Lazy Linearity for a Core Functional Language

RODRIGO MESQUITA, Well-Typed LLP, United Kingdom
BERNARDO TONINHO, Instituto Superior Técnico, University of Lisbon, Portugal and INESC-ID, Portugal

Traditionally, in linearly typed languages, consuming a linear resource is synonymous with its syntactic occurrence in the program. However, under the lens of non-strict evaluation, linearity can be further understood semantically, where a syntactic occurrence of a resource does not necessarily entail using that resource when the program is executed. While this distinction has been largely unexplored, it turns out to be inescapable in Haskell's optimising compiler, which heavily rewrites the source program in ways that break syntactic linearity but preserve the program's semantics. We introduce Linear Core, a novel system which accepts the lazy semantics of linearity statically and is suitable for lazy languages such as the Core intermediate language of the Glasgow Haskell Compiler. We prove that Linear Core is sound, guaranteeing linear resource usage, and that multiple optimising transformations preserve linearity in Linear Core while failing to do so in Core. We have implemented Linear Core as a compiler plugin to validate the system against linearity-heavy libraries, including `linear-base`.

CCS Concepts: • **Theory of computation** → **Linear logic**; **Type theory**; • **Software and its engineering** → **Functional languages**; **Compilers**.

Additional Key Words and Phrases: Linear Types, Lazy Evaluation, Haskell, GHC Core

ACM Reference Format:
Rodrigo Mesquita and Bernardo Toninho. 2026. Lazy Linearity for a Core Functional Language. *Proc. ACM Program. Lang.* 10, POPL, Article 69 (January 2026), 27 pages. https://doi.org/10.1145/3776711

1 Introduction

Linear types [Barber 1996; Girard 1987] increase the expressiveness of type systems by guaranteeing that certain resources are used *exactly once*. In programming languages with a linear type system, discarding so-called linear resources, or using them twice, is flagged as a type error. Linear types can be used to, for instance, statically enforce correct usage of file descriptors, that heap-allocated memory is freed exactly once s.t. leaks and double-frees become type errors, or guarantee deadlock freedom in channel-based communication protocols [Caires and Pfenning 2010], among other correctness properties [Bernardy et al. 2017a; Fu et al. 2020; Li et al. 2022]. The following program with a runtime error (allocated memory is freed twice) is accepted by a C-like type system. Conversely, under the lens of a linear type system where p is deemed a linear resource created by the call to `malloc`, the program is rejected:

```
let p = malloc(4) in free(p); free(p);
```

Despite their promise and extensive presence in research literature [Andreoli 1992; Cervesato et al. 2000; Wadler 1990], an effective design combining linear and non-linear typing is both challenging and necessary to bring the advantages of linear typing to mainstream languages. Consequently, few

Authors' Contact Information: Rodrigo Mesquita, Well-Typed LLP, London, United Kingdom, rodrigo@well-typed.com; Bernardo Toninho, Instituto Superior Técnico, University of Lisbon, Lisbon, Portugal and INESC-ID, Lisbon, Portugal, bernardo.toninho@tecnico.ulisboa.pt.

ACM 2475-1421/2026/1-ART69
https://doi.org/10.1145/3776711

general purpose programming languages have linear or substructural type systems. Among them are Idris 2 [Brady 2021], Rust [Matsakis and Klock 2014], whose ownership types build on linear types to guarantee memory safety without garbage collection, and, more recently, Haskell [Bernardy et al. 2017b], a pure, functional, and *lazy* general purpose language.

Linearity in Haskell stands out from linearity in other languages because linear types permeate Haskell down to Core, the intermediate language into which Haskell is translated by the Glasgow Haskell Compiler (GHC). Core is a minimal, explicitly typed, pure lazy functional language with both linear and unrestricted types, to which multiple Core-to-Core optimising transformations are applied during compilation. Notably, Core is typechecked after each transformation. This serves as a sanity check to the correction of the implementation of such transformations, since unsound ones are likely to introduce ill-typed expressions.

Just as Core's type system [Sulzmann et al. 2007] provides a degree of validation to the translation from Haskell, dubbed *desugaring*, and the subsequent optimising transformations, a linearly typed Core would guarantee that linear resource usage in the source language is not violated by desugaring and the compiler optimisation passes. Moreover, linearity information in Core can inform Core-to-Core optimising transformations [Bernardy et al. 2017b; Peyton Jones and Partain 1996; Peyton Jones and Santos 1997], such as inlining and β-reduction, to produce more efficient programs. However, despite the advantages of having linear types in Core, the status quo is that linearity is effectively ignored in Core, being only checked in the source Haskell code. The reason is that, after desugaring, the Core-to-Core optimising transformations eliminate and rearrange most of the syntactic constructs through which linearity checking is performed – often resulting in programs that are syntactically very different from the original, especially due to the flexibility provided by laziness with regard to the optimisations a compiler may perform.

While compiler optimisations aim to preserve the semantics of programs and therefore a program's resource consumption, a naive syntactic analysis of the optimised programs as performed by the current Core type checker often fails to recognize resulting programs as linear. As a trivial example, let x be a linear resource in the next two expressions, where the latter results from inlining y without eliminating the binder – a common optimising transformation in GHC. Even though the second expression no longer appears linear (as there are now two occurrences of x), it is indeed linear because the let-bound expression is never evaluated under non-strict evaluation and x is still consumed exactly once (when it is freed in the let body):

$$\textbf{let } y = \mathsf{free}\ x \textbf{ in } y \quad \Longrightarrow_{Inlining} \quad \textbf{let } y = \mathsf{free}\ x \textbf{ in } \mathsf{free}\ x$$

The Core optimising transformations expose a fundamental limitation of Core's linear type system: it does not account for the non-strict evaluation model of Core and, thus, a whole class of programs that are linear under the lens of non-strict evaluation are rejected by Core's current checker. In this work, we address this limitation by exploring how syntactic linearity breaks down in the presence of non-strictness and design a linear type system for Core which accepts the vast majority of programs and optimising transformations that, while correct under non-strict evaluation, are rejected by traditional linear typing disciplines. Additionally, we show that our system is suitable for the intermediate language of an optimising compiler by implementing it as a Glasgow Haskell Compiler plugin. Our contributions are as follows:

- We illustrate the lazy semantics of linearity, in contrast to the traditional syntactic understanding of linearity, in Haskell, by example (Section 2). A precise semantic definition of linearity, closely following that of Linear Haskell [Bernardy et al. 2017a], is given by our instrumented operational semantics under which a program using a linear resource non-linearly gets stuck (Section 4).

- We introduce Linear Core (Section 3), a lazy linear language with the key features of Core (except for type equality coercions), whose type system is designed to accept the lazy semantics of linearity. Our system achieves this via a novel combination of features: explicit tracking of captured linear resources through *usage environments* (Section 3.3); a distinct treatment of case scrutinees in weak-head normal form (Section 3.6.1); and a notion of irrelevant and tagged resources (Section 3.6.2), which allow for fine-grained control over variable usage. To the best of our knowledge, this is the first type system to accept programs which depend on non-strict evaluation to be understood as linear.
- We prove that our type system is sound, guarantees linear resource usage, and prove that multiple optimising transformations, which are rejected in Core, are validated by Linear Core (Section 4).
- We discuss our implementation of Linear Core as a GHC plugin which checks linearity in all intermediate Core programs produced during the compilation process, showing it accepts the programs resulting from transformations in libraries such as `linear-base` (Section 5).

While GHC's Core language is our main focus, we note that the ideas explored in this work can potentially be applied to any language with non-strict features, even if the language itself is strict. We assume some familiarity with Haskell and linear types for functional languages, but try to explain our work and motivations without assuming much knowledge of GHC and Core.

Additional proofs and definitions are given in the extended version of the paper [Mesquita and Toninho 2025a].

2 Linearity, Semantically

Linear type systems guarantee that linear resources are consumed exactly once. Thus, whether a program is accepted as linear intrinsically depends on the definition of consuming a resource. Traditionally, consuming a resource has been equated with its syntactic occurrence along a control-flow path in the program. However, as this section aims to make clear, in a non-strict context, a syntactic definition of linearity is overly conservative. For instance, in the following Haskell-like program, f is a linear function (a function that must use its argument exactly once), but there are two syntactic occurrences of handle: one is in a let-bound computation that closes this handle, the other is in the then branch of a conditional. If the program is evaluated lazily, the handle will, dynamically, either be used exactly once (i.e, closed) in the else branch, or be used exactly once (i.e., returned) in the then branch, since the let-bound variable is not needed in that branch and so the expression which closes the handle is never evaluated.

```
f : Handle ⊸ Handle
f handle =
    let closed = close handle in
    if <cond>
        then handle
        else closed
```

Intuitively, a computation that depends on a linear resource to produce some result consumes that resource iff the result is fully consumed; in contrast, a computation that depends on a linear resource, but is never run, will not consume that resource. We argue that a linear type system for a language with non-strict evaluation semantics should accept the above program, unlike a linear type system for the same program if it were to be evaluated eagerly, where the let-bound expression would always be evaluated. Given this observation, we turn our focus to linearity under lazy evaluation specifically since the distinction between semantically and syntactically consuming a resource is only exposed under non-strict semantics, and because GHC Core is lazy.

2.1 Semantic Linearity by Example

This section aims to develop an intuition for semantic linearity through simple Core programs that can be understood as linear but are rejected by Core's linear type system (or, in fact, by all linear type systems we are aware of prior to this work). In the examples, a light green background highlights programs that are syntactically linear and accepted by Core's linear type system. A light yellow or light orange background mark programs that are semantically linear, but not seen as linear by Core's type system. Our system accepts all light yellow programs, but not the light orange examples. A light red background indicates that the program violates linearity semantically, i.e. the program effectively discards or duplicates linear resources.

Let Bindings. The example at the start of Section 2 highlighted the subtlety of linearity in a non-strict language with a single unused let binding. In general, a let-bound expression that captures a linear variable can be conditionally needed at runtime, depending on the branch taken. Both optimising transformations (float-out) and programmers used to non-strict evaluation are keen to write programs with bindings that are selectively used in case alternatives, such as in the following program. Note that we write ⊸ for linear functions and -> for non-linear functions:

```
f :: (a ⊸ a) -> Bool -> a ⊸ a
f use bool x = let y = use x in
  case bool of
    True -> x
    False -> y
```

This example returns x in one branch and the let-bound y in the other. Semantically, as opposed to syntactically, this program is linear because the linear resource x is used exactly once regardless of which branch is taken: either directly in the **True** branch, or *via* the use of the let-bound y in the **False** branch.

That said, a let-bound y that captures a linear resource x must still be used exactly once, regardless of whether evaluation inlines the let (*à la* call-by-name) or creates a thunk (*à la* call-by-need):

```
f1 :: (a ⊸ b) -> a ⊸ (b, b)
f1 use x = let y = use x in (y, y)
```

If y is inlined, linearity is trivially violated. On the other hand, if y becomes a thunk, use x will only be evaluated once regardless of how many times y is used. Even so, this program must be rejected under call-by-need. Intuitively, linearity is violated if the computation leaks the resource or parts of it. For instance, if use = **id**, the program must clearly be rejected.

Lastly, consider an ill-typed program which defines two let bindings z and y, where z uses y which in turn uses the linear resource x:

```
f2 :: (a ⊸ a) -> a ⊸ ()
f2 use x = let y = use x in let z = use y in ()
```

Even though y occurs in z, x is never consumed because z is discarded. This example highlights that naively checking for syntactic occurrences of let-bound variables such as y is also inadequate to track resource usage. A better mental model is that using y implies using x, and thus using z implies using x too. If neither z nor y are used, then x is effectively discarded.

In essence, an unused let binding doesn't consume any resources, and a let binding used exactly once consumes the resources it captures exactly once. Let binders that depend on linear resources must be used *at most once* in mutual exclusion with the linear resources themselves (in a sense, let-bound variables are affine in the let body). We discuss how to encode this principle between let bindings and their dependencies using so called *usage environments*, in Section 3.3.

Recursive Let Bindings. Recursive let-bindings behave similarly to non-recursive lets as far as their usage in the let continuation is concerned. The challenge lies in understanding linearity in the definitions of the recursive binders and determining which resources are consumed when one of the binders in the recursive group is used. Consider the following program, which calls a recursive let-bound function defined in terms of the linear resource x and itself:

```
f3 :: Bool -> a ⊸ a
f3 bool x = let go b = case b of
                  True -> x
                  False -> go (not b)
                in go bool
```

Function f3 is semantically linear because, iff it is consumed exactly once, then x is consumed exactly once. We can see this by case analysis on go's argument: when bool is **True**, we'll use the resource x; when bool is **False**, we recurse by calling go on **True**, which in turn will use the resource x. In go's body, x is used directly in one branch and indirectly in the other (by recursing, which we know will result in using x linearly).

It so happens that go will terminate on any input, and will always consume x. However, termination is not a requirement for a binding to use a resource linearly, and we could have a similar example in which go might never terminate but still uses x linearly if evaluated:

```
f4 :: Bool -> a ⊸ a
f4 bool x = let go b = case b of
                  True -> x
                  False -> go b
                in go bool
```

The key to linearity in the presence of non-termination is Linear Haskell's definition of a linear function: *if a linear function application (f u) is consumed exactly once, then the argument (u) is consumed exactly once.* If f u doesn't terminate, it is never consumed, thus the claim holds vacuously. If go doesn't terminate, we aren't able to compute (or consume) the result of f4, so we do not promise anything about x being consumed (f4's linearity holds trivially). If it did terminate, it would consume x exactly once (e.g. if go was applied to **True**).

Naturally, a linear type system must statically reject any recursive group that could lead to non-linear resource usage. For instance, if the linear resource x is used in one case alternative and recursion is used more than once in another, x may end up being consumed more than once:

```
f5 :: Bool -> Bool ⊸ Bool
f5 bool x =
  let go b
        = case b of
            True -> x
            False -> go (not b) && go True
  in go bool
```

To determine whether the usage of a binding from a recursive group is linear, we must check the superset of resources that may end up being consumed in a single use of any of the bindings. For a strongly connected recursive group, this superset is the same for all bindings by definition. It follows from the resources used being the same for all bindings that at most a single variable from the recursive group can be used in the continuation. The treatment of recursive let bindings in our system is discussed in Section 3.5.

Case Expressions. We have observed that more programs can be accepted as linear by delaying resource consumption in (recursive) let bindings until they are evaluated. On the other hand, *case expressions drive evaluation* – and, therefore, do consume resources.

To understand exactly how, we turn to the definition of consuming a resource from Linear Haskell [Bernardy et al. 2017b]: to consume a value of atomic base type (such as `Int` or `Ptr`) exactly once, just evaluate it; to consume a function value exactly once, apply it to one argument, and consume its result exactly once; to consume a value of an algebraic datatype exactly once, pattern-match on it, and consume all its linear components exactly once. Thus, we can consume a linear resource by fully evaluating it, and evaluation happens with case expressions.

In Core, cases are of the form **case** e_s **of** z $\{\overline{\rho_i \longrightarrow e_i}\}$, where e_s is the case *scrutinee*, z is the case *binder*, and $\overline{\rho_i \longrightarrow e_i}$ are the case *alternatives*, composed of a pattern ρ_i and of the corresponding expression e_i. The case scrutinee is always evaluated to Weak Head Normal Form (WHNF) and the case binder is an alias to the result of evaluating the scrutinee to WHNF. Moreover, in Core, the alternative patterns are always exhaustive. The first example program constructs a tuple from linear resources, matches on it, then uses both linearly-bound variables from the tuple pattern. This is well-typed in Linear Haskell:

```
f6 :: a ⊸ b ⊸ (a ⊸ b ⊸ c) -> c
f6 x y use = case (x,y) of z { (a,b) -> use a b }
```

Perhaps surprisingly, a similar program which discards the pattern variables, and instead uses the resources from the scrutinee again, is still linear, despite not being accepted by Linear Haskell:

```
f7 :: a ⊸ b ⊸ (a ⊸ b ⊸ c) -> c
f7 x y use = case (x,y) of z { (a,b) -> use x y }
```

f7 can be seen as linear by realizing that the tuple being scrutinized is already in WHNF, so evaluating it again does not consume either x nor y. Even if the tuple paired two arbitrary expressions which captured x and y, because a and b are not forced in the continuation, x and y would remain unused. On the other hand, as seen in the next example, if a were used in the case body, then x is consumed via a and must not be used again. Thus, f8 must be rejected:

```
f8 :: a ⊸ b ⊸ (a ⊸ a ⊸ c) -> c
f8 x y use = case (x,y) of z { (a,b) -> use a x }
```

This idea that x and a are mutually exclusive is akin to how let bindings are mutually exclusive with the set of resources they capture, in the let-continuation. Forcing a pattern variable evaluates the expression for the corresponding constructor field, and all linear variables captured by that expression may end up being used. A third option, semantically linear, but rejected by Linear Haskell, is to use the case binder z instead of a, b or x, y:

```
f9 :: a ⊸ b ⊸ (a ⊸ b ⊸ c) -> c
f9 x y use = case (x,y) of z { (a,b) -> uncurry use z }
```

Again, z is mutually exclusive with (a, b) and with (x, y), but at least one of the three must be used to ensure the linear resources are consumed. Essentially, in this example, using a entails using the resource x, b the resource y, and the case binder z entails using both a and b.

Conversely, if the scrutinee is an expression that's not in WHNF, evaluation to WHNF may indeed consume some or all of the linear resources captured by said scrutinee. Consider the following non-WHNF scrutinee example, f10, which is accepted by our system:

```
f10 :: a ⊸ b ⊸ (a ⊸ b ⊸ (c,d)) -> (c,d)
f10 x y use = case use x y of z { (a,b) -> z }
```

In contrast to a scrutinee in WHNF, the resources captured by a non-WHNF scrutinee (here, x, y), conservatively, can no longer be used in the case alternatives, since reducing `use x y` to WHNF may consume (parts of) x and/or y. Consequently, either the case binder z or the linear pattern variables a, b must be used exactly once, since the linear resources in the scrutinee are only fully consumed if the result of evaluating the scrutinee is also fully consumed . For instance, take `use` in `f10` to be the pairing function `(,)` : it is not sufficient for `use x y` to be evaluated to WHNF to consume `x` and `y`. If all the resources were considered to be fully consumed after evaluating the scrutinee to WHNF, the pattern variables could simply be ignored and thus linear resources potentially discarded. In practice, if the scrutinee is not in WHNF then either the case binder or the linear components of the pattern must be consumed.

Alternatively, if a constructor without any linear components is matched, all linear resources used in the scrutinee must already have been fully consumed in that branch since the result of evaluating it can be consumed unrestrictedly (by definition, a constructor application to unrestricted arguments is also unrestricted). Hence, in the case-branch for a constructor without linear fields, the case binder can also be used unrestrictedly. For example, the program in Figure 1 is semantically linear. A second example of an unrestricted pattern, where K2 has no fields and K1 has one linear field, is shown in Figure 2. The use of `x` in the K2 branch is valid despite `x` also occurring in the scrutinee because this branch is never taken. In fact, any arbitrary resource could be used in absurd branches and be understood as linear (despite not being seen as such by our system).

```
f11 :: () ⊸ ()
f11 x = case x of z { () -> z <> z }
```

Fig. 1. No Linear Fields

```
f12 :: a ⊸ a
f12 x = case K1 x of z { K2 -> x; K1 a -> x }
```

Fig. 2. Absurd Branches

While many of these examples may seem artificial due to the pattern match on a known constructor, we note that in between the transformations programs undergo in an optimising compiler, many such programs naturally occur (e.g., if the definition of a function is inlined in the scrutinee).

As for *default* case alternatives, also known as *wildcards* (written _): matching a wildcard doesn't provide any linearity information, but the scrutinee is still evaluated to WHNF. When matching a wildcard, if the scrutinee is already in WHNF, programs that either use the resources from the scrutinee directly, or the case binder in that alternative, can be seen as linear. If the scrutinee is not in WHNF, we *must* use the case binder, as it's the only way to linearly consume the result of evaluating the scrutinee.

3 A Type System for Semantic Linearity in Core

In this section, we develop a pure linear calculus λ^{π}_{Δ}, dubbed *Linear Core*, that combines the linearity-in-the-arrow and multiplicity polymorphism introduced by Linear Haskell [Bernardy et al. 2017b] with all the key features of GHC Core: algebraic datatypes, case expressions, and recursive lazy let bindings; except for equality coercions, which we discuss as future work in Section 6.1.

Linear Core makes precise the various insights discussed in the previous section in a linear type system that we prove to be sound and guarantee linear resource usage at runtime. This result is obtained (see Section 4) via a soundness argument on a call-by-need big-step operational semantics that becomes stuck when linear variables in the runtime heap are used more than once, a technical approach that is natural in the literature [Bernardy and Spiwack 2021; Pierce 2004]. We show this semantics to be bisimilar to a Launchbury-style [Launchbury 1993] natural semantics. Crucially, Linear Core typing accepts all the semantically linear examples (highlighted with light yellow) from Section 2.1, which Core currently rejects. Additionally, we prove that multiple optimising Core-to-Core transformations preserve linearity in Linear Core, while violating it under Core's current type system. Despite the focus on GHC Core, the fundamental ideas for typing linearity in our call-by-need calculus may be applied to other languages which combine non-strict features or semantics with linearity.

The key ideas that make Linear Core work are:

(1) Usage environments (Section 3.3)
(2) Distinct treatment of case scrutinees in WHNF (Section 3.6.1)
(3) Irrelevant and tagged resources (Sections 3.6.2 and 3.6.3)

(1) Encodes the idea that lazy bindings do not consume resources upon definition, but rather when the bound variables themselves are consumed. Consuming a variable with some usage environment Δ equates to consuming the variables contained in Δ. (2) Case expressions in WHNF may capture ambient linear resources, but these resources can be safely used in branches since the case will not further evaluate its scrutinee. On the other hand, scrutinizing a non-WHNF expression must be treated more conservatively. (3) Irrelevant resources make existing bindings unusable while forcing the use of others. Namely, linear resources occurring in non-WHNF case scrutinees can no longer be used in the alternatives, while the pattern variables (or the case binder) must necessarily be used. Tagged resources follow the observation that some variables must be used jointly or not at all.

The reader may wonder whether the complexities of our system hinted above are a better alternative to simply specializing the optimisation passes to produce code that passes a more naive linearity check. This avenue was already explored during the implementation of Linear Haskell in GHC, exhibiting the prohibitive cost of too often restricting optimisations and producing less efficient code. Our work adheres to the principle that the type system bends to the optimisations, not the other way around[1].

We present Linear Core's syntax and type system incrementally, starting with the judgements and base linear calculi rules (Section 3.2). Then, usage environments, the rule for Δ-bound variables, and rules for (recursive) let bindings (Section 3.3). Finally, we introduce the rules to type case expressions and case alternatives, along with the key insights to do so, namely discriminating by the WHNF-ness of the scrutinee, irrelevant resources, and tagging (Section 3.6).

3.1 Language Syntax

The complete syntax of Linear Core is given in Figure 3. The types of Linear Core are algebraic datatypes, function types, and multiplicity schemes to support multiplicity polymorphism: datatypes ($T\ \overline{p}$) are parametrised by multiplicities, function types ($\varphi \to_\pi \sigma$) are also annotated with a multiplicity, which can be 1, ω (read *many*), or a multiplicity variable p introduced by a multiplicity universal scheme ($\forall p.\ \varphi$).

[1] https://gitlab.haskell.org/ghc/ghc/-/blob/c85c845dc5ad539bf28f1b8c5c1dbb349e3f3d25/compiler/GHC/Core/Lint.hs#L3225-3235

Terms are variables x, y, z, data constructors K, multiplicity abstractions $\Lambda p.\ e$ and applications $e\ \pi$, term abstractions $\lambda x{:}_\pi\tau.\ e$ and applications $e\ e'$, where lambda binders are annotated with a multiplicity π and a type σ. Then, there are non-recursive let bindings $\textbf{let}\ x{:}_\Delta\sigma = e\ \textbf{in}\ e'$, recursive let bindings $\textbf{let rec}\ \overline{x{:}_\Delta\sigma = e}\ \textbf{in}\ e'$, where the overline denotes a set of distinct bindings $x_1{:}_\Delta\sigma_1 \ldots x_n{:}_\Delta\sigma_n$ and associated expressions $e_1 \ldots e_n$, and case expressions $\textbf{case}\ e\ \textbf{of}\ z{:}_\Delta\sigma\ \{\overline{\rho \Rightarrow e'}\}$, where z is the case binder and the overline denotes a set of distinct patterns $\rho_1 \ldots \rho_n$ and corresponding right hand sides $e'_1 \ldots e'_n$. We often omit the case binder z when it is unused. Notably, (recursive) let-bound binders and case-bound binders are annotated with a so-called *usage environment* Δ – a fundamental construct for typing semantic linearity in the presence of laziness that we present in Section 3.3. Case patterns ρ can be either the *default/wildcard* pattern _, which matches any expression, or a constructor K and a set of variables that bind its arguments, where each field of the constructor has an associated multiplicity denoting which pattern-bound variables must be consumed linearly or unrestrictedly. Additionally, the set of patterns in a case expression is assumed to be exhaustive, i.e. there is always at least one pattern which matches the scrutinized expression.

Datatype declarations $\textbf{data}\ T\ \overline{p}\ \textbf{where}\ \overline{K : \overline{\sigma \to_\pi} T\ \overline{p}}$ have the name of the type being declared T parametrized over multiplicity variables $\overline{p}$, and a set of the data constructors K with signatures indicating the type and multiplicity of the constructor arguments. Note that a linear resource is used many times when a constructor with an unrestricted field is applied to it, since, dually, pattern matching on the same constructor with an unrestricted field allows it to be used unrestrictedly.

Types

τ, σ	$::=$	$T\ \overline{p}$	Datatype
	$\mid$	$\tau \to_\pi \sigma$	Function with multiplicity
	$\mid$	$\forall p.\ \tau$	Multiplicity universal scheme

Terms

e	$::=$	$x, y, z \mid K$	Variables and data constructors
	$\mid$	$\Lambda p.\ e \ \mid\ e\ \pi$	Multiplicity abstraction/application
	$\mid$	$\lambda x{:}_\pi\tau.\ e \ \mid\ e\ e'$	Term abstraction/application
	$\mid$	$\textbf{let}\ x{:}_\Delta\sigma = e\ \textbf{in}\ e'$	Let with usage environment
	$\mid$	$\textbf{let rec}\ \overline{x{:}_\Delta\sigma = e}\ \textbf{in}\ e'$	Recursive Let with usage environment
	$\mid$	$\textbf{case}\ e\ \textbf{of}\ z{:}_\Delta\tau\ \{\overline{\rho \Rightarrow e'}\}$	Case
ρ	$::=$	$K\ \overline{x{:}_\pi\tau} \mid \_$	Pattern and wildcard

Environments

Γ	$::=$	$\cdot \mid \Gamma, x{:}_\omega\tau \mid \Gamma, K{:}\tau \mid \Gamma, p \mid \Gamma, z{:}_\Delta\tau$	Unrestricted typing environment
Δ	$::=$	$\cdot \mid \Delta, x{:}_\pi\tau \mid \Delta, [x{:}_\pi\tau]$	Linear typing environment

Multiplicities **Declarations**

$\pi\ ::=\ 1 \mid \omega \mid p$ $\qquad$ $decl ::= \textbf{data}\ T\ \overline{p}\ \textbf{where}\ \overline{K : \overline{\tau \to_\pi} T\ \overline{p}}$

Fig. 3. Linear Core Syntax

3.2 Base Typing System

Linear Core (λ^π_Δ) is a linear lambda calculus akin to Linear Haskell's $\lambda^q_\to$ in that both have multiplicity polymorphism, (recursive) let bindings, case expressions, and algebraic data types. We use the π superscript to stand for multiplicity and the Δ subscript to emphasize usage environments, a key feature of our work. λ^π_Δ diverges from $\lambda^q_\to$ primarily when typing lets, case expressions, and alternatives. The core rules of the calculus for abstraction and application are similar to that of $\lambda^q_\rightarrow$ and mostly standard. We note that we handle multiplicity polymorphism differently from Linear Haskell by effectively ignoring the multiplicity semiring and treating all multiplicity polymorphic variables parametrically, for the sake of simplicity. Such variables cannot be duplicated in our system. The full type system is given in Figure 4, with auxiliary judgements given in Figure 5.

$$\boxed{\Gamma; \Delta \vdash e : \tau}$$

$$\frac{\Gamma, p; \Delta \vdash e : \tau \qquad p \notin \Gamma}{\Gamma; \Delta \vdash \Lambda p.\ e : \forall p.\ \tau}\ (\Lambda I) \qquad \frac{\Gamma; \Delta \vdash e : \forall p.\ \tau \qquad \Gamma \vdash_{mult} \pi}{\Gamma; \Delta \vdash e\ \pi : \tau[\pi/p]}\ (\Lambda E)$$

$$\frac{\Gamma; \Delta, x{:_1}\tau \vdash e : \sigma \qquad x \notin \Delta}{\Gamma; \Delta \vdash \lambda x{:_1}\tau.\ e : \tau \multimap_1 \sigma}\ (\lambda I_1) \qquad \frac{\Gamma, x{:_\omega}\tau; \Delta \vdash e : \varphi \qquad x \notin \Gamma}{\Gamma; \Delta \vdash \lambda x{:_\omega}\tau.\ e : \sigma \multimap_\omega \varphi}\ (\lambda I_\omega)$$

$$\frac{\Delta = \Delta'}{\Gamma, x{:_\Delta}\sigma; \Delta' \vdash x : \tau}\ (\textsc{Var}_\Delta) \qquad \overset{(\textsc{Split})}{\frac{\Gamma; \Delta, x :_1 \tau \vdash e : \tau \qquad K \text{ has } n \text{ linear arguments}}{\Gamma; \Delta, \overline{x{:_1}\tau\#K_i}^{n} \vdash e : \sigma}}$$

$$\frac{\rho \in \Gamma}{\Gamma; x :_\rho \tau \vdash x : \tau}\ (\textsc{Var}_p) \qquad \frac{}{\Gamma, x{:_\omega}\tau; \cdot \vdash x : \tau}\ (\textsc{Var}_\omega) \qquad \frac{\Gamma; \Delta \vdash e : \tau \multimap_1 \sigma \qquad \Gamma; \Delta' \vdash e' : \tau}{\Gamma; \Delta, \Delta' \vdash e\ e' : \sigma}\ (\lambda E_1)$$

$$\frac{}{\Gamma; x{:_1}\tau \vdash x : \tau}\ (\textsc{Var}_1) \qquad \frac{\Gamma; \Delta \vdash e : \tau \multimap_\omega \sigma \qquad \Gamma; \cdot \vdash e' : \tau}{\Gamma; \Delta \vdash e\ e' : \sigma}\ (\lambda E_\omega)$$

$$\frac{\overline{\Gamma; \cdot \vdash e_\omega : \tau_i} \qquad \overline{\Gamma; \Delta_j \vdash e_j : \tau_j} \qquad \overline{\Delta_j} = \Delta \qquad K : \overline{\tau_i \multimap_\omega}\ \overline{\tau_j \multimap} \sigma}{\Gamma; \Delta \vdash K\ \overline{e_\omega e_i} : \sigma}\ (\textsc{Constr})$$

$$\frac{\Gamma; \Delta \vdash e : \tau \qquad \Gamma, x{:_\Delta}\tau; \Delta, \Delta' \vdash e' : \sigma}{\Gamma; \Delta, \Delta' \vdash \textbf{let}\ x{:_\Delta}\tau = e\ \textbf{in}\ e' : \sigma}\ (\textsc{Let}) \qquad \frac{\overline{\Gamma, \overline{x_i{:_\Delta}\tau_i}; \Delta \vdash e_i : \sigma} \qquad \Gamma, \overline{x_i{:_\Delta}\tau_i}; \Delta, \Delta' \vdash e' : \sigma}{\Gamma; \Delta, \Delta' \vdash \textbf{let rec}\ \overline{x_i{:_\Delta}\tau_i = e_i}\ \textbf{in}\ e' : \sigma}\ (\textsc{LetRec})$$

$$\overset{(\textsc{Case}_{\text{WHNF}})}{\frac{\begin{array}{c}\text{e is in } \textit{WHNF} \qquad e \text{ matches } \rho_j \qquad \overline{\Gamma, z{:_{[\Delta]}}\tau; [\Delta], \Delta' \vdash \rho \Rightarrow_{\text{NWHNF}} e' :^{z}_{[\Delta]} \sigma} \\ \Gamma; \Delta \Vdash e : \tau \gtrdot \overline{\Delta_i} \qquad \Gamma, z{:_{\overline{\Delta_i}}}\tau; \overline{\Delta_i}, \Delta' \vdash \rho_j \Rightarrow e' :^{z}_{\overline{\Delta_i}} \sigma\end{array}}{\Gamma; \Delta, \Delta' \vdash \textbf{case}\ e\ \textbf{of}\ z{:_{\overline{\Delta_i}}}\tau\ \{\overline{\rho \Rightarrow e'}\} : \sigma}}$$

$$\overset{(\textsc{Case}_{\textsc{Not}\ \text{WHNF}})}{\frac{\text{e is not in } \textit{WHNF} \qquad \Gamma; \Delta \vdash e : \tau \qquad \overline{\Gamma, z{:_{[\Delta]}}\tau; [\Delta], \Delta' \vdash \rho \Rightarrow_{\text{NWHNF}} e' :^{z}_{[\Delta]} \sigma}}{\Gamma; \Delta, \Delta' \vdash \textbf{case}\ e\ \textbf{of}\ z{:_{[\Delta]}}\tau\ \{\overline{\rho \Rightarrow e'}\} : \sigma}}$$

$$\boxed{\Gamma; \Delta \vdash \rho \Rightarrow e :^{z}_{\Delta_s} \sigma}$$

$$\overset{(\textsc{AltN}_{\text{WHNF}})}{\frac{\Gamma, \overline{x{:_\omega}\tau}, \overline{y_i{:_{\Delta_i}}\tau_i}^{n}; \Delta \vdash e : \sigma}{\Gamma; \Delta \vdash K\ \overline{x{:_\omega}\tau}, \overline{y_i{:_1}\tau_i}^{n} \Rightarrow_{\text{WHNF}} e :^{z}_{\overline{\Delta_i}^{n}} \sigma}} \qquad \overset{(\textsc{AltN}_{\textsc{Not}\ \text{WHNF}})}{\frac{\Gamma, \overline{x{:_\omega}\tau}, \overline{y_i{:_{\Delta_i}}\tau_i}; \Delta \vdash e : \sigma \qquad \overline{\Delta_i} = \overline{\Delta_s\#K_i}^{n}}{\Gamma; \Delta \vdash K\ \overline{x{:_\omega}\tau}, \overline{y_i{:_1}\tau_i}^{n} \Rightarrow_{\text{NWHNF}} e :^{z}_{\Delta_s} \sigma}}$$

$$\overset{(\textsc{Alt}0)}{\frac{\Gamma\,[\cdot/\Delta_s]_z, \overline{x{:_\omega}\tau}; \Delta\,[\cdot/\Delta_s] \vdash e : \sigma}{\Gamma; \Delta \vdash K\ \overline{x{:_\omega}\tau} \Rightarrow e :^{z}_{\Delta_s} \sigma}} \qquad \overset{(\textsc{Alt\_})}{\frac{\Gamma; \Delta \vdash e : \sigma}{\Gamma; \Delta \vdash \_ \Rightarrow e :^{z}_{\Delta_s} \sigma}}$$

Fig. 4. Linear Core Type System

$$\boxed{\Gamma \vdash_{mult} \pi}$$

$$\frac{}{\Gamma \vdash 1}\ (1) \qquad \frac{}{\Gamma \vdash \omega}\ (\omega) \qquad \frac{}{\Gamma, \rho \vdash \rho}\ (\rho)$$

$$\boxed{\Gamma; \Delta \Vdash e : \sigma > \overline{\Delta_i}}$$

$$\frac{\overline{\Gamma; \cdot \vdash e_\omega : \tau_i} \qquad \overline{\Gamma; \Delta_j \vdash e_j : \tau_j} \qquad \overline{\Delta_j} = \Delta \qquad K : \overline{\tau_i \to_\omega}\ \overline{\tau_j \multimap} \sigma}{\Gamma; \Delta \Vdash K\ \overline{e_\omega e_j} : \sigma > \overline{\Delta_j}}\ (\text{WHNF}_K) \qquad \frac{\Gamma; \Delta \vdash \lambda x.\ e : \sigma}{\Gamma; \Delta \Vdash \lambda x.\ e : \sigma > \Delta}\ (\text{WHNF}_\lambda)$$

Fig. 5. Linear Core Auxiliary Judgements

The main judgement is written $\Gamma; \Delta \vdash e : \tau$ to denote that expression e has type τ under the unrestricted environment Γ and linear environment Δ. Variables in Γ can be freely discarded (*weakened*) and duplicated (*contracted*), while resources in Δ must be used exactly once. Despite not having explicit weakening and contraction rules in our system, they are available as admissible rules for Γ (but not for Δ), since, equivalently [Dyckhoff 1992], resources from Γ are duplicated for sub-derivations and may unrestrictedly exist in the variable rules.

Occurrences of unrestricted variables from Γ are well-typed as long as the linear environment is empty, while occurrences of linear (or multiplicity polymorphic) variables are only well-typed when the variable being typed is the only resource available in the linear context (rules Var_ω, Var_1 and Var_p, respectively). In the Var_1 and Var_p cases, the linear context must contain exactly the appropriate variable binding, whereas the unrestricted context may contain arbitrary variables. Variables in contexts are annotated with their type and multiplicity, so $x{:}_\pi\tau$ is a variable named x of type σ and multiplicity π.

Linear functions are introduced via the function type ($\sigma \to_\pi \varphi$) with $\pi = 1$, i.e. a function of type $\sigma \to_1 \varphi$ (or, equivalently, $\sigma \multimap \varphi$) introduces a linear resource of type σ in the linear environment Δ to then type an expression of type φ. Unrestricted functions are introduced via the function type ($\sigma \to_\pi \varphi$) with $\pi = \omega$, and the λ-bound variable is introduced in Γ (rules λI_1 and ΛI). Function application is standard, with linear function application splitting the linear context in two disjoint parts, one used to type the argument and the other the function. Arguments of unrestricted functions must also be unrestricted, i.e. no linear variables can be used to type them. Typing linear and unrestricted function application separately is less general than typing applications of functions of any multiplicity π by scaling (per the multiplicity semiring) the multiplicities of the resources used to type the argument by π, however, our objective of typing semantic linearity does not benefit much from doing so and therefore we opt for a simpler design.

Multiplicity abstractions (ΛI) introduce a multiplicity variable p, and construct expressions of type $\forall p.\ \sigma$, i.e. a type universally quantified over a multiplicity variable p. In the body of the abstraction, function types and datatype fields annotated with the multiplicity variable p are typed parametrically in their multiplicity, since p can be instantiated at both ω and 1. A multiplicity application (ΛE) instantiates a multiplicity-polymorphic type $\forall p.\ \sigma$ at a particular (argument) multiplicity π, resulting in an expression of type σ where occurrences of p are substituted by π, i.e. $\sigma[\pi/p]$. The rule additionally requires that π be *well-formed* in order for the expression to be well-typed, using the judgement $\Gamma \vdash_{mult} \pi$, where well-formedness is given by π either being 1, ω, or an in-scope multiplicity variable in Γ.

3.3 Usage Environments

A *usage environment* Δ encodes the idea that lazy variable bindings do not consume resources upon definition, but rather when the bindings themselves are consumed. Specifically, we annotate so-called Δ-bound variables with their *usage environment* Δ to denote that consuming such variables must be equated with consuming the resources in the annotated typing environment. Such variables are introduced in the unrestricted environment by multiple constructs such as (recursive) let binders, case binders, and case pattern variables. For example, the let-bound variable u in the following program is annotated with a usage environment $\{x{:}_1\sigma, y{:}_1\sigma\}$ tracking the resources that are used in its body. Thus, in the expression e, consuming u will equate to consuming the linear resources x and y:

$$f = \lambda x{:}_1\tau :_1 \tau.\ \lambda y{:}_1\tau.\ \textbf{let}\ u_{\{x:_1\tau, y:_1\sigma\}} = (x, y)\ \textbf{in}\ e$$

Furthermore, usage environments guarantee that using a Δ-bound variable is mutually exclusive with directly using the resources it is annotated with – using the Δ-bound variable consumes all linear resources listed in its usage environment, meaning they are no longer available for direct usage. Dually, using the linear resources directly means they are no longer available to consume through the usage environment of the Δ-bound variable. The Δ-bound variable can be left unused since it is unrestricted, regardless of the variables it is annotated with. In the example above, our type system ensures that *either* x and y are consumed or u is consumed in e, since u has usage environment $\{x :_1 \tau, y :_1 \sigma\}$.

Δ-bound Variables. A Δ-bound variable u is a variable annotated with some usage environment Δ'. For conciseness, we use the meta-variable Δ to refer to the usage environment of a Δ-bound variable. Crucially, for any Δ-bound variable u: (1) using u is equivalent to using all the linear resources in Δ; (2) using u is mutually exclusive with using the Δ resources it depends on; (3) u can be safely discarded as long as the resources in Δ are consumed. Fortunately, since linear resources must be linearly split across sub-derivations, (2) follows from (1) since consuming the linear resources in Δ to type u makes them unavailable in the context of any other derivation. Similarly, (3) also follows from (1), because if the linear resources aren't consumed by a use of u, they must be consumed in some other derivation (or otherwise the overall expression is ill-typed). These observations lead to the typing rule for Δ-bound variables:

$$\frac{\Delta = \Delta'}{\Gamma, x{:}_\Delta\sigma; \Delta' \vdash x : \tau}\ (\textsc{Var}_\Delta)$$

The rule states that an occurrence of a Δ-bound variable is well-typed if the linear environment is made up exactly of the resources in the usage environment of that variable, denoted by the $\Delta = \Delta'$ premise of the rule. Δ-bound variables are unrestricted and can be discarded and duplicated, despite multiple occurrences of the variable in an expression not being well-typed due to non-linear usage of linear resources.

3.4 Lazy Let Bindings

In Section 2.1, we discussed how linear resources used in let-bound expressions are only consumed when the respective binders are evaluated in the let body. Moreover, resources captured by a let-bound expression cannot be used simultaneously with the let-binder in its body, since those resources would end up being consumed more than once and violate the semantics of linearity. Either the binding or the resources *must* be used.

Let-bound variables are the canonical example of a Δ-bound variable. Annotating let-bound variables with their usage environment Δ delays the consumption of resources to when the variables themselves are used and guarantees the desired mutual exclusion property between the binder and the resources:

$$\frac{\Gamma;\Delta \vdash e : \tau \qquad \Gamma, x{:}_{\Delta}\tau; \Delta, \Delta' \vdash e' : \sigma}{\Gamma; \Delta, \Delta' \vdash \mathbf{let}\ x{:}_{\Delta}\tau = e\ \mathbf{in}\ e' : \sigma}\ (\textsc{Let})$$

The rule for (non-recursive) let bindings splits the linear environment in Δ and Δ'. Δ is used to type the body e of the let binding x. Perhaps surprisingly, the resources Δ are still available to type the let body e', alongside the unrestricted x binding annotated with the usage environment Δ. The resources of e being available in e' reflects how the body of a lazy binder doesn't consume resources outright, and x being an unrestricted Δ-bound variable ensures that using x will consume the resources Δ that e captures.

3.5 Recursive Let Bindings

Recursive let bindings also bind expressions lazily and so introduce a Δ-bound variable for each binding. The resources required to type the let-bindings are still available in the let body. These resources may be consumed via the corresponding Δ-bound variables, or directly if the let-bindings are unused.

The typing rule for recursive groups of bindings leverages both the assumption that all mutually recursive groups are strongly connected and the corollary that every binder in such a group must be typed with the same linear context, as observed in Section 2.1. Consequently, all bindings of the recursive group are introduced as Δ-bound variables sharing the same usage environment – using any one of the bindings in a recursive group entails consuming all resources required to type that same group – so, at most a single binder from the group can be used in the let body.

$$\frac{\overline{\Gamma, \overline{x_i{:}_{\Delta}\tau_i}; \Delta \vdash e_i : \sigma} \qquad \Gamma, \overline{x_i{:}_{\Delta}\tau_i}; \Delta, \Delta' \vdash e' : \sigma}{\Gamma; \Delta, \Delta' \vdash \mathbf{let\ rec}\ \overline{x_i{:}_{\Delta}\tau_i = e_i}\ \mathbf{in}\ e' : \sigma}\ (\textsc{LetRec})$$

This formulation is not syntax-directed since it does not describe how to annotate binders with a particular usage environment Δ. Our system simply assumes the annotations are explicitly provided in the source and checks the correctness of the overall recursive let.

In practice, determining this typing environment Δ amounts to finding a least upper bound of the resources needed to type each mutually-recursive binding that (transitively) uses all binders in the recursive group. Our implementation uses a naive $O(n^2)$ algorithm for inferring usage environments of recursive bindings. Moreover, we note that inference of usage environments for recursive binding groups bears some resemblance to the inference of principle types for recursive bindings traditionally achieved through the Hindley–Milner inference algorithm [Damas and Milner 1982]. We leave the exploration of this connection as future work.

3.6 Case Expressions

A case expression drives evaluation by evaluating its scrutinee to Weak Head Normal Form (WHNF) [Peyton Jones 1987]. Then, the case alternative whose pattern matches the result of evaluating the scrutinee is taken[2]. An expression in WHNF can either be a λ-abstraction ($\lambda x.\ e$) or a datatype constructor application ($K\ \overline{e}$). In both cases, the sub-expressions e or $\overline{e}$ occurring in the lambda body or as constructor arguments need not be evaluated for the lambda or constructor application to be in WHNF (and they may capture linear resources).

A key detail is that when the scrutinee is already in WHNF, evaluation continues in the alternative simply by substituting the pattern variables by the scrutinee sub-expressions. No resource from the

[2]In our calculus, the alternatives are assumed to be exhaustive, i.e. there always exists at least one pattern which matches the scrutinee in its WHNF, so we're guaranteed to have an expression to progress evaluation.

scrutinee can be consumed since no computation happens at all. We introduce a typing judgement $\Gamma; \Delta \Vdash e : \sigma \gg \overline{\Delta_i}$ to extract additional information from the static structure of terms in WHNF:

$$\frac{\overline{\Gamma; \cdot \vdash e_\omega : \tau_i} \qquad \overline{\Gamma; \Delta_j \vdash e_j : \tau_j} \qquad \overline{\Delta_j} = \Delta \qquad K : \overline{\tau_i \to_\omega \tau_j \multimap} \sigma}{\Gamma; \Delta \Vdash K\ \overline{e_\omega e_j} : \sigma \gg \overline{\Delta_j}}\ (\text{WHNF}_K) \qquad \frac{\Gamma; \Delta \vdash \lambda x.\, e : \sigma}{\Gamma; \Delta \Vdash \lambda x.\, e : \sigma \gg \Delta}\ (\text{WHNF}_\lambda)$$

This judgement differs from the main typing judgement in that (1) it only applies to expressions in weak head normal form, and (2) it "outputs" (to the right of $\gg$) a disjoint set of linear environments ($\overline{\Delta_i}$), where each environment corresponds to the linear resources used by a sub-expression of the WHNF expression. To type a constructor application $K\ \overline{e_\omega e_i}$, where e_ω are the unrestricted arguments and e_i the linear arguments of the constructor, we split the resources Δ into a disjoint set of resources $\overline{\Delta_i}$ required to type each linear argument individually and return exactly that split of the resources; the unrestricted e_ω expressions must be typed on an empty linear environment. A lambda expression is typed with the main typing judgement and trivially "outputs" the whole Δ environment, as there is always only a single sub-expression in lambda abstractions.

3.6.1 Discriminating by WHNF-ness. The dichotomy between evaluation (and resource usage) of a case expression whose scrutinee is in weak head normal form and one whose scrutinee is not leads to one of the key insights of λ^π_Δ: accurate typing of case expressions depends on whether the scrutinee is in WHNF. When the scrutinee is already in WHNF, the resources are unused upon evaluation and thus available in the alternatives. When it is not, resources will be consumed and cannot be used in the alternative. To illustrate, consider the following case expressions:

$$(1)\ \lambda x.\ \textbf{case}\ K\ x\ \textbf{of}\ z\ \{\_ \to x\} \qquad (2)\ \lambda x.\ \textbf{case}\ free\ x\ \textbf{of}\ z\ \{\_ \to x\}$$

The first function uses x linearly, but the second does not. Alternatives may also use the case binder or pattern variables, referring to, respectively, the whole scrutinee and all its used resources; or constructor arguments and the resources used to type them. The reader may wonder about the significance of case expressions where the scrutinee is in WHNF. We note that in an optimizing compiler (such as GHC) many intermediate programs that arise due to program transformations commonly scrutinize expressions in WHNF.

There are three competing ways to use the resources from a scrutinee in WHNF in a case alternative: directly, via the case binder, or by using pattern-bound variables. Recall how Δ-bound variables encode mutual exclusion between alternative ways of consuming resources – it follows that case binders and pattern-bound variables are yet another instance of Δ-bound variables. This suggests the following first rule for cases of scrutinees already in WHNF:

$$\frac{e \text{ is in } \mathit{WHNF} \qquad \Gamma; \Delta \Vdash e : \tau \gg \overline{\Delta_i} \qquad \overline{\Gamma, z{:}_{\overline{\Delta_i}}\tau; \overline{\Delta_i}, \Delta' \vdash \rho \ \Rightarrow\ e' :^{z}_{\overline{\Delta_i}} \sigma}}{\Gamma; \Delta, \Delta' \vdash \textbf{case}\ e\ \textbf{of}\ z{:}_{\overline{\Delta_i}}\tau\ \{\overline{\rho \ \Rightarrow\ e'}\} : \sigma}\ (\text{Case}_{\text{WHNF}})$$

First, we assert this rule is only applicable to expressions in weak head normal form. Second, we appeal to the typing judgement for expressions in WHNF to determine the split of resources amongst the scrutinee sub-expressions. Finally, we type all case alternatives with the same context, using the introducing the case binder z in the unrestricted environment as a Δ-bound variable whose usage environment is the linear resources $\overline{\Delta_i}$ used to type the scrutinee. Those same resources $\overline{\Delta_i}$ are again made available in the linear typing environment of the alternative, similarly to how the resources used to type a *Let* binder are still available in the continuation. Although the main idea for typing "WHNF case expressions" is conveyed by this rule, the full rule of our system is slightly more involved, as will be seen after discussing cases of scrutinees not in WHNF.

The alternative judgement $\Gamma; \Delta \vdash \rho \Rightarrow e :^{z}_{\Delta} \sigma$ is used to type case alternatives, encompassing three modes that are distinguished by the kind of arrow that is used: for alternatives of case expressions whose scrutinee is in WHNF ($\Rightarrow_{\mathsf{WHNF}}$), not in WHNF ($\Rightarrow_{\mathsf{NWHNF}}$), and for alternatives agnostic to the WHNF-ness of the scrutinee ($\Rightarrow$), with $\Rightarrow$ also generalizing the other two.

First, an alternative whose pattern is a constructor with $n > 0$ linear components under the $\Rightarrow_{\mathsf{WHNF}}$ mode is typed as:

$$\text{(AltN}_{\text{WHNF}}\text{)}\quad \frac{\Gamma, \overline{x{:}_{\omega}\tau}, \overline{y_i{:}_{\Delta_i}\tau_i}^{n}; \Delta \vdash e : \sigma}{\Gamma; \Delta \vdash K\ \overline{x{:}_{\omega}\tau}, \overline{y_i{:}_{1}\tau_i}^{n} \ \Rightarrow_{\mathsf{WHNF}}\ e :^{z}_{\overline{\Delta_i}^{n}} \sigma}$$

The rule states that, for such a match on a scrutinee already in WHNF, we introduce the linear components of the pattern as Δ-bound variables with usage environment matching the linear resources required to type the corresponding constructor argument in the scrutinee. The resources required to type each constructor sub-expression come annotated in the judgement (as $:_{\overline{\Delta_i}}$). Unrestricted fields of the constructor are introduced as unrestricted variables. We note that the typing environment Δ always contains the resources $\overline{\Delta_i}$ in uses of the alternative judgement.

Second, the rule for alternatives that match on the wildcard pattern, regardless of the WHNF status of the scrutinee (note the use of $\Rightarrow$, which is applicable both under $\Rightarrow_{WHNF}$ and $\Rightarrow_{NWHNF}$):

$$\text{(Alt\_)}\quad \frac{\Gamma; \Delta \vdash e : \sigma}{\Gamma; \Delta \vdash \_ \ \Rightarrow\ e :^{z}_{\Delta_s} \sigma}$$

To type a wildcard alternative we simply type the expression with the main judgement, ignoring all annotations. The case binder will have already been introduced in the environment with the appropriate usage environment by the relevant case expression rule.

Lastly, consider an alternative matching on a data constructor without any linear components. The linear resources used to type a scrutinee matching such a pattern are fully consumed during evaluation: the resulting unrestricted constructor application can only be well-typed under an empty linear environment. Consequently, the resources to type the scrutinee which are carried over to the case-alternative environments by the $Case_{WHNF}$ and $Case_{NWHNF}$ rules (see the latter below), must be reactively destroyed from branches where the pattern is unrestricted. In failing to do this, we would allow the fully-consumed resources to be used again in the branch. The $Alt0$ rule essentially encodes this insight, and is applicable regardless of the WHNF status of the scrutinee ($\Rightarrow$ mode), as long as the constructor pattern has no linear fields:

$$\text{(Alt0)}\quad \frac{\Gamma\,[\cdot/\Delta_s]_z\,, \overline{x{:}_{\omega}\tau}; \Delta\,[\cdot/\Delta_s] \vdash e : \sigma}{\Gamma; \Delta \vdash K\ \overline{x{:}_{\omega}\tau} \ \Rightarrow\ e :^{z}_{\Delta_s} \sigma}$$

The rule deletes the annotated scrutinee environment Δ_s from the linear typing context (written $\Delta[\cdot/\Delta_s]$, a substitution of the scrutinee typing environment by the empty linear environment $\cdot$) and from the usage environment of the case binder z, written $\Gamma[\cdot/\Delta_s]_z$ to delete Δ_s from the usage environment of the Δ-bound variable z in Γ. Given that the case-binder z is always annotated with Δ_s (essentially, the linear environment typing the scrutinee), this latter substitution makes the case-binder unrestricted in branches with unrestricted patterns.

This rule observes that any linear resources required to type an expression that reduces to an unrestricted constructor application are fully consumed through evaluation and the resulting unrestricted expression can be freely discarded or duplicated (e.g. via the case binder).

3.6.2 Irrelevant Resources. When the scrutinee is not in WHNF we cannot statically determine which resources are consumed upon scrutinee evaluation or which are transferred into the branch continuation e.g. as in **case** $f\ x$ **of** $z\ \{K\ a \longrightarrow e\}$:

(1) Cases with non-WHNF scrutinees must be typed such that no resource from the scrutinee can be used directly in the branch again, or we risk consuming the same resource twice. In the example, x must not directly occur in e since f *could* have consumed it.
(2) On the other hand, since evaluation to WHNF does not guarantee all resources are consumed (unless the result matches an unrestricted pattern), we must ensure that any resources that transfer from the scrutinee to the alternative are ultimately consumed (in the example, x is not consumed by evaluating the scrutinee for $f = K$).

Considering (1), we must resort to a more uniform method of guaranteeing we "finish consuming" the scrutinee. There are only two ways of uniformly referring to the remaining linear resources in the newly-evaluated scrutinee:

- The case binder, which refers to the whole result of evaluating the scrutinee;
- The linear components of a constructor pattern, which refer to expressions that may contain linear resources.

To encode linear resources that cannot be directly used (i.e., to which the Var rule is not applicable) we introduce *irrelevant* resources, written as linear resources within square brackets $[x{:}\sigma]$ and pointwise lifted to contexts and usage environments as $[\Delta]$. Irrelevant resources are linear resources in every sense, meaning they must be used exactly once. However, since irrelevant resources cannot be discarded or used directly, they have to be consumed *indirectly* via Δ-bound variables, namely, the case binder, or, in mutual exclusion, the linear pattern-bound variables.

Hence, to type a case expression whose scrutinee is not in WHNF, we type the scrutinee with linear resources Δ and the case alternatives by introducing the case binder with an irrelevant usage environment $[\Delta]$, having the same irrelevant $[\Delta]$ in the typing environment, and annotating the judgement with the irrelevant resources for use in the $\Rightarrow_{\mathsf{NWHNF}}$ judgement:

$$\text{(Case}_{\text{Not WHNF}}\text{)}\quad \frac{\text{e is not in } \textit{WHNF} \qquad \Gamma;\Delta \vdash e : \tau \qquad \overline{\Gamma, z{:}_{[\Delta]}\tau; [\Delta], \Delta' \vdash \rho \Rightarrow_{\mathsf{NWHNF}} e' :^{z}_{[\Delta]} \sigma}}{\Gamma;\Delta,\Delta' \vdash \textbf{case}\ e\ \textbf{of}\ z{:}_{[\Delta]}\tau\ \{\overline{\rho \Rightarrow e'}\} : \sigma}$$

Note how the rule is similar to the one for scrutinees in WHNF, but requires the resources in the case binder, typing environment, and judgement annotation to be made "irrelevant".

Finally, we recall the tentative *Case*$_{\text{WHNF}}$ rule presented before and highlight its flaw: the $\Gamma;\Delta \vdash \rho \Rightarrow_{\mathsf{WHNF}} e :^{z}_{\Delta_s} \sigma$ judgement is only well-defined for patterns ρ matching the WHNF form of the scrutinee, as the distribution of resources per constructor components only makes sense for the constructor pattern matching the scrutinee. Alternatives not matching the scrutinee could use resources arbitrarily as they will never be executed. We uniformly treat non-matching alternatives as if the scrutinee were not in WHNF. Having introduced irrelevant resources, we can now present the full *Case*$_{\text{WHNF}}$ rule:

$$\text{(Case}_{\text{WHNF}}\text{)}\quad \frac{\begin{array}{c}\text{e is in } \textit{WHNF} \qquad e \text{ matches } \rho_j \qquad \overline{\Gamma, z{:}_{[\Delta]}\tau; [\Delta], \Delta' \vdash \rho \Rightarrow_{\mathsf{NWHNF}} e' :^{z}_{[\Delta]} \sigma} \\ \Gamma;\Delta \Vdash e : \tau \gg \overline{\Delta_i} \qquad \Gamma, z{:}_{\overline{\Delta_i}}\tau; \overline{\Delta_i}, \Delta' \vdash \rho_j \Rightarrow e' :^{z}_{\overline{\Delta_i}} \sigma\end{array}}{\Gamma;\Delta,\Delta' \vdash \textbf{case}\ e\ \textbf{of}\ z{:}_{\overline{\Delta_i}}\tau\ \{\overline{\rho \Rightarrow e'}\} : \sigma}$$

where we appeal to the pattern judgment for WHNF scrutinees for pattern ρ_j, allowing the $\overline{\Delta_i}$ resources to be consumed either through the case-binder or outright; and to the non-WHNF judgment for the remaining branches, where the Δ resources may only be consumed via the case-binder. We again note that it might seem unusual to specialize a rule for expressions in WHNF, as programs scrutinizing an expression in WHNF are rarely written by humans. Yet, our system is designed to be suitable for optimising compilers in which intermediate programs commonly scrutinize expressions in WHNF.

3.6.3 Splitting and Tagging Fragments. As opposed to scrutinees in WHNF, where resources can be cleanly divided amongst the various sub-expressions of a constructor application and, henceforth, each pattern variable, there is no direct mapping between the resources typing a scrutinee not in WHNF and the usage environments of pattern variables in any alternative.

We introduce *tagged resources* as a means to enforce that all pattern-bound variables for a scrutinee not in WHNF are either *jointly* used to consume all resources occurring in the environment, or not at all (instead, the case binder may be used). Given irrelevant resources $[\Delta_s]$ used to type a scrutinee, and a pattern $K\ \overline{x_\omega}, \overline{y_i}$ with i linear components, we assign a usage environment Δ_i to each linear pattern variable where Δ_i is obtained from the scrutinee environment *tagged* with the constructor name and linear-variable index $[\Delta_s]\#K_i$. Then, $y_i{:}_{\Delta_i}\sigma$ is introduced in Γ, just like other Δ-bound variables.

$$\textsc{(AltN}_{\textsc{Not WHNF}}\textsc{)}\quad \frac{\Gamma, \overline{x{:}_\omega\tau}, \overline{y_i{:}_{\Delta_i}\tau_i}; \Delta \vdash e : \sigma \qquad \overline{\Delta_i} = \overline{\Delta_s\#K_i}^n}{\Gamma; \Delta \vdash K\ \overline{x{:}_\omega\tau}, \overline{y_i{:}_1\tau_i}^n \Rightarrow_{\mathsf{NWHNF}} e :^z_{\Delta_s} \sigma} \qquad \textsc{(Split)}\quad \frac{\Gamma; \Delta, x :_1 \tau \vdash e : \tau \qquad K \text{ has } n \text{ linear arguments}}{\Gamma; \Delta, \overline{x{:}_1\tau\#K_i}^n \vdash e : \sigma}$$

Having uniquely tagged the resources in the usage environment of each pattern variable, we need only express that (1) the pattern Δ-bound variables can be used (i.e., the tagged resources need be available in the linear environment in order for the Var$_\Delta$ rule to be applicable) and (2) if a K-pattern Δ-bound variable is consumed, all remaining K-Δ-bound variables, standing for the remaining linear components of the same pattern, must also be consumed.

To satisfy these two constraints, we introduce a Split rule that allows a linear resource $x{:}_1\sigma$ to be split into n resources at a given constructor K, where n is the number of linear components of the constructor, and each resource resulting from the split is *tagged* with K and the positional index of a linear component (i.e. $x{:}_1\sigma$ can be split into $\overline{x{:}_1\sigma\#K_i}^n$). By assigning to each pattern variable a *fragment* of the scrutinee resources (with a tag), we require the scrutinee resources to be Split in order to use pattern variables at all. Moreover, the remaining tagged resources cannot be used directly, yet need to still be consumed exactly once. Thus, the choice to consume the tagged resources via the usage environments of the other pattern variables is forced, ensuring that no variable for a constructor's linear component can go unused (the tagged environments are disjoint). Tagged resources are inspired by fractional permissions in separation logic [Boyland 2003].

For instance, in the term $\lambda x.$ **case** $f\ x$ **of** z $\{K\ a\ b \to (a, b)\}$, where x is a linear variable, the case alternative is typed with $[x]$, the case binder z is introduced as $z{:}_{[x]}\tau$, and the pattern variables are introduced as $a{:}_{[x]\#K_1}\tau$ and $b{:}_{[x]\#K_2}\tau$, assuming both components of K are linear. The occurrences of a and b are well-typed because we can first Split $[x{:}_1\sigma]$ into $[x{:}_1\sigma]\#K_1$, $[x{:}_1\sigma]\#K_2$, noting that Split can be applied both to relevant and proof irrelevant linear resources in Δ.

4 Metatheory of Linear Core

We develop the metatheory of Linear Core by first presenting the operational semantics of λ^π_Δ, consisting of a lazy natural semantics in the style of Launchbury [Launchbury 1993]. This semantics conveys the expected behavior of an implementation of lazy evaluation and is agnostic to any linearity information. Such a semantics is ill-suited for reasoning about linearity and so we develop an *instrumented* linearity-aware semantics that is enriched with sufficient information to allow us

$$\frac{}{\Theta : \Lambda p.e \Downarrow \Theta : \Lambda p.e} \qquad \frac{\Theta : e \Downarrow \Theta' : \Lambda p.e' \quad \Theta' : e'[\pi/p] \Downarrow \Theta'' : v}{\Theta : e\ \pi \Downarrow \Theta'' : v}$$

$$\frac{}{\Theta : \lambda x :_\pi \sigma.e \Downarrow \Theta : \lambda x :_\pi \sigma.e} \qquad \frac{\Theta : e \Downarrow \Theta' : \lambda y :_\pi \sigma.e' \quad \Theta' : e'[x/y] \Downarrow \Theta'' : v}{\Theta : e\ x \Downarrow \Theta'' : v}$$

$$\frac{(\Theta, x :_\omega \sigma = e) : e \Downarrow \Theta' : v}{(\Theta, x :_\omega \sigma = e) : x \Downarrow (\Theta', x :_\omega \sigma = v) : v} \qquad \frac{(\Theta, x :_\omega \sigma = e) : e' \Downarrow \Theta' : v}{\Theta : \mathbf{let}\ x{:}_\Delta\sigma = e\ \mathbf{in}\ e' \Downarrow \Theta' : v}$$

$$\frac{}{\Theta : K\ \overline{x_i} \Downarrow \Theta : K\ \overline{x_i}} \qquad \frac{\Theta : e \Downarrow \Theta' : K\ \overline{x_i} \quad \Theta' : e'\overline{[x_i/y_i]}[K\ \overline{x_i}/z] \Downarrow \Theta'' : v}{\Theta : \mathbf{case}\ e\ \mathbf{of}\ z{:}_{\Delta'}\sigma\ \{\dots, K\ \overline{y_i} \Rightarrow e', \dots\} \Downarrow \Theta'' : v}$$

$$\frac{\Theta : e \Downarrow \Theta' : K\ \overline{x_i} \quad \Theta' : e'[K\ \overline{x_i}/z] \Downarrow \Theta'' : v}{\Theta : \mathbf{case}\ e\ \mathbf{of}\ z{:}_{\Delta'}\sigma\ \{\dots, \_ \Rightarrow e'\} \Downarrow \Theta'' : v} \qquad \frac{(\Theta, \overline{x_i :_\omega \sigma_i = e_i}) : e' \Downarrow \Theta' : v}{\Theta : \mathbf{let\ rec}\ \overline{x_i{:}_\Delta\sigma_i = e_i}\ \mathbf{in}\ e' \Downarrow \Theta' : v}$$

Fig. 6. Natural Semantics of λ^π_Δ

to establish type safety, thus showing that linearity is preserved. We show that the natural and instrumented semantics are bisimilar for well-typed terms and so derive type safety in the natural semantics (Section 4.2). Finally, we show that multiple optimising transformations such as those performed by GHC on Linear Core are type preserving (Section 4.3), and thus linearity preserving.

4.1 Operational Semantics

The Launchbury-style natural semantics, presented in Figure 6, captures standard lazy evaluation while ignoring linearity information. The natural semantics is equipped with an evaluation environment that maps variables to expressions and is mutated in order to express shared evaluation. Terms are explicit about their sharing (in the sense of shared reductions in lazy evaluation), and so we perform a translation step that makes all sharing explicit through let-binders (see [Mesquita and Toninho 2025a]). We write **let** $\overline{x_i{:}_{\Delta_i}\sigma = e_i}$ **in** e' for the iterated let-binding of variables $\overline{x_i}$.

The natural semantics is defined via the relation $\Theta : e \Downarrow \Theta' : v$ where e is an expression, v a value, and Θ and Θ' are evaluation environments with bindings of the form $x :_\omega \sigma = e$ which assigns the expression e to the variable x of the given type σ. The rules are standard, augmenting the runtime environment with let-bound expressions which are evaluated (only once) if the corresponding let-bound variable is forced.

While the natural semantics is a mostly direct model of an implementation of lazy evaluation, it is inconvenient for formal reasoning about well-typed linear terms. We follow an approach similar to that of [Bernardy et al. 2017a,b], defining an *instrumented* operational semantics in which linear variables are erased from the runtime environment once they are forced, ensuring that any term violating linearity will result in a stuck state. The semantics is also type-aware and carries with it sufficient data to reconstruct typing derivations for the purposes of showing type safety.

The instrumented semantics is presented in Figure 7, defining the judgment $\Gamma; \Delta \vdash (\Theta \mid e) \Downarrow (\Theta' \mid v) : \sigma, \Sigma$ where Γ and Δ are typing contexts for expressions in Σ (Γ tracks unrestricted and Δ-bound variables, Δ tracks linear assumptions); Θ and Θ' are evaluation environments, consisting of bindings of the form $x :_\pi \sigma = e$ or $x :_\Delta \sigma = e$; e is the expression being evaluated and v its resulting value, both of type σ; Σ is a list of assignments of the form $e : \tau$ which are expressions in which bindings in Θ may also be used. This additional data is needed to inductively show that the overall evaluation state is well-typed (and linearity preserving). The rules mostly mirror those of the natural semantics, with the exception of the linear variable rule which erases the binding from

$$\frac{}{\Gamma; \Delta \vdash (\Theta \mid \Lambda p.e) \Downarrow (\Theta \mid \Lambda p.e) : \forall p.\sigma, \Sigma}$$

$$\frac{\Gamma; \Delta \vdash (\Theta \mid e) \Downarrow (\Theta' \mid \Lambda p.e') : \forall p.\sigma, \Sigma \quad \Gamma; \Delta \vdash (\Theta' \mid e'[\pi/p]) \Downarrow (\Theta'' \mid v) : \sigma[\pi/p], \Sigma}{\Gamma; \Delta \vdash (\Theta \mid e\ \pi) \Downarrow (\Theta'' \mid v) : \sigma[\pi/p], \Sigma}$$

$$\frac{}{\Gamma; \Delta \vdash (\Theta \mid \lambda x :_\pi \sigma.e) \Downarrow (\Theta \mid \lambda x :_\pi \sigma.e) : \sigma \to_\pi \tau, \Sigma}$$

$$\frac{\Gamma; \Delta \vdash (\Theta \mid e) \Downarrow (\Theta' \mid \lambda y :_1 \sigma.e') : \sigma \to_1 \tau, x : \sigma, \Sigma \quad \Gamma; \Delta \vdash (\Theta', y :_1 \sigma = x \mid e') \Downarrow (\Theta'' \mid v) : \tau, \Sigma}{\Gamma; \Delta \vdash (\Theta \mid e\ x) \Downarrow (\Theta'' \mid v) : \tau, \Sigma}$$

$$\frac{\Gamma; \Delta \vdash (\Theta \mid e) \Downarrow (\Theta' \mid \lambda y :_\omega \sigma.e') : \sigma \to_\omega \tau, x : \sigma, \Sigma \quad \Gamma; \Delta \vdash (\Theta' \mid e'[x/y]) \Downarrow (\Theta'' \mid v) : \tau, \Sigma}{\Gamma; \Delta \vdash (\Theta \mid e\ x) \Downarrow (\Theta'' \mid v) : \tau, \Sigma}$$

$$\frac{\Gamma; \Delta \vdash (\Theta, x :_{\Delta'} \sigma = e \mid e) \Downarrow (\Theta' \mid v) : \sigma, \Sigma \qquad \Delta'' = \Delta' \setminus (\Theta \setminus \Theta')}{\Gamma; \Delta \vdash (\Theta, x :_{\Delta'} \sigma = e \mid x) \Downarrow (\Theta', x :_{\Delta''} \sigma = v \mid v) : \sigma, \Sigma}$$

$$\frac{\Gamma; \Delta \vdash (\Theta \mid e) \Downarrow (\Theta' \mid v) : \sigma, \Sigma}{\Gamma; \Delta \vdash (\Theta, x :_1 \sigma = e \mid x) \Downarrow (\Theta' \mid v) : \sigma, \Sigma}$$

$$\frac{\Gamma; \Delta \vdash (\Theta, x :_{\Delta'} \sigma = e \mid e') \Downarrow (\Theta' \mid v) : \tau, \Sigma}{\Gamma; \Delta \vdash (\Theta \mid \textbf{let}\ x{:}_{\Delta'}\sigma = e\ \textbf{in}\ e') \Downarrow (\Theta' \mid v) : \tau, \Sigma}$$

$$\frac{}{\Gamma; \Delta \vdash (\Theta \mid K\ \overline{x_i}) \Downarrow (\Theta \mid K\ \overline{x_i}) : T, \Sigma}$$

$$\frac{\begin{array}{c} K : \overline{\sigma_i \to_\omega \sigma_j \multimap} T \qquad \Gamma, z{:}_{\overline{y_j}} T, \overline{y_i :_\omega \sigma_i}; \Delta, \overline{y_j :_1 \sigma_j} \vdash (\Theta \mid e) \Downarrow (\Theta' \mid K\ \overline{x_i}) : T, e' : \tau, \Sigma \\ \Gamma; \Delta \vdash (\Theta' \mid e'\overline{[x_i/y_{i,j}]}[K\ \overline{x_i}/z]) \Downarrow (\Theta'' \mid v) : \tau, \Sigma \end{array}}{\Gamma; \Delta \vdash (\Theta \mid \textbf{case}\ e\ \textbf{of}\ z{:}_{\Delta'} T\ \{\dots, K\ \overline{y_{i,j}} \Rightarrow e', \dots\}) \Downarrow (\Theta'' \mid v) : \tau, \Sigma}$$

$$\frac{K : \overline{\sigma_i \to_\omega \sigma_j \multimap} T \qquad \Gamma; \Delta, z{:}_1\sigma \vdash (\Theta \mid e) \Downarrow (\Theta' \mid K\ \overline{x_i}) : T, e', \Sigma \quad \Gamma; \Delta \vdash (\Theta' \mid e'[K\ \overline{x_i}/z]) \Downarrow (\Theta'' \mid v) : \tau, \Sigma}{\Gamma; \Delta \vdash (\Theta \mid \textbf{case}\ e\ \textbf{of}\ z{:}_{\Delta'} T\ \{\dots, \_ \Rightarrow e'\}) \Downarrow (\Theta'' \mid v) : \tau}$$

$$\frac{\Gamma; \Delta \vdash (\Theta, \overline{x_i :_{\Delta'} \sigma_i = e_i} \mid e') \Downarrow (\Theta' \mid v) : \tau}{\Gamma; \Delta \vdash (\Theta \mid \textbf{let rec}\ \overline{x_i{:}_{\Delta'}\sigma_i = e_i}\ \textbf{in}\ e') \Downarrow (\Theta' \mid v) : \tau}$$

Fig. 7. Instrumented Operational Semantics

the execution environment. Both semantics define evaluation relations of the form $a \Downarrow b$, where a and b are evaluation states. We also rely on a notion of partial derivations and partial evaluation (see [Mesquita and Toninho 2025a]), written $a \Downarrow^* b$, which allows us to state and reason about a progress property.

4.2 Type Safety

Our results rely on a series of lemmas relating linear, unrestricted and Δ-bound variables, as well as a lemma characterizing irrelevant resources.

Δ-bound variables. A well-typed program with a linear variable $(x{:}_1\tau)$ is equivalently well-typed if the linear variable were instead Δ-bound $(x{:}_\Delta\sigma)$ with usage environment Δ, with Δ available in the linear context and with any occurrences of x in usage environments in Γ expanded into x's usage environment Δ, with the abuse of notation $\Gamma[\Delta/x]$.

Lemma 4.1 (Linear to Δ-bound). *If* $\Gamma; \Delta', x{:}_1\tau \vdash e : \sigma$ *then* $\Gamma[\Delta/x], x{:}_\Delta\sigma; \Delta, \Delta' \vdash e : \sigma$ *(Δ fresh).*

Dually, any Δ-bound variable with an irrelevant usage environment can be made into a linear variable as long as its (irrelevant) environment is dropped from the linear context and all the occurrences of the set of variables $[\Delta]$ in usage environments in Γ are collapsed to x, written $\Gamma[x/[\Delta]]$.

Lemma 4.2 (Δ-bound to Linear). *If* $\Gamma, x{:}_{[\Delta]}\tau; [\Delta], \Delta' \vdash e : \sigma$ *then* $\Gamma[x/[\Delta]]; \Delta', x{:}_1\tau \vdash e : \sigma$.

Finally, unrestricted and Δ-bound variables with an empty usage environment are equivalent.

Lemma 4.3 (Unrestricted and Δ-bound). $\Gamma, x{:}_\omega\tau; \Delta \vdash e : \sigma$ *iff* $\Gamma, x{:}_\emptyset\tau; \Delta \vdash e : \sigma$

Irrelevance. Irrelevant resources (Section 3.6.2) can only be consumed indirectly, essentially encoding that the non-WHNF case scrutinee resources must be consumed through the case binder or the linear pattern-bound variables. As a case expression is evaluated, the scrutinee will eventually reduce to WHNF, which must then be typed with rule $Case_{\text{WHNF}}$. Crucially, the two case rules must be in harmony in the system, in the sense that case expressions typed using the $Case_{\text{Not WHNF}}$ rule must also be well-typed by the $Case_{\text{WHNF}}$ once the scrutinee is evaluated to WHNF.

Lemma 4.4 (Irrelevance). *If* $\Gamma, z{:}_{[\Delta]}\tau; [\Delta], \Delta' \vdash \rho \Rightarrow_{\text{NWHNF}} e :^z_{[\Delta]} \sigma$ *then* $\Gamma, z{:}_{\Delta^\dagger}\tau; \Delta^\dagger, \Delta' \vdash \rho \Rightarrow e :^z_{\Delta^\dagger} \sigma$, *for* any $\Delta^\dagger$

The *Irrelevance* lemma shows that if a case alternative is well-typed with irrelevant resources, then it is well-typed with arbitrary resources. However, typing a case alternative with irrelevant resources is not complete wrt using arbitrary resources – a counter example needs only to use a resource directly.

We prove type safety of the Linear Core system via the standard type preservation and progress results. As is customary, we make use of multiple substitution lemmas, one for each kind of variable: unrestricted variables $x{:}_\omega\tau$, linear variables $x{:}_1\tau$, and Δ-bound variables $x{:}_\Delta\sigma$. The development is reported in [Mesquita and Toninho 2025a], where type safety is established directly for the instrumented semantics and ported to the natural semantics via a bisimulation argument.

We adopt the standard semantic technique of dealing with badly-formed terms, such as trying to add two values that are not numbers, by having such terms be stuck. In our setting, linear variables used more than once result in stuck states. Thus, progress ensures that linear functions consume their arguments at most once if their result is consumed exactly once. Type preservation ensures that evaluation of a closed term returns an evaluation environment with no linear variables.

Theorem 4.5 (Type Preservation). *For any well-typed evaluation state a, if $a \Downarrow b$ or $a \Downarrow^* b$ then b is well-typed.*

Theorem 4.6 (Progress). *Let a be a well-typed evaluation state. For any partial evaluation $a \Downarrow^* b$ the evaluation can be extended.*

4.3 Optimisations Preserve Linearity

A main goal of the Linear Core type system is to serve as a typed intermediate representation for optimising compilers of lazy languages with linearity. In light of this, we show that multiple optimising transformations are type preserving in Linear Core.

We describe Core-to-Core transformations as $e_1 \Longrightarrow e_2$, where e_1 is an arbitrary *well-typed* expression matching a certain shape (or of a certain type), which is transformed into expression

$(\textbf{let } x{:}_\Delta\sigma = e \textbf{ in } e') \Longrightarrow \textbf{let } x{:}_\Delta\sigma = e \textbf{ in } e'[e/x]$	Inlining
$(\lambda x{:}_\pi\sigma.\ e)\ e' \Longrightarrow e[e'/x]$	β-reduction
$(\textbf{case } (K\ \overline{e}) \textbf{ of } z{:}_\Delta\sigma\ \{\overline{\rho_j \to e_j}, K\ \overline{x} \to e_i, \overline{\rho_k \to e_k}\}) \Longrightarrow e_i\overline{[e/x]}[K\ \overline{e}/z]$	Case of known constr.
$(\textbf{case } (\textbf{case } e_c \textbf{ of } z{:}_\Delta\tau\ \{\overline{\rho_{c_i} \to e_{c_i}}\}) \textbf{ of } w{:}_{[\Delta,\Delta']}\sigma'\ \{\overline{\rho_i \to e_i}\})$	
$\Longrightarrow \textbf{case } e_c \textbf{ of } z{:}_\Delta\tau\ \{\overline{\rho_{c_i} \to \textbf{case } e_{c_i} \textbf{ of } w\ \{\overline{\rho_i \to e_i}\}}\}$	Case-of-case
$(\lambda y{:}_\pi\tau.\ \textbf{let } x{:}_\Delta\sigma = e \textbf{ in } e') \Longrightarrow \textbf{let } x{:}_\Delta\sigma = e \textbf{ in } \lambda y{:}_\pi\tau.\ e'$	Commuting let-λ
$((\textbf{let } v = e \textbf{ in } b)\ a) \Longrightarrow \textbf{let } v = e \textbf{ in } b\ a$	Commuting let-app.
$(\textbf{case } (\textbf{let } v = e \textbf{ in } b) \textbf{ of } \{\overline{\rho \to e'}\}) \Longrightarrow \textbf{let } v = e \textbf{ in case } b \textbf{ of } \{\overline{\rho \to e'}\}$	Commuting let-case
$(\textbf{case } e \textbf{ of } \{\overline{\rho_j \to e_j}, \rho \to E[e_1], \overline{\rho_k \to e_k}\}) \qquad (x \text{ fresh})$	
$\Longrightarrow \textbf{let } x = e_1 \textbf{ in case } e \textbf{ of } \{\overline{\rho_j \to e_j}, \rho \to E[x], \overline{\rho_k \to e_k}\}$	Commuting case-let
$(\textbf{let } x = (\textbf{let } v = e \textbf{ in } b) \textbf{ in } c) \Longrightarrow \textbf{let } v = e \textbf{ in let } x = b \textbf{ in } c$	Commuting let-let
$f \Longrightarrow \lambda x.(f\ x)$	η-expansion
$\lambda x.(f\ x) \Longrightarrow f$	η-reduction
$(\textbf{case } x \textbf{ of } z\ \{\overline{\rho_i \to e_i}\}) \Longrightarrow \textbf{case } x \textbf{ of } z\ \{\overline{\rho_i \to e_i[z/x]}\}$	Binder swap

Fig. 8. Optimising transformations validated by Linear Core (for commuting case-let we require e_1 not to capture the case binder, variables bound by the pattern ρ and the context $E[-]$ – see [Mesquita and Toninho 2025a] for details).

e_2. Validating a transformation entails showing that e_2 is well-typed. We often annotate the arrow with the name of the transformation when describing chains of such transformations.

The list of transformations we have validated is given in Figure 8. The behaviour of most transformations can be inferred from its name: Inlining substitutes a let-bound expression in the let body; β-reduction and case of known constructor effectively perform evaluation of the given expression; case-of-case commutes cases in scrutinee position inside the outer case alternatives; the five commuting conversions allow let-bound expressions to be commuted with other Core constructs; η-expansion and reduction apply the η-laws to terms of function type; finally, binder swap substitutes occurrences of a scrutinee variable for the case binder. While many of the transformations above are not obviously optimising, they are chained in the compiler pipeline to expose optimisation opportunities.

Reverse Binder Swap. Reverse binder swap is dual to the binder swap transformation, substituting occurrences of the case binder z by the scrutinee in the special case when the scrutinee is a variable x. By using the scrutinee x instead of the case binder, we might be able to float out expressions from the alternative using the case binder:

$$\textbf{case } x \textbf{ of } z\ \{\rho \to e\} \Rightarrow \textbf{case } x \textbf{ of } z\ \{\rho \to e[x/z]\}$$

Even though GHC applies the reverse binder swap transformation in the Core-to-Core passes, this optimisation violates linearity in Linear Core because the transformed program is rejected – the scrutinee x, a single variable, is not considered to be in WHNF by our system, thus occurrences of x in any alternative are ill-typed in it. This is a conservative choice in λ^π_Δ for typing the subtle case where the scrutinee is a single variable.

Consider **case** x **of** $\_ \to x$. Perhaps surprisingly, this expression can be considered semantically linear under call-by-need but violates linearity under call-by-name: Under call-by-need, if x refers to an unevaluated expression, scrutinizing it forces the expression bound by x to WHNF. In a subsequent use of x in an alternative, x refers to the already evaluated scrutinee in WHNF. Since x is just another name (like the case binder) for the scrutinee in WHNF, its use is also valid in the alternatives. Dually, if x refers to an expression already in WHNF, no evaluation takes place and x is still just another name for the scrutinee in WHNF.

However, under call-by-name, evaluation of x is not shared:

$$(\lambda x.\ \textbf{case}\ x\ \textbf{of}\ \_ \to x)\ (use\ y) \Longrightarrow_{\text{CBName}} \textbf{case}\ use\ y\ \textbf{of}\ \_ \to use\ y$$

After the reduction, y is a linear variable consumed both in the scrutinee and in the case alternative – y has been duplicated! If the application is instead reduced call-by-need, y is not duplicated:

$$(\lambda x.\ \textbf{case}\ x\ \textbf{of}\ \_ \to x)\ (use\ y) \Longrightarrow_{\text{CBNeed}} \textbf{let}\ x = use\ y\ \textbf{in case}\ x\ \textbf{of}\ \_ \to x$$

In an optimising compiler such as GHC, there is no definitive evaluation strategy. It is mostly up to heuristics to determine what to inline *à la* call-by-name and what to create let-bindings for *à la* call-by-need. In this flexible evaluation setting, we highlight the reverse binder swap is only a linearity-preserving transformation as long as a linear variable with more than one occurrence is never substituted for an expression with free linear variables.

5 Linear Core as a GHC Plugin

We implemented Linear Core as a plugin for the Glasgow Haskell Compiler. GHC plugins allow developers to inspect and modify programs being compiled by GHC, at different stages of compilation [Pickering et al. 2019]. The Linear Core GHC plugin [Mesquita and Toninho 2025b] is a typechecker. For every single intermediate Core program produced by GHC, our plugin checks the linearity of all expressions, failing if a linear resource is not used *exactly once* according to our typing discipline. The plugin implementation of Linear Core provides us with strong confidence that the Linear Core type system is suitable for the intermediate language of an optimizing compiler, since it is successful in accepting the vast majority of real-world linear programs produced by the GHC optimizer, and even identified some which violate linearity.

The implementation of a Linear Core checker does not follow directly from the type system because Linear Core is not syntax-directed (the design aims to be as simple to reason with as possible). Specifically, the features of Linear Core that are not syntax-directed are essentially the splitting of linear resources amongst sub-derivations and consuming fragments of resources through pattern-bound variables. As standard in substructural typing systems, our rules non-deterministically split linear resources as needed. In our implementation, we thread input/output linear resources through each derivation [Cervesato et al. 2000]. To use pattern variables with usage environment $[\Delta]\ \#K_i$, the scrutinee resources must first be Split into fragments. Instead of guessing which resources need to be split, the implementation consumes tagged fragments of a resource as needed, i.e., only when consuming a resource tagged K_j do we Split that resource on K and consume the K_j fragment. Finally, our implementation must infer the usage environments of binders in a recursive let group before using them to typecheck the let body. We use a naive $O(n^2)$ algorithm (where n is the number of let bindings) to determine these usage environments. We leave the study of the soundness and completeness of this implementation strategy to future work.

The results of running the Linear Core GHC plugin on real-world libraries with linear types are reported in Figure 9. We compiled the Haskell standard library for programming with linear types, `linear-base` (4000 lines, comprising over 100 modules), `linear-smc` [Bernardy and Spiwack 2021] (1500 lines), `priority-sesh` [Kokke and Dardha 2021] (1400 lines), `text-builder-linear` [Lelechenko 2024] (1800 lines), and `linear-generics` [Spiwack et al. 2025] (2000 lines), using our plugin. We count the number of programs accepted by our implementation, where each top-level binding in a module counts as a program, and every such binding is checked once per optimisation pass, i.e. we check all intermediate programs produced by GHC. The total of rejected programs is counted in the same way, without halting compilation. We note that our implementation is not performance conscious and types every intermediate program from scratch.

Library	Total Accepted	Total Rejected	Unique Rejected	Accept Rate
linear-base	48018	735	51	98.50%
linear-smc	18759	5	3	99.97%
priority-sesh	760	95	6	88.89%
text-builder-linear	4858	34	6	99.30%
linear-generics	63332	0	0	100.00%

Fig. 9. Linear Core Plugin on Linear Libraries with GHC 9.10.3

The results indicate that our mostly direct implementation of Linear Core is successful in accepting the vast majority of the thousands of intermediate programs produced by GHC when compiling libraries that make extensive use of linear types. However, Linear Core does not accept every single program produced by the optimisation pipeline of GHC. Our manual analysis of rejected programs revealed excellent results. Programs considered ill-typed by the Linear Core implementation include programs that:

- we expected GHC to produce, but know are not seen as linear by λ^{π}_{Δ}. Namely, programs resulting from the reverse binder swap which scrutinize a variable and then use it again in the case alternatives. As discussed in Section 4.3, these programs could be understood as linear as long as all applications of linear functions are reduced call-by-need
- were rejected due to the lack of "linearity coercions" in λ^{π}_{Δ}. Extending Linear Core with these coercions would make these programs be accepted (as discussed in Section 6.1);
- actually violate linearity. These are the programs we ultimately want to spot and reject by having a linearly typed Core validate intermediate programs. We found both a program which outright discarded a linear resource and that rewrite rules implementing stream fusion [Coutts et al. 2007; Gill et al. 1993] (among other rules) could produce linearly-invalid programs (in one such case, the type of build was not general enough to accommodate linearity).

Besides validating that our implementation is faithful to the λ^{π}_{Δ} system, these programs provide insight as to what more is needed to fully understand linearity in a mature optimising compiler.

6 Related and Future Work

Linear Haskell. Linear Haskell [Bernardy et al. 2017b] augments the Haskell surface language with linear types, but it is not concerned with extending GHC's intermediate language(s), which presents its own challenges. However, the implementation of Linear Haskell in GHC does modify and extend Core with linearity/multiplicity annotations. Core's type system is unable to type semantic linearity of programs in the sense elaborated in our work, which results in Core rejecting most linear programs resulting from optimising transformations that leverage the non-strict semantics of Core. Our work overcomes the limitations of Core's linear type system derived from Linear Haskell by understanding linearity semantically in the presence of laziness, and by showing that multiple Core-to-Core optimisations employed by GHC are linearity preserving, using a linearity-aware semantics that is essentially identical to that of [Bernardy et al. 2017a]. Linear Core can also be seen as a system that validates the programs written in Linear Haskell and are compiled by GHC, by guaranteeing (through typing) that linear resources are still used exactly once throughout the optimising transformations. We note that both Linear Haskell and our work has no special treatment of exceptions. This is also the case in GHC Core, where exceptions have no special status and so there is no natural way of dealing with the interaction between exceptions and linearity in Core itself without a major overhaul of the exception mechanisms of the language.

Linearity-aware Semantics. Several other work explore linearity aware semantics, dating back to [Chirimar et al. 1996] and [Turner and Wadler 1999]. The former uses a computational interpretation of linear logic to give an account of reference counting. The latter present a heap-based operational interpretation of linear logic and explore memory usage properties under call-by-name and call-by-need, showing both satisfy different properties under this lens. Their semantics is similar to ours, but does not account for pattern matching. More recently, [Choudhury et al. 2021; Marshall et al. 2022] and [Bernardy et al. 2017a,b] all make use of usage-aware semantics. Choudhury et al. [2021] do so in the context of graded dependent type theory, using the usage-aware heap-based operational semantics to show correct accounting of resource usage in their setting, as we do in ours, but they do not model call-by-need, only call-by-name. The work of [Marshall et al. 2022] is applies the ideas of [Choudhury et al. 2021] to a non-graded call-by-name setting. Bernardy et al. [2017b] provide a similar semantics to ours, but present only a naive linear typing system that does not leverage non-strict evaluation. We also note the non-strict language Clean [Brus et al. 1987] that employs uniqueness types to ensure resources are used at most once.

Inspirations for Linear Core. Linear Mini-Core [Bernardy et al. 2020] is a specification of linear types in Core as they were being implemented in GHC, and doubles as the (unpublished) precursor to our work. Linear Mini-Core first observes the incapacity of Core's type system to accept linear programs after transformations, and introduces usage environments for let-bound variables with the same goal of Linear Core of specifying a linear type system for Core that accepts the optimising transformations. We draw from Linear Mini-Core the rule for non-recursive let expressions and how let-bound variables are annotated with a usage environment. However, our work further explores the interaction of laziness with linearity in depth, and diverges significantly in rules for typing other constructs, notably, case expressions and case alternatives. Furthermore, unlike Mini-Core, we prove Linear Core is type safe, guarantees linear resource usage, and that multiple optimising transformations, when applied to Linear Core programs, preserve linearity. Our irrelevant resources have similarities with 0-multiplicity values from QTT [Brady 2021] which are disallowed from being used directly in executable code and, as previously noted, our tagged resources are related to fractional permissions from separation logic [Boyland 2003].

Linearity-directed Optimisations. Core-to-Core transformations appear in multiple papers [Baker-Finch et al. 2004; Breitner 2016; Maurer et al. 2017; Peyton Jones and Marlow 2002; Peyton Jones and Partain 1996; Peyton Jones and Santos 1997; Santos and Peyton Jones 1995; Sergey et al. 2017], all designed in the context of a typed language (Core) which does not have linear types. However, some authors [Bernardy et al. 2017b; Peyton Jones and Partain 1996; Peyton Jones and Santos 1997] have observed that optimisations (in particular, let-floating and inlining) can greatly benefit from linearity analysis and, in order to improve those transformations, special purpose linear-type-inspired systems were created and implemented. Preserving linear types in Core throughout the compilation pipeline allows the optimiser to leverage non-heuristic linearity information where, previously, it would resort solely to ad-hoc or incomplete custom-built linearity inference passes (naturally, these passes are still necessary to optimise programs not using explicit linearity or multiplicity annotations). Linearity may potentially be used to the benefit of other transformations. An obvious candidate is *inlining*, which is applied based on heuristics from information provided by the *cardinality analysis* pass that counts occurrences of bound variables. Linearity can be used to non-heuristically inform the inliner [Bernardy et al. 2017b].

6.1 Future Work

We highlight some avenues of future work. Briefly, these include *multiplicity coercions*, optimisations leveraging linearity, and incorporating Linear Core in GHC Haskell.

Multiplicity Coercions. Linear Core does not have type equality coercions, a flagship feature of GHC Core's type system. Coercions allow the Core intermediate language to encode a panoply of Haskell source type-level features such as GADTs, type families, or newtypes. In Linear Haskell, multiplicities are introduced as annotations to function arrows which specify the linearity of the function. In practice, multiplicities are simply types of kind Multiplicity , where One and Many are the type constructors of the kind Multiplicity ; multiplicity polymorphism follows from type polymorphism, where multiplicity variables are just type variables. Encoding multiplicities as types allows Haskell programs to leverage type-level features when handling multiplicities. For instance, it is possible via the use of GADTs to define a function whose linearity of its second argument depends on the value of its first argument, internally realized through so-called *multiplicity coercions.* Currently, Core cannot make use of such coercions to determine whether the usages of linear resources match their intended multiplicity. Studying the interaction between coercions and multiplicities is a main avenue of future work for Linear Core.

Linear Core in the Glasgow Haskell Compiler. Integrating Linear Core in the Glasgow Haskell Compiler is one of the ultimate goals of our work. Core's current type system ignores linearity due to its limitation in understanding semantic linearity, and our work fills this gap and would allow Core to be linearly typed all throughout. Implementing Linear Core in GHC is a challenging endeavour, since we must account for all other Core features (e.g. strict constructor fields), propagate new or modified data throughout the entire pipeline, and accept more optimisations. Despite our initiative in this direction[3], we leave this as future work.

Generalizing Linear Core to Haskell. Linear types, despite their compile-time correctness guarantees regarding resource management, impose a burden on programmers in being a restrictive typing discipline (witnessed, e.g., by Linear Constraints [Spiwack et al. 2022]). Linear Core eases the restrictions of linear typing by being more flexible in understanding linearity for lazy languages. It is future work to apply our ideas to the surface Haskell language.

7 Data-Availability Statement

This work has a companion artifact [Toninho and Mesquita 2025] which contains the implementation described in Section 5.

Acknowledgments

We would like to thank Arnaud Spiwack for many insightful discussions on linearity in Core and the reviewers for their feedback, which improved the paper significantly. This work was supported by Well-Typed LLP and by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects UID/50021/2025 and UID/PRR/50021/2025.

[3]https://gitlab.haskell.org/ghc/ghc/-/merge_requests/10310

References

Jean-Marc Andreoli. 1992. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation* 2, 3 (06 1992), 297–347. arXiv:https://academic.oup.com/logcom/article-pdf/2/3/297/6137548/2-3-297.pdf doi:10.1093/logcom/2.3.297

Clem Baker-Finch, Kevin Glynn, and Simon Peyton Jones. 2004. Constructed Product Result Analysis for Haskell. *Journal of Functional Programming* 14, 2 (March 2004), 211–245. https://www.microsoft.com/en-us/research/publication/constructed-product-result-analysis-haskell/

Andrew Barber. 1996. *Dual Intuitionistic Linear Logic.* Technical Report ECS-LFCS-96-347. The University of Edinburgh.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017a. Linear Haskell: practical linearity in a higher-order polymorphic language. *CoRR* abs/1710.09756 (2017). arXiv:1710.09756 http://arxiv.org/abs/1710.09756

J. Bernardy, R. Eisenberg, M. Boespflug, R. Newton, S. Peyton Jones, and A. Spiwack. 2020. Linear Mini-Core. https://gitlab.haskell.org/ghc/ghc/-/wikis/uploads/355cd9a03291a852a518b0cb42f960b4/minicore.pdf.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017b. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.* 2, POPL, Article 5 (Dec. 2017), 29 pages. doi:10.1145/3158093

Jean-Philippe Bernardy and Arnaud Spiwack. 2021. Evaluating Linear Functions to Symmetric Monoidal Categories. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell.* Association for Computing Machinery, New York, NY, USA, 14–26. doi:10.1145/3471874.3472980

John Boyland. 2003. Checking interference with fractional permissions. In *Proceedings of the 10th International Conference on Static Analysis* (San Diego, CA, USA) *(SAS'03).* Springer-Verlag, Berlin, Heidelberg, 55–72.

Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:26. doi:10.4230/LIPIcs.ECOOP.2021.9

Joachim Breitner. 2016. *Lazy Evaluation: From natural semantics to a machine-checked compiler transformation.* Ph. D. Dissertation. Karlsruher Institut für Technologie (KIT). doi:10.5445/IR/1000054251

T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. 1987. Clean — A language for functional graph rewriting. In *Functional Programming Languages and Computer Architecture*, Gilles Kahn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–384.

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR 2010 - Concurrency Theory*, Paul Gastin and François Laroussinie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 222–236.

Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. 2000. Efficient resource management for linear logic proof search. *Theoretical Computer Science* 232, 1 (2000), 133–163. doi:10.1016/S0304-3975(99)00173-5

Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. 1996. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming* 6, 2 (1996), 195–244. doi:10.1017/S0956796800001660

Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. 2021. A graded dependent type system with a usage-aware semantics. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–32. doi:10.1145/3434331

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany) *(ICFP '07).* Association for Computing Machinery, New York, NY, USA, 315–326. doi:10.1145/1291151.1291199

Luís Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, Richard A. DeMillo (Ed.). ACM Press, 207–212. doi:10.1145/582153.582176

Roy Dyckhoff. 1992. Contraction-Free Sequent Calculi for Intuitionistic Logic. *The Journal of Symbolic Logic* 57, 3 (1992), 795–807. http://www.jstor.org/stable/2275431

Peng Fu, Kohei Kishida, and Peter Selinger. 2020. Linear Dependent Type Theory for Quantum Programming Languages: Extended Abstract. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20).* Association for Computing Machinery, New York, NY, USA, 440–453. doi:10.1145/3373718.3394765

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark) *(FPCA '93).* Association for Computing Machinery, New York, NY, USA, 223–232. doi:10.1145/165180.165214

Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. doi:10.1016/0304-3975(87)90045-4

Wen Kokke and Ornela Dardha. 2021. Deadlock-Free Session Types in Linear Haskell. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell* (Virtual, Republic of Korea) *(Haskell 2021).* Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3471874.3472979

John Launchbury. 1993. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93).* Association for

Computing Machinery, New York, NY, USA, 144–154. doi:10.1145/158511.158618

Andrew Lelechenko. 2024. text-builder-linear: Builder for Text and ByteString based on linear types. https://hackage.haskell.org/package/text-builder-linear

Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2022. Linear Types for Large-Scale Systems Verification. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 69 (apr 2022), 28 pages. doi:10.1145/3527313

Danielle Marshall, Michael Vollmer, and Dominic Orchard. 2022. Linearity and Uniqueness: An Entente Cordiale. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer, 346–375. doi:10.1007/978-3-030-99336-8_13

Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. *Ada Lett.* 34, 3 (oct 2014), 103–104. doi:10.1145/2692956.2663188

Luke Maurer, Zena Ariola, Paul Downen, and Simon Peyton Jones. 2017. Compiling without continuations. In *ACM Conference on Programming Languages Design and Implementation (PLDI'17)*. ACM, 482–494. https://www.microsoft.com/en-us/research/publication/compiling-without-continuations/

Rodrigo Mesquita and Bernardo Toninho. 2025a. Lazy Linearity for a Core Functional Language. *CoRR* abs/2511.10361 (2025). arXiv:2511.10361 https://arxiv.org/abs/2511.10361

Rodrigo Mesquita and Bernardo Toninho. 2025b. Linear Core: Theory and a GHC Plugin. https://hackage.haskell.org/package/linear-core-prototype-0.1.0.0

Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming* 12 (July 2002), 393–434. https://www.microsoft.com/en-us/research/publication/secrets-of-the-glasgow-haskell-compiler-inliner/

Simon Peyton Jones and Will Partain. 1996. Let-Floating: Moving Bindings to Give Faster Programs. *Proc. of ICFP'96* 31 (10 1996). doi:10.1145/232629.232630

Simon Peyton Jones and Andre Santos. 1997. A transformation-based optimiser for Haskell. *Science of Computer Programming* 32, 1 (October 1997). https://www.microsoft.com/en-us/research/publication/a-transformation-based-optimiser-for-haskell/

Simon L. Peyton Jones. 1987. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., USA.

Matthew Pickering, Nicolas Wu, and Boldizsár Németh. 2019. Working with Source Plugins. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell* (Berlin, Germany) *(Haskell 2019)*. Association for Computing Machinery, New York, NY, USA, 85–97. doi:10.1145/3331545.3342599

Benjamin C. Pierce. 2004. *Advanced Topics in Types and Programming Languages*. The MIT Press.

Andre Santos and Simon Peyton Jones. 1995. *Compilation by transformation for non-strict functional languages*. Ph. D. Dissertation. https://www.microsoft.com/en-us/research/publication/compilation-transformation-non-strict-functional-languages/

Ilya Sergey, Dimitrios Vytiniotis, Simon L. Peyton Jones, and Joachim Breitner. 2017. Modular, higher order cardinality analysis in theory and practice. *Journal of Functional Programming* 27 (2017), e11. doi:10.1017/S0956796817000016

Arnaud Spiwack, David Feuer, and José Pedro Magalhães. 2025. linear-generics: Generic programming library for generalised deriving. https://hackage.haskell.org/package/linear-generics

Arnaud Spiwack, Csongor Kiss, Jean-Philippe Bernardy, Nicolas Wu, and Richard A. Eisenberg. 2022. Linearly Qualified Types: Generic Inference for Capabilities and Uniqueness. *Proc. ACM Program. Lang.* 6, ICFP, Article 95 (aug 2022), 28 pages. doi:10.1145/3547626

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (Nice, Nice, France) *(TLDI '07)*. Association for Computing Machinery, New York, NY, USA, 53–66. doi:10.1145/1190315.1190324

Bernardo Toninho and Rodrigo Mesquita. 2025. *Lazy Linearity for a Core Functional Language (Artifact)*. doi:10.5281/zenodo.17713905

David N. Turner and Philip Wadler. 1999. Operational interpretations of linear logic. *Theoretical Computer Science* 227, 1 (1999), 231–248. doi:10.1016/S0304-3975(99)00054-7

Philip Wadler. 1990. Linear Types can Change the World!. In *Programming Concepts and Methods*.

Received 2025-07-10; accepted 2025-11-06