Type-checking Linearity in Core/System FC

Rodrigo Mesquita
Bernardo Toninho

March 29, 2023

Abstract

Linear types were added both to Haskell and to its Core intermediate language, which is used as an internal consistency tool to validate the transformations a Haskell program undergoes during compilation. However, the current Core type-checker rejects many linearly valid programs that originate from Core-to-Core optimizing transformations. As such, linearity typing is effectively disabled, for otherwise disabling optimizations would be far more devastating. The goal of our proposed dissertation is to develop an extension to Core's type system that accepts a larger amount of programs and verifies that optimizing transformations applied to well-typed linear Core produce well-typed linear Core. Our extension will be based on attaching variable *usage environments* to binders, which augment the type system with more fine-grained contextual linearity information, allowing the system to accept programs which seem to syntactically violate linearity but preserve linear resource usage. We will also develop a usage environment inference procedure and integrate the procedure with the type checker. We will validate our proposal by showing a range of Core-to-Core transformations can be typed by our system.

Resumo

Tipos lineares foram integrados ambos no Haskell e na sua linguagem intermédia, Core, que serve como uma ferramenta de consistência interna do compilador que valida as transformações feitas nos programas ao longo do processo de compilação. No entanto, o sistema de tipos do Core rejeita programas lineares válidos que são produto de optimizações Core-to-Core, de tal forma que a validação da linearidade ao nível do sistema de tipos não consegue ser desempenhada com sucesso, sendo que a alternativa, não aplicar optimizações, tem resultados bastante mais indesejáveis. O objetivo da dissertação que nos propomos a fazer é extender ao sistema de tipos do Core de forma a aceitar mais programas lineares, e verificar que as optimizações usadas não destroem a linearidade dos programas. A nossa extensão parte de adicionar *ambientes de uso* às variáveis, aumentando o sistema de tipos com informação de linearidade suficiente para aceitar programas que aparentemente violam linearidade sintaticamente, mas que a preservam a um nível semântico. Para além do sistema de tipos, vamos desenvolver um algoritmo de inferência de *ambientes de uso*. Vamos validar a nossa proposta através do conjunto de transformações Core-to-Core que o nosso sistema consegue tipificar.

Contents

List of Figures

CHAPTER 1

Introduction

Statically safe programming languages provide compile time correctness guarantees by having the compiler rule out certain classes of errors or invalid programs. Moreover, static typing allows programmers to state and enforce (compile-time) invariants relevant to their problem domain. In this sense, type safety entails that all possible executions of a type-correct program cannot exhibit behaviors deemed "wrong" by the type system design. This idea is captured in the motto "well-typed programs do not go wrong".

Linear type systems [21, 3] add expressiveness to existing type systems by enforcing that certain *resources* (e.g. a file handle) must be used *exactly once*. In programming languages with a linear type system, not using certain resources or using them twice are flagged as type errors. Linear types can thus be used to, for example, statically guarantee that file handles, socket descriptors, and allocated memory is freed exactly once (leaks and double-frees become type errors), and channel-based communication protocols are deadlock-free [7], among other high-level correctness properties [19, 30, 4].

As an example, consider the following C-like program in which allocated memory is freed twice. We know this to be the dreaded double-free error which will crash the program at runtime. Regardless, a C-like type system will accept this program without any issue.

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

Under the lens of a linear type system, consider the variable p to be a linear resource created by the call to `malloc`. Since p is linear, it must be used *exactly once*. However, the program above uses p twice, in the two different calls to `free`. With a linear type system, the program above *does not typecheck*! In this sense, linear typing effectively ensures the program does not compile with a double-free error. In Section 2.1 we give a formal account of linear types and provide additional examples.

Despite their promise and their extensive presence in research literature [49, 8, 1], the effective design of the combination of linear and non-linear typing is both challenging and necessary to bring the advantages of linear typing to mainstream languages. Consequently, few general purpose programming languages have linear type systems. Among them are Idris 2 [5], a linearly and dependently typed language based on Quantitative Type Theory, Rust [33], a language whose ownership types build on linear types to guarantee memory safety without garbage collection or reference counting, and, more recently, Haskell [4], a *purely functional* and *lazy* language.

Linearity in Haskell stands out from linearity in Rust and Idris 2 due to the following reasons:

- Linear types were only added to the language roughly *31 years after* Haskell's inception, unlike Rust and Idris 2 which were designed with linearity from the start. It is an especially difficult endeavour to add linear types to a well-established language with a large library ecosystem and many active users, rather than to develop the language from the ground up with linear types in mind, and the successful approach as implemented in GHC 9.0, the leading Haskell compiler, was based on Linear Haskell [4], where a linear type system designed with retaining backwards-compatibility and allowing code reuse across linear and non-linear users of the same libraries in mind was described. We describe Linear Haskell in detail in Section 2.3.

- Linear types permeate Haskell down to (its) **Core**, the intermediate language into which Haskell is translated. **Core** is a minimal, explicitly typed, functional language, on which multiple Core-to-Core optimizing transformations are defined. Due to Core's minimal design, typechecking Core programs is very efficient and doing so serves as a sanity check to the correction of the source transformations. If the resulting optimized Core program fails to typecheck, the optimizing transformations (are very likely to) have introduced an error in the resulting program. We present Core (and its formal basis, System F_C [47]) in Section 2.4.

Aligned with the philosophy of having a *typed* intermediate language, the integration of linearity in Haskell required extending **Core** with linear types. Just as a *typed* Core ensures that the translation from Haskell (dubbed *desugaring*) and the subsequent optimizing transformations are correctly implemented, a *linearly typed* Core guarantees that linear resource usage in the source language is not violated by the translation process and the compiler optimization passes. It is crucial that the program behaviour enforced by linear types is *not* changed by the compiler in the desugaring or optimization stages (optimizations should not destroy linearity!) and a linearity aware Core typechecker is key in providing such guarantees. Additionally, a linear Core can inform Core-to-Core optimizing transformations [38, 39, 4] in order to produce more performant programs.

While the current version of Core is linearity-aware (i.e., Core has so-called multiplicity annotations in variable binders), its type system does not (fully) validate the linearity constraints in the desugared program and essentially fails to type-check programs resulting from several optimizing transformations that are necessary to produce efficient object code. The reason for this latter point is not evidently clear: if we can typecheck linearity in the surface level Haskell why do we fail to do so in Core? The desugaring process from surface level Haskell to Core, and the subsequent Core-to-Core optimizing transformations, eliminate and rearrange most of the syntactic constructs through which linearity checking is performed – often resulting in programs completely different from the original.

However, these transformed programs that no longer type-check because of linearity are *semantically linear*, that is, linear resources are still used exactly once, despite the type-system no longer accepting those programs. In order to maintain Core linearly-typed accross transformations, Core must be extended with additional linearity information to allow type-checking linearity in Core where we currently do not.

Concluding, by extending Core / System F_C with linearity and multiplicity annotations such that we can desugar all of Linear Haskell and validate it accross transformations taking into consideration Core's call-by-need semantics, we can validate the surface level linear type's implementation, we can guarantee optimizing transformations do not destroy linearity, and we might be able to inform optimizing transformations with linearity.

Goals

From a high-level view, our goals for the dissertation include:

- Extending Core's type system and type-checking algorithm with additional linearity information in order to successfully type-check linearity in Core across transformations;
- Validating that our type-system accepts programs before and after each transformation is applied;
- Arguing the soundness of the resulting system (i.e. no semantically non-linear programs are deemed linear);
- Implementing our extensions to Core in GHC, the leading Haskell Compiler.

CHAPTER 2

Background and Related Work

In this section we review the concepts required to understand our work. In short, we discuss linear types, the Haskell programming language, linear types as they exist in Haskell (dubbed Linear Haskell), Haskell's main intermediate language (Core) and its formal foundation (System F_C) and, finally, an overview of GHC's pipeline with explanations of some Core-to-Core optimizing transformations.

2.1 Linear Types

Much the same way type systems can statically eliminate various kinds of programs that would fail at runtime, such as a program that derreferences an integer value rather than a pointer, linear type systems can guarantee that certain errors (regarding resource usage) are forbidden.

In linear type systems [21, 3], so called linear resources must be used *exactly once*. Not using a linear resource at all or using said resource multiple times will result in a type error. We can model many real-world resources such as file handles, socket descriptors, and allocated memory, as linear resources. This way, because a file handle must be used exactly once, forgetting to close the file handle is a type error, and closing the handle twice is also a type error. With linear types, avoiding leaks and double frees is no longer a programmer's worry because the compiler can guarantee the resource is used exactly once, or *linearly*.

To understand how linear types are defined and used in practice, we present two examples of anonymous functions that receive a handle to work with (that must be closed before returning), we explore how the examples could be disregarded as incorrect, and work our way up to linear types from them. The first function ignores the received file handle and returns $\star$ (read unit), which is equivalent to C's `void`.

$$\lambda h.\ \mathsf{return}\ \star; \qquad\qquad \lambda h.\ \mathsf{close}\ h;\ \mathsf{close}\ h;$$

Ignoring the file handle which should have been closed by the function makes the first function incorrect. Similarly, the second function receives the file handle and closes it twice, which is incorrect not because it falls short of the specification, but rather because the program will crash while executing it. Additionally, both functions share the same type, `Handle` $\rightarrow \star$, i.e. a function that takes a `Handle` and returns $\star$. The second function also shares this type because `close` has type `Handle` $\rightarrow \star$. Under a simple type system

such as C's, both functions are type correct (the compiler will happily succeed), but both have erroneous behaviours. The first forgets to close the handle and the second closes it twice. Our goal is to reach a type system that rejects these two programs.

The key observation to invalidating these programs is to focus on the function type going between `Handle` and $\star$ and augment it to indicate that *the argument must be used exactly once*, or, in other words, that the argument of the function must be linear. We take the function type $A \to B$ and replace the function arrow ($\to$) with the linear function arrow ($\multimap$)[1] operator to denote a function that uses its argument exactly once: $A \multimap B$. Providing the more restrictive linear function signature `Handle` $\multimap \star$ to the example programs would make both of them fail to typecheck because they do not satisfy the linearity specification that the function argument should only be used exactly once.

In order to further give well defined semantics to a linear type system, we present a linearly typed lambda calculus [21, 3], a very simple language with linear types, by defining what are syntatically valid programs through the grammar in Fig. 2.1 and what programs are well typed through the typing rules in Fig. 2.2. The language features functions and function application ($\multimap$), two flavours of pairs, additive (&) and multiplicative ($\otimes$), a disjunction operator ($\oplus$) to construct sum types, and the ! modality operator which constructs an unrestricted type from a linear one, allowing values inhabiting $!A$ to be consumed unrestrictedly. A typing judgement for the linearly typed lambda calculus has the form

$$\Gamma; \Delta \vdash M : A$$

where Γ is the context of resources that may be used unrestrictedly, that is, any number of times, Δ is the context of resources that must be used linearly (*exactly once*), M is the program to type and A is its type. When resources from the linear context are used, they are removed from the context and no longer available, and all resources in the linear context must be used exactly once.

$$\begin{array}{rcl} A, B & ::= & \star \\ & | & A \multimap B \\ & | & A \oplus B \\ & | & A \otimes B \\ & | & A \,\&\, B \\ & | & !A \end{array} \qquad \begin{array}{rcl} M, N & ::= & \star \mid \mathsf{let}\ \star = M\ \mathsf{in}\ N \\ & | & u \\ & | & \lambda u.M \mid M\ N \\ & | & \mathrm{inl}\ M \mid \mathrm{inr}\ M \\ & | & \mathsf{case}\ M\ \mathsf{of}\ \mathrm{inl}\ u_1 \to N_1; \mathrm{inr}\ u_2 \to N_2 \\ & | & M \otimes N \mid \mathsf{let}\ u_1 \otimes u_2 = M\ \mathsf{in}\ N \\ & | & M \,\&\, N \mid \mathrm{fst}\ M \mid \mathrm{snd}\ M \\ & | & !M \mid \mathsf{let}\ !u = M\ \mathsf{in}\ N \end{array}$$

Figure 2.1: Grammar for a linearly-typed lambda calculus

The function abstraction is typed according to the linear function introduction rule ($\multimap I$). The rule states that a a function abstraction, written $\lambda u.M$, is a linear function (i.e. has type $A \multimap B$) given the unrestricted context Γ and the linear context Δ, if the program M has type B given the same unrestricted context Γ and the linear context $\Delta, u{:}A$. That is, if M has type B using u of type A exactly once besides the other resources in Δ, then the lambda abstraction has the linear function type.

$$\frac{\Gamma; \Delta, u{:}A \vdash M : B}{\Gamma; \Delta \vdash \lambda u.M : A \multimap B}\ (\multimap I) \qquad \frac{\Gamma; \Delta \vdash M : A \multimap B \qquad \Gamma; \Delta' \vdash N : A}{\Gamma; \Delta, \Delta' \vdash M\ N : B}\ (\multimap E)$$

[1] Since linear types are born from a correspondence with linear logic [21] (the Curry-Howard isomorphism [14, 24]), we borrow the $\multimap$ symbol and other linear logic connectives to describe linear types.

Function application is typed according to the elimination rule for the same type ($\multimap E$). To type an application $M\ N$ as B, M must have type $A \multimap B$ and N must have type A. To account for the linear resources that might be used while proving both that $M{:}A \multimap B$ and $N{:}A$, the linear context must be split in two such that both typing judgments succeed using exactly once every resource in their linear context (while the resources in Γ might be used unrestrictedly), hence the separation of the linear context in Δ and Δ'.

The multiplicative pair ($M \otimes N$) is constructed from two linearly typed expressions that can each be typed with a division of the given linear context, as we see in its introduction rule ($\otimes I$). Upon deconstruction, the multiplicative pair elimination rule ($\otimes E$) requires that both of the pair constituents be consumed exactly once.

$$\frac{\Gamma; \Delta \vdash M : A \qquad \Gamma; \Delta' \vdash N : B}{\Gamma; \Delta, \Delta' \vdash (M \otimes N) : A \otimes B}\ (\otimes I) \qquad \frac{\Gamma; \Delta \vdash M : A \otimes B \qquad \Gamma; \Delta', u{:}A, v{:}B \vdash N : C}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ u \otimes v\ \mathsf{in}\ N : C}\ (\otimes E)$$

On the other hand, the additive pair requires that both elements of the pair can be proved with the same linear context, and upon deconstruction only one of the pair elements might be used, rather than both simultaneously.

Finally, the "of-course" operator (!) can be used to construct a resource that can be used unrestrictedly ($!M$). Its introduction rule ($!I$) states that to construct this resource means to add a resource to the unrestricted context, which can then be used freely. To construct an unrestricted value, however, the linear context *must be empty* – an unrestricted value can only be constructed if it does not depend on any linear resource.

$$\frac{\Gamma; \cdot \vdash M : A}{\Gamma; \cdot \vdash !M : !A}\ (!I) \qquad \frac{\Gamma; \Delta \vdash M : !A \qquad \Gamma, u{:}A; \Delta' \vdash N : C}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ !u = M\ \mathsf{in}\ N : C}\ (!E)$$

To utilize an unrestricted value M, we must bind it to u with $\mathsf{let}\ !u = M\ \mathsf{in}\ N$ which can then be used in N unrestrictedly, because u extends the unrestricted context rather than the linear context as we have seen thus far.

In section 2.3 we describe how linear types are defined in Haskell, a programming language more featureful than the linearly typed lambda calculus. We will see that the theoretical principles underlying the linear lambda calculus and linear Haskell are the same, and by studying them in this minimal setting we can understand them at large.

2.2 Haskell

Haskell is a functional programming language defined by the Haskell Report [28, 31] and whose *de-facto* implementation is GHC, the Glasgow Haskell Compiler [32]. Haskell is a *lazy*, *purely functional* language, i.e., functions cannot have side effects or mutate data, and, contrary to many programming languages, arguments are *not* evaluated when passed to functions, but rather are only evaluated when its value is needed. The combination of purity and laziness is unique to Haskell among mainstream programming languages.

Haskell is a large feature-rich language but its relatively small core is based on a typed lambda calculus. As such, there exist no statements and computation is done simply through the evaluation of functions. Besides functions, one can define types and their constructors and pattern match on said constructors. Function application is denoted by the juxtaposition of the function expression and its arguments, which often means empty space between terms (`f a` means `f` applied to `a`). Pattern matching is done with the `case` keyword followed by the enumerated alternatives. All variable names start with lower case

$$\frac{}{\Gamma; u{:}A \vdash u : A}\ (u) \qquad \frac{}{\Gamma, u{:}A; \cdot \vdash u : A}\ (u)$$

$$\frac{\Gamma; \Delta \vdash M : A \qquad \Gamma; \Delta' \vdash N : B}{\Gamma; \Delta, \Delta' \vdash (M \otimes N) : A \otimes B}\ (\otimes I) \qquad \frac{\Gamma; \Delta \vdash M : A \otimes B \qquad \Gamma; \Delta', u{:}A, v{:}B \vdash N : C}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ u \otimes v\ \mathsf{in}\ N : C}\ (\otimes E)$$

$$\frac{\Gamma; \Delta, u{:}A \vdash M : B}{\Gamma; \Delta \vdash \lambda u.M : A \multimap B}\ (\multimap I) \qquad \frac{\Gamma; \Delta \vdash M : A \multimap B \qquad \Gamma; \Delta' \vdash N : A}{\Gamma; \Delta, \Delta' \vdash M\ N : B}\ (\multimap E)$$

$$\frac{\Gamma; \Delta \vdash M : A \qquad \Gamma; \Delta \vdash N : B}{\Gamma; \Delta \vdash M \,\&\, N : A \,\&\, B}\ (\&I) \qquad \frac{\Gamma; \Delta \vdash M : A \,\&\, B}{\Gamma; \Delta \vdash \mathrm{fst}\ M : A}\ (\&E_L) \qquad \frac{\Gamma; \Delta \vdash M : A \,\&\, B}{\Gamma; \Delta \vdash \mathrm{snd}\ M : B}\ (\&E_R)$$

$$\frac{\Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash \mathrm{inl}\ M : A \oplus B}\ (\oplus I_L) \qquad \frac{\Gamma; \Delta \vdash M : B}{\Gamma; \Delta \vdash \mathrm{inr}\ M : A \oplus B}\ (\oplus I_R)$$

$$\frac{\Gamma; \Delta \vdash M : A \oplus B \qquad \Gamma; \Delta', w_1{:}A \vdash N_1 : C \qquad \Gamma; \Delta', w_2{:}B \vdash N_2 : C}{\Gamma; \Delta, \Delta' \vdash \mathsf{case}\ M\ \mathsf{of}\ \mathrm{inl}\ w_1 \to N_1 \mid \mathrm{inr}\ w_2 \to N_2 : C}\ (\oplus E)$$

$$\frac{}{\Gamma; \cdot \vdash \star : \star}\ (\star I) \qquad \frac{\Gamma; \Delta \vdash M : \star \qquad \Gamma; \Delta' \vdash N : B}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ \star = M\ \mathsf{in}\ N : B}\ (\star E)$$

$$\frac{\Gamma; \cdot \vdash M : A}{\Gamma; \cdot \vdash !M : !A}\ (!I) \qquad \frac{\Gamma; \Delta \vdash M : !A \qquad \Gamma, u{:}A; \Delta' \vdash N : C}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ !u = M\ \mathsf{in}\ N : C}\ (!E)$$

Figure 2.2: Typing rules for a linearly-typed lambda calculus

and types start with upper case (excluding type variables). To make explicit the type of an expression, the `::` operator is used (e.g. `f :: Int → Bool` is read `f` *has type* function from `Int` to `Bool`).

Because Haskell is a pure programming language, input/ouptut side-effects are modelled at the type-level through the non-nullary[2] type constructor `IO`. A value of type `IO a` represents a *computation* that when executed will perform side-effects and produce a value of type `a`. Computations that do I/O can be composed into larger computations using so-called monadic operators, which are like any other operators but grouped under the same abstraction. Some of the example programs will look though as if they had statements, but, in reality, the sequential appearence is just syntatic sugar to an expression using monadic operators. The main take away is that computations that do I/O may be sequenced together with other operations that do I/O while retaining the lack of statements and the language purity guarantees.

As an example, consider these functions that do I/O and their types. The first opens a file by path and returns its handle, the second gets the size of a file from its handle, and the third closes the handle. It is important that the handle be closed exactly once, but currently nothing enforces that usage policy.

```
openFile :: FilePath → IOMode → IO Handle
hFileSize :: Handle → IO Integer
hClose :: Handle → IO ()
```

The following function makes use of the above definitions to return the size of a file

[2] `IO` has kind `Type → Type`, that is, it is only a type after another type is passed as a parameter (e.g. `IO Int`, `IO Bool`); `IO` by itself is a *type constructor*

given its path. Note that the function silently leaks the handle to the file, despite compiling successfully.

```
countWords :: FilePath → IO Integer
countWords path = do
  handle ← openFile path ReadMode
  size ← hFileSize handle
  return size
```

Another defining feature of Haskell is its powerful type system. In contrast to most mainstream programming languages, such as OCaml and Java, Haskell supports a myriad of advanced type level features, such as:

- Multiple forms of advanced polymorphism: where languages with whole program type inference usually stick to Damas–Hindley–Milner type inference [15], Haskell goes much further with, e.g., arbitrary-rank types [40], type-class polymorphism [23], levity polymorphism [17], multiplicity polymorphism [4], and, more recently, impredicative polymorphism [46].

- Type level computation by means of type classes [22] and Haskell's type families [9, 10, 18], which permit a direct encoding of type-level functions resembling rewrite rules.

- Local equality constraints and existential types by using GADTs, which we explain ahead in more detail. A design for first class existential types with bi-directional type inference in Haskell has been published in [16], despite not being yet implemented in GHC.

These advanced features have become commonplace in Haskell code, enforcing application level invariants and program correctness through the types. As an example to work through this section while we introduce compile-time invariants with GADTs, consider the definition of `head` in the standard library, a function which takes the first element of a list by pattern matching on the list constructors.

```
head :: [a] → a
head [] = error "List is empty!"
head (x : xs) = x
```

When applied to the empty list, `head` terminates the program with an error. This function is unsafe – our program might crash if we use it on an invalid input. Leveraging Haskell's more advanced features, we can use more expressive types to assert properties about the values and get rid of the invalid cases (e.g. we could define a `NonEmpty` type to model a list that can not be empty). A well liked motto is "make invalid states unrepresentable". In this light, we introduce Generalized Algebraic Data Types (GADTs) and create a list type indexed by size for which we can write a completely safe `head` function by expressing that the size of the list must be at least one, at the type level.

2.2.1 Generalized Algebraic Data Types

GADTs [11, 41, 48] are an advanced Haskell feature that allows users to define data types as they would common algebraic data types, with the added abilitiy to give explicit type signatures for the data constructors where the result type may differ in the type

parameters (e.g., we might have two constructors of the same data type `T a` return values of type `T Bool` and `T Int`). This allows for additional type invariants to be represented with GADTs through their type parameters, which restricts the use of specific constructors and their subsequent deconstruction through pattern matching. Pattern matching against GADTs can introduce local type refinements, that is, refines the type information used for typechecking individual case alternatives. We develop the length-indexed lists example without discussing the type system and type inference details of GADTs as desribed in [48].

We define the data type in GADT syntax for length-index lists which takes two type parameters. The first type parameter is the length of the list and the type of the type parameter (i.e. the kind of the first type parameter) is `Nat`. To construct a type of kind `Nat` we can only use the type constructors `Z` and `S`. The second type parameter is the type of the values contained in the list, and any type is valid, hence the `Type` kind.

```
data Vec (n :: Nat) (a :: Type) where
  Nil :: Vec Z a
  Cons :: a → Vec m a → Vec (S m) a
```

The length-indexed list is defined inductively as either an empty list *of size zero*, or the construction of a list by appending a new element to an existing list *of size m* whose final size is $m + 1$ (`S m`). The list `Cons 'a' (Cons 'b' Nil)` has type `Vec (S (S Z)) Char` because `Nil` has type `Vec Z Char` and `Cons 'a' Nil` has type `Vec (S Z) Char`. GADTs make possible different data constructors being parametrized over different type parameters as we do with `Vec`'s size parameter being different depending on the constructor that constructs the list.

To define the safe `head` function, we must specify the type of the input list taking into account that the size must not be zero. To that effect, the function takes as input a `Vec (S n) a`, that is, a vector with size (n+1) for all possible n's, making a call to `head` on a list of type `Vec Z a` (an empty list) a compile-time type error.

```
head :: Vec (S n) a → a
head (Cons x xs) = x
```

Pattern matching on the `Nil` constructor is not needed, despite it being a constructor of `Vec`. The argument type doesn't match the type of the `Nil` constructor (`S n` $\neq$ `Z`), so the corresponding pattern case alternative is innacessible because the typechecker does not allow calling `head` on `Nil` (once again, its type, `Vec Z a`, does not match the input type of `head`, `Vec (S n) a`).

In practice, the idea of using more expressive types to enforce invariants at compile time, that is illustrated by this simple example, can be taken much further, e.g., to implement type-safe artificial neural networks[29], enforce size compatibility in operations between matrices and vectors[43], to implement red-black trees guaranteeing its invariants at compile-time, or to implement a material system in a game engine[35].

Linear types are, similarly, an extension to Haskell's type system that makes it even more expressive, by providing a finer control over the usage of certain resources at the type level.

2.3 Linear Haskell

The introduction of linear types to Haskell's type system is originally described in Linear Haskell [4]. While in Section 2.6 we discuss the reasoning and design choices behind

retrofitting linear types to Haskell, here we focus on linear types solely as they exist in the language, and re-work the file handle example seen in the previous section to make sure it doesn't typecheck.

A linear function ($f :: A \multimap B$) guarantees that if ($f\ x$) is consumed exactly once, then the argument x is consumed exactly once. The precise definition of *consuming a value* depends on the value as follows, paraphrasing Linear Haskell [4]:

- 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. For example, a linear pair (equivalent to $\otimes$) is consumed exactly once if pattern-matched on *and* both the first and second element are consumed once.

In Haskell, linear types are introduced through *linearity on the function arrow*. In practice, this means function types are annotated with a linearity that defines whether a function argument must be consumed *exactly once* or whether it can be consumed *unrestrictedly* (many times). As an example, consider the function f below, which doesn't typecheck because it is a linear function (annotated with `1`) that consumes its argument more than once, and the function g, which is an unrestricted function (annoted with `Many`) that typechecks because its type allows the argument to be consumed unrestrictedly.

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

```
g :: a %Many -> (a, a)
g x = (x, x)
```

The function annotated with the *multiplicity* annotation of `1` is equivalent to the linear function type ($\multimap$) seen in the linear lambda calculus (Section 2.1). Additionally, algebraic data type constructors can specify whether their arguments are linear or unrestricted, requiring that, when pattern matched on, linear arguments be consumed once while unrestricted arguments need not be consumed exactly once. To encode the multiplicative linear pair ($\otimes$) we must create a pair data type with two linear components. To consume an algebraic data type is to consume all its linear components once, so, to consume said pair data type, we need to consume both its linear components – successfully encoding the multiplicative pair elimination rule ($\otimes E$). To construct said pair data type we must provide two linear elements, each consuming some required resources to be constructed, thus encoding the multiplicative pair introduction rule ($\otimes I$). As such, we define `MultPair` as an algebraic data type whose constructor uses a linear arrow for each of the arguments[3].

```
data MultPair a b where
  MkPair :: a %1 -> b %1 -> MultPair a b
```

The linearity annotations `1` and `Many` are just a specialization of the more general so-called *multiplicity annotations*. A multiplicity of `1` entails that the function argument must be consumed once, and a function annotated with it ($\rightarrow_1$) is called a linear function (often

[3] By default, constructors defined without GADT syntax have linear arguments. We could have written `data MultPair a b = MkPair a b` to the same effect.

written with $\multimap$). A function with multiplicity of `Many` ($\to_\omega$) is an unrestricted function, which may consume its argument 0 or more times. Unrestricted functions are equivalent to the standard function type and, in fact, the usual function arrow ($\to$) implicitly has multiplicity `Many`. Multiplicities naturally allow for *multiplicity polymorphism*, which we explain below.

Consider the functions f and g which take as an argument a function from `Bool` to `Int`. Function f expects a linear function ($Bool \to_1 Int$), whereas g expects an unrestricted function ($Bool \to_\omega Int$). Function h is a function from `Bool` to `Int` that we want to pass as an argument to both f and g.

```
f :: (Bool % 1 → Int) → Int
f c = c True

g :: (Bool → Int) → Int
g c = c False
```

```
h :: Bool % m → Int
h x = case x of
  False → 0
  True → 1
```

For the application of f and g to h to be well typed, the multiplicity of h ($\to_?$) should match the multiplicity of both f ($\to_1$) and g ($\to_\omega$). Multiplicity polymorphism allows us to use *multiplicity variables* when annotating arrows to indicate that the function can both be typed as linear and as an unrestricted function, much the same way type variables can be used to define polymorphic functions. Thus, we define h as a multiplicity polymorphic function ($\to_m$), making h a well-typed argument to both f and g (m will unify with 1 and ω at the call sites).

2.4 Core and System F_C

Haskell is a large and expressive language with many syntatic constructs and features. However, the whole of Haskell can be desugared down to a minimal, explicitly typed, intermediate language called **Core**. Desugaring allows the compiler to focus on the small desugared language rather than on the large surface one, which can greatly simplify the subsequent compilation passes. Core is a strongly-typed, lazy, purely functional intermediate language akin to a polymorphic lambda calculus, that GHC uses as its key intermediate representation. To illustrate the difference in complexity, in GHC's implementation of Haskell, the abstract syntax tree is defined through dozens of datatypes and hundreds of constructors, while the GHC's implementation of Core is defined in 3 types (expressions, types, and coercions) and 15 constructors [13]. The existence of Core and its use is a major design decision in GHC Haskell with significant benefits which have proved themselves in the development of the compiler.

- Core allows us to reason about the entirety of Haskell in a much smaller functional language. Performing analysis, optimizing transformations, and code generation is done on Core, not Haskell. The implementation of these compiler passes is significantly simplified by the minimality of Core.
- Since Core is an (explicitly) typed language (c.f. System F [20, 42]), type-checking Core serves as an internal consistency check for the desugaring and optimization passes. The Core typechecker provides a verification layer for the correctness of desugaring and optimizing transformations (and their implementations) because both desugaring and optimizing transformations must produce well-typed Core.
- Finally, Core's expressiveness serves as a sanity-check for all the extensions to the source language – if we can desugar a feature into Core then the feature must be

sound by reducibility. Effectively, any feature added to Haskell is only syntatic sugar if it can be desugared to Core.

The implementation of Core's typechecker differs significantly from the Haskell typechecker because Core is explicitly typed and its type system is based on the *System* F_C [47] type system (i.e., System F extended with a notion of type coercion), while Haskell is implicitly typed and its type system is based on the constraint-based type inference system $OutsideIn(X)$ [48]. Therefore, typechecking Core is simple, fast and requires no type inference, whereas Haskell's typechecker must account for almost the entirety of Haskell's syntax, and must perform type-inference in the presence of arbitrary-rank polymorphism, existential types, type-level functions, and GADTs, which are known to introduce significant challenges for type inference algorithms [48]. Haskell is typechecked in addition to Core to elaborate the user program. This might involve performing type inference to make implicit types explicit and solving constraints to pass implicit dictionary arguments explicitly. Furthermore, type-checking the source language allows us to provide meaningful type errors. If Haskell wasn't typechecked and instead we only typechecked Core, everything (e.g. all binders) would have to be explicitly typed and type error messages would refer to the intermediate language rather than the written program.

The Core language is based on *System* F_C, a polymorphic lambda calculus with explicit type-equality coercions that, like types, are erased at compile time (i.e. types and coercions alike don't incur any cost at run-time). The syntax of System F_C [47] terms is given in Figure 2.3, which corresponds exactly to the syntax of System F [20, 42] with term and (kind-annotated) type abstraction as well as term and type application, extended with algebraic data types, let-bound expressions, pattern matching and coercions or casts.

u	$::=$	$x \mid K$	Variables and data constructors
e	$::=$	u	Term atoms
	$\mid$	$\Lambda a{:}\kappa.\ e \mid e\ \varphi$	Type abstraction/application
	$\mid$	$\lambda x{:}\sigma.\ e \mid e_1\ e_2$	Term abstraction/application
	$\mid$	$\mathsf{let}\ x{:}\sigma = e_1\ \mathsf{in}\ e_2$	
	$\mid$	$\mathsf{case}\ e_1\ \mathsf{of}\ \overline{p \to e_2}$	
	$\mid$	$e \blacktriangleright \gamma$	Cast
p	$::=$	$K\ \overline{b{:}\kappa}\ \overline{x{:}\sigma}$	Pattern

Figure 2.3: System F_C's Terms

Explicit type-equality coercions (or simply coercions), written $\sigma_1 \sim \sigma_2$, serve as evidence of equality between two types σ_1 and σ_2. A coercion $\sigma_1 \sim \sigma_2$ can be used to safely *cast* an expression e of type σ_1 to type σ_2, where casting e to σ_2 using $\sigma_1 \sim \sigma_2$ is written $e \blacktriangleright \sigma_1 \sim \sigma_2$. The syntax of *coercions* is given by Figure 2.4 and describes how coercions can be constructed to justify new equalities between types (e.g. using symmetry and transitivity). For example, given $\tau \sim \sigma$, the coercion $\mathbf{sym}\ (\tau \sim \sigma)$ denotes a type-equality coercion from σ to τ using the axiom of symmetry of equality. Through it, the expression $e{:}\sigma$ can be cast to $e{:}\tau$, i.e. $(e{:}\sigma \blacktriangleright \mathbf{sym}\ \tau \sim \sigma) : \tau$.

System F_C's coercions are key in desugaring advanced type-level Haskell features such as GADTs, type families and newtypes [47]. In short, these three features are desugared as follows:

$$
\begin{array}{lcll}
\sigma, \tau & ::= & d \mid \tau_1\ \tau_2 \mid S_n\ \overline{\tau}^n \mid \forall a{:}\kappa.\tau & \text{Types} \\
\gamma, \delta & ::= & g \mid \tau \mid \gamma_1\ \gamma_2 \mid S_n\ \overline{\gamma}^n \mid \forall a{:}\kappa.\gamma & \text{Coercions} \\
 & \mid & \mathbf{sym}\ \gamma \mid \gamma_1 \circ \gamma_2 \mid \gamma@\sigma \mid \mathbf{left}\ \gamma \mid \mathbf{right}\ \gamma & \\
\varphi & ::= & \tau \mid \gamma & \text{Types and Coercions}
\end{array}
$$

Figure 2.4: System F_C's Types and Coercions

- GADTs local equality constraints are desugared into explicit type-equality evidence that are pattern matched on and used to cast the branch alternative's type to the return type.

- Newtypes such as `newtype BoxI = BoxI Int` introduce a global type-equality `BoxI` $\sim$ `Int` and construction and deconstruction of said newtype are desugared into casts.

- Type family instances such as `type instance F Int = Bool` introduce a global coercion `F Int` $\sim$ `Bool` which can be used to cast expressions of type `F Int` to `Bool`.

Core further extends *System* F_C with *jumps* and *join points* [34], allowing new optimizations to be performed which ultimately result in efficient code using labels and jumps, and with a construct used for internal notes such as profiling information.

In the context of Linear Haskell, and recalling that Haskell is fully desugared into Core / System F_C as part of its validation and compilation strategy, we highlight the inherent incompatibility of linearity with Core / System F_C as a current flaw in GHC that invalidates all the benefits of Core wrt linearity. Thus, we must extend System F_C (and, therefore, Core) with linearity in order to adequately validate the desugaring and optimizing transformations as linearity preserving, ensuring we can reason about Linear Haskell in its Core representation.

2.5 GHC Pipeline

The GHC compiler processes Haskell source files in a series of phases that feed each other in a pipeline fashion, each transforming their input before passing it on to the next stage. This pipeline is crucial in the overall design of GHC. We now give a high-level overview of the phases.

2.5.1 Haskell to Core

Parser. The Haskell source files are first processed by the lexer and the parser. The lexer transforms the input file into a sequence of valid Haskell tokens. The parser processes the tokens to create an abstract syntax tree representing the original code, as long as the input is a syntactically valid Haskell program.

Renamer. The renamer's main tasks are to resolve names to fully qualified names, resolve name shadowing, and resolve namespaces (such as the types and terms namespaces), taking into consideration both existing identifiers in the module being compiled and identifiers exported by other modules. Additionally, name ambiguity, variables out of scope, unused bindings or imports, etc., are checked and reported as errors or warnings.

Type-checker. With the abstract syntax tree validated by the renamer and with the names fully qualified, the Haskell program is type-checked before being desugared into Core. Type-checking the Haskell program guarantees that the program is well-typed. Otherwise, type-checking fails with an error reporting where in the source typing failed. Furthermore, every identifier in the program is annotated with its type. Haskell is an implicitly typed language and, as such, type-inference must be performed to type-check programs. During type inference, every identifier is typed and we can use its type to decorate said identifier in the abstract syntax tree produced by the type-checker. First, annotating identifiers is *required* to desugar Haskell into Core because Core is explicitly typed – to construct a Core abstract syntax tree the types are indispensable (i.e. we cannot construct a Core expression without explicit types). Secondly, names annotated with their types are useful for tools manipulating Haskell, e.g. for an IDE to report the type of an identifier.

Desugaring. The type-checked Haskell abstract syntax tree is then transformed into Core by the desugarer. We've discussed in Section 2.4 the relationship between Haskell and Core in detail, so we refrain from repeating it here. It suffices to say that the desugarer transforms the large Haskell language into the small Core language by simplifying all syntactic constructs to their equivalent Core form (e.g. `newtype` constructors are transformed into coercions).

2.5.2 Core-To-Core Transformations

The Core-to-Core transformations are the most important set of optimizing transformations that GHC performs during compilation. By design, the frontend of the pipeline (parsing, renaming, typechecking and desugaring) does not include any optimizations – all optimizations are done in Core. The transformational approach focused on Core, known as *compilation by transformation*, allows transformations to be both modular and simple. Each transformation focuses on optimizing a specific set of constructs, where applying a transformation often exposes opportunities for other transformations to fire. Since transformations are modular, they can be chained and iterated in order to maximize the optimization potential (as shown in Figure 2.5).

However, due to the destructive nature of transformations (i.e. applying a transformation is not reversible), the order in which transformations are applied determines how well the resulting program is optimized. As such, certain orderings of optimizations can hide optimization opportunities and block them from firing. This phase-ordering problem is present in most optimizing compilers.

Foreshadowing the fact that Core is the main object of our study, we want to type-check linearity in Core before *and* after each optimizing transformation is applied (Section 2.4). In that light, we describe below some of the individual Core-to-Core transformations, using $\Longrightarrow$ to denote a program transformation. In the literature, the first set of Core-to-Core optimizations was described in [44, 39]. These were subsequently refined and expanded [36, 2, 34, 6, 45]. In Figure 2.5 we present an example that is optimized by multiple transformations to highlight how the compilation by transformation process produces performant programs.

Inlining. Inlining is an optimization common to all compilers, but especially important in functional languages [39]. Given Haskell's pure and lazy semantics, inlining can be employed in Haskell to a much larger extent because we needn't worry about evaluation

order or side effects, contrary to most imperative and strict languages. *Inlining* consists of replacing an occurrence of a let-bound variable by its right-hand side:

$$\mathsf{let}\ x = e\ \mathsf{in}\ \ldots x \ldots \implies \mathsf{let}\ x = e\ \mathsf{in}\ \ldots e \ldots$$

Effective inlining is crucial to optimization because, by bringing the definition of a variable to the context in which it is used, many other local optimizations are unlocked. The work [36] further discusses the intricacies of inlining and provides algorithms used for inlining in GHC.

β-reduction. β-reduction is an optimization that consists of reducing an application of a term λ-abstraction or type-level Λ-abstraction (Figure 2.3) by replacing the λ-bound variable with the argument the function is applied to:

$$(\lambda x{:}\tau.\ e)\ y \implies e[y/x] \qquad (\Lambda a{:}\kappa.\ e)\ \varphi \implies e[\varphi/a]$$

If the λ-bound variable is used more than once in the body of the λ-abstraction we must be careful not to duplicate work, and we can let-bound the argument, while still removing the λ-abstraction, to avoid doing so:

$$(\lambda x{:}\tau.\ e)\ y \implies \mathsf{let}\ x = y\ \mathsf{in}\ e$$

β-reduction is always a good optimization because it effectively evaluates the application at compile-time (reducing heap allocations and execution time) and unlocks other transformations.

Case-of-known-constructor. If a `case` expression is scrutinizing a known constructor $K\ \overline{x{:}\sigma}$, we can simplify the case expression to the branch it would enter, substituting the pattern-bound variables by the known constructor arguments ($\overline{x{:}\sigma}$):

$$\begin{array}{l}\textbf{case}\ K\ v_1\ \ldots\ v_n\ \textbf{of} \\ \quad K\ x_1\ \ldots\ x_n \to e \quad \implies e[v_i/x_i]_{i=1}^{n} \\ \quad \ldots\end{array}$$

Case-of-known-constructor is an optimization mostly unlocked by other optimizations such as inlining and β-reduction, more so than by code written as-is by the programmer. As β-reduction, this optimization is also always good – it eliminates evaluations whose result is known at compile time and further unblocks for other transformations.

Let-floating. A let-binding in Core entails performing *heap-allocation*, therefore, let-related transformations directly impact the performance of Haskell programs. In particular, let-floating transformations are concerned with best the position of let-bindings in a program in order to improve efficiency. Let-floating is an important group of transformations for non-strict (lazy) languages described in detail by [38]. We distinguish three let-floating transformations:

- *Float-in* consists of moving a let-binding as far *inwards* as possible. For example, it could be moving a let-binding outside of a case expression into the branch alternative that uses the let-bound variable:

$$\begin{array}{l}\mathsf{let}\ x = y + 1 \\ \ \mathsf{in\ case}\ z\ \mathsf{of} \\ \ \ [] \to x * x \\ \ \ (p : ps) \to 1\end{array} \implies \begin{array}{l}\mathsf{case}\ z\ \mathsf{of} \\ \ \ [] \to \mathsf{let}\ x = y + 1\ \mathsf{in}\ x * x \\ \ \ (p : ps) \to 1\end{array}$$

This can improve performance by not performing let-bindings (e.g. if the branch the let was moved into is never executed); improving strictness analysis; and further unlocking other optimizations such as [38]. However, care must be taken when floating a let-binding inside a λ-abstraction because every time that abstraction is applied the value (or thunk) of the binding will be allocated in the heap.

- *Full laziness* transformation, also known as *float-out*, consists of moving let-bindings outside of enclosing λ-abstractions. The warning above regarding λ-abstractions recomputing the binding every time they are applied is valid even if bindings are not purposefully pushed inwards. In such a situation, floating the let binding out of the enclosing lambda can make it readily available across applications of said lambda.

- The *local transformations* are the third type of let-floating optimizations. In this context, the local transformations are local rewrites that improve the placement of bindings. There are three local transformations:

  1. $(\mathsf{let}\ v = e\ \mathsf{in}\ b)\ a \implies \mathsf{let}\ v = e\ \mathsf{in}\ b\ a$
  2. $\mathsf{case}\ (\mathsf{let}\ v = e\ \mathsf{in}\ b)\ \mathsf{of}\ \ldots \implies \mathsf{let}\ v = e\ \mathsf{in}\ \mathsf{case}\ b\ \mathsf{of}\ \ldots$
  3. $\mathsf{let}\ x = (\mathsf{let}\ v = e\ \mathsf{in}\ b)\ \mathsf{in}\ c \implies \mathsf{let}\ v = e\ \mathsf{in}\ \mathsf{let}\ x = b\ \mathsf{in}\ c$

  These transformations do not change the number of allocations but potentially create opportunities for other optimizations to fire, such as expose a lambda abstraction [38].

η-expansion and η-reduction. η-expansion is a transformation that expands a function expression f to $(\lambda x.f\ x)$, where x is not free in f. This transformation can improve efficiency because it can fully apply functions which would previously be partially applied by using the variable bound to the expanded λ. A partially applied function is often more costly than a fully saturated one because it entails a heap allocation for the function closure, while a fully saturated one equates to a function call. η-reduction is the inverse transformation to η-expansion, i.e., a λ-abstraction $(\lambda x.f\ x)$ can be η-reduced to simply f.

Case-of-case. The case-of-case transformation fires when a case expression is scrutinizing another case expression. In this situation, the transformation duplicates the outermost case into each of the inner case branches:

$$\mathsf{case}\ \left(\begin{array}{c}\mathsf{case}\ e_c\ \mathsf{of}\\ alt_{c_1} \to e_{c_1}\\ \ldots\\ alt_{c_n} \to e_{c_n}\end{array}\right)\ \mathsf{of}\ \begin{array}{l} alt_1 \to e_1\\ \ldots\\ alt_n \to e_n\end{array} \implies \begin{array}{l}\mathsf{case}\ e_c\ \mathsf{of}\\ alt_{c_1} \to \left(\begin{array}{c}\mathsf{case}\ e_{c_1}\ \mathsf{of}\\ alt_1 \to e_1\\ \ldots\\ alt_n \to e_n\end{array}\right)\\ \ldots\\ alt_{c_n} \to \left(\begin{array}{c}\mathsf{case}\ e_{c_n}\ \mathsf{of}\\ alt_1 \to e_1\\ \ldots\\ alt_n \to e_n\end{array}\right)\end{array}$$

This transformation exposes other optimizations, e.g., if e_{c_n} is a known constructor we can readily apply the *case-of-known-constructor* optimization. However, this transformation also potentially introduces significant code duplication. To this effect, we apply a transformation that creates *join points* (i.e., shared bindings outside the case expressions

that are used in the branch alternatives) that are compiled to efficient code using labels and jumps.

Common sub-expression elimination. Common sub-expression elimination (CSE) is a transformation that is effectively inverse to *inlining*. This transformation factors out expensive expressions into a shared binding. In practice, lazy functional languages don't benefit nearly as much as strict imperative languages from CSE and, thus, it isn't very important in GHC [12].

Static argument and lambda lifting. *Lambda lifting* is a transformation that abstracts over free variables in functions by making them λ-bound arguments [25, 44]. This allows functions to be "lifted" to the top-level of the program (because they no longer have free variables). Lambda lifting may unlock inlining opportunities and allocate less function closures, since the definition is then created only once at the top-level and shared across uses. The *static argument* transformation identifies function arguments which are *static* across calls, and eliminates said *static argument* to avoid passing the same fixed value as an argument in every function call, which is especially significant in recursive functions. To this effect, the *static argument* is bound outside of the function definition and becomes a free variable in its body. It can be thought of as the transformation inverse to *lambda lifting*.

Strictness analysis and worker/wrapper split. The strictness analysis, in lazy programming languages, identifies functions that always evaluate their arguments, i.e. functions with (morally) *strict arguments*. Arguments passed to functions that necessarily evaluate them can be evaluated before the call and therefore avoid some heap allocations. The strictness analysis may be used to apply the worker/wrapper split transformation [37]. This transformation creates two functions from an original one: a worker and a wrapper. The worker receives unboxed values [27] as arguments, while the wrapper receives boxed values, unwraps them, and simply calls the worker function (hence the wrapper being named as such). This allows the worker to be called in expressions other than the wrapper, saving allocations and being possibly much faster, especially if the worker recursively ends up calling itself rather than the wrapper.

2.5.3 Code Generation

The code generation needn't be changed to account for the work we will do in the context of this thesis, so we only briefly describe it.

After the core-to-core pipeline is run on the Core program and produces optimized Core, the program is translated down to the Spineless Tagless G-Machine (STG) language [26]. STG language is a small functional language that serves as the abstract machine code for the STG abstract machine that ultimately defines the evaluation model and compilation of Haskell through operational semantics.

From the abstract state machine, we generate C-- (read C minus minus), a C-like language designed for native code generation, which is finally passed as input to one of the code generation backends[4], such as LLVM, x86 and x64, or (recently) JavaScript and WebAssembly.

[4]GHC is not *yet* runtime retargetable, i.e. to use a particular native code generation backend the compiler must be built targetting it.

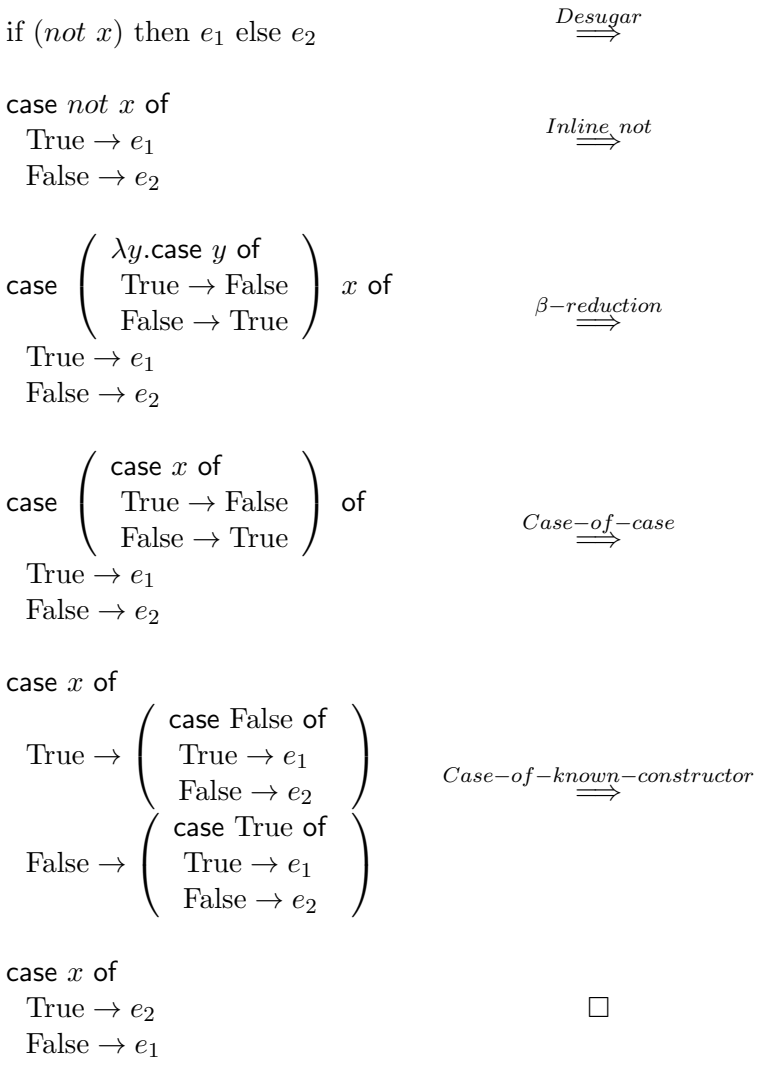

Figure 2.5: Example sequence of transformations

2.6 Related Work

Formalization of Core System F_C [47] (Section 2.4) does not account for linearity in its original design, and, to the best of our knowledge, no extension to System F_C with linearity and non-strict semantics exists. As such, there exists no formal definition of Core that accounts for linearity. In this context, we intend to introduce a linearly typed System F_C with multiplicity annotations and typing rules to serve as a basis for a linear Core. Critically, this Core linear language must account for call-by-need evaluation semantics and be valid in light of Core-to-Core optimizing transformations.

Linear Haskell Haskell, contrary to most programming languages with linear types, has existed for 31 years of its life *without* linear types. As such, the introduction of linear types to Haskell comes with added challenges that do not exist in languages that were designed with linear types from the start:

- Backwards compatibility. The addition of linear types shouldn't break all existing Haskell code.

- Code re-usability. The linearly-typed part of Haskell's ecosystem and its non-linearly-typed counterpart should fit in together and it must be possible to define functions readily usable by both sides simultaneously.
- Future-proofing. Haskell, despite being an industrial-strength language, is also a petri-dish for experimentation and innovation in the field of programming languages. Therefore, Linear Haskell takes care to accomodate possible future features, in particular, its design is forwards compatible with affine and dependent types.

Linear Haskell [4] is thus concerned with retrofitting linear types in Haskell, taking into consideration the above design goals, but is not concerned with extending Haskell's intermediate language(s), which presents its own challenges.

Nonetheless, while the Linear Haskell work keeps Core unchanged, its implementation in GHC does modify and extend Core with linearity/multiplicity annotations, and said extension of Core with linear types does not account for optimizing transformations and the non-strict semantics of Core.

Our work on linear Core intends to overcome the limitations of linear types as they exist in Core, i.e. integrating call-by-need semantics and validating the Core-to-Core passes, ultimately doubling as a validation of the implementation of Linear Haskell.

Linearity-influenced optimizations Core-to-Core transformations appear in many works across the research literature [38, 39, 44, 36, 2, 34, 6, 45], all designed in the context of a typed language (Core) which does not have linear types. However, [38, 39, 4] observe that certain optimizations (in particular, let-floating and inlining) greatly benefit from linearity analysis and, in order to improve those transformation, linear-type-inspired systems were created specifically for the purpose of the transformation.

By fully supporting linear types in Core, these optimizing transformations could be informed by the language inherent linearity, and, consequently, avoid an ad-hoc or incomplete linear-type inference pass custom-built for optimizations. Additionally, the linearity information may potentially be used to the benefit of optimizing transformations that currently don't take any linearity into account.

CHAPTER 3

Proposed Work

3.1 Motivation

Since the publication of Linear Haskell [4] and its implementation in GHC 9.0, Haskell's type system supports linearity annotations in functions. Linear Haskell effectively brings linear types to a mainstream, pure, and lazy functional language. Concretely, Haskell function types can be annotated with a multiplicity, where a multiplicity of `1` requires the argument to be consumed exactly once and a multiplicity of `Many` allows the function argument to be consumed unrestrictedly, i.e., zero or more times.

As mentioned in Section 2.4, GHC Haskell features two typed languages in its pipeline: Haskell and its main intermediate language, Core (Core is a much smaller language than the whole of Haskell, even though we can compile the whole of Haskell to Core). The addition of linear types to GHC Haskell required changing the type system of both languages. However, Linear Haskell only describes an extension to the surface-level Haskell type system, not Core. Nonetheless, in practice, Core is linearity-aware.

We want Core and its type system to give us guarantees about desugaring and optimizing transformations with regard to linearity just as Core does for types – a linearly typed Core ensures that linearly-typed programs remain correct both after desugaring and across all GHC optimizing transformations, i.e. linearity is preserved when desugaring and optimisations should not destroy linearity.

In this sense, Core is already annotated with linearity, but its type-checker **currently ignores linearity annotations**. In spite of the strong formal foundations of linear types driving the implementation, their interaction with the whole of GHC is still far from trivial. The implemented type system cannot accomodate several optimising transformations that produce programs which violate linearity *syntactically* (i.e. multiple occurrences of linear variables in the program text), but ultimately preserve it in a *semantic* sense, where a linear term is still *consumed exactly once* – this is compounded by lazy evaluation driving a non-trivial mismatch between syntactic and semantic linearity.

Consider the example in Figure 3.1, an expression in which y and z are variables bound by a λ-abstraction, and both are annotated with a multiplicity of `1`. Note that let-binding x doesn't necessarily consume y and z because of Core's call-by-need semantics. In the example, it might not seem as though y and z are both being consumed linearly but, in fact, they both are. Given that in the first branch we use x – which entails using y and z linearly – and in the second branch we use y and z directly, in both branches we are consuming both y and z linearly.

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

Figure 3.1: Example Inlining

Similarly, consider the program in Figure 3.1 with a let-binding that uses x, a linearly bound λ-variable. In surface level Haskell, let-bindings always consume linear variables `Many` times to avoid dealing with the complexity of call-by-need semantics, so this program would not type-check, because x is being consumed `Many` times instead of `1`. Despite not

```
f :: a %1 → a
f x = let y = x + 2 in y
```

Figure 3.2: Example Let

being accepted by the surface-level language, linear programs using lets occur naturally in Core due to optimising transformations that create let bindings, such as β-reduction. In a similar fashion, programs which violate syntatic linearity for other reasons other than let bindings are produced by Core transformations.

The current solution to valid programs being rejected by Core's linear type-checker is to effectively disable the linear type-checker since, alternatively, disabling optimizing transformations which violate linearity incurs significant performance costs. However, we believe that GHC's transformations are correct, and it is the linear type-checker and its underlying type system that cannot sufficiently accommodate the resulting programs.

Additionally, some Core-to-Core transformations such as let-floating and inlining already depend on custom linear type systems to produce more performant programs. Valid linearity annotations in Core could potentially inform and define more optimizations.

3.2 Goals

In the upcoming dissertation we will:

- Propose an extension of Core's type system and type checking algorithm with additional linearity information in order to accommodate linear programs in Core that result from the optimising transformations described in Section 2.5;
- Argue the soundness of the resulting system (i.e. no semantically non-linear programs are deemed linear);
- Show how it validates Core-to-Core optimisation passes;
- Implement the extension into GHC's Core type-checker (internally known as the Core linter).

3.2.1 Extending Core's type system

The key design goal of the extension of Core's type system is to provide enough information so that *syntactically* non-linear but *semantically* linear programs are equipped with enough typing information so that they are deemed linearly well-typed, while ruling out programs

that violate linearity from being considered linearly typed. To this end, we propose to extend Core's linear type system with *usage annotations* for let, letrec and case binder bound variables. Given time, we will also explore a new kind of coercion for multiplicities to validate programs that combine GADTs with multiplicities.

3.2.2 Typing Usage Environments

A solution to a handful of type-checking issues regarding certain variable binders is to extend said binders with a *usage environment*. A usage environment is a mapping from variables to multiplicities. The idea is to annotate let, recursive lets and case binders with a usage environment rather than a multiplicity (in contrast, a λ-bound variable has a multiplicity instead of a usage environment).

There are two sides to the usage environment solution. First, our type system must be able to type-check Core programs in a context where let, recursive let and case binders have usage environments. Secondly, the usage environments must be inferred by the type-checking algorithm when relevant binders are created and maintained throughout the Core-to-Core passes to ultimately be used by the typing rules.

With usage environments, the example in Figure 3.1, in which x is a linear variable, would type-check by annotating y with a usage environment of $[x \to 1]$, because the expression bound by y uses x exactly once, *and* emitting that x is used once when y is used, that is, emitting y's usage environment upon using y.

The example in Figure 3.1 would similarly type-check by annotating x with the usage environment $[y \to 1, z \to 1]$ – to linearly type-check that y and z are used linearly, both branches must use y and z linearly. Using usage environments, in the first branch, using x amounts to using its usage environment (using y and z once) and, in the second branch, y and z are used exactly once; meaning y and z are used exactly once in both branches.

We have explored preliminary typing rules with usage environment inference rules for let, recursive let, and case binders. As we will show below, calculating usage environments is not trivial, especially for recursive let bindings. Yet, at a high level, computing the usage environment of a definition entails collecting the usages, where using a λ-bound variable emits a mapping from that variable to its multiplicity and using a variable annotated with a usage environment emits all mappings from variables to their corresponding multiplicities from that usage environment.

Let Binder Regardless of the original Haskell programs desugared to Core, let bindings are always common in Core due to a myriad of optimizing transformations that create and manipulate let-bindings (Section 2.5). Currently, every let-bound variable in Core is annotated with a multiplicity at its binding site. However, the multiplicity for let-bound variables must be ignored by the type-checker throughout the transformation pipeline (or otherwise too many valid programs would be rejected for violating linearity). Programs with let bindings can be correctly typed by annotating the let-bound variables with a usage environment. Instead of annotating every binder with a multiplicity, only λ-bound variables should be annotated with a multiplicity, while let-bound variables should be annotated with a usage environment. To compute a usage environment for a let-bound variable, compute the usages of free variables in the body of the binder. To type-check an occurrence of a let-bound variable, emit that binder's usage environment. The typing rule combines these two statements:

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

Algorithm 1: computeRecUsages

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

The rule is read "An expression $\mathsf{let}\ x = t\ \mathsf{in}\ u$ has type C with usage environment V under a Γ context, if the expression t has type A with usage environment U under Γ *and* the expression u has type C with usage environment V under the context Γ extended with the let-bound variable x which has type A and usage environment U".

Recursive Let Binder A variable bound by a recursive-let is in scope in its own definition, allowing for self-reference. Just as let bindings, recursive-let bindings with free linear variables in their assignment body can form in Core during the Core-to-Core passes, and, similarly, the linearity is ignored by the type-checker as the lesser of the two unfavorable options. When the recursive-let-binder is annotated with a usage environment, to type-check t in $\mathsf{letrec}\ x{:}_U A = v\ \mathsf{in}\ t$, where x has type A with usage environment U, simply emit x's usage environment when x occurs in t.

However, calculating the usage environment of a recursive-let binder is much more challenging – a recursive-let-bound variable in its own definition does not yet have a usage environment when it's being computed. The following example uses y linearly if we compute the usage environment of f to be $[y \to 1]$, but can we programmatically reach that solution?

```
letrec f z = case z of
  True → f False
  False → y
in f True
```

The preliminary idea to calculate the usage environment of a set of mutually recursive let binders is to perform the computation in two separate passes. First, calculate a *naive usage environment* by emitting a multiplicity when λ-bound variables are used, usage environments when let-bound and case-bound variables are used, and, conversily, a multiplicity for each recursive-let-bound variable (rather than a usage environment). Second, run algorithm 1 to produce a *final usage environment*. The algorithm receives the recursive binders names and their corresponding naive environment. Intuitively, the algorithm, for each recursive binder, iterates over all (initially naive) usage environments and substitutes the recursive binder by the corresponding usage environment (and possibly scaled up by the amount of times that recursive binder is used in the environment being updated[1]). The type-checking and usage environment inference algorithm combine in the following rule:

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

[1] A point of contention is using a usage-environment-annotated variable more than once, a problem for which a solution is not evidently clear. Let-bound variables are heap-allocated and executed only once when evaluated. Should using a let-bound variable twice entail using the resources of the binder's definition twice? Tentatively no, because the value is only effectively computed once.

Case Binder In Core, all case expressions create a bound variable, the value of which is the evaluated case scrutinee. This case binder is used by some optimizations to refer to the value pattern matched on. If the case scrutinee must be used linearly, we must consider how to type-check linearity when the case-binder is also used in the case alternatives.

Our preliminary idea is to annotate the case binder with *independent usage environments for each pattern match*. When type-checking the case alternatives, using the case binder equates to using the usage environment associated with that particular alternative. To construct the usage environment of the complete case expression, we take a usage environment which is a superset of the usage environment constructed in each branch. Finally, the usage environment of the case binder in a branch that matched a nullary data constructor is empty, while in branches that matched a non-nullary constructor, the usage environment is a mapping from the variables bound by the pattern to their multiplicities. The typing rule that merges these ideas is:

$$\frac{\Gamma \vdash t : D_{\pi_1 \ldots \pi_n} \rightsquigarrow \{U\} \qquad \Gamma; z :_{U_k} D_{\pi_1 \ldots \pi_n} \vdash b_k : C \rightsquigarrow \{V_k\} \qquad V_k \leqslant V}{\Gamma \vdash \mathsf{case}\ t\ \mathsf{of}\ z :_{(U_1 \ldots U_n)} D_{\pi_1 \ldots \pi_n} \{b_k\}_1^m : C \rightsquigarrow \{U + V\}}\ (\textsc{case})$$

3.2.3 Validating the Work

The greatest measure of success is validating linearity in all programs compiled with GHC, by enabling the linear type-checker after desugaring and each Core-to-Core transformation. In its current implementation, the linear Core type-checker, which is enabled through a flag, rejects many linearly valid programs. Ideally, by the end of our research and implementation, this flag could be enabled by default and accommodate all programs to which existing transformations are applied. Realistically, we want to accept as many diverse transformations as possible while still preserving linearity, even if we are unable to account for all of them.

For each transformation we successfully support in our type system, we will argue that it does indeed preserve linearity by type-checking the input program to the transformation and the ouptut program.

Quantitively, we will benchmark the variation in compilation time and heap-allocations when our extended type-checker is enabled, in comparison to the compiler with the Core type-checker that ignores linearity annotations being run.

The linear Core type-checker validation and benchmarks will be done by running the GHC testsuite and compiling the *head.hackage* package set[2] with the flag which enables the linear Core type-checker. The GHC project also automatically runs tests and benchmarks through its continuous integration (CI) pipeline, which we intend to use to further validate our implementation continuously.

3.2.4 Tasks and Chronogram

In the dissertation, we will propose an extension to Core / System F_C's type system and type-checking algorithm with additional linearity information (such as *usage environments*) in order to accommodate linear programs in Core throughout the GHC pipeline (Section 2.5) stages of desugaring and optimization. This type-system entails augmenting Core's syntax to support additional linearity information and extending existing typing judgments and rules to account for linearity.

[2] *head.hackage* is a package set comprised of relevant libraries of the Haskell ecosystem which are compiled by, and patched against, GHC's latest commit.

Furthermore, we will argue the soundness of our system, that is, our type-system must provably not accept any programs which aren't linear.

Because we want to ensure our type system validates programs before and after optimizing transformations are applied, we will validate that each optimizing transformation does not destroy linearity in programs wrt our type system.

We will implement this extension to Core in GHC, the leading Haskell compiler. Core's type-system implementation, internally known as the Core linter, will serve as the base for our extension. Running the Core linter will enforce an iterative approach to implementing the extensions and allow us to validate our progress continuously.

Defining and implementing this type-system in GHC can be done iteratively because each binder can be handled separately. The typing rules for let bindings, recursive let bindings, and case bindings are distinct, and optimizing transformations seldom interact with all three at the same time. Consequently, we can interweave the formal specification, implementation in Core, and validation of individual optimizations and of our implementation.

Throughout this, we will write the final dissertation document, using it as a driving force for the research which will cristalize our ideas and help communicate them. GHC is a large project with many involved parties – it is crucial that we communicate our ideas and changes clearly, so we can benefit from other contributors expert feedback, and ultimately merge our changes to Core upstream.

CHAPTER 4

Linear Core*

4.1 Type Soundness

Theorem 1 (Type preservation)**.** If $\Gamma \vdash e : \sigma$ and $e \to^* e'$ then $\Gamma \vdash e' : \sigma$

Proof. By structural induction on the small-step reduction.

Case: $(\lambda x{:}_\pi \sigma.\ e)\ e' \longrightarrow e[e'/x]$

(1) $\Gamma, \Gamma' \vdash (\lambda x{:}_\pi \sigma.\ e)\ e' : \varphi$	by assumption
(2) $\Gamma \vdash (\lambda x{:}_\pi \sigma.\ e) : \sigma \to_\pi \varphi$	by inversion on (λE)
(3) $\Gamma, x{:}_\pi \sigma \vdash e : \varphi$	by inversion on (λI)
(4) $\Gamma' \vdash e' : \sigma$	by inversion on (λE)
(5) $\Gamma, \Gamma' \vdash e[e'/x] : \varphi$	by subst. lemma (3,4)

Case: $(\Lambda p.\ e)\ \pi \longrightarrow e[\pi/p]$

(1) $\Gamma \vdash (\Lambda p.\ e)\ \pi : \sigma[\pi/p]$	by assumption
(2) $\Gamma \vdash (\Lambda p.\ e) : \forall p.\ \sigma$	by inversion on (ΛE)
(3) $\Gamma \vdash_{mult} \pi$	by inversion on (ΛE)
(4) $\Gamma, p \vdash e : \sigma$	by inversion on (ΛI)
(5) $p \notin \Gamma$	by inversion on (ΛI)
(6) $\Gamma \vdash e[\pi/p] : \sigma[\pi/p]$	by subst. lemma (3,4)

Case: $\mathbf{let}\ x{:}_\Delta \sigma = e\ \mathbf{in}\ e' \longrightarrow e'[e/x]$

(1) $\Gamma \vdash \mathbf{let}\ x{:}_\Delta \sigma = e\ \mathbf{in}\ e' : \varphi$	by assumption
(2) $\Gamma, x{:}_\Delta \sigma \vdash e' : \varphi$	by inversion on (let)
(3) $\Delta \vdash e : \sigma$	by inversion on (let)
(4) $\Delta \subseteq \Gamma$	by inversion on (let)
(5) $\Gamma \vdash e'[e/x] : \varphi$	by subst. lemma (2,3,4?)

Case: $\mathbf{let}\ \overline{x{:}_\Delta \sigma = e}\ \mathbf{in}\ e' \longrightarrow e'[\overline{\mathbf{let\ rec}\ \overline{x{:}_\Delta \sigma = e}\ \mathbf{in}\ e/x}]$

Types

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

Terms

u	::=	$x, y, z \mid K$	Variables and data constructors
e	::=	u	Term atoms
	$\mid$	$\Lambda p.\ e \mid e\ \pi$	Multiplicity abstraction/application
	$\mid$	$\lambda x{:}_\pi \sigma.\ e \mid e_1\ e_2$	Term abstraction/application
	$\mid$	**let** $x{:}_\Delta \sigma = e_1$ **in** e_2	Let
	$\mid$	**let rec** $\overline{x{:}_\Delta \sigma = e_1}$ **in** e_2	Recursive Let
	$\mid$	**case** e_1 **of** $z{:}_{\overline{\Delta}}$ $\{\overline{\rho \to e_2}\}$	Case
ρ	::=	$K\ \overline{x{:}_\pi \sigma} \mid \_$	Pattern and wildcard

Environments

Γ	::=	$\cdot$	Empty environment
	$\mid$	$\Gamma, x{:}_\pi \sigma$	Lambda bound variable
	$\mid$	$\Gamma, x{:}_\Delta \sigma$	Let(rec) bound variable
	$\mid$	$\Gamma, K{:}\sigma$	Data constructor
	$\mid$	Γ, p	Multiplicity variable

Multiplicities

π, μ ::= $1 \mid \omega \mid p \mid \pi + \mu \mid \pi \cdot \mu$

Usage Environments

Δ ::= $\cdot \mid \Delta, x{:}_1 \sigma \mid \Delta + \Delta' \mid \pi\Delta$

Declarations

pgm	::=	$\overline{decl}; e$
$decl$	::=	**data** $T\ \overline{p}$ **where** $\overline{K : \overline{\sigma \to_\pi}\ T\ \overline{p}}$

Figure 4.1: Linear Core* Syntax

(1) $\Gamma \vdash$ **let** $\overline{x{:}_\Delta \sigma = e}$ **in** $e' : \varphi$ — by assumption
(2) $\Gamma, \overline{x{:}_\Delta \sigma} \vdash e : \varphi$ — by inversion on (letrec)
(3) $\overline{\Delta, \overline{x{:}_\Delta \sigma \vdash e' : \sigma}}$ — by inversion on (letrec)
(4) $\overline{\Delta \subseteq \Gamma}$ — by inversion on (letrec)
(5) $\Gamma \vdash e'[\textbf{let rec } \overline{x{:}_\Delta \sigma = e} \textbf{ in } e/x]$ — by subst. lemma (2,3,4?)

Case: **case** $K\ \overline{e}$ **of** $z{:}_{\overline{\Delta}}\sigma\ \{\ldots, K\ \overline{x{:}_\pi \sigma}\} \longrightarrow e'[\overline{e}/\overline{x}][K\ \overline{e}/z]$

(1) $\Gamma, \Gamma' \vdash$ **case** $K\ \overline{e}$ **of** $z{:}_{\overline{\Delta}}\sigma\ \{\ldots, K\ \overline{x{:}_\pi \sigma} \to e'\} : \varphi$ — by assumption
(2) $\Gamma \vdash K\ \overline{e} : \sigma$ — by inversion on (case)

$$\boxed{\Gamma \vdash e : \sigma}$$

$$\frac{\Gamma \vdash e : \varphi}{\Gamma, x{:}_{\omega}\sigma \vdash e : \varphi}\ (Weaken_{\omega}) \qquad \frac{\Gamma \vdash e : \varphi}{\Gamma, x{:}_{\Delta}\sigma \vdash e : \varphi}\ (Weaken_{\Delta})$$

$$\frac{\Gamma, x{:}_{\omega}\sigma, x{:}_{\omega}\sigma \vdash e : \varphi}{\Gamma, x{:}_{\omega}\sigma \vdash e : \varphi}\ (Contract_{\omega}) \qquad \frac{\Gamma, x{:}_{\Delta}\sigma, x{:}_{\Delta}\sigma \vdash e : \varphi}{\Gamma, x{:}_{\Delta}\sigma \vdash e : \varphi}\ (Contract_{\Delta})$$

$$\frac{}{\cdot, x{:}_{1}\sigma \vdash x : \sigma}\ (Var_1) \qquad \frac{\forall y{:}_{\pi}\varphi \in \Gamma.\ \pi \neq 1}{\Gamma, x{:}_{\omega}\sigma \vdash x : \sigma}\ (Var_{\omega}) \qquad \frac{}{\Delta, x{:}_{\Delta}\sigma \vdash x : \sigma}\ (Var_{\Delta})$$

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

$$\frac{\Gamma, x{:}_{\pi}\sigma \vdash e : \varphi}{\Gamma \vdash \lambda x{:}_{\pi}\sigma.\ e : \sigma \to_{\pi} \varphi}\ (\lambda I) \qquad \frac{\Gamma \vdash e : \varphi \to_{\pi} \sigma \qquad \Gamma' \vdash e' : \varphi}{\Gamma, \Gamma' \vdash e\ e' : \sigma}\ (\lambda E)$$

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

$$\frac{\Gamma, \overline{x{:}_{\Delta}\sigma} \vdash e : \varphi \qquad \overline{\Gamma', \Delta, \overline{x{:}_{\Delta}\sigma} \vdash e' : \sigma}}{\Gamma, \overline{\Gamma'}, \overline{\Delta} \vdash \mathbf{let\ rec}\ \overline{x{:}_{\Delta}\sigma = e'}\ \mathbf{in}\ e : \varphi}\ (LetRec)$$

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \overline{\Gamma', z{:}_{\Delta_i}\sigma \vdash_{alt} \rho_i \to e_i : \sigma \Rightarrow \varphi}}{\Gamma, \Gamma' \vdash \mathbf{case}\ e_1\ \mathbf{of}\ z{:}_{\overline{\Delta}^n}\sigma\ \{\overline{\rho_i \to e_i}^n\} : \varphi}\ (Case)$$

$$\boxed{\Gamma \vdash_{alt} \rho \to e : \sigma \Rightarrow \varphi}$$

$$\frac{K : \overline{\sigma \to_{\pi}}\ T\ \overline{p} \in \Gamma \qquad \Gamma, \overline{x{:}_{\pi}\sigma} \vdash e : \varphi}{\Gamma \vdash_{alt} K\ \overline{x{:}_{\pi}\sigma} \to e : T\ \overline{p} \Rightarrow \varphi}\ (Alt) \qquad \frac{\Gamma \vdash e : \varphi}{\Gamma \vdash_{alt} \_ \to e : T\ \overline{p} \Rightarrow \varphi}\ (Alt\_)$$

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

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

$$\boxed{\Gamma \vdash decl : \Gamma'} \qquad \boxed{\Gamma \vdash pgm : \sigma}$$

$$\frac{}{\Gamma \vdash (\mathbf{data}\ T\ \overline{p}\ \mathbf{where}\ \overline{K : \sigma}) : (\overline{K : \sigma})}\ (Data) \qquad \frac{\overline{\Gamma \vdash decl : \Gamma_d} \qquad \Gamma = \Gamma_0, \overline{\Gamma_d} \qquad \Gamma\ \mathsf{is\ consistent?} \qquad \Gamma \vdash e : \sigma}{\Gamma_0 \vdash \overline{decl}; e : \sigma}\ (Pgm)$$

Figure 4.2: Linear Core* Typing Rules

(3) $\Gamma', z{:}_{\Delta}\sigma \vdash_{alt} K\ \overline{x{:}_{\pi}\sigma} \to e' : \sigma \Longrightarrow \varphi$ by inversion on (case)
(4) $K : \overline{\sigma \to_{\pi}} T\ \overline{p} \in \Gamma'$ by inversion on (alt)
(5) $\Gamma', z{:}_{\Delta}\sigma, \overline{x{:}_{\pi}\sigma} \vdash e' : \varphi$ by inversion on (alt)
(6) $\Gamma, z{:}_{\Delta_i}\sigma, \Gamma' \vdash e'[\overline{e}/\overline{x}] : \varphi$ by subst. lemma (2,5)
(7) $\Gamma, \Gamma' \vdash e'[\overline{e}/\overline{x}][???/z]$ by subst. lemma (6,2)

Values

$v \quad ::= \quad \Lambda p.\ e \mid \lambda x.\ e \mid K\ \overline{v}$

Evaluation Contexts

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad E \quad ::= \quad \Box \mid E\ e \mid E\ \pi \mid \textbf{case}\ E\ \textbf{of}\ z{:}\overline{\Delta}\sigma\{\overline{\rho \to e}\}$$

Expression Reductions

$(\Lambda p.\ e)\ \pi$	$\longrightarrow$	$e[\pi/p]$
$(\lambda x.\ e)\ e'$	$\longrightarrow$	$e[e'/x]$
$\textbf{let}\ x{:}\Delta\sigma = e\ \textbf{in}\ e'$	$\longrightarrow$	$e'[e/x]$
$\textbf{let rec}\ \overline{x{:}\Delta\sigma = e}\ \textbf{in}\ e'$	$\longrightarrow$	$e'[\textbf{let rec}\ \overline{x{:}\Delta\sigma = e}\ \textbf{in}\ e/x]$
$\textbf{case}\ K\ \overline{e}\ \textbf{of}\ \ldots K\ \overline{x{:}\pi\sigma} \to e'$	$\longrightarrow$	$e'[\overline{e}/\overline{x{:}\pi\sigma}][K\ \overline{e}/z]$

Figure 4.3: Linear Core* Operational Semantics (call-by-name)

Case: $e_1\ e_2 \longrightarrow e_1'\ e_2$

(1) $e_1 \longrightarrow e_1'$	by inversion on β-reduction
(2) $\Gamma, \Gamma' \vdash e_1\ e_2 : \varphi$	by assumption
(3) $\Gamma \vdash e_1 : \sigma \to_\pi \varphi$	by inversion on (λE)
(4) $\Gamma' \vdash e_2 : \sigma$	by inversion on (λE)
(5) $\Gamma \vdash e_1' : \sigma \to_\pi \varphi$	by induction hypothesis in (3,1)
(6) $\Gamma, \Gamma' \vdash e_1'\ e_2 : \varphi$	by (λE) (4,3)

Case: $e\ \pi \longrightarrow e'\ \pi$

(1) $e \longrightarrow e'$	by inversion on mult. β-reduction
(2) $\Gamma \vdash e\ \pi : \sigma[\pi/p]$	by assumption
(3) $\Gamma \vdash e : \forall p.\ \sigma$	by inversion on (ΛE)
(4) $\Gamma \vdash_{mult} \pi$	by inversion on (ΛE)
(5) $\Gamma \vdash e' : \forall p.\ \sigma$	by induction hypothesis (3,1)
(6) $\Gamma \vdash e'\ \pi : \sigma[\pi/p]$	by (ΛE) (5,4)

Case: $\textbf{case}\ e\ \textbf{of}\ z{:}\overline{\Delta}\sigma\ \{\rho_i \to e_i'\} \longrightarrow \textbf{case}\ e'\ \textbf{of}\ z{:}\overline{\Delta}\sigma\ \{\rho_i \to e_i'\}$

(1) $e \longrightarrow e'$	by inversion on case reduction
(2) $\Gamma, \Gamma' \vdash \textbf{case}\ e\ \textbf{of}\ z{:}\overline{\Delta}\sigma\ \{\rho_i \to e_i'\} : \varphi$	by assumption
(3) $\Gamma \vdash e : \sigma$	by inversion on case
(4) $\overline{\Gamma', z{:}_{\Delta_i}\sigma \vdash_{alt} \rho_i \to e_i' : \varphi}$	by inversion on case
(5) $\Gamma \vdash e' : \sigma$	by induction hypothesis (3,1)
(6) $\Gamma, \Gamma' \vdash \textbf{case}\ e'\ \textbf{of}\ z{:}\overline{\Delta}\sigma\ \{\rho_i \to e_i'\} : \varphi$	by case (4,3)

□

$$\boxed{\Gamma \vdash e : \sigma \rightsquigarrow \Delta}$$

$$\frac{}{\Gamma, x{:}_{\pi}\sigma \vdash x : \sigma \rightsquigarrow \cdot, x{:}_{\pi}\sigma}\ (Var_{\pi}) \qquad \frac{}{\Gamma, x{:}_{\Delta}\sigma \vdash x : \sigma \rightsquigarrow \Delta}\ (Var_{\Delta})$$

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

$$\frac{\Gamma, x{:}_1\sigma_1 \vdash e : \sigma_2 \rightsquigarrow \Delta, x{:}_1\sigma_1 \qquad x{:}_1\sigma_1 \notin \Delta}{\Gamma \vdash \lambda x{:}_1\sigma_1.\ e : \sigma_1 \to_{\pi} \sigma_2 \rightsquigarrow \Delta}\ (\lambda I_1)$$

$$\frac{\Gamma, x{:}_{\omega}\sigma_1 \vdash e : \sigma_2 \rightsquigarrow \Delta}{\Gamma \vdash \lambda x{:}_{\omega}\sigma_1.\ e : \sigma_1 \to_{\pi} \sigma_2 \rightsquigarrow \Delta \restriction_{\neq x}}\ (\lambda I_{\omega})$$

$$\frac{\Gamma \vdash e_1 : \sigma_2 \to_{\pi} \sigma_1 \rightsquigarrow \Delta \qquad \Gamma \vdash e_2 : \sigma_2 \rightsquigarrow \Delta'}{\Gamma \vdash e_1\ e_2 : \sigma_1 \rightsquigarrow \Delta + \Delta'}\ (\lambda E)$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \rightsquigarrow \Delta \qquad \Gamma, x{:}_{\Delta}\sigma_1 \vdash e_2 : \sigma_2 \rightsquigarrow \Delta'}{\Gamma \vdash \mathbf{let}\ x{:}_{\Delta}\sigma_1 = e_1\ \mathbf{in}\ e_2 : \sigma_2 \rightsquigarrow \Delta'}\ (Let)$$

$$\frac{\overline{\Delta} = \mathsf{computeRecUsages}(\overline{\Delta_{naive}}) \qquad \overline{\Gamma, \overline{x{:}_{1}\sigma} \vdash e' : \sigma \rightsquigarrow \Delta_{naive}} \qquad \Gamma, \overline{x{:}_{\Delta}\sigma} \vdash e : \varphi \rightsquigarrow \Delta'}{\Gamma \vdash \mathbf{let\ rec}\ \overline{x{:}_{\Delta}\sigma = e'}\ \mathbf{in}\ e : \varphi \rightsquigarrow \Delta'}\ (LetRec)$$

$$\frac{\Gamma \vdash e_1 : \sigma \rightsquigarrow \Delta \qquad \overline{\Gamma \vdash_{pat} \rho_i : \sigma \rightsquigarrow \Delta_i} \qquad \overline{\Gamma', z{:}_{\Delta_i}\sigma \vdash_{alt} \rho_i \to e_i : \sigma \Rightarrow \varphi \rightsquigarrow \Delta'} \qquad \overline{\Delta' \leqslant \Delta''}}{\Gamma, \Gamma' \vdash \mathbf{case}\ e_1\ \mathbf{of}\ z{:}_{\overline{\Delta}^n}\sigma\ \{\overline{\rho_i \to e_i}^n\} : \varphi \rightsquigarrow \Delta + \Delta''}\ (Case)$$

$$\boxed{\Gamma \vdash_{pat} \rho : \sigma \rightsquigarrow \Delta}$$

$$\frac{}{\Gamma, K{:}\overline{\sigma \to_{\pi}}\varphi \vdash_{pat} K\ \overline{x{:}_{\pi}\sigma} : \varphi \rightsquigarrow \cdot, \overline{x{:}_{\pi}\sigma}}\ (pat)$$

$$\boxed{\Gamma \vdash_{alt} \rho \to e : \sigma \Rightarrow \varphi \rightsquigarrow \Delta}$$

$$\frac{K : \overline{\sigma \to_{\pi}}\ T\ \overline{p} \in \Gamma \qquad \Gamma, \overline{x{:}_{\pi}\sigma} \vdash e : \varphi \rightsquigarrow \Delta}{\Gamma \vdash_{alt} K\ \overline{x{:}_{\pi}\sigma} \to e : T\ \overline{p} \Rightarrow \varphi \rightsquigarrow \Delta}\ (Alt)$$

$$\frac{\Gamma \vdash e : \varphi \rightsquigarrow \Delta}{\Gamma \vdash_{alt} \_ \to e : T\ \overline{p} \Rightarrow \varphi \rightsquigarrow \Delta}\ (Alt_{\_})$$

Figure 4.4: Linear Core* - Infer Usage Environments

Theorem 2 (Progress)**.** Evaluation of a well-typed term does not block. If $\cdot \vdash e : \sigma$ then e is a value or there exists e' such that $e \to e'$.

Proof. By structural induction on the (only) typing derivation

Case: ΛI

(1) $\cdot \vdash (\Lambda p.\ e) : \forall p.\ \sigma$	by assumption
(2) $(\Lambda p.\ e)$ is a value	by definition

Case: ΛE

(1) $\cdot \vdash e_1\ \pi : \sigma[\pi/p]$ by assumption
(2) $\cdot \vdash e_1 : \forall p.\ \sigma$ by inversion on (ΛE)
(3) $\cdot \vdash_{mult} \pi$ by inversion on (ΛE)
(4) e_1 is a value or $\exists e_1'.e_1 \longrightarrow e_1'$ by the induction hypothesis (2)
Subcase e_1 is a value:
(5) $e_1 = \Lambda p.\ e_2$ by the canonical forms lemma (2)
(6) $(\Lambda p.\ e_2)\ \pi \longrightarrow e_2[\pi/p]$ by β-reduction on multiplicity (5,3)
Subcase $e_1 \longrightarrow e_1'$:
(5) $e_1\ \pi \longrightarrow e_1'\ \pi$ by context reduction on mult. application

Case: λI

(1) $\cdot \vdash (\lambda x{:}_\pi \sigma.\ e) : \sigma \to_\pi \varphi$ by assumption
(2) $(\lambda x{:}_\pi \sigma.\ e)$ is a value by definition

Case: λE

(1) $\cdot \vdash e_1\ e_2 : \varphi$ by assumption
(2) $\cdot \vdash e_1 : \sigma \to_\pi \varphi$ by inversion on (λE)
(3) $\cdot \vdash e_2 : \sigma$ by inversion on (λE)
(4) e_1 is a value or $\exists e_1'.e_1 \longrightarrow e_1'$ by the induction hypothesis (2)
Subcase e_1 is a value:
(5) $e_1 = \lambda x{:}_\pi \sigma.\ e$ by the canonical forms lemma
(6) $e_1\ e_2 \longrightarrow e[e_2/x]$ by term β-reduction (5,3)
Subcase $e_1 \longrightarrow e_1'$:
(5) $e_1\ e_2 \longrightarrow e_1'\ e_2$ by context reduction on term application

Case: *Let*

(1) $\cdot \vdash \mathbf{let}\ x{:}_\Delta \sigma = e\ \mathbf{in}\ e' : \varphi$ by assumption
(2) $\mathbf{let}\ x{:}_\Delta \sigma = e\ \mathbf{in}\ e' \longrightarrow e'[e/x]$ by definition of reduction. Wait, what?

Case: *LetRec*

(1) $\cdot \vdash \mathbf{let\ rec}\ \overline{x{:}_\Delta \sigma = e}\ \mathbf{in}\ e' : \varphi$ by assumption
(2) $\mathbf{let\ rec}\ \overline{x{:}_\Delta \sigma = e}\ \mathbf{in}\ e' \longrightarrow e'[\overline{\mathbf{let\ rec}\ \overline{x{:}_\Delta \sigma = e}\ \mathbf{in}\ e/x}]$ by definition of reduction. ∎

Case: *Case*

(1) $\cdot \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_{\overline{\Delta}} \sigma\ \{\overline{\rho_i \to e_i}\} : \varphi$ by assumption
(2) $\cdot \vdash e_1 : T\ \overline{p}$ by inversion of (case)
(3) $\overline{\cdot, z{:}_{\overline{\Delta}} \sigma \vdash_{alt} \rho_i \to e_i : \sigma \Longrightarrow \varphi}$ by inversion of (case)
(4) e_1 is a value or $\exists e_1'.e_1 \longrightarrow e_1'$ by induction hypothesis (2)

Subcase e_1 is a value:
(5) $e_1 = K\ \overline{e}$ by canonical forms lemma
(6) **case** e_1 **of** $z{:}_{\overline{\Delta}}\sigma\ \{\overline{\rho_i \to e_i}\} \longrightarrow e'[\overline{e}/\overline{x{:}_{\pi}\sigma}][K\ \overline{e}/z]$ by case reduction (5)
Subcase $e_1 \to e_1'$:
(5) **case** e_1 **of** $z{:}_{\overline{\Delta}}\sigma\ \{\overline{\rho_i \to e_1}\} \longrightarrow$ **case** e_1' **of** $z{:}_{\overline{\Delta}}\sigma\ \{\rho_i \to e_i\}$ by context reduction

□

Lemma 3 (Substitution of linear variables preserves typing)**.** If $\Gamma, x{:}_1\sigma \vdash e : \varphi$ and $\cdot \vdash e' : \sigma$ then $\Gamma \vdash e[e'/x] : \varphi$

Proof. By structural induction on the first derivation.

Case: Var_1

(1) $\cdot, x{:}_1\sigma \vdash x : \sigma$ by assumption
(2) $\cdot \vdash e' : \sigma$ by assumption
(3) $x[e'/x] = e'$ by def. of substitution
(4) $\cdot \vdash e' : \sigma$ by (1,2,3)

Case: $Weaken_\omega$

(1) $\Gamma, x{:}_1\sigma, y{:}_\omega\sigma' \vdash e : \varphi$ by assumption
(2) $\cdot \vdash e' : \sigma$ by assumption
(3) $\Gamma, x{:}_1\sigma \vdash e : \varphi$ by inversion on $Weaken_\omega$
(4) $\Gamma \vdash e[e'/x] : \varphi$ by induction hypothesis by (2,3)
(5) $\Gamma, y{:}_\omega\sigma' \vdash e[e'/x] : \varphi$ by $Weaken_\omega$ (4)

Case: $Weaken_\Delta$
Just like $Weaken_\omega$ with `s/omega/Delta`

Case: $Contract_\omega$

(1) $\Gamma, x{:}_1\sigma, y{:}_\omega\sigma' \vdash e : \varphi$ by assumption
(2) $\cdot \vdash e' : \sigma$ by assumption
(3) $\Gamma, x{:}_1\sigma, y{:}_\omega\sigma', y{:}_\omega\sigma' \vdash e : \varphi$ by inversion on $Contract_\omega$
(4) $\Gamma, y{:}_\omega\sigma', y{:}_\omega\sigma' \vdash e[e'/x] : \varphi$ by induction hypothesis by (2,3)
(5) $\Gamma, y{:}_\omega\sigma' \vdash e[e'/x] : \varphi$ by $Contract_\omega$ (4)

Case: $Contract_\Delta$
Just like $Contract_\omega$ with `s/omega/Delta`

Case: Var_ω
Impossible, $x{:}_1\sigma$ can't be in this rule's context

Case: Var_Δ

(1) $\Gamma, x{:}_1\sigma, y{:}_\Delta\sigma' \vdash e : \varphi$ by assumption
(2) $\cdot \vdash e' : \sigma$ by assumption
(3) $\Delta \restriction_1 = \Gamma$ by inversion on Var_Δ
(4) Stuck

Case: ΛI

(1) $\Gamma, x{:}_1\sigma \vdash \Lambda p.\ e : \forall p.\ \varphi$ by assumption
(2) $\cdot \vdash e' : \sigma$ by assumption
(3) $\Gamma, x{:}_1\sigma, p \vdash e : \varphi$ by inversion on ΛI
(4) $p \notin \Gamma$ by inversion on ΛI
(5) $\Gamma, p \vdash e[e'/x] : \varphi$ by induction hypothesis by (2,3)
(6) $\Gamma \vdash \Lambda p.\ e[e'/x] : \forall p.\ \varphi$ by ΛI (4,5)
(7) $(\Lambda p.\ e)[e'/x] = (\Lambda p.\ e[e'/x])$ by def. of substitution

Case: ΛE

(1) $\Gamma, x{:}_1\sigma \vdash e\ \pi : \varphi[\pi/p]$ by assumption
(2) $\cdot \vdash e' : \sigma$ by assumption
(3) $\Gamma, x{:}_1\sigma \vdash e : \forall p.\ \varphi$ by inversion on ΛE
(4) $\Gamma, x{:}_1\sigma \vdash_{mult} \pi$ by inversion on ΛE
(5) $\Gamma \vdash e[e'/x] \forall p.\ \varphi$ by induction hypothesis by (2,3)
(6) $\Gamma \vdash e[e'/x]\ \pi : \varphi[\pi/p]$ by ΛE (4,5)
(7) $(e\ \pi)[e'/x] = e[e'/x]\pi$ by def. of substitution

Case: λI

(1) $\Gamma, x{:}_1\sigma \vdash \lambda y{:}_\pi\sigma'.\ e : \sigma' \to_\pi \varphi$ by assumption
(2) $\cdot \vdash e' : \sigma$ by assumption
(3) $\Gamma, x{:}_1\sigma, y{:}_\pi\sigma' \vdash e : \varphi$ by inversion on λI
(4) $\Gamma, y{:}_\pi\sigma' \vdash e[e'/x] : \varphi$ by induction hypothesis by (2,3)
(5) $\Gamma \vdash \lambda y{:}_\pi\sigma'.\ e[e'/x] : \sigma' \to_\pi \varphi$ by $lLambdaI$ (4)
(6) $(\lambda y{:}_\pi\sigma'.\ e)[e'/x] = (\lambda y{:}_\pi\sigma'.\ e[e'/x])$ by def. of substitution

Case: λE_1 (linear variable occurs in function)

(1) $\Gamma, x{:}_1\sigma, \Gamma' \vdash e\ e'' : \varphi$ by assumption
(2) $\cdot \vdash e' : \sigma$ by assumption
(3) $\Gamma, x{:}_1\sigma \vdash e : \sigma' \to_\pi \varphi$ by inversion on λE
(4) $\Gamma' \vdash e'' : \sigma'$ by inversion on λE
(5) $\Gamma \vdash e[e'/x] : \sigma' \to_\pi \varphi$ by induction hypothesis by (2,3)
(6) $\Gamma, \Gamma' \vdash e[e'/x]\ e'' : \varphi$ by λE
(7) $(e''[e'/x] == e'')$ because the linear variable x cannot occur in e''
(8) $(e\ e'')[e'/x] = (e[e'/x]\ e'')$ by in def. of substitution and (7)

Case: λE_2 (linear variable occurs in argument)

(1) $\Gamma, \Gamma', x{:}_1\sigma \vdash e\ e'' : \varphi$	by assumption
(2) $\cdot \vdash e' : \sigma$	by assumption
(3) $\Gamma \vdash e : \sigma' \to_\pi \varphi$	by inversion on λE
(4) $\Gamma', x{:}_1\sigma \vdash e'' : \sigma'$	by inversion on λE
(5) $\Gamma' \vdash e''[e'/x] : \sigma'$	by induction hypothesis by (2,4)
(6) $\Gamma, \Gamma' \vdash e\ e''[e'/x] : \varphi$	by λE
(7) $(e[e'/x] == e)$	because the linear variable x cannot occur in e
(8) $(e\ e'')[e'/x] = (e\ e''[e'/x])$	by def. of substitution and (7)

□

Lemma 4 (Substitution of unrestricted variables preserves typing)**.** If $\Gamma, x{:}_\omega\sigma \vdash e : \varphi$ and $\Gamma' \vdash e' : \sigma$ and $\forall y{:}_\pi\varphi \in \Gamma'.\ \pi \neq 1$ then $\Gamma, \Gamma' \vdash e[e'/x] : \varphi$.

Proof. By structural induction on the first derivation.

Case: Var_ω

(1) $\Gamma, x{:}_\omega\sigma \vdash x : \sigma$	by assumption
(2) $\Gamma' \vdash e' : \sigma$	by assumption
(3) $\forall y{:}_\pi\varphi \in \Gamma'.\ \pi \neq 1$	by assumption
(4) $\forall y{:}_\pi\varphi \in \Gamma.\ \pi \neq 1$	by inversion on (Var_ω)
(5) $x[e'/x] = e'$	by def. of substitution
(6) $\Gamma, \Gamma' \vdash e' : \sigma$	by (1,2,5)
(7) We can only have 6 because of (3,4), how to add to proof?	

Case: Var_ω

(1) $\Gamma, x{:}_\omega\sigma \vdash y : \varphi$	by assumption
(2) $\Gamma' \vdash e' : \sigma$	by assumption
(3) $y[e'/x] = y$	by def. of substitution
(4) $\Gamma, \Gamma' \vdash y : \varphi$	by (1)
(5) Where does Γ' come from in (4)? Def. of substitution?	

Case: Var_Δ

(1) $\Gamma, y{:}_\Delta\sigma', x{:}_\omega\sigma \vdash y : \varphi$	by assumption
(2) $\Gamma' \vdash e' : \sigma$	by assumption
(3) $\forall y{:}_\pi\varphi \in \Gamma'.\ \pi \neq 1$	by assumption
(4) $\Delta \restriction_1 = \Gamma$	by inversion on (Var_Δ)
(5) $y[e'/x] = y$	by def. of substitution
(6) $\Gamma, \Gamma' \vdash y : \varphi$	by ???

Case: Var_1

(1) We couldn't have reached Var_1 with an unrestricted variable in the context, the split cannot allo

Case: $Weaken_\omega$

(1) $\Gamma, x{:}_\omega\sigma, y{:}_\omega\sigma' \vdash e : \varphi$ by assumption
(2) $\Gamma' \vdash e' : \sigma$ by assumption
(3) $\forall y{:}_\pi\varphi \in \Gamma'.\ \pi \neq 1$ by assumption
(4) $\Gamma, x{:}_\omega\sigma \vdash e : \varphi$ by inversion on $Weaken_\omega$
(5) $\Gamma, \Gamma' \vdash e[e'/x] : \varphi$ by induction hypothesis (2,3,4)
(6) $\Gamma, y{:}_\omega\sigma', \Gamma' \vdash e[e'/x] : \varphi$ by $Weaken_\omega$

Case: $Weaken_\Delta$

Just like $Weaken_\omega$ with `s/omega/Delta/g`.

Case: $Contract_\omega$

(1) $\Gamma, x{:}_\omega\sigma, y{:}_\omega\sigma' \vdash e : \varphi$ by assumption
(2) $\Gamma' \vdash e' : \sigma$ by assumption
(3) $\forall y{:}_\pi\varphi \in \Gamma'.\ \pi \neq 1$ by assumption
(4) $\Gamma, x{:}_\omega\sigma, y{:}_\omega\sigma', y{:}_\omega\sigma' \vdash e : \varphi$ by inversion on $Contract_\omega$
(5) $\Gamma, y{:}_\omega\sigma', y{:}_\omega\sigma', \Gamma' \vdash e[e'/x] : \varphi$ by induction hypothesis (2,3,4)
(6) $\Gamma, y{:}_\omega\sigma', \Gamma' \vdash e[e'/x] : \varphi$ by $Contract_\omega$

Case: $Contract_\Delta$

Just like $Contract_\omega$ with `s/omega/Delta/g`.

Case: ΛI

(1) $\Gamma, x{:}_\omega\sigma \vdash \Lambda p.\ e : \forall p.\ \varphi$ by assumption
(2) $\Gamma' \vdash e' : \sigma$ by assumption
(3) $\forall y{:}_\pi\varphi \in \Gamma'.\ \pi \neq 1$ by assumption
(4) $\Gamma, x{:}_\omega\sigma, p \vdash e : \varphi$ by inversion on ΛI
(5) $p \notin \Gamma$ by inversion on ΛI
(6) $\Gamma, p, \Gamma' \vdash e[e'/x] : \varphi$ by induction hypothesis by (2,3,4)
(7) $\Gamma, \Gamma' \vdash \Lambda p.\ e[e'/x] : \forall p.\ \varphi$ by ΛI (5,6)
(8) $(\Lambda p.\ e)[e'/x] = (\Lambda p.\ e[e'/x])$ by def. of substitution

Case: ΛE

(1) $\Gamma, x{:}_\omega\sigma \vdash e\ \pi : \varphi[\pi/p]$ by assumption
(2) $\Gamma' \vdash e' : \sigma$ by assumption
(3) $\forall y{:}_\pi\varphi \in \Gamma'.\ \pi \neq 1$ by assumption
(4) $\Gamma, x{:}_\omega\sigma \vdash e : \forall p.\ \varphi$ by inversion on ΛE
(5) $\Gamma, x{:}_\omega\sigma \vdash_{mult} \pi$ by inversion on ΛE
(6) $\Gamma, \Gamma' \vdash e[e'/x] \forall p.\ \varphi$ by induction hypothesis by (2,3,4)
(7) $\Gamma, \Gamma' \vdash e[e'/x]\ \pi : \varphi[\pi/p]$ by ΛE (5,6)
(8) $(e\ \pi)[e'/x] = e[e'/x]\ \pi$ by def. of substitution

Case: λI

(1) $\Gamma, x{:}_{\omega}\sigma \vdash \lambda y{:}_{\pi}\sigma'.\ e : \sigma' \to_{\pi} \varphi$ by assumption
(2) $\Gamma' \vdash e' : \sigma$ by assumption
(3) $\forall y{:}_{\pi}\varphi \in \Gamma'.\ \pi \neq 1$ by assumption
(4) $\Gamma, x{:}_{\omega}\sigma, y{:}_{\pi}\sigma' \vdash e : \sigma'$ by inversion on λI
(5) $\Gamma, \Gamma', y{:}_{\pi}\sigma' \vdash e[e'/x] : \sigma' \to_{\pi} \varphi$ by induction hypothesis
(6) $\Gamma, \Gamma' \vdash \lambda y{:}_{\pi}\sigma'.\ e[e'/x] : \sigma' \to_{\pi} \varphi$ by λI
(7) $(\lambda y{:}_{\pi}\sigma'.\ e)[e'/x] = (\lambda y{:}_{\pi}\sigma'.\ e[e'/x])$ by def. of subst.

Case: λE

(1) $\Gamma, x{:}_{\omega}\sigma, \Gamma'' \vdash e\ e'' : \varphi$ by assumption
(2) $\Gamma' \vdash e' : \sigma$ by assumption
(3) $\forall y{:}_{\pi}\varphi \in \Gamma'.\ \pi \neq 1$ by assumption
(4) $\Gamma, x{:}_{\omega}\sigma, \Gamma'', x{:}_{\omega}\sigma, \vdash$ by assumption
(5) $\Gamma, x{:}_{\omega}\sigma \vdash e : \sigma' \to_{\pi} \varphi$ by inversion
(6) $\Gamma'', x{:}_{\omega}\sigma$
(7) Auch, não posso duplicar a unrestricted value... como é que faço??

Case: Let

(1) $\Gamma, x{:}_{\omega}\sigma \vdash \mathbf{let}\ y{:}_{\Delta}\sigma' = e\ \mathbf{in}\ e'' : \varphi$ by assumption
(2) $\Gamma' \vdash e' : \sigma$ by assumption
(3) $\forall y{:}_{\pi}\varphi \in \Gamma'.\ \pi \neq 1$ by assumption
(4) $\Gamma, x{:}_{\omega}\sigma, y{:}_{\Delta}\sigma' \vdash e'' : \varphi$ by inversion
(5) $\Delta \vdash e : \sigma'$ by inversion
(6) $\Delta \subseteq \Gamma$ by inversion
(7) $\Delta, x{:}_{\omega}\sigma \vdash e : \sigma'$ by $Weaken$
(8) $\Gamma, y{:}_{\Delta}\sigma', \Gamma' \vdash e''[e'/x] : \varphi$ by induction hypothesis (4,2,3)
(9) $\Delta, \Gamma' \vdash e[e'/x] : \sigma'$ by induction hypothesis
(10) $\Delta, \Gamma' \subseteq \Gamma, \Gamma'$ by (6) and def. of $\subseteq$
(11) Stuck, we need to be able to say Δ, Γ' is the same usage environment as the one annotated in $y[$

□

Lemma 5 (Substitution of variables with usage environments preserves typing)**.** If $\Gamma, \Delta, x{:}_{\Delta}\sigma \vdash e : \varphi$ and $\Gamma', \Delta \vdash e' : \sigma$ then $\Gamma, \Gamma', \Delta \vdash e[e'/x] : \varphi$

Proof. By structural induction on the first derivation.

Case: Var_{Δ}

Case: ΛI

(1) $\Gamma, \Delta, x{:}_{\Delta}\sigma \vdash \Lambda p.\ e : \forall p.\ \varphi$
(2) $\Gamma, \Delta' \vdash e' : \sigma$
(3) $\Gamma, p, \Delta, x{:}_{\Delta}\sigma \vdash e : \varphi$ by inversion on ΛI
(4) $\Gamma, \Gamma', \Delta, p \vdash e[e'/x]$ by induction hypothesis (2,3)
(5) $\Gamma, \Gamma', \Delta \vdash \Lambda p.\ e[e'/x] : \forall p.\ \varphi$ by ΛI
(6) $(\Lambda p.\ e)[e'/x] = (\Lambda p.\ e[e'/x])$ by def. of subst.
(7) $\Gamma, \Gamma', \Delta \vdash (\Lambda p.\ e)[e'/x] : \forall p.\ \varphi$ by (5,6)

Case: ΛE

(1) $\Gamma, \Delta, x{:}_{\Delta}\sigma \vdash e\ \pi : \varphi$
(2) $\Gamma', \Delta \vdash e' : \sigma$
(3) $\Gamma, \Delta, x{:}_{\Delta}\sigma \vdash e : \forall p.\ \varphi$ by inversion on ΛE
(4) $\Gamma, \Delta \vdash_{mult} \pi$ by inversion on ΛE
(5) $\Gamma, \Gamma', \Delta, x{:}_{\Delta}\sigma$ by induction hypothesis (2,3)
(6) $\Gamma, \Gamma', \Delta \vdash e[e'/x]\ \pi : \varphi$ by ΛE
(7) $(e\ \pi)[e'/x] = (e[e'/x]\ \pi)$ by def. of subst.
(6) $\Gamma, \Gamma', \Delta \vdash (e\ \pi)[e'/x] : \varphi$ by (6,7)

Case: λI

(1) $\Gamma, \Delta, x{:}_{\Delta}\sigma \vdash \lambda y{:}_{\pi}\sigma'.\ e : \sigma' \to_{\pi} \varphi$
(2) $\Gamma', \Delta \vdash e' : \sigma$
(3) $\Gamma, \Delta, x{:}_{\Delta}\sigma, y{:}_{\pi}\sigma' \vdash e : \varphi$ by inversion on λI
(4) $\Gamma, \Gamma', \Delta, y{:}_{\pi}\sigma' \vdash e[e'/x] : \varphi$ by induction hypothesis (2,3)
(5) $\Gamma, \Gamma', \Delta \vdash \lambda y{:}_{\pi}\sigma'.\ e[e'/x] : \sigma' \to_{\pi} \varphi$ by λI
(6) $(\lambda y{:}_{\pi}\sigma'.\ e[e'/x]) = (\lambda y{:}_{\pi}\sigma'.\ e)[e'/x]$ by def. of subst.
(7) $\Gamma, \Gamma', \Delta \vdash \lambda(y{:}_{\pi}\sigma'.\ e)[e'/x] : \sigma' \to_{\pi} \varphi$ by (5,6)

Case: λE

(1) $\Gamma, \Gamma'', \Delta, x{:}_{\Delta}\sigma \vdash e\ e'' : \varphi$
(2) $\Gamma', \Delta \vdash e' : \sigma$
(3) $\Gamma, \Delta, x{:}_{\Delta}\sigma \vdash e : \sigma' \to_{\pi} \varphi$ by inversion on λE
(4) $\Gamma'' \vdash e'' : \sigma'$ by inversion on λE
(5) $\Gamma, \Gamma', \Delta \vdash e[e'/x] : \sigma' \to_{\pi} \varphi$ by induction hypothesis (2,3)
(6) $\Gamma, \Gamma', \Gamma'', \Delta \vdash (e[e'/x]\ e'') : \varphi$ by λE
(7) $(e[e'/x]\ e'') = (e\ e'')[e'/x]$ when the resources to Δ are needed to prove $x{:}_{\Delta}\sigma$ on the left??? Stuck

□

Bibliography

[1] J.-M. ANDREOLI. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation*, 2(3):297–347, 06 1992.

[2] C. Baker-Finch, K. Glynn, and S. Peyton Jones. Constructed product result analysis for haskell. *Journal of Functional Programming*, 14(2):211–245, March 2004.

[3] A. Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, The University of Edinburgh, 1996.

[4] J. Bernardy, M. Boespflug, R. R. Newton, S. P. Jones, and A. Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *CoRR*, abs/1710.09756, 2017.

[5] E. Brady. Idris 2: Quantitative Type Theory in Practice. In A. Møller and M. Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:26, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[6] J. Breitner. *Lazy Evaluation: From natural semantics to a machine-checked compiler transformation*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2016.

[7] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In P. Gastin and F. Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, pages 222–236, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[8] I. Cervesato, J. S. Hodas, and F. Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1):133–163, 2000.

[9] M. M. T. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. *SIGPLAN Not.*, 40(9):241–253, sep 2005.

[10] M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 1–13, New York, NY, USA, 2005. Association for Computing Machinery.

[11] J. Cheney and R. Hinze. First-class phantom types. 2003.

[12] O. Chitil. Common subexpressions are uncommon in lazy functional languages. In C. Clack, K. Hammond, and T. Davie, editors, *Implementation of Functional Languages*, pages 53–71, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[13] T. G. H. C. contributors. Glasgow haskell compiler / ghc source. https://gitlab.haskell.org/ghc/ghc, 2022.

[14] H. Curry. Functionality in combinatory logic. volume 20, pages 584–590. Department of Mathematics, The Pennsylvania State College, 1934.

[15] L. Damas and R. Milner. Principal type-schemes for functional programs. In R. A. DeMillo, editor, *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, pages 207–212. ACM Press, 1982.

[16] R. A. Eisenberg, G. Duboc, S. Weirich, and D. Lee. An existential crisis resolved: Type inference for first-class existential types. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.

[17] R. A. Eisenberg and S. Peyton Jones. Levity polymorphism. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 525–539, New York, NY, USA, 2017. Association for Computing Machinery.

[18] R. A. Eisenberg and J. Stolarek. Promoting functions to type families in haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, page 95–106, New York, NY, USA, 2014. Association for Computing Machinery.

[19] P. Fu, K. Kishida, and P. Selinger. Linear dependent type theory for quantum programming languages: Extended abstract. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20, page 440–453, New York, NY, USA, 2020. Association for Computing Machinery.

[20] J.-Y. Girard. Interpretation fonctionelle et elimination des coupures dans l'aritmetique d'ordre superieur. 1972.

[21] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[22] C. Hall, K. Hammond, S. P. Jones, and P. Wadler. Type classes in haskell. In D. Sannella, editor, *Programming Languages and Systems — ESOP '94*, pages 241–256, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[23] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, mar 1996.

[24] W. A. Howard. The formulae-as-types notion of construction. pages 479–490. 1980 (originally circulated 1969).

[25] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 190–203, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[26] S. L. P. Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[27] S. L. P. Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 636–666, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[28] S. P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, et al. Haskell 98, 1999.

[29] J. Le. Practical dependent types in haskell. https://blog.jle.im/entry/practical-dependent-types-in-haskell-1.html, 2016.

[30] J. Li, A. Lattuada, Y. Zhou, J. Cameron, J. Howell, B. Parno, and C. Hawblitzel. Linear types for large-scale systems verification. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022.

[31] S. Marlow et al. Haskell 2010 language report. 2010.

[32] S. Marlow and S. Peyton Jones. *The Glasgow Haskell Compiler*. Lulu, the architecture of open source applications, volume 2 edition, January 2012.

[33] N. D. Matsakis and F. S. Klock. The rust language. *Ada Lett.*, 34(3):103–104, oct 2014.

[34] L. Maurer, Z. Ariola, P. Downen, and S. Peyton Jones. Compiling without continuations. In *ACM Conference on Programming Languages Design and Implementation (PLDI'17)*, pages 482–494. ACM, June 2017.

[35] R. Mesquita. Ghengin: A vulkan-based, shader-centric, type-heavy, haskell game engine. https://github.com/alt-romes/ghengin, 2022.

[36] S. Peyton Jones and S. Marlow. Secrets of the glasgow haskell compiler inliner. *Journal of Functional Programming*, 12:393–434, July 2002.

[37] S. Peyton Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser, January 1993. Functional Programming, Glasgow 1993.

[38] S. Peyton Jones and W. Partain. Let-floating: Moving bindings to give faster programs. *Proc. of ICFP'96*, 31, 10 1996.

[39] S. Peyton Jones and A. Santos. A transformation-based optimiser for haskell. *Science of Computer Programming*, 32(1), October 1997.

[40] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, jan 2007.

[41] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, page 50–61, New York, NY, USA, 2006. Association for Computing Machinery.

[42] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, pages 408–425, Berlin, Heidelberg, 1974. Springer Berlin Heidelberg.

[43] A. Ruiz and D. Steinitz. hmatrix: Numeric linear algebra. https://hackage.haskell.org/package/hmatrix, 2021.

[44] A. Santos and S. Peyton Jones. *Compilation by transformation for non-strict functional languages*. PhD thesis, July 1995.

[45] I. SERGEY, D. VYTINIOTIS, S. L. P. JONES, and J. BREITNER. Modular, higher order cardinality analysis in theory and practice. *Journal of Functional Programming*, 27:e11, 2017.

[46] A. Serrano, J. Hage, S. Peyton Jones, and D. Vytiniotis. A quick look at impredicativity. *Proc. ACM Program. Lang.*, 4(ICFP), aug 2020.

[47] M. Sulzmann, M. Chakravarty, S. Peyton Jones, and K. Donnelly. System f with type equality coercions. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*, pages 53–66. ACM, January 2007.

[48] D. VYTINIOTIS, S. PEYTON JONES, T. SCHRIJVERS, and M. SULZMANN. Outsidein(x) modular type inference with local assumptions. *Journal of Functional Programming*, 21(4-5):333–412, 2011.

[49] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.