);
```

Case #2

```
var ret1 := Method(inout thing.field);
```

translates to:

```
var _tmp00, ret1 := Method(thing.field);
thing := thing.(field := _tmp00);
```

design considerations

alternatives considered

`after`

We considered using `after` to refer to the value after execution; unfortunately this breaks the semantics of `old` from within the body.

```
method LoadPassengers(linear inout self: Car, count: nat)
ensures after(self).passengers == self.passengers + count
{
  var newCount := self.passengers + count;
  ghost var beforeLoad := self;
  Assign(inout self.passengers, newCount);
  assert beforeLoad.passengers == self.passengers - count;
}
```

[WIP]

[todo discuss how borrow checking is only useful when you have mutable references]

```
                linearity
             /              \
      mutable refs           |
            \                |
             -----------\    |
                   borrow checker
```

[discuss alternative compiler optimisation pass for `function method`s]

[todo discuss improvement over `old` and labels in dynamic frames dafny]

```
method (x: Datatype)  (x: Datatype) -> (x': Datatype)

ensures old(x).y == 22  ensures x.y == 22
ensures old(x.y) == 22  ensures x.y == 22

label here:
ghost var x_snapshot := x;

old@here(x.y)

Mutate(x.somefield) // x := x.(somefield := Mutate(x.somefield))
```