2 Implementation strategy

In order to gauge how effective in-place mutation would be for linear function I decided to start by adding a keyword that would tell the compiler to perform mutation for variable that are matched, irrespective of their linearity properties.

While this results in unsafe programs (since arbitrary mutation breaks referential transparency) , when used carefully in our benchmarks it will allow use to test how promising linearity improvements might be.

Implementation details

The goal is to be able to write this program

```
%mutating
update : Ty -> Ty
update (ValOnce v) = ValOnce (S v)
update (ValTwice v w) = ValTwice (S v) (S (S w))
```

where the `%mutating` annotation indicates that the value manipulated will be subject to mutation rather than construction.

If we were to write this code in a low-level c-like syntax we would like to go from the non-mutating version here

```
void * update(v * void) {
    Ty* newv;
    if v->tag == 0 {
        newv = malloc(sizeof(ValOnce));
        newv->tag = 0;
        newv->val1 = 1 + v->val1;
    } else {
        newv = malloc(sizeof(ValTwice));
        newv->tag = 1;
        newv->val1 = 1 + v->val1;
        newv->val2 = 1 + 1 + v->val2;
    }
    return newv;
}
```

to the more efficient mutating version here

```
void * update(v * void) {
    if v->tag == 0 {
        v->val1 = 1 + v->val1;
    } else {
        v->val1 = 1 + v->val1;
        v->val2 = 1 + 1 + v->val2;
    }
    return nv;
}
```

The two programs are very similar but the second one mutate the argument directly instead of mutating a new copy of it.

There is however a very important limitation:

We only mutate uses of the constructor we are matching on

The following program would see no mutation

```
%mutating
update : Ty -> Ty
update (ValTwice v) = ValOnce (S v)
update (ValOnce v) = ValTwice (S (S v))
```

Since the constructor we are matching on the left side of the clause does not appear on the right.

This is to avoid implicit allocation when we mutate a constructor which has more fields than the one we are given. Imagine representing data as a records:

```
ValOnce = { tag : Int , val1 : Int }
ValTwice = { tag : Int , val1 : Int val2 : Int }
```

if we are given a value `ValOnce` and asked to mutate it into a value `ValTwice` we would have to allocate more space to accomodate for the extra `val2` field.

Similarly if we are given a `ValTwice` and are asked to mutate it into a value `ValOnce` we would have to carry over extra memory space that will remain unused.

Ideally our compiler would be able to identify data declaration that share the same layout and replace allocation for them by mutation, but for the purpose of this thesis we will ignore this optimisation and carefully design our benchmarks to make use of it. Which brings us to the next section

Mutating branches

For this to work we need to add a new constructor to the AST that represents *compiled* programs `CExp`. We add the consturctor

```
CMut : (ref : Name) -> (args : List (CExp vars)) -> CExp vars
```

which represents mutation of a variable identified by its `Name` in context and using the argument list to modify each of its fields.

(This new constructor has to be carried of to tress `NamedExp` `ANF` and `Lifted`, the details are irrelevants and the changes trivial)

Once this change reached the code generator it needs to output a `mutation` instructon rather than an allocation operation. Here is the code for the scheme backend

show scheme backend implementation for CMut

AS you can see we generate one instruction per field to mutate as well ad a final instruction to *return* the value passed in argument, this to keep the semantics of the existing assumption about constructing new values.

Reference nightmare

There is however an additional details that isn't as easy to implement and this is related to getting a reference to the term we are mutating.

Let's look at our `update` function once again and update it slightly

```
%mutating
update : (1 _ : Ty) -> Ty
update arg = case arg of
                  ValTwice v => ValTwice (S (S v))
                  ValOnce v => ValOnce (S v)
```

This version makes use of there temporary variable `arg` before matching on the function argument directly. Otherwise it's the same as what we showed before.

What needs to happen is that `ValTwice` on the first clause needs to access the variable `arg` and mutate it directly. And similarly for `ValOnce`.

```
case arg of
     ValTwice v => -- access `arg` and mutate it with S (S v)
                   ValTwice (S (S v))
     ValOnce v => -- access `arg` and mutate it with S v
                  ValOnce (S v)
```

However looking at the AST for pattern match clauses we see that it does not carry any information about the original value that was matched:

```
ConCase : Name -> (tag : Int) -> (args : List Name) ->
                  CaseTree (args ++ vars) -> CaseAlt vars
```

Thankfully this reference can be found earlier in the `CaseTree` part of the AST.

```
  public export
  data CaseTree : List Name -> Type where

       Case : {name, vars : _} ->
              (idx : Nat) ->
              (0 p : IsVar name idx vars) ->
              (scTy : Term vars) -> List (CaseAlt vars) ->
              CaseTree vars
       STerm : Int -> Term vars -> CaseTree vars
```

```
Unmatched : (msg : String) -> CaseTree vars

Impossible : CaseTree vars
```

A `CaseTree` is either a case containing other cases, or a term, or a missing case, or an impossible case.

We note that the `Case` constructor contains the reference to the variable that is being matched. Therefore we can get it from here and then carry it to our tree transformation.

The tree transformation itself is pretty simple and can be sumarised with this excerpt:

```
replaceConstructor : (cName : Name) -> (tag : Int) ->
                     (rhs : Term vars) ->
                     Core (Term vars)
replaceConstructor cName tag (App fc (Ref fc' (DataCon nref t arity) nm) arg) =
    if cName == nm then pure (App fc (Ref fc' (DataCon (Just ref) t arity) nm) arg)
                   else App fc (Ref fc' (DataCon nref t arity) nm) <$>
    replaceConstructor cName tag arg
```

Which checks that for every application of a data constructor if it is the same as the one we matched on, if it is, then the `CCon` instruction is replaced by a `CMut` which will tell the backend to *reuse* the memory space taken by the argument.