Todo list

Type-checking Linearity in Core
or: Semantic Linearity for a Lazy Optimising Compiler

Rodrigo Mesquita
Bernardo Toninho

August 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 [**?**, **?**] 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 [**?**], among other high-level correctness properties [**?**, **?**, **?**].

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 [**?**, **?**, **?**], 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 [**?**], a linearly and dependently typed language based on Quantitative Type Theory, Rust [**?**], a language whose ownership types build on linear types to guarantee memory safety without garbage collection or reference counting, and, more recently, Haskell [**?**], 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 [**?**], 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 [**?**]) in Section 2.5.

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 [**?**, **?**, **?**] 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.

In reality, the Core optimizing transformations only expose a more fundamental issue in the existing linear types in Haskell – its mismatch with the evaluation model. In call-by-need languages a mention of a variable might not entail using that variable - it depends on it being required or not. This is not explored at all and we're the first to do so as far as we know

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.

Change some of the second part of the introduction

We should discuss the alternative motivation of figuring out how to typecheck linearity in the presence of laziness on its own, why its hard and how it allows simpler use of linear types since the compiler doesn't constrain the programmer so much

Rather, the linearity x call-by-need should be the original motivation, with linear core as the prime example?

Explain examples of non-trivial interaction of linearity with laziness, with both lets and also with case expressions not evaluating expressions in WHNF, and otherwise

Glimpse at how core optimizations can get us into these situations where we have to see this linearity

Saying, finally, what we are going to do, and that our system is capable of seeing linearity in all of these programs, and more – it is capable of typechecking almost all optimizing transformations we studied

Changing our Goals into things we actually did

Conclude by explaining that the document is structured in such a way that the payload starts in chapter 3 after delivering the background knowledge necessary to read through it (enumerate), and that we revise the introduction there, more in depth, assuming understanding of the background concepts

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; DONE
- Validating that our type-system accepts programs before and after each transformation is applied; WIP Proofs of optimizations
- Arguing the soundness of the resulting system (i.e. no semantically non-linear programs are deemed linear); DONE (modulo 1)
- Implementing our extensions to Core in GHC, the leading Haskell Compiler. NOPE.

CHAPTER 2

Background

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), evaluation strategies, Haskell's main intermediate language (Core) and its formal foundation (System F_C) and, finally, an overview of GHC's pipeline with explanations of crucial Core-to-Core optimizing transformations that we'll try to validate.

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 [**?**, **?**], 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` $\to \star$. 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 [**?**, **?**], 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 [**?**] (the Curry-Howard isomorphism [**?**, **?**]), 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 [**?**, **?**] and whose *de-facto* implementation is GHC, the Glasgow Haskell Compiler [**?**]. 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 in the type system 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. In this example Haskell program, the use of linear types could eventually prevent the handle from being leaked by requiring it to be used exactly once.

```
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 [**?**], Haskell goes much further with, e.g., arbitrary-rank types [**?**], type-class polymorphism [**?**], levity polymorphism [**?**], multiplicity polymorphism [**?**], and, more recently, impredicative polymorphism [**?**].
- Type level computation by means of type classes [**?**] and Haskell's type families [**?**, **?**, **?**], 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 [**?**], 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 [**?**, **?**, **?**] 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, i.e. refine 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 [**?**].

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. This definition makes 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[**?**], enforce size compatibility in operations between matrices and vectors[**?**], to implement red-black trees guaranteeing its invariants at compile-time, or to implement a material system in a game engine[**?**].

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.

As we'll see after introducing linear haskell and multiplicities, GADTs/local equality evidence and multiplicity polymorphism interact in a non-trivial manner, and the way

2.3 Linear Haskell

The introduction of linear types to Haskell's type system is originally described in Linear Haskell [**?**]. While in Section 6.1 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 when the handle is forgotten.

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 [**?**]:

- 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.

Either hint at a the importance of the definition of consuming values and expressions here, or be very clear about it in the Linear Core section

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].

[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.

```
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 written with $\multimap$). A function with a multiplicity of `Many` ($\rightarrow_\omega$) 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 ($\rightarrow$) implicitly has multiplicity `Many`. Multiplicities naturally allow for *multiplicity polymorphism*.

Consider the functions f and g which take as an argument a function from `Bool` to `Int`. Function f expects a linear function ($Bool \rightarrow_1 Int$), whereas g expects an unrestricted function ($Bool \rightarrow_\omega Int$). 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 ($\rightarrow_?$) should match the multiplicity of both f ($\rightarrow_1$) and g ($\rightarrow_\omega$). 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 ($\rightarrow_m$), making h a well-typed argument to both f and g (m will unify with 1 and ω at the call sites).

2.4 Evaluation Strategies

Edit first draft. Não gosto particularmente do que escrevi

Cite Ariola's work, Plotkins? Who to cite for call-by-value?

Unlike most mainstream programming languages, Haskell has a so called *lazy* evaluation strategy, namely *call-by-need*. *Call-by-need* evaluation dictates that an expression is only evaluated when it is needed (so no work is done to evaluate expressions that are unused at runtime), and the values that are indeed evaluated are memoized and shared across use sites. For example, the following program will only compute *factorial* 2500 if *expr* evaluates to *True*, and in that case the work to compute it is only done once, despite being used twice:

```
let f res = if expr then res * res else 0
in f (factorial 2500)
```

In contrast, mainstream languages commonly use an *eager* evaluation strategy called *call-by-value*, in which expressions are eagerly evaluated to a value. In the above example, under *call-by-value*, *factorial* 2500 would always be evaluated and passed as a value, regardless of being used in the body. It is out of the scope of this work to discuss the merits and tradeoffs of eager vs. lazy evaluation.

A third option is the *call-by-name* evaluation strategy. In *call-by-name*, expressions are only evaluated when needed, however, there is no sharing. In the above example,

it would only evaluate *factorial* 2500 *twice* if *expr* were *True*. Despite being similar to *call-by-need*, in practice, language implementors prefer *call-by-need* over *call-by-name* to achieve *non-strict* semantics, because the latter duplicates a lot of work, while the former only does work once and then shares the result.

Call-by-value, *call-by-name*, and *call-by-need* are the most common evaluation strategies used to describe a language's execution model. Each has a different definition of β-reduction for the operational semantics of the language:

- Under *call-by-value*, a function application is reduced to a value by evaluating the function and the argument to a *value*, then substituting occurrences of the function argument variable by the argument value:

$$\frac{e \longrightarrow e'}{(\lambda x.b)\ e \longrightarrow v\ e'} \qquad (\lambda x.b)\ v \longrightarrow b[v/x]$$

- Under *call-by-name*, a function application is reduced to a value by evaluating the function to a *value* (a lambda), then substituting occurrences of the function argument variable by the whole argument expression:

$$(\lambda x.e)\ e' \longrightarrow e[e'/x]$$

- Under *call-by-need*, a function application is reduced to a value by evaluating the function to a *value*, then transforming the function application into a **let** binding:

$$(\lambda x.e)\ e' \longrightarrow \mathsf{let}\ x = e'\ \mathsf{in}\ e$$

  The **let** binding introduces a suspended computation known as a *thunk*, whose value is only computed when the binding is *forced*. After evaluating the expression, the binding is overwritten with the result of the computation and subsequent uses of the binding use the computed value without additional work.

In essence, *call-by-value* gives the language *strict* evaluation semantics ($f(\bot) = \bot$) by evaluating the arguments in order before the body of the function, whereas *call-by-name* and *call-by-need* give the language *non-strict* semantics, by evaluating the arguments only when they are needed ($f(\bot)$ is not $\bot$ if f's argument isn't used). *Call-by-need* additionally guarantees work is only done once by suspending computations into a *thunk* that is overwritten with the value of running the suspended computation when it is *forced*, ensuring sharing.

The subtleties of suspending computations (i.e. creating *thunks*) and *forcing* them under *call-by-need* evaluation are especially relevant in the context of our work regarding linearity in Core, so we discuss them in more depth below.

2.4.1 Evaluation in Haskell

When do we require an argument for the call-by-need lambda calculus without case? I suppose when arguments appear in function position

In Haskell, applying a function to an expression, in general[4], results in a **let** binding that suspends the evaluation of the expression/computation of the result (seen in the β-reduction rule under call-by-need).

[4] Optimisations such as ocurrence analysis, allow us to substitute some expressions in a *call-by-name*-style without creating a let binding if the argument is only used once in the body.

Suspending a computation amounts to giving a name to the unevaluated expression. When this name itself is evaluated then the computation is said to be *forced*, and a result is computed by evaluating the expression associated to that name. The result of the computation then overwrites the unevaluated expression the name was associated with, and subsequent uses of the same name will now refer to the computed result. The suspended computation is called a *thunk*, and call-by-need evaluation is usually modelled with a mutable heap for the existing thunks or values they are overwritten with [**?**].

On second thought, it might be better to discuss the notion of consuming resources in the call-by-need lambda calculus in the first section of the Linear Core chapter (note, all resources are consumed when something is evaluated to a lambda, so function application is already correct

As for the call-by-need linear lambda calculus paper, it might be better to discuss it there

Connection between forcing thunks and consuming resources

Cite https://www.sciencedirect.com/science/article/pii/S1571066104000222

Introduce evaluation semantics, call-by-value as the most common, call-by-need, call-by-name, call-by-let?

Call-by-need, refer to background work, discuss sharing etc?

We've already introduced linear haskell, so be sure to include at least one example of linearity with laziness, and how the linear typechecker doesn't see it

Foreshadowing para que syntatically coisas aparecem muitas vezes, but not a problem

Also refer that paper about call-by-need linear lambda calculus, but only discuss it later in Related Work

Falar de thunks + sharing, de suspended computations e de coisas em memória serem overwritten. Talvez deixar a discussão sobre interação com linearidade para mais tarde

Falar de WHNF vs NF, possivelmente numa subsecção (maybe not and leave this for third chapter)

Haskell isn't really call-by-need, it's like a mix of things, right? e.g. linear applications are evaluated w/ call by name without allocating closures

2.5 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 main types (expressions, types, and coercions) corresponding to 15 constructors [**?**]. 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 [**?**, **?**]), 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 [**?**] 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 $OutsideIn(X)$ [**?**]. 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 [**?**]. 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 [**?**] terms is given in Figure 2.3, which corresponds exactly to the syntax of System F [**?**, **?**] 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
	|	$\Lambda a{:}\kappa.\ e \mid e\ \varphi$	Type abstraction/application
	|	$\lambda x{:}\sigma.\ e \mid e_1\ e_2$	Term abstraction/application
	|	$\mathsf{let}\ x{:}\sigma = e_1\ \mathsf{in}\ e_2$	
	|	$\mathsf{case}\ e_1\ \mathsf{of}\ \overline{p \to e_2}$	
	|	$e \blacktriangleright \gamma$	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 **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. $(e{:}\sigma \blacktriangleright \textbf{sym}\ \tau \sim \sigma) : \tau$.

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

Figure 2.4: System F_C's Types and Coercions

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

- 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* [**?**], 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.

We could mention Sequent Core as the origin of jumps and joins here, or simply cite it

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.

2.6 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.

2.6.1 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 2.5 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).

2.6.2 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 2.5). In light of it, 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 [?, ?]. These were subsequently refined and expanded [?, ?, ?, ?, ?]. 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 [?]. 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:

$$\mathsf{let}\ x = e\ \mathsf{in}\ e' \implies \mathsf{let}\ x = e\ \mathsf{in}\ e'[e/x]$$

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 [?] 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 ($\overline{x{:}\sigma}$):

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

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 [?]. 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:

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

  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 [**?**]. 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:

$$\begin{array}{lll}1. & (\mathsf{let}\ v = e\ \mathsf{in}\ b)\ a & \Longrightarrow\ \mathsf{let}\ v = e\ \mathsf{in}\ b\ a \\ 2. & \mathsf{case}\ (\mathsf{let}\ v = e\ \mathsf{in}\ b)\ \mathsf{of}\ \ldots & \Longrightarrow\ \mathsf{let}\ v = e\ \mathsf{in\ case}\ b\ \mathsf{of}\ \ldots \\ 3. & \mathsf{let}\ x = (\mathsf{let}\ v = e\ \mathsf{in}\ b)\ \mathsf{in}\ c & \Longrightarrow\ \mathsf{let}\ v = e\ \mathsf{in\ let}\ x = b\ \mathsf{in}\ c\end{array}$$

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

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:

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

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 [**?**].

Static argument and lambda lifting. *Lambda lifting* is a transformation that abstracts over free variables in functions by making them λ-bound arguments [**?**, **?**]. 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 [**?**]. This transformation creates two functions from an original one: a worker and a wrapper. The worker receives unboxed values [**?**] 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.

Add examples/definitions to the three previous transformations

Add a paragraph about the binder swap, and another about the reverse binder swap, possibly foreshadowing how it is important in our study as something that is only linearity preserving because of the way other optimizations are defined.

```
f x = letrec go y = case x of z {(a, b) → ...(expensive z)...}
  in ...
```

If we do the reverse binder-swap we get

```
f x = letrec go y = case x of z {(a, b) → ...(expensive x)...}
  in ...
```

and now we can float out:

```
f x = let t = expensive x
  in letrec go y = case x of z {(a, b) → ...(t)...}
  in ...
```

Now (expensive x) is computed once, rather than once each time around the 'go' loop..

Add another example sequence of transformations with the example Simon provided using reverse-binder-swap

2.6.3 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 [**?**]. 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[5], such as LLVM, x86 and x64, or (recently) JavaScript and WebAssembly.

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

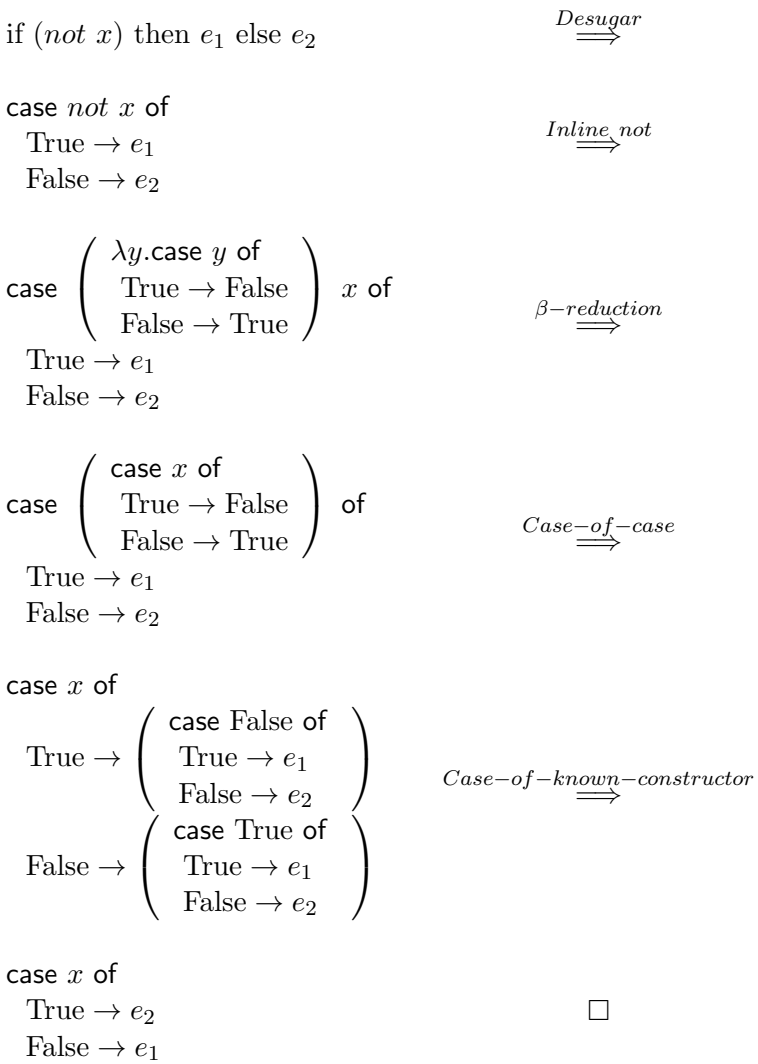

Figure 2.5: Example sequence of transformations

CHAPTER 3

A Type System for Semantic Linearity in Core

- A linear type system statically guarantees that linear resources are consumed *exactly once*.
- With linear types, programmers can create powerful abstractions that enforce certain resources to be used linearly, such as file handles or allocated memory.
- Linear Haskell brings the promises of linear types to the Haskell, and is available in the Glasgow Haskell Compiler (GHC) starting from version 9.0.
- For example: trivial linear haskell program
- GHC desugars Linear Haskell into an intermediate language called Core, a System FC, and then applies multiple so-called Core-to-Core optimising transformations in succession.
- For example: some simple inlining
- Linear functions are likewise optimised by the compiler but, crucially, the optimisations must preserve linearity! It would indeed be disastrous if the compiler discarded or duplicated resources such as a file handle or some allocated memory.
- From the above example, consider x linear. It might seem as though x is being used twice, but since Core's evaluation strategy is call-by-need, we will only evaluate the let when the value is required, and since after optimisation it is never used, we'll never compute the value and thus never use the resource
- In essence, in a lazy language such as Haskell, there's a mismatch between syntactic and semantic linearity, with the former referring to syntactic occurrences of a linear resource in the program text, and the latter to whether the program consumes the resource exactly once in a semantic sense, accounting for the evaluation strategy such as in the let example above.
- The problem: Core is not aware of this. Its linear type system is still very naive, in that it doesn't understand semantic linearity, and will reject most transformed linear programs, since often transformations turn programs that are syntactically linear to programs that are only semantically linear.

- Despite using Core's type system to validate the implementation of the optimising transformations preserve types (and all their combinations), we don't use the Core's linear type system to validate whether the optimisations and their implementation preserves *linearity*. We only *think* that the current optimisations preserve linearity, but we don't check it.
- Understanding semantic linearity isn't trivial, and, in fact, we aren't aware of any linear type system that accounts for non-strict evaluation semantics besides our own.
- We develop a linear type system for Core that understands semantic linearity. We prove the usual preservation and progress theorems with it, and then prove that multiple optimisations preserve types With it, we can accept all intermediate programs GHC produces while applying Core-to-core optimising transformations.
- We implemented a GHC plugin that validates every Core program produced by the optimising pipeline using our type system
- Contributions (ref section for each):
  - We explain and build intuition for semantic linearity in Core programs by example
  - We present a linear type system for call-by-need Core which understands all axis of semantic linearity we previously exposed; and we prove type safety of that system
  - We prove that multiple Core-to-core optimising transformations preserve linearity + types under the type system; and discuss
  - We develop a GHC plugin that checks all intermediate Core programs using our type system

This is the Introduction. We should start elsewhere

The introduction needs a lot of motivation!

Inicío deve motivar o leitor, e temos de explicar qual é o problema da linearidade sintatica em Haskell (vs semantica), e a interação disso com o call-by-need/lazy evaluation. Quase como se fosse um paper.

Interação de linearity com a call-by-need/laziness

Compiler optimizations put programs in a state where linearity mixed with call-by-need is pushed to the limits. That is, the compiler preserves linearity, but in a non-trivial semantic way.

We present a system that can understand linearity in the presence of lazy evaluation, and validate multiple GHC core-to-core optimizations in this system, showing they can preserve types in our system where in the current implemented Core type system they don't preserve linearity.

Note that programmers won't often write these programs in which let binders are unused, or where we pattern match on a known constructor – but these situations happen all the time in the intermediate representation of an optimising compiler ...

This was the old introduction to chapter 3. I think we can do better

Continue introduction

Our contributions are (to rewrite):

- We expose a connection between the definition of *consuming* a resource in a linear type system and with the fundamental definition of *evaluation/progress* and the evaluation strategy of a language, making the *consuming* more precise across languages (in fact, generalizing the definition of consuming a resource from Linear Haskell).

3.1 Linearity, Semantically

A linear type system statically guarantees that linear resources are *consumed* exactly once. Consequently, whether a program is well-typed under a linear type system intrinsically depends on the precise definition of *consuming* a resource. Even though *consuming* a resource is commonly regarded as synonymous with *using* a resource, i.e. with the syntactic occurrence of the resource in the program, that is not always the case. In fact, this section highlights the distinction between using resources syntactically and semantically as a principal limitation of linear type systems for (especially) non-strict languages, with examples motivated by the constructs available in GHC Core and how they're evaluated.

Consider the following program in a functional pseudo-language, where a computation that closes the given *handle* is bound to *x* before the *handle* is returned:

$$f : \mathit{Handle} \multimap \mathit{Handle}$$
$$f\ \mathit{handle} = \textbf{let}\ x = \mathit{close}\ \mathit{handle}\ \textbf{in}\ \mathit{handle}$$

In this seemingly innocent example, the *handle* appears to be closed before returned, whereas in fact the handle will only be closed if the let bound computation is effectively run (i.e. evaluated). The example illustrates that *consuming* a resource is not necessarily synonymous with using it syntactically, as depending on the evaluation strategy of the pseudo-language, the computation that closes the handle might or not be evaluated, and if it isn't, the *handle* in that unused computation is not consumed. Expanding on this, consider the above example program under distinct evaluation strategies:

Call-by-value With *eager* evaluation, the let bound expression *close handle* is eagerly evaluated, and the *handle* will be closed before being returned. It is clear that a linear type system should not accept such a program since the linear resource *handle* is duplicated – it is used in a computation that closes it, while still being made available to the caller of the function.

Call-by-need On the other hand, with *lazy* evaluation, the let bound expression will only be evaluated when the binding *x* is needed. We return the *handle* right away, and the let binding is forgotten as it cannot be used outside the scope of the function, so the handle is not closed by *f*. Under the lens of *call-by-need* evaluation, *using* a resource in a let binding only results in the resource being *consumed* if the binding itself is *consumed*. We argue that a linear type system under *call-by-need* evaluation should accept the above program, unlike a linear type system for the same program evaluated *call-by-value*.

Intuitively, a computation that depends on a linear resource to produce a result consumes that resource iff the result is effectively computed; in contrast, a computation that depends on a linear resource, but is never run, will not consume that resource.

From this observation, and exploring the connection between computation and evaluation, it becomes clear that *linearity* and *consuming resources*, in the above example and for programs in general, should be defined in function of the language's evaluation strategy. We turn our focus to *linearity* under *call-by-need*, not only because GHC Core is

Alguma dificuldade em dizer exatamente como é que evaluation drives/is computa-

call-by-need, but also because the distinction between semantically and syntactically consuming a resource is only exposed under non-strict semantics. Indeed, under *call-by-value*, syntactic occurrences of a linear resource directly correspond to semantically using that resource[1] because *all* expressions are eagerly evaluated – if all computations are eagerly run, all linear resources required by computations are *eagerly consumed*.

3.1.1 Semantic Linearity by Example

Aligned with our original motivation of typechecking linearity in GHC Core such that optimising transformations preserve linearity, and with the secondary goal of understanding linearity in a non-strict context, this section helps the reader build an intuition for semantic linearity through examples of Core programs that are semantically linear but rejected by Core's linear type system. In the examples, a light green background highlights programs that are syntactically linear and are accepted by Core's naive linear type system. A light yellow background marks programs that are semantically linear, but are not seen as linear by Core's naive linear type system. Notably, the linear type system we develop in this work accepts all light yellow programs. A light red background indicates that the program simply isn't linear, not even semantically, i.e. the program effectively discards or duplicates linear resources.

Let bindings

We start our discussion with non-strict (non-recursive) let bindings, i.e. let bindings whose body is evaluated only when the binding is needed, rather than when declared. In Core, a let binding entails the creation of a *thunk* that suspends the evaluation of the let body (for background, see Section 2.4.1). When the *thunk* is *forced*, the evaluation is carried out, and the result overrides the *thunk* – the let binding now points to the result of the evaluation. A *thunk* is *forced* (and the suspended computation is evaluated) when the binding itself is evaluated.

In a linear type system, a non-strict let binding that depends on a linear resource x doesn't consume the resource as long as the binding isn't evaluated – the suspended computation only uses the resource if it is run. For this reason, we can't naively tell whether x is consumed just by looking at the let binding body. In the following example, we assign a computation that depends on the resource x to a binder, which is then returned:

```
f1 :: (a ⊸ b) → a ⊸ b
f1 use x =
  let y = use x
    in y
```

The linear resource x is used exactly once, since it is used exactly once in the body of the binding and the binding is used exactly once in the let body. According to Linear Haskell's core calculus $\lambda^{q}_{\rightarrow}$ [**?**], let bound variables are annotated with a multiplicity which is multiplied (as per the multiplicities semiring) with the multiplicities of the variables that are free in the binder's body. In short, if a let binder is linear (has multiplicity 1) then the linear variables free in its body are only used once; if the let binder is unrestricted (has multiplicity ω) then the resources in its body are consumed many times, meaning no linear

[1] With the minor exception of trivial aliases, which don't entail any computation even in *call-by-value*. In theory, we could use in mutual exclusion any of the aliases to refer to a resource without loss of linearity

variables can occur in that let binder's body. Unfortunately, GHC's implementation of Linear Haskell doesn't seem to infer multiplicities for lets yet, so while the above program should typecheck in Linear Haskell, it is rejected by GHC.

The next example exposes the case in which the let binder is ignored in the let body. Here, the linear resource x is used in y's body and in the let body, however, the resource is still used linearly semantically because y isn't used at all, thus x is consumed just once in the let body:

```
f2 :: (a ⊸ a) → a ⊸ a
f2 use x =
  let y = use x
  in use x
```

Programmers don't often write bindings that are completely unused, yet, an optimising compiler will produce intermediate programs with unused bindings[2] from transformations such as inlining, which can substitute out occurrences of the binder (e.g. y is inlined in the let body).

Let bindings can also go unused if they are defined before branching on case alternatives. At runtime, depending on the branch taken, the let binding will be evaluated only if it occurs in that branch. Both optimising transformations (float-out), and programmers used to non-strict evaluation, can produce programs with bindings that are selectively used in the case alternatives, for instance:

```
f3 :: (a ⊸ a) → Bool → a ⊸ a
f3 use bool x =
  let y = use x
  in case bool of
    True → x
    False → y
```

This example essentially merges $f1$ with $f2$, using x directly in one branch and using y in the other. Semantically, this program is linear because the linear resource x ends up being used exactly once in both case alternatives, directly or indirectly.

Shifting our focus from not using a let binding to using it (more than once), we reiterate that a let binding creates a *thunk* which is only evaluated once, and re-used subsequently. Despite the binder body only being evaluated once, and thus its resources only used once to compute a result, we can still only consume said result of the computation once – perhaps surprisingly, as the perception so far is that "resources are consumed during computation" and multiple uses of the same let binder share the result that was computed only once. Illustratively, the following program must *not* typecheck:

```
f4 :: (a ⊸ b) → a ⊸ (b, b)
f4 use x =
  let y = use x
  in (y, y)
```

Intuitively, the result of the computation must also be used exactly once, despite being

[2] Unused bindings are then also dropped by the optimising compiler

effectively computed just once, because said result may still contain (parts of) the linear resource. The trivial example is *f4* applied to *id* – the result of computing *id x* is *x*, and *x* must definitely not be shared! Indeed, if the result of the computation involving the linear resource was, e.g., an unrestricted integer, then sharing the result would not involve consuming the resource more than once. Concretely, the result of evaluating a let binder body using linear resources, if computed, must be consumed exactly once, or, otherwise, we risk discarding or duplicating said resources.

Lastly, consider a program which defines two let bindings *z* and *y*, where *z* uses *y* which in turn uses the linear resource *x*:

```
f5 :: (a ⊸ a) → a ⊸ ()
f5 use x =
  let y = use x
  let z = use y
  in ()
```

Even though the binding *y* is used in *z*, *x* is still never consumed because *z* isn't evaluated in the let body, and consequently *y* isn't evaluated either – never consuming *x*. We use this example to highlight that even for let bound variables, the syntactic occurrence of a variable isn't enough to determine whether it is used. Instead, we ought to think of uses of *y* as implying using *x*, and therefore uses of *z* imply using *x*, however, if neither is used, then *x* isn't used. Since *x* is effectively discarded, this example also violates linearity.

Summary The examples so far build an intuition for semantic linearity in the presence of lazy let bindings. In essence, an unused let binding doesn't consume any resources, and a let binding used exactly once consumes its resources exactly once. Let binders that depend on linear resources must be used *at most once* – let bound variables are *affine* in the let body. Moreover, if the let binding (y) isn't used in the let body, then the resources it depends on ($\overline{x}$) must still be used – the binding y is mutually exclusive with the resources $\overline{x}$ (for the resources to be used linearly, either the binder occurs exactly once y, or the resources $\overline{x}$ do). We'll later see how we can encode this principle of mutual exclusivity between let bindings and their dependencies using so called *usage environments*, in Section 3.2.2.

Recursive let bindings

Second, we look into recursive let bindings. For the most part, recursive let bindings behave as non-recursive let bindings, i.e. we must use them *at most once* because, when evaluated, the linear resources used in the binders bodies are consumed. The defining property of a group of mutually recursive let bindings is that the binders bound in that group can occur, without restrictions, in the bodies of those same binders. The same way that, in a let body, evaluating a binding that uses some resource exactly once consumes that resource once, using the binding in its own definition also entails using that resource once. Consider the following program, that calls a recursive let-bound function defined in terms of the linear resource *x* and itself:

```
f6 :: Bool → a ⊸ a
f6 bool x =
  let go b
```

```
      = case b of
          True → x
          False → go (¬ b)
  in go bool
```

Function *f6* is semantically linear because, iff it is consumed exactly once, then *x* is consumed exactly once. We can see this by case analysis on *go*'s argument

- When *bool* is *True*, we'll use the resource *x*
- When *bool* is *False*, we recurse by calling *go* on *True*, which in turn will use the resource *x*.

In *go*'s body, *x* is used directly in one branch and indirectly in the other, by recursively calling *go* (which we know will result in using *x* linearly). It so happens that *go* will terminate on any input, and will always consume *x*. However, termination is not a requirement for a binding to use *x* linearly, and we could have a similar example in which *go* might never terminate but still uses *x* linearly if evaluated:

```
f7 :: Bool → a ⊸ a
f7 bool x =
  let go b
      = case b of
          True → x
          False → go b
  in go bool
```

The key to linearity in the presence of non-termination is Linear Haskell's definition of a linear function: *if a linear function application (f u) is consumed exactly once, then the argument (u) is consumed exactly once.* If *f u* doesn't terminate, it is never consumed, thus the claim holds vacuously; that's why *f8* typechecks:

```
f8 :: a ⊸ b
f8 x = f8 x
```

If *go* doesn't terminate, we aren't able to compute (nor consume) the result of *f7*, so we do not promise anything about *x* being consumed (*f7*'s linearity holds trivially). If it did terminate, it would consume *x* exactly once (e.g. if *go* was applied to *True*).

Determining the linear resources used in a recursive binding might feel peculiar since we need to know the linear resources used by the binder to determine the linear resources it uses. The paradoxical definition is difficult to grasp, just how learning that a function can be defined in terms of itself is perplexing when one is first introduced to general recursion. Informally, we *assume* the binding will consume some linear resources exactly once, and use that assumption when reasoning about recursive calls such that those linear resources are used exactly once.

Generalizing, we need to find a set of linear resources (Δ) that satisfies the recursive equation [3] arising from given binding x, such that:

[3] This set of resources will basically be the least upper bound of the sets of resources used in each mutually recursive binding scaled by the times each binding was used

1. Occurrences of x in its own body are synonymous with using all resources in Δ exactly once,

2. And if the binding x is fully evaluated, then all resources in Δ are consumed exactly once

Finding a solution to this equation is akin to finding a (principle) type for a recursive binding: the binding needs to be given a type such that occurrences of that binding in its own body typecheck using that type. Foreshadowing, the core system we developed assumes recursive let bindings to be annotated with a set of resources satisfying the equations; but we also present an algorithm to determine this solution, and distinguish between an *inference* and a *checking* phase, where we first determine the linear resources used by a group of recursive bindings and only then check whether the binding is linear, in our implementation of checking of recursive lets.

There might not be a solution to the set of equations. In this case, the binding undoubtedly entails using a linear resource more than once. For example, if we use a linear resource *x* in one case alternative, and invoke the recursive call more than once, we might eventually consume *x* more than once:

```
f9 :: Bool → Bool ⊸ Bool
f9 bool x =
  let go b
      = case b of
        True → x
        False → go (¬ b) ∧ go True
  in go bool
```

Note that if returned *x* instead of *go bool* in the let body, then, despite the binding using *x* more than once, we would still consume *x* exactly once, since recursive bindings are still lazy.

Lastly, we extend our single-binding running example to use two mutually recursive bindings that depend a linear resource:

```
f10 :: Bool → a ⊸ a
f10 bool x =
  let go1 b
        = case b of
          True → go2 b
          False → go1 (¬ b)
      go2 b
        = case b of
          True → x
          False → go1 b
  in go1 bool
```

As before, we must find a solution to the set of equations defined by the mutually recursive bindings to determine which resources will be consumed. In this case, *go1* and *go2* both consume *x* exactly once if evaluated. We additionally note that a strongly connected group of recursive bindings (i.e. all bindings are transitively reachable from any of them)

will always consume the same set of resources – if all bindings are potentially reachable, then all linear resources are too.

Summary Recursive let bindings behave like non-recursive let bindings in that if they aren't consumed, the resources they depend on aren't consumed either. However, recursive let bindings are defined in terms of themselves, so the set of linear resources that will be consumed when the binder is evaluated is also defined in terms of itself (we need it to determine what resources are used when we recurse). We can intuitively think of this set of linear resources that will be consumed as a solution to a set of equations defined by a group of mutually recursive bindings, which we are able to reason about without an algorithm for simpler programs. In our work, the core type system isn't concerned with deriving said solution, but we present a simple algorithm for inferring with our implementation.

Case expressions

Finally, we discuss semantic linearity for case expressions, which have been purposefully left for last as the key ingredient that brings together the semantic linearity insights developed thus far, because, essentially, *case expressions drive evaluation* and semantic linearity can only be understood in function of how expressions are evaluated.

Up until now, the example functions have always linearly transformed linear resources, taking into consideration how expressions will be evaluated (and thus consumed) to determine if resources are being used linearly. However, there have been no examples in which linear resources are *fully consumed* in the bodies of linear functions. In other words, all example functions so far return a value that has to be itself consumed exactly once to ensure the linear argument is, in turn, consumed exactly once – as opposed to functions whose application simply needs to be evaluated to guarantee its linear argument is consumed (functions that return an unrestricted value). For example, the entry point to the linear array API presented in Linear Haskell takes such a function as its second argument:

$$newMArray :: Int \to (MArray\ a \multimap Unrestricted\ b) \multimap b$$

In short, case expressions enable us to consume resources and thus write functions that fully consume their linear arguments. To understand exactly how, we turn to the definition of *consuming* a resource from Linear Haskell [**?**]:

- 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.

That is, we can consume a linear resource by fully evaluating it, through case expressions. In section 3.1.2 we generalize the idea that consuming a resource is deeply tied to evaluation. Here, we continue building intuition for semantic linearity, first reviewing how case expressions evaluate expressions, and then exploring how they consume resources, by way of example.

In Core, case expressions are of the form **case** e_s **of** z $\{\overline{\rho_i \to e_i}\}$, where e_s is the case *scrutinee*, z is the case *binder*, and $\overline{\rho_i \to e_i}$ are the case *alternatives*, composed of a pattern ρ_i and of the corresponding expression e_i. Critically:

1. The case scrutinee is always evaluated to Weak Head Normal Form (WHNF).
2. Evaluating to WHNF an expression that is already in WHNF is a no-op, that is, no computation whatsoever occurs, unlike evaluating a e.g. function application.
3. The case binder is an alias to the result of evaluating the scrutinee to WHNF.
4. The alternative patterns are always exhaustive, i.e. there always exists a pattern that matches the WHNF of a value resulting from evaluating the scrutinee, where a pattern is either a wildcard that matches all expressions ($\_$), or a constructor and its linear and non-linear component binders ($K\ \overline{x}\,\overline{y}$, with $\overline{x}$ as linearly-bound variables and $\overline{y}$ as unrestricted ones).

To explore these case properties in the presence of linearity, we start with an example of a program that constructs a tuple from linear resources then pattern matches on it, then uses both linearly-bound variables from the tuple pattern match. This is well-typed in Linear Haskell:

$$f11 :: a \multimap b \multimap (a \multimap b \multimap c) \to c$$
$$f11\ x\ y\ use = \textbf{case}\ (x, y)\ \textbf{of}\ \{(a, b) \to use\ a\ b\}$$

What might be more surprising is that a similar program which discards the pattern variables and instead uses the resources in the scrutinee is also semantically linear, despite not being accepted by Linear Haskell:

$$f12 :: a \multimap b \multimap (a \multimap b \multimap c) \to c$$
$$f12\ x\ y\ use = \textbf{case}\ (x, y)\ \textbf{of}\ z\ \{(a, b) \to use\ x\ y\}$$

We justify that *f12* is linear by appealing to property 2 – since the expression (the tuple) being scrutinized is already in WHNF, evaluating it will not consume neither x nor y. Even if the tuple was constructed with two expressions using x and y respectively, no computation would happen since we aren't using neither a nor b (thereby never forcing the arguments of the tuple). However, if we did use a in the case body, then x would be unavailable:

$$f13 :: a \multimap a \multimap (a \multimap a \multimap c) \to c$$
$$f13\ x\ y\ use = \textbf{case}\ (x, y)\ \textbf{of}\ z\ \{(a, b) \to use\ a\ x\}$$

This idea that x and a are mutually exclusive is the same behind let bindings being mutually exclusive to the resources that define them. By forcing the pattern variable (or the let binding) we run the computations defined in terms of the linear variables used for that constructor argument (or let binder body), but otherwise, if we don't use those binders, then we don't run the computation thus no resources are consumed.

A third option for this example is to use the case binder z instead of a, b or x, y:

$$f14 :: a \multimap b \multimap (a \multimap b \multimap c) \to c$$

```
f14 x y use = case (x, y) of z {(a, b) → uncurry use z}
```

Again, z is mutually exclusive with a, b and with x, y, but at least one of the three must occur to ensure the linear resources are consumed. In this example, we can think that using a entails using the resource x, b the resource y, and the case binder z entails using both a and b.

Dually, consider the scrutinee to be an expression that's not in WHNF, s.t. evaluating it to WHNF will require doing computation and thus consume linear resources that are used in it:

```
f15 :: a ⊸ b ⊸ (a ⊸ b ⊸ (c, d)) → (c, d)
f15 x y use = case use x y of z {(a, b) → z}
```

Unlike when the scrutinee was in WHNF, we can no longer use x, y in the case alternatives, but we *must* still use either the case binder z or the linear pattern variables a, b, e.g. it would be quite disastrous if any of the following typechecked:

```
doubleFree :: Ptr ⊸ (Ptr ⊸ Result) → Result
doubleFree x free = case free x of z {Result v → free x}
```

```
leakPointer :: Ptr → ()
leakPointer x = case id x of z {_ → ()}
```

The result of evaluating the scrutinee must be consumed exactly to guarantee that the resources used in the scrutinee are fully consumed, or risk them being only "almost" consumed. Take for example *use* in *f15* to simply be $(,)$: it is not sufficient for *use x y* to be evaluated to WHNF to consume *x* and *y*. Otherwise, if all the resources were considered to be fully consumed after the scrutinee were evaluated in a case expression, we could simply ignore the pattern variables, effectively discarding linear resources (for cases such as the *use* $= (,)$ example). In short, if the scrutinee is not in WHNF we must either consume the case binder or the linear components of the pattern.

However, it is crucial to also consider pattern matches on constructors without any linear components. If the constructor has no linear fields, it means the result can be consumed unrestrictedly and, therefore, all linear resources used in the computation have been fully consumed. For instance, the following program, where *K2* has no fields and *K1* has one linear field, shouldn't typecheck because in case alternatives that matched a constructor without linear fields the scrutinee resources are no longer available:

```
f :: a ⊸ a
f x = case K1 x of z {K2 → x; K1 a → x}
```

Actually, this example is fine; in practice, it only matters that resources are consumed to be able to use the case binder unrestrictedly. This one doesn't typecheck, but is still semantically linear

This particular example has a known constructor being scrutinized which might seem like an unrealistic example, but we recall that during the transformations programs undergo in an optimizing compiler, many programs such as this naturally occur (e.g. if the definition of a function were inlined in the scrutinee).

Moreover, in a branch of a constructor without linear fields we also know the result of evaluating the scrutinee to be unrestricted, so we can also use the case binder unrestrictedly and refer to it zero or more times. For example, this program is also linear:

```
f16 :: () ⊸ ()
f16 x = case x of z {() → z <> z}
```

Further exploring that each linear field must be consumed exactly once, and that resources in WHNF scrutinees aren't consumed, we are able to construct more contrived examples, the following two of which the first doesn't typecheck because the same linear field is used twice, but the second one does since it uses each linear field exactly once (despite pattern matching on the same components twice)

```
f w = case w of z
  (a, b) →
    case (a, b) of z'
      (c, d) →
        (a, c)
```

```
f w = case w of z
  (a, b) →
    case (a, b) of z'
      (c, d) →
        (a, d)
```

Finally, we consider the default case alternatives, also known as wildcards (written _), in the presence of linearity: Matching against the wildcard doesn't provide new information, so we linearity is seen as before but without fully consuming linear resources (in non-linear patterns) nor binding new linear resources (in linear patterns). If the scrutinee is in WHNF, we can either use the resources from the scrutinee or the case binder in that alternative, if the scrutinee is not in WHNF, we *must* use the case binder, as it's the only way to linearly consume the result of evaluating the scrutinee to WHNF.

Summary Case expressions evaluate their scrutinees to WHNF, introduce a case binder, and bind pattern variables. If the scrutinee is already in WHNF, all resources occurring in it are still available in the case alternative, alongside the case binder and the pattern-bound variables. In the case alternative, either the resources of the scrutinee, the case binder, or the linearly bound pattern variables must be used exactly once, but mutually exclusively. For scrutinees not in WHNF, in the case alternative, either the case binder or the linear pattern variables must be used mutually exclusively. If the pattern doesn't bind any linear resources, then it may be consumed unrestrictedly, and therefore the case binder may also be used unrestrictedly.

3.1.2 Generalizing linearity in function of evaluation

Indeed, as hinted towards in the previous section, there's a deep connection between *evaluation* and *consuming resources*.

Definition X.Y: A linear resource is consumed when it is either fully evaluated (NF) to a value, or when it is returned s.t. an application of that function being fully evaluated would fully evaluate the resource to a value. Or something like that. Note how this generalizes Linear Haskell's definition of consuming a resource: ...

- We could almost say that eventually everything all linear resources must be evaluated to NF to be consumed, or returned by a function s.t. a continuation of that function has to evaluate the result to NF., or something.

- Discuss our own generalized (call-by-value, call-by-name, etc) definition of consuming resources by evaluation. Something like, if an expression is fully evaluated, all linear resources that expression depends on to compute a result are consumed, or something...

- How does this relate to strictness? Reference the section of Linear Haskell about linearity and strictness, and basically revisit what they say.

3.2 Linear Core

In this section, we develop a linear calculus λ^{π}_{Δ}, dubbed *Linear Core*, that combines the linearity-in-the-arrow and multiplicity polymorphism introduced by Linear Haskell [**?**] with all the key features from GHC's Core language, except for type equality coercions[4]. Specifically, our core calculus is a linear lambda calculus with algebraic datatypes, case expressions, recursive let bindings, and multiplicity polymorphism.

Linear Core makes much more precise the various insights discussed in the previous section by crystalizin them together in a linear type system for which we prove soundness via the usual preservation and progress theorems. Crucially, the Linear Core type system accepts all the *semantically linear* example programs (highlighted with light yellow) from Section 3.1.1, which Core currently rejects.

We also note that despite the focus on GHC Core, the fundamental ideas for understanding linearity in a call-by-need calculus can be readily applied to other call-by-need languages.

Explicar algumas das ideias fundamentais, e apresentar as regras iterativamente. Podemos começar com as triviais e avançar para os dois pontos mais difíceis : Lets e Cases

3.2.1 Linear Core Overview

Linear Core is a

Syntax, examples

Remember to mention we assume all patterns are exhaustive

[4] We explain a main avenue of future work, multiplicity coercions, in Section 6.2

3.2.2 Usage environments

Usage environments as a way to encode mutual exclusivity between using a variable and the resources it is comprised of. Explain definition, it is literally the variables used to type the binder in the case of the let. A bit more complicated for cases i.e. "which resources are we using when using the case-binder? effectively, the scrutinee. what about case pat vars?..." and so on

3.2.3 Lazy let bindings

Let bindings are hard, if they are used then we use resources. If they don't get used then we use no resources! In practice, resources that show up in the body of the let must be used, be it by using the let binder, or by using them directly. This makes the let binder and the resources in its body mutual exclusive.

Explain the idea of suspended computation, and how resources will be consumed to some extent when we force the computation – also foreshadowing that evaluation to WHNF doesn't necessarily consume all resources

Assign usage environments to let-bound variables, trivial usage of usage environments (in contrast with case expressions)

Recursive let bindings

3.2.4 Case expressions evaluate to WHNF

Case expressions are the means by which we do evaluation and pattern matching – when things are scrutinized, we evaluate them (if they aren't evaluated – tag is 0), and then compare the result against multiple alternatives

When things are evaluated, that's when consumption of resources really happen. For example, closing a handle is only closed when we pattern match on the result of closing the handle (a state token). This means two things

Item 1. Pattern matching on an expression in WHNF does no computation, so no resources are used

Item 2. Pattern matching an expression that is evaluated will not consume all the resources that define that computation – because of laziness, we only evaluate things to WHNF. To fully consume a value, we need to consume all the linear components of that pattern.

In practice, we can't know which resources are consumed by evaluating a given expression. The resources become in a limbo state – they cannot be used directly because they might have been consumed, but they mustn't be considered as consumed, because they might not have been. We say these resources enter a proof irrelevant state. They must still be tracked as though they weren't consumed, but they cannot be used directly to construct the program. How can we ensure these proof irrelevant resource variables are fully consumed? With usage environments – for the case binder and for the pattern variables, and otherwise propagate

The trick here is to separate the case rules into two separate rules, one that fires when the scrutinee is in WHNF, the other when it isn't.

The case binder and pattern variables will consume the scrutinee resources, be those irrelevant or relevant resources

Splitting

3.3 Metatheory

Consider making type safety and optimizations a section of their own, so we can have a reverse-binder-swap subsection

3.3.1 Type safety

We proved soundness of our system...

The harder cases are for the interesting ones - lets, cases, and case alternatives

Theorem 1 (Type preservation)**.** If $\Gamma; \Delta \vdash e : \sigma$ and $e \longrightarrow e'$ then $\Gamma; \Delta \vdash e' : \sigma$

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

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

(1) $\Gamma; \Delta, \Delta' \vdash (\lambda x{:}_{\pi}\sigma.\ e)\ e' : \varphi$
(2) $\Gamma; \Delta \vdash (\lambda x{:}_{\pi}\sigma.\ e) : \sigma \to_{\pi} \varphi$ — by inversion on (λE)
(3) $\Gamma; \Delta' \vdash e' : \sigma$ — by inversion on (λE)
Subcase $\pi = 1, p$:
(4) $\Gamma; \Delta, x{:}_{1,p}\sigma \vdash e : \varphi$ — by inversion on (λI)
(5) $\Gamma[\Delta'/x]; \Delta, \Delta' \vdash e[e'/x] : \varphi$ — by linear subst. lemma (3,4)
(6) $\Gamma[\Delta'/x] = \Gamma$ — since Γ is well defined before x's binding (1)
Subcase $\pi = \omega$:
(4) $\Delta' = \cdot$ — by inversion on (λE_{ω})
(5) $\Gamma, x{:}_{\omega}\sigma; \Delta \vdash e : \varphi$ — by inversion on (λI)
(6) $\Gamma; \Delta, \cdot \vdash e[e'/x] : \varphi$ — by unrestricted subst. lemma (3,4,5)

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

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

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

(1) $\Gamma; \Delta, \Delta' \vdash \mathbf{let}\ x{:}_{\Delta}\sigma = e\ \mathbf{in}\ e' : \varphi$
(2) $\Gamma; \Delta \vdash e : \sigma$ — by inversion on *Let*
(3) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash e' : \varphi$ — by inversion on *Let*
(4) $\Gamma; \Delta, \Delta' \vdash e'[e/x] : \varphi$ — by Δ-var subst. lemma (2,3)

Case: $\textbf{let rec } \overline{x_i{:}_\Delta \sigma_i = e_i} \textbf{ in } e' \longrightarrow e'[\overline{\textbf{let rec } \overline{x_i{:}_\Delta \sigma_i = e_i} \textbf{ in } e_i / x}]$

(1) $\Gamma; \Delta, \Delta' \vdash \textbf{let rec } \overline{x_i{:}_\Delta \sigma_i = e_i} \textbf{ in } e' : \varphi$
(2) $\overline{\Gamma, \overline{x_i{:}_\Delta \sigma_i}; \Delta \vdash e_i : \sigma_i}$ by inversion on *LetRec*
(3) $\Gamma, \overline{x_i{:}_\Delta \sigma_i}; \Delta, \Delta' \vdash e' : \varphi$ by inversion on *LetRec*
(4) $\overline{\Gamma; \Delta, \cdot \vdash \textbf{let rec } \overline{x_i{:}_\Delta \sigma_i = e_i} \textbf{ in } e_i : \sigma_i}$ by *LetRec* (2,2)
(6) $\Gamma; \Delta, \Delta' \vdash e'[\overline{\textbf{let rec } \overline{x_i{:}_\Delta \sigma_i = e_i} \textbf{ in } e_i / x}] : \varphi$ by Δ-var subst. (3,4)

Case: $\textbf{case } K\ \overline{e} \textbf{ of } z{:}_\Delta \sigma\ \{\ldots, K\ \overline{x{:}_\pi \sigma} \to e'\} \longrightarrow e'[\overline{e/x}][K\ \overline{e}/z]$ This is definitely another of the most interesting cases. We must invoke split, use Alt0 or AltN, use delta and unr. substitution, subcases where the case binder is used and isn't, use CaseWHNF and realize that if it were an expression reduced first with CaseNotWHNF then the usage environment is proof irrelevant, rearrange the usage environments of the case pattern alternatives, etc...

(1) $\Gamma; \Delta, \Delta' \vdash \textbf{case } K\ \overline{e_i} \textbf{ of } z{:}_\Delta \sigma\ \{\ldots, K\ w{:}_\pi \sigma \to e'\} : \varphi$
(2) $K\ \overline{e_i}$ is in WHNF by def. of WHNF
(3) $\Gamma; \Delta \vdash K\ \overline{e_i} : \sigma$ by inv. on $CaseWHNF$
(4) $\Gamma, z{:}_\Delta \sigma; \Delta, \Delta' \vdash_{alt} K\ \overline{w{:}_\pi \sigma} \to e' :^z_\Delta \sigma \Rightarrow \varphi$ by inv. on $CaseWHNF$
Subcase $K\ \overline{w{:}_\pi \sigma} = K\ \overline{x{:}_\omega \sigma}, \overline{y_i{:}_1 \sigma_i}^n, n > 0$
(5) $K : \overline{\sigma_i \to_\pi} \sigma \in \Gamma$ by inv. on $AltN$
(6) $\overline{\Gamma; \cdot \vdash e_i : \sigma}, \overline{\Gamma; \Delta_i \vdash e_i : \sigma}, \overline{\Delta_i} = \Delta$ by constructor application lemma (3,5)
(7) $\Gamma, z{:}_\Delta \sigma, \overline{x{:}_\omega \sigma}, \overline{y_i{:}_{\Delta_i} \sigma_i}; \Delta, \Delta' \vdash e' : \varphi$ by inv. on $AltN$
(8) $\Gamma, z{:}_\Delta \sigma, \overline{y_i{:}_{\Delta_i} \sigma_i}; \Delta, \Delta' \vdash e' : \varphi$ by inv. on $AltN$
TODO: Δ_i must be empty in order to type AltN/invoke substitution

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

(1) $\Gamma; \Delta, \Delta' \vdash \textbf{case } K\ \overline{e} \textbf{ of } z{:}_\Delta \sigma\ \{\ldots, \_ \to e'\} : \varphi$
(2) $\Gamma; \Delta \vdash K\ \overline{e} : \sigma$
(3) $K\ \overline{e}$ is in WHNF
(4) $\Gamma, z{:}_\Delta \sigma; \Delta, \Delta' \vdash_{alt} \_ \to e' :^z_\Delta \sigma \Rightarrow \varphi$ by inv on $CaseWHNF$
(5) $\Gamma, z{:}_\Delta \sigma; \Delta, \Delta' \vdash e' : \varphi$ by inv on $Alt\_$
(6) $\Gamma; \Delta, \Delta' \vdash e'[K\ \overline{e}/z] : \varphi$ by Δ-subst.

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

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

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

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

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

(1) $e \longrightarrow e'$ by inversion on case reduction
(2) $\Gamma; \Delta, \Delta' \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_\Delta\sigma\ \{\rho_i \to e_i'\} : \varphi$
(3) e is not in WHNF since it can evaluate further by (1)
TODO: that's not quite the definition of WHNF!
(4) $\Gamma; \Delta \vdash e : \sigma$ by inv.
(5) $\overline{\Gamma, z{:}_{[\Delta]}\sigma; [\Delta], \Delta' \vdash_{alt} \rho \to e'' :^{z}_{[\Delta]} \sigma \Rightarrow \varphi}$
(6) $\Gamma; \Delta \vdash e' : \sigma'$ by i.h. (1,4)
(7) $\Gamma; \Delta, \Delta' \vdash \mathbf{case}\ e'\ \mathbf{of}\ z{:}_\Delta\sigma\ \{\rho_i \to e_i'\} : \varphi$ by $CaseNotWHNF$

□

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

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

Case: ΛI

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

Case: ΛE

(1) $\cdot;\cdot \vdash e_1\ \pi : \sigma[\pi/p]$ by assumption
(2) $\cdot;\cdot \vdash e_1 : \forall p.\ \sigma$ by inversion on (ΛE)
(3) $\cdot;\cdot \vdash_{mult} \pi$ 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) $\cdot;\cdot \vdash (\lambda x{:}_\pi\sigma.\ e) : \sigma \to_\pi \varphi$ by assumption
(2) $(\lambda x{:}_\pi\sigma.\ e)$ is a value by definition

Case: λE

(1) $\cdot;\cdot \vdash e_1\ e_2 : \varphi$ by assumption
(2) $\cdot;\cdot \vdash e_1 : \sigma \to_\pi \varphi$ by inversion on (λE)
(3) $\cdot;\cdot \vdash e_2 : \sigma$ 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

Case: *LetRec*

(1) $\cdot;\cdot \vdash \mathbf{let\ rec}\ \overline{x_i{:}_\Delta \sigma_i = e_i}\ \mathbf{in}\ e' : \varphi$ by assumption
(2) $\mathbf{let\ rec}\ \overline{x_i{:}_\Delta \sigma_i = e_i}\ \mathbf{in}\ e' \longrightarrow e'[\mathbf{let\ rec}\ \overline{x_i{:}_\Delta \sigma_i = e_i}\ \mathbf{in}\ e_i/x]$ by definition

Case: $CaseWHNF$ and $CaseNotWHNF$

(1) $\cdot;\cdot \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}.\sigma\ \{\overline{\rho_i \to e_i}\} : \varphi$ by assumption
(2) $\cdot;\cdot \vdash e : \sigma$ by inversion of $CaseWHNF$ or $CaseNotWHNF$
(4) $\overline{\cdot, z{:}.;\cdot\sigma \vdash_{alt} \rho_i \to e_i :^z \sigma \Rightarrow \varphi}$ by inversion of $CaseWHNF$ or $CaseNotWHNF$
(5) e is a value or $\exists e'.\ e \longrightarrow e'$ by induction hypothesis (2)
Subcase e is a value
TODO: this should rather be whether e is in WHNF,
and there should be a better connection between values and WHNF explicit.
(6) $e_1 = K\ \overline{e}$ by canonical forms lemma
(7) e is in WHNF by (6) (TODO: match correctly value and WHNF)
(8) $\overline{\rho_i \to e_i}$ is a complete pattern by coverage checker
(9) $\mathbf{case}\ K\ \overline{e}\ \mathbf{of}\ z{:}.\sigma\ \{\overline{\rho_i \to e_i}\} \longrightarrow e_i\overline{[e/x]}[K\ \overline{e}/z]$
by case reduction on pattern or wildcard
Subcase $\exists e'.\ e \longrightarrow e'$
(6) e is definitely not in WHNF
(7) $\mathbf{case}\ e\ \mathbf{of}\ z{:}.\sigma\ \{\overline{\rho_i \to e_1}\} \longrightarrow \mathbf{case}\ e'\ \mathbf{of}\ z{:}.\sigma\ \{\rho_i \to e_i\}$ by ctx. case reduction

□

Lemma 1 (Substitution of linear variables preserves typing)**.** *If* $\Gamma; \Delta, x{:}_1\sigma \vdash e : \varphi$ *and* $\Gamma; \Delta' \vdash e : \sigma$ *then* $\Gamma\,[\Delta'/x]\,; \Delta, \Delta' \vdash e[e'/x] : \varphi$*, where* $\Gamma[\Delta'/x]$ *substitutes all occurrences of* x *in the usage environments of variables in* Γ *by the linear variables in* Δ'*. (really,* x *couldn't appear anywhere else since* x *is linear).*

Proof. By structural induction on the first derivation.

Case: ΛI

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

Case: ΛE

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

Case: λI_1

(1) $\Gamma; \Delta, x{:}_1\sigma \vdash \lambda y{:}_1\sigma'.\ e : \sigma' \to_1 \varphi$
(2) $\Gamma; \Delta' \vdash e' : \sigma$
(3) $\Gamma; \Delta, x{:}_1\sigma, y{:}_1\sigma' \vdash e : \varphi$ by inversion on λI
(4) $\Gamma[\Delta'/x]; \Delta, y{:}_1\sigma', \Delta' \vdash e[e'/x] : \varphi$ by induction hypothesis by (2,3)
(5) $\Gamma[\Delta'/x]; \Delta, \Delta' \vdash \lambda y{:}_1\sigma'.\ e[e'/x] : \sigma' \to_1 \varphi$ by λI (4)
(6) $(\lambda y{:}_1\sigma'.\ e)[e'/x] = (\lambda y{:}_1\sigma'.\ e[e'/x])$ by def. of substitution

Case: λI_ω

(1) $\Gamma; \Delta, x{:}_1\sigma \vdash \lambda y{:}_\omega\sigma'.\ e : \sigma' \to_\omega \varphi$
(2) $\Gamma; \Delta' \vdash e' : \sigma$
(3) $\Gamma, y{:}_\omega\sigma'; \Delta, x{:}_1\sigma \vdash e : \varphi$ by inversion on λI
(4) $\Gamma[\Delta'/x], y{:}_\omega\sigma'; \Delta, \Delta' \vdash e[e'/x] : \varphi$ by induction hypothesis by (2,3)
(5) $\Gamma[\Delta'/x]; \Delta, \Delta' \vdash \lambda y{:}_\omega\sigma'.\ e[e'/x] : \sigma' \to_1 \varphi$ by λI (4)
(6) $(\lambda y{:}_\omega\sigma'.\ e)[e'/x] = (\lambda y{:}_\omega\sigma'.\ e[e'/x])$ by def. of substitution

Case: Var_1

(1) $\Gamma; x{:}_1\sigma \vdash x : \sigma$
(2) $\Gamma; \Delta' \vdash e' : \sigma$
(3) $\Gamma[\Delta'/x]; \Delta' \vdash e' : \sigma$ by weaken
(4) $x[e'/x] = e'$ by def. of substitution
(5) $\Gamma[\Delta'/x]; \Delta' \vdash e' : \sigma$ by (3)

Case: Var_ω

(1) Impossible. $x{:}_1\sigma$ can't be in the context.

Case: Var_Δ

(1) $\Gamma, y{:}_{\Delta, x:_1\sigma}\varphi; \Delta, x{:}_1\sigma \vdash y : \varphi$
(2) $\Gamma; \Delta' \vdash e' : \sigma$
(3) $y[e'/x] = y$
(4) $\Gamma[\Delta'/x], y{:}_{\Delta,\Delta'}\varphi; \Delta, \Delta'; \vdash y : \varphi$ by Var_Δ

Case: *Split*

Trivial induction

Case: λE_1

(1) $\Gamma; \Delta, \Delta'', x{:}_1\sigma \vdash e\ e'' : \varphi$
(2) $\Gamma; \Delta' \vdash e' : \sigma$
Subcase x occurs in e
(3) $\Gamma; \Delta, x{:}_1\sigma \vdash e : \sigma' \to_1 \varphi$ by inversion on λE_1
(4) $\Gamma; \Delta'' \vdash e'' : \sigma'$ by inversion on λE_1
(5) $\Gamma[\Delta'/x]; \Delta, \Delta' \vdash e[e'/x] : \sigma' \to_1 \varphi$ by induction hypothesis (2,3)
(6) $\Gamma[\Delta'/x]; \Delta, \Delta', \Delta'' \vdash e[e'/x]\ e'' : \varphi$ by λE_1
(7) $(e[e'/x]\ e'') = (e\ e'')[e'/x]$ because x does not occur in e''
Subcase x occurs in e''
(3) $\Gamma; \Delta \vdash e : \sigma' \to_1 \varphi$ by inversion on λE_1
(4) $\Gamma; \Delta'', x{:}_1\sigma \vdash e'' : \sigma'$ by inversion on λE_1
(5) $\Gamma[\Delta'/x]; \Delta'', \Delta' \vdash e''[e'/x] : \sigma'$ by induction hypothesis (2,4)
(6) $\Gamma[\Delta'/x]; \Delta, \Delta', \Delta'' \vdash e\ e''[e'/x] : \varphi$ by λE_1
(7) $(e\ e''[e'/x]) = (e\ e'')[e'/x]$ because x does not occur in e

Case: λE_ω

(1) $\Gamma; \Delta, x{:}_1\sigma \vdash e\ e'' : \varphi$
(2) $\Gamma; \Delta' \vdash e' : \sigma$
(3) x does not occur in e'' by e'' linear context is empty
(4) $\Gamma; \Delta, x{:}_1\sigma \vdash e : \sigma' \to_\omega \varphi$ by inversion on λE_ω
(5) $\Gamma; \cdot \vdash e'' : \sigma'$ by inversion on λE_ω
(6) $\Gamma[\Delta'/x]; \Delta, \Delta' \vdash e[e'/x] : \sigma' \to_\omega \varphi$ by induction hypothesis (2,4)
(7) $\Gamma[\Delta'/x]; \Delta, \Delta' \vdash e[e'/x]\ e'' : \varphi$ by λE_ω
(8) $(e[e'/x]\ e'') = (e\ e'')[e'/x]$ because x does not occur in e''

Case: *Let*

(1) $\Gamma; \Delta' \vdash e' : \sigma$
Subcase x occurs in e
(2) $\Gamma; \Delta, x{:}_1\sigma, \Delta'' \vdash \mathbf{let}\ y{:}_{\Delta, x:_1\sigma}\sigma' = e\ \mathbf{in}\ e'' : \varphi$
(3) $\Gamma, y{:}_{\Delta, x:_1\sigma}\sigma'; \Delta, x{:}_1\sigma, \Delta'' \vdash e'' : vp$ by inversion on *Let*
(4) $\Gamma; \Delta, x{:}_1\sigma \vdash e : \sigma'$ by inversion on *Let*
(5) $\Gamma[\Delta'/x], y{:}_{\Delta, \Delta'}\sigma'; \Delta, \Delta', \Delta'' \vdash e''[e'/x]$ by induction hypothesis $(1, 3)$
(6) $\Gamma[\Delta'/x]; \Delta, \Delta' \vdash e[e'/x] : \sigma'$ by induction hypothesis $(1, 4)$
(7) $\Gamma[\Delta'/x]; \Delta, \Delta', \Delta'' \vdash \mathbf{let}\ y{:}_{\Delta, \Delta'}\sigma' = e[e'/x]\ \mathbf{in}\ e''[e'/x] : \varphi$ (5,6) by *Let*
(8) $(\mathbf{let}\ y{:}_{\Delta, \Delta'}\sigma' = e[e'/x]\ \mathbf{in}\ e''[e'/x]) = (\mathbf{let}\ y{:}_{\Delta, \Delta'}\sigma' = e\ \mathbf{in}\ e'')[e'/x]$ by subst.
Subcase x does not occur in e
(2) $\Gamma; \Delta, \Delta'', x{:}_1\sigma \vdash \mathbf{let}\ y{:}_{\Delta}\sigma' = e\ \mathbf{in}\ e'' : \varphi$
(3) $\Gamma, y{:}_{\Delta}\sigma'; \Delta, \Delta'', x{:}_1\sigma \vdash e'' : \varphi$ by inversion on *Let*
(4) $\Gamma; \Delta \vdash e : \sigma'$ by inversion on *Let*
(5) $\Gamma[\Delta'/x], y{:}_{\Delta}\sigma'; \Delta, \Delta', \Delta'' \vdash e''[e'/x] : \varphi$ by induction hypothesis (1,3)
(6) $\Gamma[\Delta'/x]; \Delta, \Delta', \Delta'' \vdash \mathbf{let}\ y{:}_{\Delta}\sigma' = e\ \mathbf{in}\ e''[e'/x] : \varphi$ by *Let* (2,5,6)
(7) $\mathbf{let}\ y{:}_{\Delta}\sigma' = e\ \mathbf{in}\ e''[e'/x] = (\mathbf{let}\ y{:}_{\Delta}\sigma' = e\ \mathbf{in}\ e'')[e'/x]$ by x does not occur in e

Case: *LetRec*

(1) $\Gamma; \Delta' \vdash e' : \sigma$
Subcase $x{:}_1\sigma$ occurs in some e_i
(2) $\Gamma; \Delta, x{:}_1\sigma, \Delta'' \vdash \mathbf{let\ rec}\ \overline{y_i{:}_{\Delta, x:_1\sigma}\sigma_i = e_i}\ \mathbf{in}\ e'' : \varphi$
(3) $\Gamma, \overline{y_i{:}_{\Delta, x:_1\sigma}\sigma_i}; \Delta, x{:}_1\sigma, \Delta'' \vdash e'' : \varphi$ by inversion on *LetRec*
(4) $\overline{\Gamma, \overline{y_i{:}_{\Delta, x:_1\sigma}\sigma_i}; \Delta, x{:}_1\sigma \vdash e_i : \sigma_i}$ by inversion on *LetRec*
(5) $\Gamma[\Delta'/x], \overline{y_i{:}_{\Delta, \Delta'}\sigma_i}; \Delta, \Delta', \Delta'' \vdash e''[e'/x] : \varphi$ by induction hypothesis (1,3)
(6) $\overline{\Gamma, \overline{y_i{:}_{\Delta, \Delta'}\sigma_i}; \Delta, \Delta' \vdash e_i[e'/x] : \sigma_i}$ by induction hypothesis (1,4)
(7) $\Gamma[\Delta'/x]; \Delta, \Delta', \Delta'' \vdash \mathbf{let\ rec}\ \overline{y_i{:}_{\Delta, \Gamma_1'}\sigma_i = e_i[e'/x]}\ \mathbf{in}\ e''[e'/x] : \varphi$ by *LetRec*
(8) $(\mathbf{let\ rec}\ \overline{y_i{:}_{\Delta, \Delta'}\sigma_i = e_i}\ \mathbf{in}\ e'')[e'/x] = \mathbf{let\ rec}\ \overline{y_i{:}_{\Delta, \Delta'}\sigma_i = e_i[e'/x]}\ \mathbf{in}\ e''[e'/x]$
Subcase $x{:}_1\sigma$ does not occur in any e_i
(2) $\Gamma; \Delta, x{:}_1\sigma, \Delta'' \vdash \mathbf{let\ rec}\ \overline{y_i{:}_{\Delta}\sigma_i = e_i}\ \mathbf{in}\ e'' : \varphi$
(3) $\Gamma, \overline{y_i{:}_{\Delta}\sigma_i}; \Delta, x{:}_1\sigma, \Delta'' \vdash e'' : \varphi$ by inversion on *LetRec*
(4) $\overline{\Gamma, \overline{y_i{:}_{\Delta}\sigma_i}; \Delta \vdash e_i : \sigma_i}$ by inversion on *LetRec*
(5) $\Gamma[\Delta'/x], \overline{y_i{:}_{\Delta}\sigma_i}; \Delta, \Delta', \Delta'' \vdash e''[e'/x] : \varphi$ by i.h. (1,3)
(6) $\Gamma[\Delta'/x]; \Delta, \Delta', \Delta'' \vdash \mathbf{let\ rec}\ \overline{y_i{:}_{\Delta}\sigma_i = e_i}\ \mathbf{in}\ e''[e'/x] : \varphi$ by *LetRec*
(7) $\mathbf{let\ rec}\ \overline{y_i{:}_{\Delta}\sigma_i = e_i}\ \mathbf{in}\ e''[e'/x] = (\mathbf{let\ rec}\ \overline{y_i{:}_{\Delta}\sigma_i = e_i}\ \mathbf{in}\ e'')[e'/x]$ by subcase

Case: *CaseWHNF*

(1) $\Gamma; \Delta' \vdash e' : \sigma$
Subcase x occurs in e
(2) $\Gamma; \Delta, x{:}_1\sigma, \Delta'' \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_{\Delta, x:_1\sigma}\sigma'\ \{\overline{\rho \to e''}\} : \varphi$
(3) e is in WHNF
(4) $\Gamma; \Delta, x{:}_1\sigma \vdash e : \sigma'$
(5) $\overline{\Gamma, z{:}_{\Delta, x:_1\sigma}\sigma'; \Delta, x{:}_1\sigma, \Delta'' \vdash_{alt} \rho \to e'' :^{z}_{\Delta, x:_1\sigma} \sigma' \Longrightarrow \varphi}$
(6) $\Gamma\,[\Delta'/x]\,; \Delta, \Delta' \vdash e[e'/x] : \varphi$ by i.h.
(7) $\overline{\Gamma\,[\Delta'/x]\,, z{:}_{\Delta, \Delta'}\sigma'; \Delta, \Delta', \Delta'' \vdash_{alt} \rho \to e''[e'/x] :^{z}_{\Delta, \Delta'} \sigma' \Longrightarrow \varphi}$ by lin. subst. alts.
(8) $\Gamma\,[\Delta'/x]\,; \Delta, \Delta', \Delta'' \vdash \mathbf{case}\ e[e'/x]\ \mathbf{of}\ z{:}_{\Delta, \Delta'}\sigma'\ \{\overline{\rho \to e''[e'/x]}\} : \varphi$

Subcase x occurs in $\overline{e''}$
(2) $\Gamma; \Delta, \Delta'', x{:}_1\sigma \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_\Delta\sigma'\ \{\overline{\rho \to e''}\} : \varphi$
(3) e is in WHNF
(4) $\Gamma; \Delta \vdash e : \sigma'$
(5) $\overline{\Gamma, z{:}_\Delta\sigma'; \Delta, \Delta'', x{:}_1\sigma \vdash_{alt} \rho \to e'' :^z_\Delta \sigma' \Longrightarrow \varphi}$
(6) $e[e'/x] = e$ by x does not occur in e
(7) $\overline{\Gamma\,[\Delta'/x]\,, z{:}_\Delta\sigma'; \Delta, \Delta'', \Delta' \vdash_{alt} \rho \to e''[e'/x] :^z_\Delta \sigma' \Longrightarrow \varphi}$ by i.h.
(8) $\Gamma\,[\Delta'/x]\,; \Delta, \Delta'', \Delta' \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_\Delta\sigma'\ \{\overline{\rho \to e''[e'/x]}\} : \varphi$

Case: *CaseNotWHNF*

(1) $\Gamma; \Delta' \vdash e' : \sigma$
Subcase x occurs in e
(2) $\Gamma; \Delta, x{:}_1\sigma, \Delta'' \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_{[\Delta, x:_1\sigma]}\sigma'\ \{\overline{\rho \to e''}\} : \varphi$
(3) e is definitely not in WHNF
(4) $\Gamma; \Delta, x{:}_1\sigma \vdash e : \sigma'$ by inv.
(5) $\overline{\Gamma, z{:}_{[\Delta, x:_1\sigma]}\sigma'; [\Delta, x{:}_1\sigma]\,, \Delta'' \vdash_{alt} \rho \to e'' :^z_{[\Delta, x:_1\sigma]} \sigma' \Longrightarrow \varphi}$ by inv.
(6) $\Gamma\,[\Delta'/x]\,; \Delta, \Delta' \vdash e[e'/x] : \varphi$ by i.h.
(7) $\overline{\Gamma\,[\Delta'/x]\,, z{:}_{[\Delta,\Delta']}\sigma'; [\Delta, \Delta']\,, \Delta'' \vdash_{alt} \rho \to e''[e'/x] :^z_{[\Delta,\Delta']} \sigma' \Longrightarrow \varphi}$
by subst. of p. irr. vars in alt.
or, simply, congruence?
(x only occurs in ctxts, so replace all xs by Δ', starting by Γ)?
(8) $\Gamma\,[\Delta'/x]\,; \Delta, \Delta', \Delta'' \vdash \mathbf{case}\ e[e'/x]\ \mathbf{of}\ z{:}_{\Delta,\Delta'}\sigma'\ \{\overline{\rho \to e''[e'/x]}\} : \varphi$
Subcase x occurs in $\overline{e''}$
(2) $\Gamma; \Delta, \Delta'', x{:}_1\sigma \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_{[\Delta]}\sigma'\ \{\overline{\rho \to e''}\} : \varphi$
(3) e is definitely not in WHNF
(4) $\Gamma; \Delta \vdash e : \sigma'$ by inv.
(5) $\overline{\Gamma, z{:}_{[\Delta]}\sigma'; [\Delta]\,, \Delta'', x{:}_1\sigma \vdash_{alt} \rho \to e'' :^z_{[\Delta]} \sigma' \Longrightarrow \varphi}$ by inv.
(6) $e[e'/x] = e$ by x does not occur in e
(7) $\overline{\Gamma\,[\Delta'/x]\,, z{:}_{[\Delta]}\sigma'; [\Delta]\,, \Delta'', \Delta' \vdash_{alt} \rho \to e''[e'/x] :^z_{[\Delta]} \sigma' \Longrightarrow \varphi}$ by lin. subst. on alts
(8) $\Gamma\,[\Delta'/x]\,; \Delta, \Delta'', \Delta' \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_{[\Delta]}\sigma'\ \{\overline{\rho \to e''[e'/x]}\} : \varphi$

□

Lemma 1.1 (Substitution of linear variables on case alternatives preserves typing)**.** *If* $\Gamma; \Delta, x{:}_1\sigma \vdash_{alt} \rho \to e :^z_{\Delta_s} \sigma \Longrightarrow \varphi$ *and* $\Gamma; \Delta' \vdash e' : \sigma$ *and* $\Delta_s \subseteq \Delta, x$ *then* $\Gamma\,[\Delta'/x]\,; \Delta, \Delta' \vdash_{alt} \rho \to e[e'/x] :^z_{\Delta_s[\Delta'/x]} \sigma \Longrightarrow \varphi$

Proof. By structural induction on the *alt* judgment.

Case: *AltN*

(1) $\Gamma; \Delta' \vdash e' : \sigma$
(2) $\Gamma; \Delta, x{:}_1\sigma \vdash_{alt} K\ \overline{x{:}_\omega\sigma}, \overline{y_i{:}_1\sigma_i}^n\ \to e :^z_{\Delta_s} \sigma' \Longrightarrow \varphi$
(3) $n > 0$ by inv.
(4) $\overline{\Delta_i} = \overline{\Delta_s \# K_j}^n$ by inv.
(5) $\overline{\Delta_i \neq \cdot}$

(6) $\Gamma, \overline{x{:}_{\omega}\sigma}, \overline{y_i{:}_{\Delta_i}\sigma_i}; \Delta, x{:}_1\sigma \vdash e : \varphi$ by inv.
(7) $\Gamma\left[\Delta'/x\right], \overline{x{:}_{\omega}\sigma}, \overline{y_i{:}_{\Delta_i[\Delta'/x]}\sigma_i}; \Delta, \Delta' \vdash e[e'/x] : \varphi$ by i.h.
(8) $\overline{\Delta_i[\Delta'/x]} = \overline{\Delta_s[\Delta'/x]\#K_j}^{n}$ by (4) and cong.
(8) $\Gamma\left[\Delta'/x\right]; \Delta, \Delta' \vdash_{alt} \rho \to e[e'/x] :^{z}_{\Delta_s[\Delta'/x]} \sigma' \Longrightarrow \varphi$

Case: *Alt*0 This is one of the most interesting proof cases, and particularly hard to prove.

- The first insight is to divide the proof into two subcases, accounting for when the scrutinee (and hence Δ_s) contains the linear resource and when it does not.
- The second insight is to recall that Δ and Δ' are disjoint to be able to prove the subcase in which x does not occur in the scrutinee
- The third insight is to *create* linear resources seemingly out of nowhere *under a substitution that removes them.* We see this happen in the case where x occurs in the scrutinee, for both the linear and affine contexts (see (5,6)). We must also see that we can swap x for Δ' if neither can occur (see (7)).

(1) $\Gamma; \Delta' \vdash e' : \sigma$
Subcase x occurs in scrutinee
(2) $\Gamma; \Delta, x{:}_1\sigma \vdash_{alt} K\ \overline{x{:}_{\omega}\sigma}\ \to e :^{z}_{\Delta_s, x{:}_1\sigma} \sigma' \Longrightarrow \varphi$
(2.5) $\Gamma\left[\cdot/\Delta_s, x\right]_z, \overline{x{:}_{\omega}\sigma}; (\Delta, x{:}_1\sigma)\left[\cdot/\Delta_s, x\right] \vdash e : \varphi$ by inv.
(3) $\Gamma\left[\cdot/\Delta_s, x\right]_z, \overline{x{:}_{\omega}\sigma}; \Delta\left[\cdot/\Delta_s\right] \vdash e : \varphi$
(4) $e[e'/x] = e$ since x cannot occur in e (erased from cx)
(5) $\Delta[\cdot/\Delta_s] = (\Delta, \Delta')[\cdot/\Delta_s, \Delta']$ by cong. of subst.
(6) $\Gamma[\cdot/\Delta_s, x]_z[\Delta'/x] = \Gamma[\Delta'/x][\cdot/\Delta_s, \Delta']_z$ by cong. of subst. and more
(7) $\forall x, \Delta, \Delta', \Gamma : x \notin \Delta \wedge \Delta' \not\subset \Delta \wedge \Gamma; \Delta \vdash e : \sigma \Longrightarrow \Gamma[\Delta'/x]; \Delta \vdash e : \sigma$ by Weaken and variables in Γ cannot occur in e if they mention x nor if they mention Δ'
(8) $\Gamma\left[\Delta'/x\right]\left[\cdot/\Delta_s, \Delta'\right]_z, \overline{x{:}_{\omega}\sigma}; (\Delta, \Delta')\left[\cdot/\Delta_s, \Delta'\right] \vdash e[e'x] : \varphi$ by (4,5,6,7) and x and Δ' are erased from ctx
(9) $\Gamma\left[\Delta'/x\right]; \Delta, \Delta' \vdash_{alt} K\ \overline{x{:}_{\omega}\sigma}\ \to e[e'/x] :^{z}_{\Delta_s, \Delta'} \sigma' \Rightarrow \varphi$ by *Alt*0
Subcase x does not occur in scrutinee
(2) $\Gamma; \Delta, x{:}_1\sigma \vdash_{alt} K\ \overline{x{:}_{\omega}\sigma}\ \to e :^{z}_{\Delta_s} \sigma' \Longrightarrow \varphi$
(3) $\Gamma\left[\cdot/\Delta_s\right]_z, \overline{x{:}_{\omega}\sigma}; \Delta\left[\cdot/\Delta_s\right], x{:}_1\sigma \vdash e : \varphi$ by x does not occur in Δ_s and inv.
(4) $\Gamma\left[\Delta'/x\right]\left[\cdot/\Delta_s\right]_z, \overline{x{:}_{\omega}\sigma}; \Delta\left[\cdot/\Delta_s\right], \Delta' \vdash e[e'/x] : \varphi$ by i.h. and x does not occur in Δ_s
(5) $\Gamma\left[\Delta'/x\right]\left[\cdot/\Delta_s\right]_z, \overline{x{:}_{\omega}\sigma}; (\Delta, \Delta')\left[\cdot/\Delta_s\right] \vdash e[e'/x] : \varphi$ by Δ and Δ' being disjoint by hypothesis, and Δ_s being a subset of Δ
(6) $\Delta_s[\Delta'/x] = \Delta_s$ by x does not occur in Δ_s
(7) $\Gamma\left[\Delta'/x\right]; \Delta, \Delta' \vdash_{alt} K\ \overline{x{:}_{\omega}\sigma}\ \to e[e'/x] :^{z}_{\Delta_s[\Delta'/x]} \sigma' \Rightarrow \varphi$

Case: *Alt*_ (trivial induction)

(1) $\Gamma; \Delta' \vdash e' : \sigma$
(2) $\Gamma; \Delta, x{:}_1\sigma \vdash_{alt} \_\ \to e :^{z}_{\Delta_s} \sigma' \Longrightarrow \varphi$
(3) $\Gamma; \Delta, x{:}_1\sigma \vdash e : \varphi$
(4) $\Gamma\left[\Delta'/x\right]; \Delta, \Delta' \vdash e[e'/x] : \varphi$
(5) $\Gamma\left[\Delta'/x\right]; \Delta, \Delta' \vdash_{alt} \_\ \to e[e'/x] :^{z}_{\Delta_s, \Delta'} \sigma' \Rightarrow \varphi$

□

Lemma 2 (Substitution of unrestricted variables preserves typing)**.** If $\Gamma, x{:}_\omega\sigma; \Delta \vdash e : \varphi$ and $\Gamma; \cdot \vdash e' : \sigma$ then $\Gamma, \Delta \vdash e[e'/x] : \varphi$.

Proof. By structural induction on the first derivation.

Case: ΛI

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

Case: ΛE

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

Case: λI_1

(1) $\Gamma, x{:}_\omega\sigma; \Delta \vdash \lambda y{:}_1\sigma'.\ e : \sigma' \to_1 \varphi$	
(2) $\Gamma; \cdot \vdash e' : \sigma$	
(3) $\Gamma, x{:}_\omega\sigma; \Delta, y{:}_1\sigma' \vdash e : \varphi$	by inversion on λI_1
(4) $\Gamma; \Delta, y{:}_1\sigma' \vdash e[e'/x] : \varphi$	by induction hypothesis (2,3)
(5) $\Gamma; \Delta \vdash \lambda y{:}_1\sigma'.\ e[e'/x] : \sigma' \to_1 \varphi$	by λI_1
(6) $(\lambda y{:}_\pi\sigma'.\ e)[e'/x] = (\lambda y{:}_\pi\sigma'.\ e[e'/x])$	by def. of subst.

Case: λI_ω

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

Case: Var_ω

(1) $\Gamma, x :_\omega ; \cdot \sigma \vdash x : \sigma$
(2) $\Gamma; \cdot \vdash e' : \sigma$
(4) $x[e'/x] = e'$ by def. of substitution
(5) $\Gamma; \cdot \vdash e' : \sigma$ by (2)

Case: Var_ω

(1) $\Gamma, x :_\omega \sigma; \cdot \vdash y : \varphi$
(2) $\Gamma; \cdot \vdash e' : \sigma$
(3) $y[e'/x] = y$ by def. of substitution
(4) $\Gamma; \cdot \vdash y : \varphi$ by inversion on $Weaken_\omega$ (1)

Case: Var_1

(1) Impossible. The context in Var_1 is empty.

Case: Var_Δ

(1) Impossible. The context in Var_Δ only contains linear variables.

Case: *Split*

Trivial induction

Case: $\lambda E_{1,p}$

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

Case: λE_ω

(1) $\Gamma, x :_\omega \sigma; \Delta \vdash e\ e'' : \varphi$
(2) $\Gamma; \cdot \vdash e' : \sigma$
(3) $\Gamma, x :_\omega \sigma; \Delta \vdash e : \sigma' \to_\omega \varphi$ by inversion on λE_ω
(4) $\Gamma, x :_\omega \sigma; \cdot \vdash e'' : \sigma'$ by inversion on λE_ω
(5) $\Gamma; \Delta \vdash e[e'/x] : \sigma' \to_1 \varphi$ by induction hypothesis (2,3)
(6) $\Gamma; \cdot \vdash e''[e'/x] : \sigma'$ by induction hypothesis (2,4)
(7) $\Gamma; \Delta \vdash e[e'/x]\ e''[e'/x] : \varphi$ by λE_ω (5,6)
(8) $(e\ e'')[e'/x] = (e[e'/x]\ e''[e'/x])$ by def. of subst.

Case: *Let*

(1) $\Gamma, x{:}_{\omega}\sigma; \Delta, \Delta' \vdash \mathbf{let}\ y{:}_{\Delta}\sigma' = e\ \mathbf{in}\ e'' : \varphi$
(2) $\Gamma; \cdot \vdash e' : \sigma$
(3) $\Gamma, x{:}_{\omega}\sigma, y{:}_{\Delta}\sigma'; \Delta, \Delta' \vdash e''\varphi$ by inversion on *Let*
(4) $\Gamma, x{:}_{\omega}\sigma; \Delta \vdash e : \sigma'$ by inversion on *Let*
(5) $\Gamma, y{:}_{\Delta}\sigma'; \Delta \vdash e''[e'/x] : \varphi$ by induction hypothesis (2,3)
(6) $\Gamma; \Delta \vdash e[e'/x] : \sigma'$ by induction hypothesis (2,4)
(7) $\Gamma; \Delta, \Delta' \vdash \mathbf{let}\ y{:}_{\Delta}\sigma' = e[e'/x]\ \mathbf{in}\ e''[e'/x]$ by *Let* (5,6)
(8) $(\mathbf{let}\ y{:}_{\Delta}\sigma' = e\ \mathbf{in}\ e'')[e'/x] = (\mathbf{let}\ y{:}_{\Delta}\sigma' = e[e'/x]\ \mathbf{in}\ e''[e'/x])$

Case: *LetRec*

(1) $\Gamma, x{:}_{\omega}\sigma; \Delta, \Delta' \vdash \mathbf{let\ rec}\ \overline{y{:}_{\Delta}\sigma' = e}\ \mathbf{in}\ e'' : \varphi$
(2) $\Gamma'; \cdot \vdash e' : \sigma$
(3) $\Gamma, x{:}_{\omega}\sigma, \overline{y{:}_{\Delta}\sigma'}; \Delta, \Delta' \vdash e'' : \varphi$ by inversion on *LetRec*
(4) $\overline{\Gamma, x{:}_{\omega}\sigma, y{:}_{\Delta}\sigma'}; \Delta \vdash e : \sigma'$ by inversion on *LetRec*
(5) $\overline{\Gamma, y{:}_{\Delta}\sigma'}; \Delta, \Delta' \vdash e''[e'/x] : \varphi$ by induction hypothesis (2,3)
(6) $\overline{\Gamma, y{:}_{\Delta}\sigma'}; \Delta \vdash e[e'/x] : \sigma'$ by induction hypothesis (2,4)
(7) $\Gamma; \Delta, \Delta' \vdash \mathbf{let\ rec}\ \overline{y{:}_{\Delta}\sigma' = e[e'/x]}\ \mathbf{in}\ e''[e'/x] : \varphi$ by *LetRec* (5,6)
(8) $(\mathbf{let\ rec}\ \overline{y{:}_{\Delta}\sigma' = e}\ \mathbf{in}\ e'')[e'/x] = (\mathbf{let\ rec}\ \overline{y{:}_{\Delta}\sigma' = e[e'/x]}\ \mathbf{in}\ e''[e'/x])$

Case: *CaseWHNF*

(1) $\Gamma; \cdot \vdash e' : \sigma$
(2) $\Gamma, x{:}_{\omega}\sigma; \Delta, \Delta' \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_{\Delta}\sigma'\ \{\overline{\rho \to e''}\} : \varphi$
(3) $\Gamma, x{:}_{\omega}\sigma; \Delta \vdash e : \sigma$
(4) $\Gamma; \Delta \vdash e[e'/x] : \sigma'$ by i.h.
(5) e is in WHNF
(6) $\overline{\Gamma, x{:}_{\omega}\sigma, z{:}_{\Delta}\sigma'; \Delta, \Delta' \vdash \rho \to e'' :_{\Delta} \sigma' \Rightarrow \varphi}$
(7) $\overline{\Gamma, z{:}_{\Delta}\sigma'; \Delta, \Delta' \vdash \rho \to e''[e'/x] :_{\Delta} \sigma' \Rightarrow \varphi}$ by unr. subst. on alts lemma
(8) $\Gamma; \Delta, \Delta' \vdash \mathbf{case}\ e[e'/x]\ \mathbf{of}\ z{:}_{\Delta}\sigma'\ \{\overline{\rho \to e''[e'/x]}\} : \varphi$

Case: *CaseNotWHNF*

(1) $\Gamma; \cdot \vdash e' : \sigma$
(2) $\Gamma, x{:}_{\omega}\sigma; \Delta, \Delta' \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_{[\Delta]}\sigma'\ \{\overline{\rho \to e''}\} : \varphi$
(3) $\Gamma, x{:}_{\omega}\sigma; \Delta \vdash e : \sigma$
(4) $\Gamma; \Delta \vdash e[e'/x] : \sigma'$ by i.h.
(5) e is definitely not in WHNF
(6) $\overline{\Gamma, x{:}_{\omega}\sigma, z{:}_{[\Delta]}\sigma'; [\Delta]\,, \Delta' \vdash \rho \to e'' :_{[\Delta]} \sigma' \Rightarrow \varphi}$
(6) $\overline{\Gamma, z{:}_{[\Delta]}\sigma'; [\Delta]\,, \Delta' \vdash \rho \to e''[e'/x] :_{[\Delta]} \sigma' \Rightarrow \varphi}$ by unr. subst. on alts lemma
(8) $\Gamma; \Delta, \Delta' \vdash \mathbf{case}\ e[e'/x]\ \mathbf{of}\ z{:}_{[\Delta]}\sigma'\ \{\overline{\rho \to e''[e'/x]}\} : \varphi$

□

Lemma 2.1 (Substitution of unrestricted variables on case alternatives preserves typing)**.** *If* $\Gamma, x{:}_\omega\sigma; \Delta \vdash_{alt} \rho \to e :^z_{\Delta_s} \sigma' \Rightarrow \varphi$ *and* $\Gamma; \Delta \vdash e' : \sigma$ *and then* $\Gamma; \Delta \vdash_{alt} \rho \to e[e'/x] :^z_{\Delta_s} \sigma' \Rightarrow \varphi$

Proof. By structural induction on the *alt* judgment.

Case: *AltN* (trivial induction)

(1) $\Gamma; \cdot \vdash e : \sigma$
(2) $\Gamma, x{:}_\omega\sigma; \Delta \vdash_{alt} K\ \overline{x{:}_\omega\sigma}, \overline{y_i{:}_1\sigma_i}^n \to e :^z_{\Delta_s} \sigma' \Rightarrow \varphi$
(3) $\Gamma, x{:}_\omega\sigma, \overline{x{:}_\omega\sigma}, \overline{y_i{:}_{\Delta_i}\sigma_i}; \Delta \vdash e : \varphi$
(4) $\Gamma, \overline{x{:}_\omega\sigma}, \overline{y_i{:}_{\Delta_i}\sigma_i}; \Delta \vdash e[e'/x] : \varphi$ by i.h.
(5) $\Gamma; \Delta \vdash_{alt} K\ \overline{x{:}_\omega\sigma}, \overline{y_i{:}_1\sigma_i}^n \to e[e'/x] :^z_{\Delta_s} \sigma' \Rightarrow \varphi$ by *AltN*

Case: *Alt*0

(1) $\Gamma; \cdot \vdash e' : \sigma$
(2) $\Gamma, x{:}_\omega\sigma; \Delta \vdash_{alt} K\ \overline{x{:}_\omega\sigma} \to e :^z_{\Delta_s} \sigma' \Rightarrow \varphi$
(3) $\Gamma[\cdot/\Delta_s]_z, x{:}_\omega\sigma, \overline{x{:}_\omega\sigma}; \Delta[\cdot/\Delta_s] \vdash e : \varphi$ by inv.
(4) $\Gamma[\cdot/\Delta_s]_z, \overline{x{:}_\omega\sigma}; \Delta[\cdot/\Delta_s] \vdash e[e'/x] : \varphi$ by i.h.
(5) $\Gamma; \Delta \vdash_{alt} K\ \overline{x{:}_\omega\sigma} \to e :^z_{\Delta_s} \sigma' \Rightarrow \varphi$ by *Alt*0

Case: $Alt_\_$ (trivial induction)

(1) $\Gamma; \cdot \vdash e : \sigma$
(2) $\Gamma, x{:}_\omega\sigma; \Delta \vdash_{alt} \_ \to e :_{\Delta_s} \sigma' \Rightarrow \varphi$
(3) $\Gamma, x{:}_\omega\sigma; \Delta \vdash e : \varphi$
(4) $\Gamma; \Delta \vdash e[e'/x] : \varphi$ by i.h.
(5) $\Gamma; \Delta \vdash_{alt} \_ \to e[e'/x] :^z_{\Delta_s} \sigma' \Rightarrow \varphi$ by $Alt_\_$

□

Lemma 3 (Substitution of variables with usage environments preserves typing)**.** *If* $\Gamma, x{:}_\Delta\sigma; \Delta, \Delta' \vdash e : \varphi$ *and* $\Gamma; \Delta \vdash e' : \sigma$ *then* $\Gamma; \Delta, \Delta' \vdash e[e'/x] : \varphi$

Proof. By structural induction on the first derivation.

Case: ΛI

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

Case: ΛE

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

Case: λI_1

(1) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash \lambda y{:}_1\sigma'.\ e : \sigma' \to_1 \varphi$
(2) $\Gamma; \Delta \vdash e' : \sigma$
(3) $\Gamma, x{:}_{\Delta}\sigma; \Delta, y{:}_1\sigma', \Delta' \vdash e : \varphi$ by inversion on λI
(4) $\Gamma; \Delta, y{:}_1\sigma', \Delta' \vdash e[e'/x] : \varphi$ by induction hypothesis (2,3)
(5) $\Gamma; \Delta, \Delta' \vdash \lambda y{:}_1\sigma'.\ e[e'/x] : \sigma' \to_1 \varphi$ by λI
(6) $(\lambda y{:}_1\sigma'.\ e[e'/x]) = (\lambda y{:}_1\sigma'.\ e)[e'/x]$ by def. of subst.
(7) $\Gamma; \Delta, \Delta' \vdash \lambda(y{:}_1\sigma'.\ e)[e'/x] : \sigma' \to_1 \varphi$ by (4,5)

Case: λI_ω

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

Case: Var_ω

(1) $\Gamma, y{:}_\omega\sigma', x{:}_{\cdot}\sigma; \cdot \vdash y : \sigma'$
(2) $\Gamma; \cdot \vdash e' : \sigma$
(3) $y[e'/x] = y$
(4) $\Gamma, y{:}_\omega\sigma'; \cdot \vdash y : \sigma'$ by (1) and $Weaken_\Delta$

Case: Var_1

(1) $\Gamma, x{:}_{y{:}_1\sigma}\sigma; y{:}_1\sigma \vdash y : \sigma$
(2) $\Gamma; y{:}_1\sigma \vdash e' : \sigma$
(3) $y[e'/x] = y$
(4) $\Gamma, x{:}_{y{:}_1\sigma}\sigma; y{:}_1\sigma \vdash y : \sigma$ by 1
(5) $\Gamma; y{:}_1\sigma \vdash y : \sigma$ by $Weaken_\Delta$

Case: Var_Δ

(1) $\Gamma, x{:}_{\Delta}\sigma; \Delta \vdash y : \sigma$
(2) $\Gamma'; \Delta \vdash e' : \sigma$
(3) $x[e'/x] = e'$
(4) $\Gamma'; \Delta \vdash e' : \sigma$ by (2)

Case: *Split*

Trivial induction

Case: $\lambda E_{1,p}$

(1) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta', \Delta'' \vdash e\ e'' : \varphi$
(2) $\Gamma; \Delta \vdash e' : \sigma$
Subcase Δ occurs in e
(3) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash e : \sigma' \to_{1,p} \varphi$
(4) $\Gamma, x{:}_{\Delta}\sigma; \Delta'' \vdash e'' : \sigma'$
(5) $\Gamma; \Delta'' \vdash e'' : \sigma'$ by $Weaken_{\Delta}$
(6) $\Gamma; \Delta, \Delta' \vdash e[e'/x] : \sigma' \to_{1,p} \varphi$ by i.h.
(7) $\Gamma; \Delta, \Delta', \Delta'' \vdash e[e'/x]\ e'' : \varphi$ by $(\lambda E_{1,p})$
(8) $(e[e'/x]\ e'') = (e\ e'')[e'/x]$ since x cannot occur in e''
Subcase Δ occurs in e''
(3) $\Gamma, x{:}_{\Delta}\sigma; \Delta' \vdash e : \sigma' \to_{1,p} \varphi$
(4) $\Gamma; \Delta' \vdash e : \sigma' \to_{1,p} \varphi$ by $Weaken_{\Delta}$
(5) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta'' \vdash e'' : \sigma'$
(6) $\Gamma; \Delta, \Delta'' \vdash e''[e'/x] : \sigma'$ by i.h.
(7) $\Gamma; \Delta, \Delta', \Delta'' \vdash (e\ e''[e'/x]) : \varphi$ by $(\lambda E_{1,p})$
(8) $e\ e''[e'/x] = (e\ e'')[e'/x]$ since x does not occur in e
Subcase Δ is split between e and e''
x cannot occur in either, so the substitution is trivial, and x can be weakened.

Case: λE_{ω}

(1) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash e\ e'' : \varphi$
(2) $\Gamma; \Delta \vdash e' : \sigma$
(3) Δ cannot occur in e''
(4) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash e : \sigma' \to_{\omega} \varphi$ by inv. on λE_{ω}
(5) $\Gamma; \cdot \vdash e'' : \sigma'$ by inv. on λE_{ω}
(6) $\Gamma; \Delta, \Delta' \vdash e[e'/x] : \sigma' \to_{\omega} \varphi$ by induction hypothesis (2,4)
(7) $\Gamma; \Delta, \Delta' \vdash e[e'/x]\ e'' : \varphi$ by λE_{ω} (5,6)
(8) $e[e'/x]\ e'' = (e\ e'')[e'/x]$ x does not occur in e'' by (3)

Case: *Let*

(1) $\Gamma; \Delta \vdash e' : \sigma$
Subcase Δ occurs in e
(2) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta', \Delta'' \vdash \mathbf{let}\ y{:}_{\Delta,\Delta'}\sigma' = e\ \mathbf{in}\ e'' : \varphi$
(3) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash e : \sigma'$ by inversion on (let)
(4) $\Gamma, x{:}_{\Delta}\sigma, y{:}_{\Delta,\Delta'}\sigma'; \Delta, \Delta', \Delta'' \vdash e'' : \varphi$ by inversion on (let)

(5) $\Gamma, y{:}_{\Delta,\Delta'}\sigma'; \Delta, \Delta', \Delta'' \vdash e'' : \varphi$ by $Weaken_\Delta$ (admissible)
(6) $\Gamma; \Delta, \Delta' \vdash e[e'/x] : \sigma'$ by induction hypothesis (1,3)
(7) $\Gamma; \Delta, \Delta', \Delta'' \vdash \mathbf{let}\ y{:}_{\Delta,\Delta'}\sigma' = e[e'/x]\ \mathbf{in}\ e'' : \varphi$ by (let) (5,6)
(8) $\mathbf{let}\ y{:}_{\Delta,\Delta'}\sigma' = e[e'/x]\ \mathbf{in}\ e'' = (\mathbf{let}\ y{:}_{\Delta,\Delta'}\sigma' = e\ \mathbf{in}\ e'')[e'/x]$ since x cannot occur in e''

Subcase Δ occurs in e''
(2) $\Gamma, x{:}_\Delta\sigma; \Delta, \Delta', \Delta'' \vdash \mathbf{let}\ y{:}_{\Delta'}\sigma' = e\ \mathbf{in}\ e'' : \varphi$
(3) $\Gamma, x{:}_\Delta\sigma; \Delta' \vdash e : \sigma'$ by inversion on (let)
(4) $\Gamma; \Delta' \vdash e : \sigma'$ by $Weaken_\Delta$
(5) $\Gamma, x{:}_\Delta\sigma, y{:}_{\Delta'}\sigma'; \Delta, \Delta', \Delta'' \vdash e'' : \varphi$ by inversion on (let)
(6) $\Gamma, y{:}_{\Delta'}\sigma'; \Delta, \Delta', \Delta'' \vdash e''[e'/x] : \varphi$ by i.h. (1,5)
(7) $\Gamma; \Delta, \Delta', \Delta'' \vdash \mathbf{let}\ y{:}_{\Delta'}\sigma' = e\ \mathbf{in}\ e''[e'/x] : \varphi$ by (let)
(8) $\mathbf{let}\ y{:}_{\Delta'}\sigma' = e\ \mathbf{in}\ e''[e'/x] = (\mathbf{let}\ y{:}_{\Delta'}\sigma' = e\ \mathbf{in}\ e'')[e'/x]$ since x cannot occur in e

Subcase Δ is split between e and e''
x cannot occur in either, so the substitution is trivial, and x can be weakened.

Case: LetRec

(1) $\Gamma; \Delta \vdash e' : \sigma$
Subcase Δ occurs in $\overline{e_i}$
(2) $\Gamma, x{:}_\Delta\sigma; \Delta, \Delta', \Delta'' \vdash \mathbf{let\ rec}\ \overline{y_i{:}_{\Delta,\Delta'}\sigma'_i = e_i}\ \mathbf{in}\ e'' : \varphi$
(3) $\Gamma, x{:}_\Delta\sigma, \overline{y_i{:}_{\Delta,\Delta'}\sigma'_i}; \Delta, \Delta', \Delta'' \vdash e'' : \varphi$ by inversion on (let)
(4) $\Gamma, \overline{y_i{:}_{\Delta,\Delta'}\sigma'_i}; \Delta, \Delta', \Delta'' \vdash e'' : \varphi$ by $Weaken_\Delta$
(5) $\Gamma, x{:}_\Delta\sigma, \overline{y_i{:}_{\Delta,\Delta'}\sigma'_i}; \Delta, \Delta' \vdash e_i : \sigma'_i$ by inversion on (let)
(6) $\Gamma, \overline{y_i{:}_{\Delta,\Delta'}\sigma'_i}; \Delta, \Delta' \vdash e_i[e'/x] : \sigma'_i$ by induction hypothesis (1,5)
(7) $e''[e'/x] = e''$ since x cannot occur in e''
(8) $\Gamma; \Delta, \Delta', \Delta'' \vdash \mathbf{let\ rec}\ \overline{y_i{:}_{\Delta,\Delta'}\sigma'_i = e_i[e'/x]}\ \mathbf{in}\ e'' : \varphi$ by (let) (4,6)
Subcase Δ occurs in e''
(2) $\Gamma, x{:}_\Delta\sigma; \Delta, \Delta', \Delta'' \vdash \mathbf{let\ rec}\ \overline{y_i{:}_{\Delta'}\sigma'_i = e_i}\ \mathbf{in}\ e'' : \varphi$
(3) $\Gamma, x{:}_\Delta\sigma, \overline{y_i{:}_{\Delta'}\sigma'_i}; \Delta' \vdash e_i : \sigma'_i$ by inversion on (let)
(4) $\Gamma, \overline{y_i{:}_{\Delta'}\sigma'_i}; \Delta' \vdash e_i : \sigma'_i$ by $Weaken_\Delta$
(6) $\Gamma, x{:}_\Delta\sigma, \overline{y_i{:}_{\Delta'}\sigma'_i}; \Delta, \Delta', \Delta'' \vdash e'' : \varphi$ by inversion on (let)
(7) $\Gamma, \overline{y_i{:}_{\Delta'}\sigma'_i}; \Delta, \Delta', \Delta'' \vdash e''[e'/x] : \varphi$ by i.h. (1,6)
(8) $\overline{e_i[e'/x] = e_i}$ since x cannot occur in $\overline{e_i}$
(9) $\Gamma; \Delta, \Delta', \Delta'' \vdash \mathbf{let\ rec}\ \overline{y_i{:}_{\Delta'}\sigma'_i = e_i}\ \mathbf{in}\ e''[e'/x] : \varphi$ by (let)
Subcase Δ is split between e and e''
x cannot occur in either, so the substitution is trivial, and x can be weakened.

Case: CaseWHNF

(1) $\Gamma; \Delta \vdash e' : \sigma$
Subcase Δ occurs in e
(2) $\Gamma, x{:}_\Delta\sigma; \Delta, \Delta', \Delta'' \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_{\Delta,\Delta'}\sigma'\ \{\overline{\rho \to e''}\} : \varphi$
(3) e is in WHNF
(4) $\Gamma, x{:}_\Delta\sigma; \Delta, \Delta' \vdash e : \sigma'$
(5) $\overline{\Gamma, x{:}_\Delta\sigma, z{:}_{\Delta,\Delta'}\sigma'; \Delta, \Delta', \Delta'' \vdash_{alt} \rho \to e'' :^z_{\Delta,\Delta'} \sigma' \Rightarrow \varphi}$
(6) $\Gamma; \Delta, \Delta' \vdash e[e'/x] : \sigma'$ by i.h.

(7) $\overline{\Gamma, z{:}_{\Delta,\Delta'}\sigma'; \Delta, \Delta', \Delta'' \vdash_{alt} \rho \to e''[e'/x] :^{z}_{\Delta,\Delta'} \sigma' \Rightarrow \varphi}$by subst. of Δ vars on case alts
(8) $\Gamma; \Delta, \Delta', \Delta'' \vdash \mathbf{case}\ e[e'/x]\ \mathbf{of}\ z{:}_{\Delta,\Delta'}\sigma'\ \{\overline{\rho \to e''[e'/x]}\} : \varphi$ by $CaseWHNF$
Subcase Δ does not occurs in e
(2) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta', \Delta'' \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_{\Delta'}\sigma'\ \{\overline{\rho \to e''}\} : \varphi$
(3) e is in WHNF
(4) $\Gamma, x{:}_{\Delta}\sigma; \Delta' \vdash e : \sigma'$
(5) $\Gamma; \Delta' \vdash e : \sigma'$ by (admissible) $Weaken_{\Delta}$
(6) $e[e'/x] = e$ by x cannot occur in e
(7) $\overline{\Gamma, x{:}_{\Delta}\sigma, z{:}_{\Delta'}\sigma'; \Delta, \Delta', \Delta'' \vdash_{alt} \rho \to e'' :^{z}_{\Delta'}: \sigma' \Rightarrow \varphi}$
(8) $\overline{\Gamma, z{:}_{\Delta'}\sigma'; \Delta, \Delta', \Delta'' \vdash_{alt} \rho \to e''[e'/x] :^{z}_{\Delta'} \sigma' \Rightarrow \varphi}$ by subst. of Δ vars on case alts
(9) $\Gamma; \Delta, \Delta', \Delta'' \vdash \mathbf{case}\ e[e'/x]\ \mathbf{of}\ z{:}_{\Delta'}\sigma'\ \{\overline{\rho \to e''[e'/x]}\} : \varphi$ by $CaseWHNF$
Subcase Δ is partially used in e
This is like the subcase above, but consider Δ'
to contain some of part of Δ and Δ to refer to the other part only.

Case: CaseNotWHNF

(1) $\Gamma; \Delta \vdash e' : \sigma$
Subcase Δ occurs in e
(2) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta', \Delta'' \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_{[\Delta,\Delta']}\sigma'\ \{\overline{\rho \to e''}\}$
(3) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash e : \sigma'$ by inv.
(4) $\Gamma; \Delta, \Delta' \vdash e[e'/x] : \sigma'$ by i.h.
(5) $\overline{\Gamma, x{:}_{\Delta}\sigma, z{:}_{[\Delta,\Delta']}\sigma'; [\Delta, \Delta'], \Delta'' \vdash_{alt} \rho \to e'' :^{z}_{[\Delta,\Delta']} \sigma' \Rightarrow \varphi}$ by inv.
(6) $e''[e'/x] = e$ by x cannot occur in $\overline{e''}$ since Δ is not available (only $[\Delta]$)
(7) $\overline{\Gamma, z{:}_{[\Delta,\Delta']}\sigma'; [\Delta, \Delta'], \Delta'' \vdash_{alt} \rho \to e'' :^{z}_{[\Delta,\Delta']} \sigma' \Rightarrow \varphi}$ by (admissible) $Weaken_{\Delta}$
(8) $\Gamma; \Delta, \Delta', \Delta'' \vdash \mathbf{case}\ e[e'/x]\ \mathbf{of}\ z{:}_{[\Delta,\Delta']}\sigma'\ \{\overline{\rho \to e''}\}$ by $CaseNotWHNF$
Subcase Δ does not occur in e
(2) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta', \Delta'' \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_{[\Delta']}\sigma'\ \{\overline{\rho \to e''}\}$
(3) $\Gamma, x{:}_{\Delta}\sigma; \Delta' \vdash e : \sigma'$ by inv.
(4) $\Gamma; \Delta' \vdash e : \sigma'$ by weaken
(5) $e[e'/x] = e$ by x cannot occur in e
(5) $\overline{\Gamma, x{:}_{\Delta}\sigma, z{:}_{[\Delta']}\sigma'; \Delta, [\Delta'], \Delta'' \vdash_{alt} \rho \to e'' :^{z}_{[\Delta']} \sigma' \Rightarrow \varphi}$ by inv.
(7) $\overline{\Gamma, z{:}_{[\Delta']}\sigma'; \Delta, [\Delta'], \Delta'' \vdash_{alt} \rho \to e''[e'/x] :^{z}_{[\Delta']} \sigma' \Rightarrow \varphi}$
by subst. of Δ-vars in case alternatives
(8) $\Gamma; \Delta, \Delta', \Delta'' \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_{[\Delta,\Delta']}\sigma'\ \{\overline{\rho \to e''[e'/x]}\}$ by $CaseNotWHNF$
Subcase Δ is partially used in e
This is like the subcase above, but consider Δ'
to contain some of part of Δ and Δ to refer to the other part only.

□

Lemma 3.1 (Substitution of Δ-bound variables on case alternatives preserves typing)**.** *If* $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash_{alt} \rho \to e :^{z}_{\Delta_s} \sigma' \Rightarrow \varphi$ *and* $\Gamma; \Delta \vdash e' : \sigma$ *and* $\Delta_s \subseteq (\Delta, \Delta')$ *then* $\Gamma; \Delta, \Delta' \vdash_{alt} \rho \to e[e'/x] :^{z}_{\Delta_s} \sigma' \Rightarrow \varphi$

Proof. By structural induction on the *alt* judgment.

Case: *AltN* (trivial induction)

(1) $\Gamma; \Delta \vdash e' : \sigma$
(2) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash_{alt} K\ \overline{x{:}_{\omega}\sigma}, \overline{y_i{:}_1\sigma_i}^n\ \to e :^z_{\Delta_s} \sigma' \Rightarrow \varphi$
(3) $n > 0$ by inv.
(4) $\overline{\Delta_i} = \overline{\Delta_s \# K_j}$ by inv.
(5) $\Gamma, x{:}_{\Delta}\sigma, \overline{x{:}_{\omega}\sigma}, \overline{y_i{:}_{\Delta_i}\sigma_i}; \Delta, \Delta' \vdash e : \varphi$ by inv.
(6) $\Gamma, \overline{x{:}_{\omega}\sigma}, \overline{y_i{:}_{\Delta_i}\sigma_i}; \Delta, \Delta' \vdash e[e'/x] : \varphi$ by i.h.
(7) $\Gamma; \Delta, \Delta' \vdash_{alt} K\ \overline{x{:}_{\omega}\sigma}, \overline{y_i{:}_1\sigma_i}^n\ \to e[e'/x] :^z_{\Delta_s} \sigma' \Rightarrow \varphi$ by *AltN*

Case: *Alt*0

(1) $\Gamma; \Delta \vdash e' : \sigma$
Subcase Δ occurs in scrutinee
(2) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta', \Delta'' \vdash_{alt} K\ \overline{x{:}_1\sigma}\ \to e :^z_{\Delta,\Delta'} \sigma' \Rightarrow \varphi$
(3) $(\Gamma, x{:}_{\Delta}\sigma)[\cdot/\Delta, \Delta']_z; (\Delta, \Delta', \Delta'')[\cdot/\Delta, \Delta'] \vdash e : \varphi$
(4) $\Gamma[\cdot/\Delta, \Delta']_z, x{:}_{\Delta}\sigma; \Delta'' \vdash e : \varphi$
by def. of $[]_z$ subst. and [] subst.
(5) $\Gamma[\cdot/\Delta, \Delta']_z, x{:}_{\Delta}\sigma; \Delta'' \vdash e[e'/x] : \varphi$
by x cannot occur in e by Δ not available
(6) $\Gamma[\cdot/\Delta, \Delta']_z; \Delta'' \vdash e[e'/x] : \varphi$
by (admissible) $Weaken_{\Delta}$
(7) $\cdot = (\Delta, \Delta')[\cdot/\Delta, \Delta']$
(8) $\Gamma[\cdot/\Delta, \Delta']_z; (\Delta, \Delta', \Delta'')[\cdot/\Delta, \Delta'] \vdash e[e'/x] : \varphi$
by (6,7)
(9) $\Gamma; \Delta, \Delta', \Delta'' \vdash_{alt} e[e'/x] :^z_{\Delta,\Delta'} \sigma' \Rightarrow \varphi$
Subcase Δ does not occur in the scrutinee
(2) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta', \Delta'' \vdash_{alt} K\ \overline{x{:}_1\sigma}\ \to e :^z_{\Delta'} \sigma' \Rightarrow \varphi$
(3) $(\Gamma, x{:}_{\Delta}\sigma)[\cdot/\Delta']_z; (\Delta, \Delta', \Delta'')[\cdot/\Delta'] \vdash e : \varphi$
(4) $\Gamma[\cdot/\Delta']_z, x{:}_{\Delta}\sigma; \Delta, \Delta'' \vdash e : \varphi$
by def. of $[]_z$ subst. and [] subst.
(5) $\Gamma[\cdot/\Delta']_z; \Delta, \Delta'' \vdash e[e'/x] : \varphi$ by Δ-subst. lemma
(6) $\cdot = \Delta'[\cdot/\Delta']$
(7) $\Gamma[\cdot/\Delta']_z; (\Delta, \Delta', \Delta'')[\cdot/\Delta'] \vdash e[e'/x] : \varphi$ by (5,6)
(8) $\Gamma; \Delta, \Delta', \Delta'' \vdash_{alt} \vdash e[e'/x] :^z_{\Delta'} \sigma' \Rightarrow \varphi$ by *Alt*0
Subcase Δ is partially used in the scrutinee
This is like the subcase above, but consider Δ'
to contain some of part of Δ and Δ to refer to the other part only.

Case: *Alt*$_{\_}$ (trivial induction)

(1) $\Gamma; \Delta \vdash e' : \sigma$
(2) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash_{alt} \_\ \to e :^z_{\Delta_s} \sigma' \Rightarrow \varphi$
(3) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash e : \varphi$ by inv.
(4) $\Gamma; \Delta, \Delta' \vdash e[e'/x]] : \varphi$ by i.h.
(5) $\Gamma; \Delta, \Delta' \vdash_{alt} \_\ \to e[e'/x] :^z_{\Delta_s} \sigma' \Rightarrow \varphi$ by *Alt*$_{\_}$

□

TODO! Substitution of proof-irrelevant linear variables preserves typing. The term always remains the same because x cannot occur in any term, however, all variables that refer to x in their usage environment must now refer the usage env. of the substitee (e.g. $[x] => [\Delta]$). This seems trivial to see correct, since all occurrences are in environments, so we get some equivalence similar to the one we need for the proof of Alt0.

Lemma 4 (Substitution of proof-irrelevant linear variables preserves typing)**.** *If* $\Gamma; \Delta, [x{:}_1\sigma] \vdash [: \delta] e\varphi$ *and* $\Gamma; \Delta' \vdash [: \delta'] e'\sigma$ *then* $\Gamma; \Delta, [\Delta'] \vdash [: \delta\,[\Delta'/x]\,, \delta'] e\varphi$

TODO: Multiplicity substitution preserves typing lemma
TODO: Canonical forms lemma
TODO: Corollary of Δ-var subst. for $\overline{\Delta}$
TODO: Constructor app typing: If $\Gamma, \Delta \vdash K\ \overline{e}$ and $K{:}\overline{\sigma \to \pi}\ T\ \overline{p} \in \Gamma$ and Γ has no linear vars then $\overline{\Gamma, \Delta_i \vdash e_i : \sigma_i}$

3.3.2 Core-to-Core optimisations preserve linearity

We proved multiple optimizing transformations preserve linearity

Inlining

To the best of our knowledge, there is no linear type system for which inlining preserves linearity[5]

Reverse Binder Swap Considered Harmful

(Link to ticket)

(1) All optimisations preserve (semantic) linearity
(2) If a function is (semantically) linear, then we can evaluate it using call-by-name and preserve linearity
(3) Reverse binder swap is an optimisation
(4) If reverse binder swap is applied to a case scrutinizing a linear resource in the body (‘e‘) of a linear fun
(5) If we evaluate ‘f‘, we do it call-by-name because of (2)
(6) Call-by-name substitution of the linear argument in the body of a function has been reverse binder swa
(7) Contradiction: By 3 and 1, ‘f‘ is linear after reverse-binder-swap. By 2, we can substitute arguments to

Conclusion: Either we need to forfeit that we can always substitute call-by-name linear function applications, or we forfeit that reverse binder swap preserves linearity (instead, it preserves a weaker notion of linearity understood by the syntatic-occurrence-analyzer)

Reverse-binder-swap is only well-defined in certain scenarios where the optimizations don’t apply call-by-name beta-reduction after the reverse-binder-swap optimization – otherwise we would duplicate resources. In this case, it is not a matter of syntatic vs semantic linearity

[5] https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0111-linear-types.rst#id90

On the reverse binder swap, mention From Call-by-name, call-by-value, call-by-need and the linear lambda calculus: The call-by-name calculus is not entirely suitable for reasoning about functional programs in lazy languages, because the beta rule may copy the argument of a function any number of times. The call-by-need calculus uses a diferent notion of reduction, observationally equivalent to the call-by-name calculus. But call-by-need, like call-by-value, guarantees that the argument to a function is not copied before it is reduced to a value.

It's also interesting to note that reverse-binder-swap preserves linearity under pure call-by-need but not under call-by-name, because (In the sense that if EVEN linear functions reduce call-by-need rather than call-by-name, then it would preserve optimisations)

Reduce call-by-name linear function application

$$
\begin{aligned}
& f\ y = (\lambda x \circ \mathbf{case}\ x\ \mathbf{of}\ \_ \to x : 1)\ (h\ y) \\
& ===> \\
& f\ y = (\mathbf{case}\ x\ \mathbf{of}\ \_ \to x)\ [h\ y\ /\ x] \\
& ===> \\
& f\ y = (\mathbf{case}\ h\ y\ \mathbf{of}\ \_ \to h\ y) \quad \text{-- consume y twice.}
\end{aligned}
$$

Vs. call-by-need

$$
\begin{aligned}
& (\lambda y = (\lambda x \circ \mathbf{case}\ x\ \mathbf{of}\ \_ \to x : 1)\ (h\ y))\ e \\
& ===> \\
& \quad \mathbf{let}\ y = e\ \mathbf{in} \\
& \qquad \mathbf{let}\ x = h\ y\ \mathbf{in} \\
& \qquad\quad \mathbf{case}\ x\ \mathbf{of}\ \_ \to x \\
& ===> \\
& \qquad \mathbf{let}\ x = h\ e\text{`}v\ \mathbf{in} \\
& \qquad\quad \mathbf{case}\ x\ \mathbf{of}\ \_ \to x \\
& ===> \\
& \qquad\quad \mathbf{case}\ x\text{`}v\ \mathbf{of}\ \_ \to x\text{`}v
\end{aligned}
$$

3.4 Syntax Directed System

In the other system we assume that the recursive lets are strongly connected, i.e. the expressions always

3.4.1 Inferring usage environments

The difference between this and the previous section is a bit blurry

There's one more concern: usage environments aren't readily available, especially in recursive lets. We must perform inference of usage environments before we can type-check using them. This is how:

Rather, we define a syntax directed type system that infers usage environments while checking...

$$\boxed{\Gamma; \Delta_1/\Delta_2 \vdash e : \sigma}$$

$$\frac{\Gamma, p; \Delta_1/\Delta_2 \vdash e : \sigma \qquad p \notin \Gamma}{\Gamma; \Delta_1/\Delta_2 \vdash \Lambda p.\ e : \forall p.\ \sigma}\ (\Lambda I) \qquad \frac{\Gamma; \Delta_1/\Delta_2 \vdash e : \forall p.\ \sigma \qquad \Gamma \vdash_{mult} \pi}{\Gamma; \Delta_1/\Delta_2 \vdash e\ \pi : \sigma[\pi/p]}\ (\Lambda E)$$

$$\frac{\Gamma; \Delta_1, x{:}_1\sigma/\Delta_2 \vdash e : \varphi \qquad x \notin \Delta_1, \Delta_2}{\Gamma; \Delta_1/\Delta_2 \vdash \lambda x{:}_1\sigma.\ e : \sigma \to_1 \varphi}\ (\lambda I_1) \qquad \frac{\Gamma, x{:}_\omega\sigma; \Delta_1/\Delta_2 \vdash e : \varphi \qquad x \notin \Gamma}{\Gamma; \Delta_1/\Delta_2 \vdash \lambda x{:}_\omega\sigma.\ e : \sigma \to_\omega \varphi}\ (\lambda I_\omega)$$

$$\frac{}{\Gamma, x{:}_\Delta\sigma; \Delta, \Delta'/\Delta' \vdash x : \sigma}\ (Var_\Delta) \qquad \frac{\Gamma; \Delta_1, x{:}_1\sigma/\Delta_2 \vdash e : \varphi \qquad K \text{ has } n \text{ linear arguments}}{\Gamma; \Delta_1, \overline{x{:}_1\sigma \# K_i}^n/\Delta_2 \vdash x : \sigma}\ (Split)$$

$$\frac{}{\Gamma, x{:}_\omega\sigma; \Delta/\Delta \vdash x : \sigma}\ (Var_\omega) \qquad \frac{\Gamma; \Delta_1/\Delta_2 \vdash e : \sigma \to_1 \varphi \qquad \Gamma; \Delta_2/\Delta_3 \vdash e' : \sigma}{\Gamma; \Delta_1/\Delta_3 \vdash e\ e' : \varphi}\ (\lambda E_1)$$

$$\frac{}{\Gamma; \Delta, x{:}_1\sigma/\Delta \vdash x : \sigma}\ (Var_1) \qquad \frac{\Gamma; \Delta_1/\Delta_2 \vdash e : \sigma \to_\omega \varphi \qquad \Gamma; \Delta_2/\Delta_2 \vdash e' : \sigma}{\Gamma; \Delta_1/\Delta_2 \vdash e\ e' : \varphi}\ (\lambda E_\omega)$$

$$\frac{\Gamma; \Delta_1/\Delta_2 \vdash e : \sigma \qquad \Gamma, x{:}_\Delta\sigma; \Delta_1/\Delta_3 \vdash e' : \varphi}{\Gamma; \Delta_1/\Delta_3 \vdash \mathbf{let}\ x{:}_{\Delta_1 \setminus \Delta_2}\sigma = e\ \mathbf{in}\ e' : \varphi}\ (Let)$$

$$\frac{\overline{\Gamma, \overline{x{:}_\Delta\sigma}; \Delta_1/\Delta_i \vdash e : \sigma} \qquad \Gamma, \overline{x{:}_\Delta\sigma}; \Delta, \Delta' \vdash e' : \varphi}{\Gamma; \Delta_1/\Delta_2 \vdash \mathbf{let\ rec}\ \overline{x{:}_\Delta\sigma = e}\ \mathbf{in}\ e' : \varphi}\ (LetRec)$$

(CaseWHNF)
$$\frac{\text{e is in } \mathit{WHNF} \qquad \Gamma; \Delta_1/\Delta_2 \vdash e : \sigma \qquad \overline{\Gamma, z{:}_\Delta\sigma; \Delta, \Delta' \vdash_{alt} \rho \to e' :^z_\Delta \sigma \Longrightarrow \varphi}}{\Gamma; \Delta, \Delta' \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_\Delta\sigma\ \{\overline{\rho \to e'}\} : \varphi}$$

(CaseNotWHNF)
$$\frac{\text{e is definitely not in } \mathit{WHNF} \qquad \Gamma; \Delta_1/\Delta_2 \vdash e : \sigma \qquad \overline{\Gamma, z{:}_{[\Delta]}\sigma; [\Delta], \Delta' \vdash_{alt} \rho \to e' :^z_{[\Delta]} \sigma \Longrightarrow \varphi}}{\Gamma; \Delta, \Delta' \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_{[\Delta]}\sigma\ \{\overline{\rho \to e'}\} : \varphi}$$

$$\boxed{\Gamma; \Delta_1/\Delta_2 \vdash_{alt} \rho \to e :^z_{\Delta_s} \sigma \Rightarrow \varphi}$$

(AltN)
$$\frac{\Gamma, \overline{x{:}_\omega\sigma}, \overline{y_i{:}_{\Delta_i}\sigma_i}; \Delta \vdash e : \varphi \qquad \overline{\Delta_i} = \overline{\Delta_s \# K_j}^n \qquad \overline{\Delta_i \neq \cdot} \qquad n > 0 \qquad K : \overline{\sigma_i \to_\pi} \sigma \in \Gamma}{\Gamma; \Delta_1/\Delta_2 \vdash_{alt} K\ \overline{x{:}_\omega\sigma}, \overline{y_i{:}_1\sigma_i}^n \to e :^z_{\Delta_s} \sigma \Longrightarrow \varphi}$$

(Alt0)
$$\frac{\Gamma\,[\cdot/\Delta_s]_z, \overline{x{:}_\omega\sigma}; \Delta\,[\cdot/\Delta_s] \vdash e : \varphi \qquad K : \overline{\sigma_i \to_\omega} \sigma \in \Gamma}{\Gamma; \Delta_1/\Delta_2 \vdash_{alt} K\ \overline{x{:}_\omega\sigma} \to e :^z_{\Delta_s} \sigma \Longrightarrow \varphi}$$

(Alt_)
$$\frac{\Gamma; \Delta_1/\Delta_2 \vdash e : \varphi}{\Gamma; \Delta_1/\Delta_2 \vdash_{alt} \_ \to e :^z_{\Delta_s} \sigma \Longrightarrow \varphi}$$

Figure 3.1: Linear Core* Syntax Directed

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

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

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

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

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

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

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

$$\frac{\overline{\Gamma, \overline{x{:}_1\sigma} \vdash e' : \sigma \rightsquigarrow \Delta_{naive}} \qquad \overline{\Delta} = \mathsf{computeRecUsages}(\overline{\Delta_{naive}}) \qquad \Gamma, \overline{x{:}_{\Delta}\sigma} \vdash e : \varphi \rightsquigarrow \Delta'}{\Gamma \vdash \mathbf{let\ rec}\ \overline{x{:}_{\Delta}\sigma = e'}\ \mathbf{in}\ e : \varphi \rightsquigarrow \Delta'}\ (LetRec)$$

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

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

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

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

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

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

Figure 3.2: WIP: Linear Core* - Infer Usage Environments

CHAPTER 4

Linear Core as a GHC Plugin

This section discusses the implementation of Linear Core as a GHC Plugin, with a dash of painful history in the attempt of implementing Linear Core directly into GHC

4.1 Consuming tagged resources

As discussed in Section **??**, constructor pattern bound linear variables are put in the context with a *tagged* usage environment with the resources of the scrutinee. In a *tagged* usage environment environment, all resources are tagged with a constructor and an index into the many fields of the constructor.

In practice, a resource might have more than one tag. For example, in the following program, after the first pattern match, a and b have, respectively, usage environments $\{x\#K_1\}$ and $\{x\#K_2\}$:

```
f x = case x of
  K a b → case a of
    Pair n p → (n, p)
```

However, in the following alternative, n has usage environment $\{x\#K_1\#Pair_1\}$ and p has $\{x\#K_1\#Pair_2\}$. To typecheck (n, p), one has to *Split* x first on K and then on $Pair$, in order for the usage environments to match.

In our implementation, we split resources on demand (and don't directly allow splitting linear resources), i.e. when we use a tagged resource we split the linear resource in the linear environment (if available), but never split otherwise. Namely, starting on the innermost tag (the closest to the variable name), we substitute the linear resource for its split fragments, and then we iteratively further split those fragments if there are additional tags. We note that it is safe to destructively split the resource (i.e. removing the original and only leaving the split fragments) because we only split resources when we need to consume a fragment, and as soon as one fragment is consumed then using the original "whole" variable would violate linearity.

In the example, if n is used, we have to use its usage environment, which in turn entails using $x\#K_1\#Pair_1$, which has two tags. In this order, we:

- Split x into $x\#K_1$ and $x\#K_2$
- Split $x\#K_1$ and $x\#K_2$ into

  - $x\#K_1\#Pair_1$ and $x\#K_1\#Pair_2$
  - Leave $x\#K_2$ untouched, as we only split on demand, and we aren't using a fragment of $x\#K_2$.

- Consume $x\#K_1\#Pair_1$, the usage environment of n, by removing it from the typing environment.

CHAPTER 5

More ToDos

- We know the case binder to ALWAYS be in WHNF, perhaps there could be some annotation on the case binder s.t. we know nothing happens when we scrutinize it as a single variable
- Should we discuss this? It would be fine, but we're not able to see this because of call-by-name substitution

  $$f\ x = \mathbf{case}\ x\ \mathbf{of}\ z\ \{\_ \to x\}$$

- Generalizing linearity section...
- We kind of ignore the multiplicity semiring. We should discuss how we don't do ring operations ... but that's kind of wrong.

CHAPTER 6

Discussion

Preamble

6.1 Related Work

terrible paragraph name

Formalization of Core

System F_C [**?**] (Section 2.5) 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 [**?**] 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 [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**], all designed in the context of a typed language (Core) which does not have linear types. However, [**?**, **?**, **?**] 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.

6.2 Future Work

- Linear(X), a linear type system defined by the underlying definition of evaluation (which in turn implies how consuming a resource is defined)
- Implementation in Core
- Generalization to source level language, being more permissive in the handling of resources imposes less burden on the programmer
- It's harder to typecheck linearity like this in the source level because of the interaction with other source features, but seems feasible and an improvement to the usability of linear types. It allows more lazy functional programming idioms with linear types (also because laziness and strictness is less well defined as in Core, bc opts)

6.3 Conclusions