# Linting Linearity in Core/System FC

Rodrigo Mesquita

November 8, 2022

## 1 Introduction

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.

# 2 Motivation

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

# 3 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 $\delta$ 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?)

- Transformations, occurrences?
- Recursive lets
- Empty case expression

### 3.1 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,

### 3.2 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 $\delta$, 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 $\delta$ 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.

### 3.3 Empty Case

- https://gitlab.haskell.org/ghc/ghc/-/issues/20058
- https://gitlab.haskell.org/ghc/ghc/-/issues/18768

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

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

## 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?).

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 (...)

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

```
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
```