Linting Linearity in Core/System FC

Rodrigo Mesquita

November 25, 2022

1 Foreword

This document is a work in progress proposal that I will deliver at the end of the semester as part of my master thesis preparation. The document needs a proper introduction and lengthy background and related work section besides the content (to be extended) that follows, such that an outsider might understand what I propose to do. However, this initial iteration is instead targeted at those already familiar with the problem and serves as a (less bureaucratic) research proposal on linting linearity in Core and to showcase the progress I have made so far.

2 Introduction

Since Linear Haskell's[] publication and implementation release in GHC 9.0, Haskell's type system supports linearity annotations in functions – bringing linear types into a mainstream pure and lazy functional language. Concretely, function types can be annotated with a multiplicity (a multiplicity of One requires the argument to be consumed exactly once, a multiplicity of Many allows the argument to be consumed unrestrictedly, i.e., zero or more times). A function is linear if it consumes its arguments exactly once when it itself is consumed exactly once (its arguments have multiplicity of one), for some definition of *consume* that I should revise here.

Linearly typed programs are proven/checked to be linear by the type system, so the addition of linear types evidently implied changing the type checker to support linearity. There exist, however, two distinct type checkers in Haskell. The first is run on the surface language, i.e. the Haskell we write, and is a very complex type checker that now also supports typing linearity. The second is run on the intermediate language *Core* that we obtain from desugaring Haskell.

Core is a much smaller language than the whole of Haskell (even though we can compile the whole of Haskell to it!) and the type checker for it is small and fast due to *Core* being explicitly typed and much closer to a polymorphic lambda calculus than its surface-level counterpart. This type checker (called *Lint*) gives us guarantees of correctness in face of all the complex transformations a Haskell program undergoes, such as desugaring and core-to-core passes, because the linter is always run on the resulting code before being compiled to untyped machine code.

We want Core to give us guarantees about our desugaring and transformations with regard to linearity too – a linearly typed Core will ensure that

linearly-typed programs remain correct (linearity is preserved) after desugaring and all GHC transformations (optimisations shouldn't destroy linearity).

Core is already annotated with linearity, but the **linter currently ignores it**! In spite of the strong formal foundations of linear types driving the implementation, their interaction with the whole of GHC is still far from trivial, and the implemented type system cannot accomodate various optimising transformations and, from them resulting, programs that seemingly violate linearity but which do preserve it semantically; for example, laziness makes it harder to reason about linearity. Therefore, we reject various valid programs with regard to linearity after desugaring: the current solution, because disabling optimisations can inccur great performance costs, is to disable the linear linter. However, we believe that GHC's transformations are correct, and it is the linear type system that can't accommodate the resulting programs.

We propose to extend Core's type system with additional linearity information to be able to accommodate linear programs as seen in Core after optimising transformations. We will prove the system's soundness/type safety and validate some core-to-core optimisation passes. Ultimately, we will implement the extension into GHC's linter. Concretely, we'll extend Core's linear type system with rules based on usage annotations for let, letrec and case binder bound variables. Additionally, we will explore a new kind of coercion for multiplicities to validate programs that combine GADTs with multiplicities. The ultimate measure of success is the -dlinear-core-lint flag that, right now, when enabled, rejects many linearly valid programs in the linter. Ideally, by the end of our research and implementation, this flag could be enabled by default and accommodate all existing transformations. Realistically, we want to accept as many diverse transformations as possible while still preserving linearity, even if we are unable to fix them all.

In section 3 we describe usage environments and how they solve three distinct problems. In section 4 we discuss the beginning of multiplicity coercions. In section 5 we take programs that are currently rejected and show how the type system with our extensions can accept them.

3 Typing Usage Environments

Haskell has call-by-need semantics that entail that an expression isn't evaluated when it is let bound but rather when the binding variable is used. This makes it hard to reason about linearity for let bound variables. The following example doesn't typecheck in Haskell, but semantically it is indeed linear because the let binding is lazy.

```
f :: a %1 -> a
f x = let y = x in y
```

Despite not being accepted by the surface-level language, linear programs using lets can occur in Core due to optimising transformations that create let bindings. In a similar fashion, programs which violate syntatic linearity for other reasons other than let bindings are formed from Core transformations.

The solution we found to a handful of problems regarding binders is to have a *usage environment*. A usage environment is a mapping from variables to multiplicities. The main idea is to annotate those binders with a usage annotation

rather than a multiplicity (in contrast, a lambda-bound variable has exactly one multiplicity). When we find a bound variable with a usage environment, we type linearity as if we were using all variables with corresponding multiplicities from that usage environment.

In the above example, this would amount to annotating y with a usage environemnt $x := 1$ (because the expression bound by y uses x one time). Upon using y, we are actually using x one time, and therefore linearity is preserved.

3.1 Let

Let bindings in Core are the first family of problems we tackle with usage environment annotations. Multiple transformations can introduce let-bindings, such as CSE and join points. By extending the type system to allow lets in core we start paving our way to a linear linter.

Right now, Core annotates every variable with a multiplicity at its binding site. The multiplicity for let-bound variables must be ignored thorought the transformation pipeline or otherwise too many programs valid programs would be rejected for violating linearity in its transformed type.

However, programs with let bindings can be correctly typed by associating a usage environment to the bound variables. Instead of associating a multiplicity to every binder, we want to associate a multiplicity if the variable is lambda bound and a usage environment when it is let bound. We then instance the usage environment solution to lets in particular – a let bound variable is annotated with the usage environment computed from the binder expression; in the let body expression, when we find an occurrence of the let bound variable we emit its usage environment. The typing rule is the following:

$$\frac{\Gamma \vdash t : A \rightsquigarrow \{U\} \qquad \Gamma; x :_U A \vdash u : C \rightsquigarrow \{V\}}{\Gamma \vdash \text{let } x :_U A = t \text{ in } u : C \rightsquigarrow \{V\}} \text{ (LET)}$$

Take for example an expression in which y and z are lambda bound with a multiplicity of exactly one. In the following code it might not look as if y and z are both being consumed linearly, but indeed they are since in the first branch we use x – which means using y and z linearly – and we use y and z directly on the second branch. Note again that let binding x doesn't consume y and z because of laziness. Only *using* x consumes y and z.

```
let x = (y, z) in
case e of
  Pat1 -> ... x ...
  Pat2 -> ... y ... z ...
```

If we annotate the x bound by the let with a usage environment δ mapping all free variables in its binder to a multiplicity ($\delta = [y := 1, z := 1]$), we could, upon finding x, simply emit that y and z are consumed once. When typing the second branch we'd also see that y and z are used exactly once. Because both branches use y and z linearly, the whole case expression uses y and z linearly.

3.2 Recursive Lets

A recursive let binds a variable to an expression that might use the bound variable in its body. Certain optimisations can create a recursive let that uses

linearly bound variables in its binder body. We want to treat recursive lets similarly to lets: attribute to bound variables a usage environment and emit it when the variable is used.

The tricky part about determining the usage environment of the variables bound by a recursive let is knowing what usage to emit inside their own body when computing said usage environment. The example below uses x linearly but how can we prove it? If we are able to somehow compute f's usage environment to $x := 1$, we know x is used once when f is applied to *True*.

```
letrec f = \case
        True -> f False
        False -> x
in f True
```

The key idea to computing the usage environment of multiple variables that use themselves in their body is to do the computation in two separate passes. First, calculate a naive environment by emiting free variables multiplicities as usual while treating the recursive-let bound variables as free ones. Second, pass the variables names and their corresponding naive environment as input to algorithm 1 to get a final usage environment. Intuitively, the algorithm, for each recursive binder, goes over all (initially naive) usage environment and substitutes the recursive binder by the corresponding usage environment, scaled up by the amount of times that recursive binder is used in the environment being updated.

We present ahead an algorithm for computing the usage environment of a set of recursive definitions, and its textual description. An (at least informal) proof that the algorithm works should eventually follow this.

- Given a list of functions and their naive environment computed from the body and including the recursive function names $((f, F), (g, G), (h, H)$ in which there might exist multiple occurrences of f, g, h in F, G, H
- For each bound rec var and its environment, update all bindings and their usage as described in the algorithm
- After iterating through all bound rec vars, all usage environments will be free of recursive bind usages, and hence "final"

Note that the difficulty of calculating a usage environment for n recursive-let bound variables increases quadratically, i.e. the algorithm has complexity $O(n^2)$, but this is not a problem since it's rare to have more than a handful of binders in the same recursive let block.

Algorithm 1: computeRecUsages

$usageEnvs \leftarrow naiveUsageEnvs.map(fst)$;
for $(bind, U) \in naiveUsageEnvs$ **do**
  for $V \in usageEnvs$ **do**
    $V \leftarrow sup(V[bind] * U \setminus \{bind\}, V \setminus \{bind\})$

Putting the usage environment idea together with the algorithm we get the following typing rule:

$$\frac{\begin{array}{c}\Gamma; x_1 : A_1 \ldots x_n : A_n \vdash t_i : A_i \rightsquigarrow \{U_{i_{\text{naive}}}\} \\ (U_1 \ldots U_n) = \mathit{computeRecUsages}(U_{1_{\text{naive}}} \ldots U_{n_{\text{naive}}}) \\ \Gamma; x_1 :_{U_1} A_1 \ldots x_n :_{U_n} A_n \vdash u : C \rightsquigarrow \{V\}\end{array}}{\Gamma \vdash \text{let } x_1 :_{U_1} A_1 = t_1 \ldots x_n :_{U_n} A_n = t_n \text{ in } u : C \rightsquigarrow \{V\}} \text{ (LETREC)}$$

Instancing the usage environment solution according to the above description, here's an informal proof that the previous example is well typed: We compute the naive usage environment of f to be $x := 1, f := 1$. We compute its actual usage environment by scaling the naive environment of f without f by the amount of times f appears in the naive environment of f $(1 * [x := 1])$ and get $x := 1$. Finally, we emit $x := 1$ from applying f in the let's body.

The next example is a linearly invalid program because x is a linearly bound variable that is used more than once, and shows how this method correctly rejects it.

```
letrec f = \case
         True -> f False + f False
         False -> x
    in f True
```

To compute the usage environment of f we take its naive environment to be $x := 1, f := 2$. Then, we compute the final environment to be $x := 1$ scaled by the usage of f, $2 * [x := 1]$, resulting in $x := 2$. Now, to lint the linearity of the whole let we must ensure the body of the let uses x linearly. We emit from the body the usage environment of f $(x := 2)$ which uses x more than once, i.e. not linearly.

3.3 Handling the case binder

The current typing rule for case expressions describes that using the case binder is as using the case scrutinee multiple times (though it seems like it actually just starts counting after the second usage which would make the rule wrong for a case expression that doesn't use the binder – the scrutinee should also be consumed), so we scale the usage of the case scrutinee by the superior multiplicity of using the case binder across all branches. However, we hypothetize that this prevents us from typing many valid programs and ultimately doesn't capture correctly the usage of variable, and present a seemingly viable alternative.

Firstly, we remember the definition of *consuming a value* from Linear Haskell as

- Consuming an atomic value is forcing it to Normal Form (NF)
- Consuming a value that is constructed with more than 1 argument is consuming all of its arguments

and further note that when an expression is scrutinized by a case expression it is evaluated to Weak Head Normal Form (WHNF), which in the case of a nullary data constructor is equal to NF.

Now, with the following example, my interpretation of the current rule says that the following case expression consumes the scrutinee twice because the case binder is used. However, we know for sure that in the branch where z is used, z is equal to False, and using False doesn't violate linearity since we can

unrestrictedly create nullary data constructors. A situation similar to this one below happens after the CSE transformation.

```
case <complex expression> of z {
    False -> case y of z' { DEFAULT -> z }
    True  -> y
}
```

A different example is using the case binder z instead of the arguments we pattern matched on x, y. This currently violates linearity because both x and y aren't used and because z is used twice. This doesn't typecheck in the frontend either. However, we know that in this branch, using z is effectively the same as using (x, y)!

```
case <complex expression> of z {
    (x,y) -> z
}
```

The good news is both these two programs and a handful of others listed in Section 5 are accepted with a new rule we've devised for the linear linter. We still need to prove we don't accept any invalid programs as valid.

The key idea of the case-binder-usage solution is annotating the case binder with independent usage environments for each pattern match. That is, for each of the possible branches, using the case binder z will equate to using its usage environment in that branch. That makes it so that using the case binder instead of a nullary data constructor is the same, using the case binder instead of reconstructing the value with the bound pattern variables too, and other situations in which we previously couldn't typecheck linearity are now accepted.

The typing rule for the case expression using the case binder solution:

$$\frac{\begin{array}{c}\Gamma \vdash t : D_{\pi_1 \ldots \pi_n} \rightsquigarrow \{U\} \\ \Gamma; z :_{U_k} D_{\pi_1 \ldots \pi_n} \vdash b_k : C \rightsquigarrow \{V_k\} \qquad V_k \leq V\end{array}}{\Gamma \vdash \text{case } t \text{ of } z :_{(U_1 \ldots U_n)} D_{\pi_1 \ldots \pi_n} \{b_k\}_1^m : C \rightsquigarrow \{U + V\}} \text{ (CASE)}$$

4 Multiplicity Coercions

The second set of problems arises from our inability to coerce a multiplicity into another (or say that one is submultiple of another?).

Taking the following example we can see that we don't know how to say that x is indeed linear in one case and unrestricted in the other, even though it is according to its type. We'd need some sort of coercion to coerce through the multiplicity to the new one we uncover when we pattern match on the GADT evidence (...)

```
data SBool :: Bool -> Type where
  STrue :: SBool True
  SFalse :: SBool False

type family If b t f where
  If True t _ = t
  If False _ f = f
```

```
dep :: SBool b -> Int %(If b One Many) -> Int
dep STrue x = x
dep SFalse _ = 0
```

5 Examples

In this section we present worked examples of programs that currently fail to compile with **-dlinear-core-lint** but would succeed according to the novel typing rules we introduced above. Each example belongs to a subsection that indicates after which transformation the linting failed.

5.1 After the desugarer (before optimizations)

The definition for $! in **linear-base**[] fails to lint after the desugarer (before any optimisation takes place) with *Linearity failure in lambda: x 'Many ⊈ p*

```
($!) :: forall {rep} a (b :: TYPE rep) p q. (a %p -> b) %q -> a %p -> b
($!) f !x = f x
```

The desugared version of that function follows below. It violates (naive?) linearity by using x twice, once to force the value and a second time to call f. However, the x passed as an argument is actually the case binder and it must be handled in its own way. The case binder is the key (as seen in **??**) to solving this and many other examples.

```
($!)
  :: forall {rep :: RuntimeRep} a (b :: TYPE rep) (p :: Multiplicity)
            (q :: Multiplicity).
     (a %p -> b) %q -> a %p -> b
($!)
  = \ (@(rep :: RuntimeRep))
      (@a)
      (@(b :: TYPE rep))
      (@(p :: Multiplicity))
      (@(q :: Multiplicity))
      (f :: a %p -> b)
      (x :: a) ->
      case x of x { __DEFAULT -> f x }
```

The case binder usage rule typechecks this program because x is consumed once, the usage environment of the case binder is empty ($x :_\emptyset a$) and, therefore, when x is used in $f(x)$, we use its usage environment which is empty, so we don't use anything new.

How are strictness annotations typed in the frontend? The issue is this program consumes to value once to force it, and then again to determine the return value.

As a finishing note on this example, we show the resulting program from **-ddump-simpl** that simply uses a different name for the case binder.

```
($!)
  :: forall {rep :: RuntimeRep} a (b :: TYPE rep)
            (p :: Multiplicity) (q :: Multiplicity).
```

```
    (a %p -> b) %q -> a %p -> b
($!)
  = \ (@(rep :: RuntimeRep))
      (@a)
      (@(b :: TYPE rep_aSJ))
      (@(p :: Multiplicity))
      (@(q :: Multiplicity))
      (f :: a %p -> b)
      (x :: a) ->
      case x of y { __DEFAULT -> f_aDV y }
```

5.2 Common Sub-expression Elimination

Currently, the CSE seems to transform a linear program that pattern matches on constant and returns the same constant into a program that breaks linearity that pattern matches on the argument and returns the argument (where in the frontend a constant equal to the argument would be returned)

The definition for && in **linear-base** fails to lint after the common sub-expression elimination (CSE) pass transforms the program.

```
(&&) :: Bool %1 -> Bool %1 -> Bool
False && False = False
False && True = False
True && x = x
```

The issue with the program resulting from the transformation is x being used twice in the first branch of the first case expression. We pattern match on y to force it (since it's a type without constructor arguments, forcing is consuming) and then return x rather than the constant $False$

```
(&&) :: Bool %1 -> Bool %1 -> Bool
(&&) = \ (x :: Bool) (y :: Bool) ->
  case x of w1 {
    False -> case y of w2 { __DEFAULT -> x };
    True -> y
  }
```

At a first glance, the resulting program is impossible to typecheck linearly.

The key observation is that after x is forced into either $True$ or $False$, we know x to be a constructor without arguments (which can be created without restrictions) and know that where we see x we can as well have $True$ or $False$ depending on the branch. However, using x here is very unsatisfactory (and linearly unsound?) because x might be an expression, and we can't really associate x to the value we pattern matched on (be it $True$ or $False$). What we really want to have instead of x is the $w1$ case binder – it relates directly to the value we pattern matched on, it's a variable rather than an expression, and, most importantly, our case-binder-usage idea could be applied here as well

The solution with the unifying case binder usage idea means annotating the case binder with its usage environment. For $True$ and $False$ (and other data constructors without arguments) the usage environment is always empty (using them doesn't entail using any other variable), meaning we can always use the case binder instead of the actual constant constructor without issues. Concretely, if we had $w1$ instead of the second occurrence of x, we'd have an

empty usage environment for $w1$ in the $False$ branch ($w1 :_\emptyset Bool$) and upon using $w1$ we wouldn't use any extra resources

To make this work, we'd have to look at the current transformations and see whether it would be possible to ensure that CSEing the case scrutinee would entail using the case binder instead of the scrutinee. I don't know of a situation in which we'd really want the scrutinee rather than the case binder, so I hypothetize the modified transformation would work out in practice and solve this linearity issue.

Curiously, the optimised program resulting from running all transformations actually does what we expected it with regard to using the constant constructors and preserving linearity. So is the issue from running linear lint after a particular CSE but it would be fine in the end?

```
(&&) :: Bool %1 -> Bool %1 -> Bool
(&&) = \ (x :: Bool) (y :: Bool) ->
  case x of {
    False -> case y of { __DEFAULT -> GHC.Types.False };
    True -> y
  }
```

5.3 Compiling ghc with -dlinear-core-lint

From the definition of *groupBy* in *GHC.Data.List.Infinite*:

```
groupBy :: (a -> a -> Bool) -> Infinite a -> Infinite (NonEmpty a)
groupBy eq = go
  where
    go (Inf a as) = Inf (a:|bs) (go cs)
      where (bs, cs) = span (eq a) as
```

We get a somewhat large resulting core expression, in the middle of which the following expression with a linear type – which currently syntatically violates linearity.

```
jp :: ([a], Infinite a) \%1 -> (# [a], Infinite a #)
jp (wwj :: ([a], Infinite a)) =
    case wwj of wwj {
        DEFAULT -> case wwj of { (wA, wB) -> (# wA, wB #) }
    }
```

Using the case binder usage solution, the case binder is annotated with a usage environment for the only branch ($U_w wj = \emptyset$) and using wwj is equal to using the $\emptyset$. Here it would mean that second **case of** wwj doesn't actually use any variables and hence the first wwj isn't used twice.

5.4 Artificial examples

Following the difficulties in consuming or not consuming the case binder and its associated bound variables linearly we constructed some additional examples that bring out the problem quite clearly.

```
case x of w1
    (x1,x2) -> case y:Bool of
                DEFAULT -> (x1,x2)
```

Is equal to (note that the case binder is being used rather than the x case scrutinee which could be an arbitrary expression and hence really cannot be consumed multiple times?)

```
case x of w1
    (x1,x2) -> case y:Bool of
                DEFAULT -> w1
```

We initially wondered whether x was being consumed if the pattern match ignored some of its variables. We pose that if it does have wildcards then it isn't consuming x fully

```
case x of w1
    (_,x2) -> Is x being consumed? no because not all of its components are
    being consumed?
```

Can we have a solution that even handles weird cases in which we pattern match twice on the same expression but only on one constructor argument per time? I don't think so.

```
case exp of w1
    (x1,_) -> case w1 of w2
                (_, x2) -> (x1, x2)
```

A double let rec in which both use a linear bound variable y

```
letrec f = \case
        True  -> y
        False -> g True
       g = \case
        True -> f True
        False -> y
    in f False
```

We have to compute the usage environment of f and g. For f, we have the first branch with usage environment y and the second with usage g, meaning we have $F = \{g := 1, y := 1\}$ For g, we have $G = \{f := 1, y := 1\}$. To calculate the usage environment of f to emit upon $fFalse$, we calculate ...

Refer to the algorithm for computing recursive environment

6 Work in progress

6.1 In Result of TcGblEnv axioms

I don't understand this one yet. I would guess it has to do with multiplicity coercions. Incorrect incompatible branch: CoAxBranch (src/Prelude/Linear/GenericUtil.hs:112:3-51):

```
type family Fixup (f :: Type -> Type) (g :: Type -> Type) :: Type -> Type where
  Fixup (D1 c f) (D1 _c g) = D1 c (Fixup f g)
  Fixup (C1 c f) (C1 _c g) = C1 c (Fixup f g)
  Fixup (S1 c f) (S1 _c (MP1 m f)) = S1 c (MP1 m f) -- error in this constructor
  Fixup (S1 c f) (S1 _c f) = S1 c f
  Fixup (f :*: g) (f' :*: g') = Fixup f f' :*: Fixup g g'
  Fixup (f :+: g) (f' :+: g') = Fixup f f' :+: Fixup g g'
  Fixup V1 V1 = V1
  Fixup _ _ = TypeError ('Text "FixupMetaData: representations do not match.")
```

6.2 Inlining

6.3 CSE

6.4 Join Points

When duplicating a case (in the case-of-case transformation), to avoid code explosion, the branches of the case are first made into join points

```
case e of
  Pat1 -> u
  Pat2 -> v
~~>
let j1 = u in
let j2 = v in
case e of
  Pat1 -> j1
  Pat2 -> j2
```

If there is any linear variable in u and v, then the standard let rule above will fail (since j1 occurs only in one branch, and so does j2).

However, if j1 and j2 were annotated with their usage environment,

6.5 Empty Case

For case expressions, the usage environment is computed by checking all branches and taking sup. However, this trick doesn't work when there are no branches.

- https://gitlab.haskell.org/ghc/ghc/-/issues/20058
- https://gitlab.haskell.org/ghc/ghc/-/issues/18768
- (1) Just like a case expression remembers its type (Note [Why does Case have a 'Type' field?] in Core.hs), it should remember its usage environment. This data should be verified by Lint.
- (2) Once this is done, we can remove the Bottom usage and the second field of UsageEnv. In this step, we have to infer the correct usage environment for empty case in the typechecker.

```
{-# LANGUAGE LinearTypes, EmptyCase #-}
module M where

{-# NOINLINE f #-}
f :: a %1-> ()
f x = case () of {}
```

This example is well typed: a function is linear if it consumes its argument exactly once when it's consumed exactly once. It seems like the function isn't linear since it won't consume x because of the empty case, however, that also means f won't be consumed due to the same empty case, thus linearity is preserved.

```
* In the case of empty types (see Note [Bottoming expressions]), say
  data T
we do NOT want to replace
  case (x::T) of Bool {}   -->   error Bool "Inaccessible case"
because x might raise an exception, and *that*'s what we want to see!
(#6067 is an example.) To preserve semantics we'd have to say
   x `seq` error Bool "Inaccessible case"
but the 'seq' is just such a case, so we are back to square 1.
```

There are three different problems:

- castBottomExpr converts (case x :: T of) :: T to x.
- Worker/wrapper moves the empty case to a separate binding
- CorePrep eliminates empty case, just like point 1 (See – Eliminate empty case in GHC.CoreToStg.Prep

With castBottomExpr, we get the example above to

```
f = \ @a (x (%1) :: a) -> ()
```

And if we don't,

```
f = \ @a (x (%1) :: a) -> case () of {}
```

And that supposedly if we had a usage environment in the case expression we could avoid the error. How is it typed without the transformation in face of the bottom? (Even knowing that theoretically it's because of divergence?)