Linting Linearity in Core/System FC

Rodrigo Mesquita

November 18, 2022

1 Introduction

1.1 Motivation

- Are we really preserving linearity?
- Inform/unlock other optimisations that take into account linearity

2 Background

- Linear types
- Linear Haskell
- Core
- Sistemas de inferência
- GADTs e Coercions

3 Related Work

4 Technical Details

Since Linear Haskell's**??** publication and implementation release in GHC 9.2, Haskell's type system supports linearity annotations in functions – bringing linear types into a mainstream pure and lazy functional language.

System FC is the formal system in which the implementation of GHC's intermediate representation language *Core* is based on.

There are at least two distinct typecheckers in core. The first is run on the frontend language, i.e. the Haskell we write, and is a big and complex typechecker. The second is run on the intermediate language *Core* that we obtain from desugaring Haskell.

Core is a much smaller and more principled language than the whole of Haskell (even though we can compile the whole of Haskell to it), and the typechecker for it is small and fast due to *Core* being explicitly typed and having a very small abstract syntax tree. This typechecker is called *Lint* and gives us guarantees of correctness (i.e. sanity) in face of the complexity of all the transformations a Haskell program undergoes, such as type inference, desugaring and optimising transformations (and other Core related passes?).

- In GHC, the type Type is shared between the surface language and Core. So modifying the types implies that Core is modified as well.

- More importantly, because Core is a good check that our implementation works as intended. That is, a linearly typed Core will ensure that linearly-typed programs are indeed desugared to linearly-typed programs in Core. And that optimisations do not destroy linearity.

The addition of linear types comes at two levels: frontend and Core. In the frontend, we can now annotate with linearity type declarations and typecheck programs that use them accordingly (a linear fuction will consume its argument exactly once if it is consumed exactly once, for some definition of *consume* that I should revise here). In Core, we still have linearity annotations and ideally *Lint* would check that all the GHC transformations to our linear program preserved its linearity. However, **it can't**!

Despite the strong formal foundations of linear types driving the implementation, their interaction with the whole of GHC is still far from trivial. Diverse problems with linearity spring up when we get past the desugarer. In particular, optimizing transformations, coercions from GADTs and type families, recursive lets, and empty case expressions don't currently fit in with linearity.

We believe that GHC's transformations are correct, and it is the linear type system that can't accommodate the resulting programs. We aim to formalise a type system that is able to type check *Core/System FC* at all points in the core pipeline and implement it into GHC to be able to preserve linearity accross the stages and to enable *Lint* to preserve our sanity regarding linearity.

5 Typing Usage Environments

The first set of problems appears in the core-to-core optimisation passes. GHC applies many optimising transformations to *Core* and we believe those transformations preserve linearity. However, our linear type system cannot check that they indeed preserve linearity.

The simpler examples come from straightforward and common optimising transformations. Then we have recursive let definitions that don't accommodate linearity even though it might converge to only use the value once. Finally, we have the empty case expression introduced with the *EmptyCase* language extension that we currently can't typecheck either.

A usage environment is a mapping from variables to multiplicities.

The key idea is to annotate every variable with either its multiplicity, if it's lambda bound, or with its usage environment, if it's let bound.

For example, if y and z are linear, in the following code it might not look as if y and z were 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 that let binding x doesn't consume y and z, only using x itself does, because of laziness.

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

If we annotate the x bound by 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, hence the linearity in the two branches match.

Currently, in GHC, we don't annotate let-bound variables with a usage environment, but we already calculate a usage environment and use it to check some things (which things?)

- occurrences? store usage environment in Ids (vars)
- Recursive lets (can it be rewritten using fix)
- Empty case expression

What do we currently do?

When we lint a core expression, we get both its type and its usage environment. That means that to lint linearity in an expression, whenever we come across a free variable we compute its usage environment and take it into account

5.1 Inlining

If we annotate the let bound variables with their usage and emit that usage when we come across those variables, we can solve the linearity issues with inlining.

5.2 Recursive Lets

Optimisations can create a letrec which uses a variable linearly. The following example uses 'x' linearly, but this is not seen by the linter.

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

Following the idea of making let-bound variables remember the usage environment here's an informal description that that example is well typed. We want to anotate the let-bound variable f with a usage environment delta δ, and use the binding body to compute it.

Now, how to compute the usage environment of a case expression? It's described here if I'm not mistaken https://gitlab.haskell.org/ghc/ghc/-/wikis/uploads/355cd9a03291a852a518b0

However, not understanding it, my take would be that we can compute the usage environment of every branch of the case expression and make sure that they all unify (wrt to submultiplicities). The special case is when the branch happens to call f itself while computing the usage environment of f. If it were another let bound variable we'd add its own usage environment to the one we're computing; if it were a lambda bound variable we'd add [itself := its multiplicity].

My idea is that we can emit a special usage $[rec := p]$, which, when unified against the other case branches δ will always succeed with a unification mapping from $rec \to p\delta$ scaled by the multiplicity of rec

So taking the example, to compute the usage environment of f, we'd compute for the second branch $[x := 1]$ and for the first branch $[rec := 1]$. Then, we'd unify them with $rec \to [x := 1 * 1]$, and somehow result in $[x := 1]$

An example which should break linearity because x is not linear in the first branch:

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

To compute the usage environment of f we take the second branch usage environment $[x := 1]$ and the first branch usage $[rec := 1 + 1]$ and unify them, somehow resulting in $[x := 2]$. Now, to lint the linearity in the whole let expression we must ensure that the body of the let uses x linearly. The body is $fTrue$, which is a let-bound variable (how to distinguish functions that must be applied vs variables) so we take its usage environment $[x := 2]$ which does not use x linearly and thus breaks linearity.

Re-explained: to compute the usage environment of the recursive let-bound function f when applied, we compute Z such that for all branch alternatives $U, V, ...,$ $U \subset Z$, $V \subset Z$ and so on by tracking the multiplicities and usage environments of variables that show up in the body and by emitting a special keyword rec everytime we find a saturated call of f (how to handle unsaturated calls?) (how to have a more general solution that doesn't require a keyword, perhaps something to do with fixed points); Then we scale $Z \setminus \{(rec,_) \}$ by $Z[rec]$ and get T which is the usage environment of f when saturated.

Example 1 revisited:

1. Take $U = \{rec := 1\}$

2. Take $V = \{x := 1\}$

3. Take $Z = \{x := 1, rec := 1\}$

4. Take $\pi = Z[rec] = 1$

5. Take $W = Z \setminus \{rec\} = \{x := 1\}$

6. Take $T = \pi W = \{x := \pi * 1\} = \{x := 1\}$

7. Linearity OK

Example 2 revisited:

1. Take $U = \{rec := 2\}$

2. Take $V = \{x := 1\}$

3. Take $Z = \{x := 1, rec := 1\}$

4. Take $\pi = Z[rec] = 2$

5. Take $W = Z \setminus \{rec\} = \{x := 1\}$

6. Take $T = \pi W = \{x := \pi * 1\} = \{x := 2\}$
7. Linearity not OK

Draft typing rule:

$$\frac{\Gamma; \Delta/\Delta'; \Omega \vdash M : A \Uparrow \qquad \Gamma; \Delta/\Delta''; \Omega \vdash N : B \Uparrow \qquad \Delta' = \Delta''}{\Gamma; \Delta/\Delta'; \Omega \vdash (M \mathbin{\&} N) : A \mathbin{\&} B \Uparrow} \ (\textsc{letrec})$$

5.3 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?)

6 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?).

When we pattern match on a GADT we ...

Taking this example (copy example over) 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
```

7 Typing After Transformations

7.1 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,

8 Novel rules

8.1 The usage environment in a Recursive Let

8.2 The usage environment of a Case Binder

$$\frac{\Gamma;\Delta/\Delta';\Omega \vdash M : A \Uparrow \qquad \Gamma;\Delta/\Delta'';\Omega \vdash N : B \Uparrow \qquad \Delta' = \Delta''}{\Gamma;\Delta/\Delta';\Omega \vdash (M \mathbin{\&} N) : A \rightsquigarrow U}\ (\textsc{letrec})$$

9 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.

9.1 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* $\not\subseteq$ *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 8.2) 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 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 }
```

9.2 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. In concrete, 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
  }
```

9.3 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 the following core which violates linearity.

```
groupBy = \ (@a) (eq :: a -> a -> Bool) (eta :: Infinite a) ->
  letrec {
    go :: Infinite a -> Infinite (NonEmpty a)
    go = \ (inf :: Infinite a) -> case inf of {
      Inf x xs -> let {
        ds :: ([a], Infinite a)
        ds = let {
            parteq :: a -> Bool
            parteq = eq x
        } in
    letrec {
     goZ :: Infinite a -> ([a], Infinite a)
     goZ = \ (inf’ :: Infinite a) -> case wgo inf’ of { (# wA, wB #) -> (wA, wB) };

     wgo :: Infinite a -> (# [a], Infinite a #)
     wgo = \ (inf’ :: Infinite a) -> case inf’ of wildX2 {
      Inf y ys ->
       join {
         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 #) } }
         } in
          case parteq y of {
           False ->
             let {
               ww :: ([a], Infinite a)
               ww = ([] @a, wildX2)
             } in jump jp ww;
           True ->
             let {
               dy :: ([a], Infinite a)
               dy = case wgo ys of { (# wA, wB #) -> (wA, wB)
               }
             } in
               let {
                 wwB :: [a]
                 wwB = case dy of { (bs, cs) -> bs }
               } in
                 let {
                   wwA :: [a]
                   wwA = : @a y wwB
                 } in let {
                   wwC :: Infinite a
                   wwC = case dy of { (bs, cs) -> cs }
                 } in let {
                   ww :: ([a], Infinite a)
                   ww = (wwA, wwC)
                 } in jump jp ww
               }
      };
    } in
      case wgo xs of { (# wA, wB #) -> (wA, wB) }
  } in
    Inf @(NonEmpty a) (:| @a x (case ds of { (bs, cs) -> bs })) (case ds of { (bs, cs) -> go c
 };
} in go eta
```

The issue is in the linear function that shows up in the core output (how does the linearity end up there?). The wwj variable is used once in the case expression, bound as the case binder, and used in the case body once again. Why do we do that instead of pattern matching right away? Seems a bit redundant.

Observation: If the branch is DEFAULT, then the case binder binds the case scrutinee which was just forced?, but hasn’t been actually consumed, because we haven't consumed its components. As long as the same branch doesn't consume the pattern matching result and the case binder at the same time it should be fine?

In this example, we force wwj with the case binder, but we don't really consume it (more precise definition of consume...), so we can use the case binder meaning we're simply using x for the first time. Needs to be formalised....

```
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 #) }
    }
```

Solution with the case binder idea that unifies a different problems: The case binder is annotated with a usage environment $U_w wj = []$, and its usage is equal to using the usage environment

Here it would mean that second **case of** wwj doesn't actually use any variables and hence it isn't used twice.

9.4 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.")
```

9.5 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 put 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?

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