Todo list

We need to handle EmptyCase

And discuss how we didn't handle multiplicity coercions

Consider dropping some bits about GADTs?

Symbol to stand for both 1 and p, and notation to make proof irrelevant stuff in the types so we can also refer to relevant and irrelevant at the same time with some symbol (e.g. for Split)

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

Rodrigo Mesquita
Adviser: Bernardo Toninho

September 28, 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 optimising 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 optimising 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

Linear type systems [24, 4] 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 is flagged as a type error. Linear types can thus be used to, for instance, statically guarantee that socket descriptors are closed or heap-allocated memory is freed, exactly once (leaks and double-frees become type errors), or guarantee channel-based communication protocols are deadlock-free [9], among other correctness properties [22, 32, 5].

Consider the following C-like program in which allocated memory is freed twice. Regardless of the *double-free* error, 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 extensive presence in research literature [49, 10, 1], an effective design combining linear and non-linear typing is both challenging and necessary to bring the advantages of linear typing to mainstream languages. Consequently, few general purpose programming languages have linear type systems. Among them are Idris 2 [7], Rust [35], a language whose ownership types build on linear types to guarantee memory safety without garbage collection or reference counting, and, more recently, Haskell [5], a pure, functional, and *lazy* general purpose programming language.

It's the language we focus on

Linearity in Haskell stands out from linearity in other languages due to the following reasons:

- Linear types permeate Haskell down to (its) Core, the intermediate language into which Haskell is translated. Core is a minimal, explicitly typed, functional language, to which multiple Core-to-Core optimising transformations are applied during compilation. 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 optimised Core program fails to typecheck, the optimising transformations (are very likely to) have introduced an error in the resulting program. We present Core (and its formal basis, System F_C [47]) in Section 2.5.

- Both Haskell and its intermediate language Core are *lazily* evaluated, i.e. expressions in Haskell are only evaluated when needed, unlike expressions in Idris or Rust that are *eagerly* evaluated. Laziness allows an optimising compiler to aggressively transform the source program without changing its semantics, and indeed, GHC heavily transforms Core by leveraging its laziness. However, lazy evaluation interacts non-trivially with linearity. Intuitively, since expressions are not necessarily evaluated, an occurrence of a linear resource in an expression does not necessarily entail consuming that resource (i.e. if the expression isn't evaluated, the resource isn't used).

  In eagerly evaluated languages, the distinction between syntactic uses of a resource and the actual use of linear resources at runtime does not exist – an occurrence of a variable in the program always entails consuming it. This interaction is unique to Haskell since, as far as we know, it is the only language featuring both laziness and linearity. We review lazy and eager evaluation strategies in Section 2.4.

Much like a typed Core ensures that the translation from Haskell (dubbed *desugaring*) and the subsequent optimising transformations applied to Core 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 a program's 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 type-checker is key in providing such guarantees – it would be disastrous if the compiler, e.g., duplicated a pointer to heap-allocated memory that was previously just used once and then freed in the original program. Even more, linearity in Core can inform Core-to-Core optimising transformations [39, 40, 5], including inlining and β-reduction, to produce more performant programs.

In spite of a linearly typed Core ultimately being the desired intermediate language for the Glasgow Haskell Compiler, both in its expressiveness to completely represent a Haskell with linear types and in enabling more extensive compiler validations plus improved optimisations, linearity is effectively ignored in Core. The reason is not evidently clear: if we can typecheck linearity in the surface level Haskell, it should also be possible, and natural, to do so in Core.

The desugaring process from surface level Haskell to Core, and the subsequent Core-to-Core optimising transformations, eliminate, and rearrange, most of the syntactic constructs through which linearity checking is performed – often resulting in programs completely different from the original, especially due to the flexibility laziness provides a compiler in the optimisations it may perform. Crucially, since optimisations do not destroy linearity, the resulting program is still linear *semantically*, however, the current linear type system fails to recognize it as linear. For instance, let x be a linear resource in the two following programs, where the latter results from inlining y in the let body of the former. Despite the result no longer *looking* linear (as there are now two syntactic occurrences of the linear resource x), the program *is* indeed linear because the let-bound expression freeing x is never evaluated, so x is consumed exactly once when it is freed in the let body:

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

The Core optimising transformations expose a fundamental limitation of Core's linear type system: it does not account for the call-by-need evaluation model of Core, and thus a whole class of programs that are linear under the lens of lazy evaluation are rejected by Core's current linear type system.

In this work, we address this limitation (and its implications on validating and influencing optimising transformations) by designing a type system which understands semantic linearity in the presence of laziness and is suitable for the intermediate language of an optimising compiler. In detail, our contributions are:

- We explain and provide insights into *semantic* linearity in contrast to *syntactic* linearity, in Haskell, by example (§ 3.1).
- We introduce Linear Core, a type system for a linear lazy language with all key features from Core (except for type equality coercions), which, crucially, understands semantic linearity in the presence of laziness. To the best of our knowledge, this is the first type system to understand linearity semantically in the context of lazy evaluation (§ 3.2).
- We prove Linear Core to be sound (well-typed Linear Core programs do not get *stuck*) and prove that multiple optimising transformations (which currently violate linearity in Core) preserve types and linearity in Linear Core (§ 3.3).
- We implement Linear Core as a GHC plugin which typechecks linearity in all intermediate Core programs produced during the compilation process, showing it accepts the intermediate programs resulting from (laziness-abusing) transformations in linearity-heavy Haskell libraries, such as `linear-base` (§ 3.4).

We review background concepts fundamental to our work in Chapter 2, including linear type systems, linear types in Haskell, evaluation strategies, Core's type system, and multiple optimising transformations applied by GHC in its compilation process. We compare our contributions to related work and discuss possible avenues for further research (highlighting so-called *multiplicity coercions*) in Chapter 4, which concludes the document.

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 multiple Core-to-Core optimising transformations that we prove type-preserving in our type system.

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 [24, 4], 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` $\to \star$, 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 [24, 4], 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.

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

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 [24] (the Curry-Howard isomorphism [15, 27]), 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 [31, 33] and whose *de-facto* implementation is GHC, the Glasgow Haskell Compiler [34]. 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 [16], Haskell goes much further with, e.g., arbitrary-rank types [41], type-class polymorphism [26], levity polymorphism [20], multiplicity polymorphism [5], and, more recently, impredicative polymorphism [45].
- Type level computation by means of type classes [25] and Haskell's type families [11, 12, 21], 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 [19], 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.

2.3 Linear Haskell

The introduction of linear types to Haskell's type system is originally described in Linear Haskell [5]. While in Section 4.1.1 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 [5]:

- To consume a value of atomic base type (such as `Int` or `Ptr`) exactly once, just evaluate it.
- To consume a function value exactly once, apply it to one argument, and consume its result exactly once.
- To consume a value of an algebraic datatype exactly once, pattern-match on it, and consume all its linear components exactly once. For example, a linear pair (equivalent to $\otimes$) is consumed exactly once if pattern-matched on *and* both the first and second element are consumed once.

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

```
f :: a % 1 → (a, a)          g :: a % Many → (a, a)
f x = (x, x)                 g x = (x, x)
```

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

```
data MultPair a b where
  MkPair :: a % 1 → b % 1 → MultPair a b
```

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

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

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

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

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

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

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

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

2.4 Evaluation Strategies

Edit first draft. Acho que ainda está confuso

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

Unlike most mainstream programming languages, Haskell has so called *non-strict* evaluation semantics due to its *lazy* evaluation strategy, also known as *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. These execution models can be rigorously described through the operational semantics of the language. The so-called small-step operational semantics define the valid evaluation steps (reductions) an expression can use to evaluate to a value. The operational semantics of these three evaluation models primarily differ in how the reduction rule for function application, also known as β-reduction, is defined:

- Under *call-by-value*, a function application is reduced to a *value* by evaluating the function $\lambda x.\ b$ and the argument e to a *value* v, then substituting occurrences of the function argument variable by the argument value:

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

- 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. Evaluation then progresses in the let body until the value of a let-bound variable is needed, being then computed and substituted in the let body. The call-by-need lambda calculus [2] is, accordingly, further characterized by the following evaluation axioms:

$$\begin{aligned}
&\mathsf{let}\ x = V\ \mathsf{in}\ C[x] \longrightarrow \mathsf{let}\ x = V\ \mathsf{in}\ C[V] \\
&(\mathsf{let}\ x = L\ \mathsf{in}\ M)\ N \longrightarrow \mathsf{let}\ x = L\ \mathsf{in}\ M\ N \\
&\mathsf{let}\ y = (\mathsf{let}\ x = L\ \mathsf{in}\ M)\ \mathsf{in}\ N \longrightarrow \mathsf{let}\ x = L\ \mathsf{in}\ \mathsf{let}\ y = M\ \mathsf{in}\ N
\end{aligned}$$

In essence, *call-by-value* gives the language *strict* evaluation semantics, where expressions are evaluated in the order they occur, s.t. using a non-terminating computation (e.g. $f(x) = f(x)$, usually written $\bot$) in the program text will necessarily make the program not terminate upon reaching that expression (i.e. written $g(\bot) = \bot$ for all g). In contrast, *call-by-name* and *call-by-need* evaluation give the language *non-strict* semantics, where the order in which expressions are evaluated is undefined, since they are evaluated only when needed. Consequently, in non-strict semantics, using $\bot$ only results in a non-termination if $\bot$ is evaluated, which isn't necessary for the program to reach a result (i.e. $g(\bot)$ is not necessarily $\bot$).

Call-by-need additionally guarantees we only do "work" to compute an expression bound to a variable *once*, when it is first required, and share the result of the computation amongst subsequent occurrences of the variable. Intuitively:

- When an expression is bound by a `let`, a *thunk* is created, representing the (suspended) computation that evaluates the expression to a value
- *If* the let-bound variable is used in a computation, we must evaluate the *thunk*-suspended computation to determine the value this variable represents.
- The *thunk* is overwritten with the resulting value, s.t. subsequent uses of the same variable now refer to the already computed value instead of the suspended computation.

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 review laziness in that context:

In Haskell and in its intermediate language Core, 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 (as per the β-reduction rule under call-by-need). Suspending a computation amounts to giving a name to the unevaluated expression. As before, when the value associated to this name is required, then the suspended computation is said to be *forced*, a result is computed by evaluating the expression, and the *thunk* is overwritten with the result. Call-by-need evaluation is traditionally modelled with a mutable heap for the existing thunks or values they are overwritten with [**?**]. Beyond lambdas and let-bindings, Haskell and Core also feature algebraic datatypes and case expressions to match on datatype constructors:

- A datatype defines a (user-defined) *type* through a set of constructors $\overline{K_i}$ that can be used to build values of the type they define. A constructor applied to a set of argument expressions, $K\ \overline{e}$, is said to be in Weak Head Normal Form, and does not reduce any further on its own.
- A case expression $\mathsf{case}\ e_s\ \mathsf{of}\ \overline{\rho_i \to e_i}$ is defined by a scrutinee e_s and a list of alternatives comprised of a pattern ρ (either a wildcard $\_$, or a constructor and a set of variables to bind the constructor arguments $K\ \overline{x}$) and a right hand side expression e_i. Case expressions are lazily evaluated by
  1. Evaluating the scrutinee to Weak Head Normal Form, resulting in either a lambda expression or a constructor of arity n applied to n unevaluated expressions $\overline{e_i}^n$
  2. Matching the weak head normal form of the scrutinee against the patterns, (possibly) substituting the pattern-bound variables of a matching constructor by the unevaluated arguments of a scrutinee constructor application in Weak Head Normal Form, e.g.:

$$\mathsf{case}\ K\ \overline{e_i}^n\ \mathsf{of}\ K\ \overline{x_i}^n \to e' \implies e'\overline{e_i/x_i}^n$$

Finally, briefly accounting for linearity in this context, we note that a linear resource occurring in a suspended computation is only consumed if that computation is executed, foreshadowing the distinction we will explore in Section 3.1 between multiple *syntactic* occurrences of a linear resource and the *semantic* usages of the same resource, where we may have multiple *syntactic* occurrences of a linear resource in suspended computations, but *semantically* consume the resource exactly once as long as we run just one of them exactly once.

[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.

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 [14]. 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, optimising 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 [23, 42]), type-checking Core serves as an internal consistency check for the desugaring and optimization passes. The Core typechecker provides a verification layer for the correctness of desugaring and optimising transformations (and their implementations) because both desugaring and optimising transformations must produce well-typed Core.

- Finally, Core's expressiveness serves as a sanity-check for all the extensions to the source language – if we can desugar a feature into Core then the feature must be sound by reducibility. Effectively, any feature added to Haskell is only syntatic sugar if it can be desugared to Core.

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

The Core language is based on *System* F_C, a polymorphic lambda calculus with explicit type-equality coercions that, like types, are erased at compile time (i.e. types and coercions alike don't incur any cost at run-time). The syntax of System F_C [47] terms is given in Figure 2.3, which corresponds exactly to the syntax of System F [23, 42] with

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

Figure 2.3: System F_C's Terms

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.

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

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

Figure 2.4: System F_C's Types and Coercions

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

- 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* [36], allowing new optimizations to be performed which ultimately result in efficient code using labels and jumps, and with a construct used for internal notes such as profiling information.

In the context of Linear Haskell, and recalling that Haskell is fully desugared into Core / System F_C as part of its validation and compilation strategy, we highlight the inherent incompatibility of linearity with Core / System F_C as a current flaw in GHC

that invalidates all the benefits of Core wrt linearity. Thus, we must extend System F_C (and, therefore, Core) with linearity in order to adequately validate the desugaring and optimising 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 optimising 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 optimising 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 optimised. As such, certain orderings of optimizations can hide optimization opportunities and block them from firing. This phase-ordering problem is present in most optimising 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 optimising 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 [43, 40]. These were subsequently refined and expanded [37, 3, 36, 8, 44]. In Figure 2.5 we present an example that is optimised 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 [40]. 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 [37] 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 [39]. 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:

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

  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 [39]. 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.

$$\lambda y.\ \mathsf{let}\ x = e\ \mathsf{in}\ e' \implies \mathsf{let}\ x = e\ \mathsf{in}\ \lambda y.\ e'$$

- 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:

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

  These transformations do not change the number of allocations but potentially create opportunities for other optimizations to fire, such as expose a lambda abstraction [39].

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:

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

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 [13].

Static argument and lambda lifting. *Lambda lifting* is a transformation that abstracts over free variables in functions by making them λ-bound arguments [28, 43]. 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 [38]. This transformation creates two functions from an original one: a worker and a wrapper. The worker receives unboxed values [30] 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.

Binder-swap. The binder swap transformation applies to a case expression whose scrutinee is a variable x, and consists of swapping the case binder z for x in all case alternatives:

$$\mathsf{case}\ x\ \mathsf{of}\ z\ \{\overline{\rho_i \to e_i}\} \Longrightarrow \mathsf{case}\ x\ \mathsf{of}\ z\ \{\overline{\rho_i \to e_i[z/x]}\}$$

By removing occurrences of x in the case alternatives we might end up with the case scrutinee being the only occurrence of x, which allows us to inline x and possibly save an allocation, for example:

$$\begin{aligned}
&\mathsf{let}\ x = factorial\ y\ \mathsf{in}\ \mathsf{case}\ x\ \mathsf{of}\ b\ \{I\#\ v \to \ldots x \ldots\} \\
&\Longrightarrow_{Binder\ swap} \\
&\mathsf{let}\ x = factorial\ y\ \mathsf{in}\ \mathsf{case}\ x\ \mathsf{of}\ b\ \{I\#\ v \to \ldots b \ldots\} \\
&\Longrightarrow_{Inlining} \\
&\mathsf{case}\ factorial\ y\ \mathsf{of}\ b\ \{I\#\ v \to \ldots b \ldots\}
\end{aligned}$$

Reverse binder-swap. The reverse binder swap is (unsurprisingly) the reverse of the binder swap transformation. For a case whose scrutinee is a variable x, reverse binder swaps occurrences of the case binder z by the variable x:

$$\mathsf{case}\ x\ \mathsf{of}\ z\ \{\overline{\rho_i \to e_i}\} \Longrightarrow \mathsf{case}\ x\ \mathsf{of}\ z\ \{\overline{\rho_i \to e_i[x/z]}\}$$

It is not entirely obvious why this might optimise a program, however, z is bound in the case alternative, so expressions involving z may not be floated out of the case alternative. If z is substituted by x, which isn't bound to the case, we might float out an expensive operation out of the case alternatives and, for example, out of a loop:

$$\begin{aligned}
&\mathsf{letrec}\ go\ y = \mathsf{case}\ x\ \mathsf{of}\ z\ \{(a,b) \to \ldots (expensive\ z) \ldots\}\ \mathsf{in}\ \ldots go \ldots \\
&\Longrightarrow_{Reverse\ binder\ swap} \\
&\mathsf{letrec}\ go\ y = \mathsf{case}\ x\ \mathsf{of}\ z\ \{(a,b) \to \ldots (expensive\ x) \ldots\}\ \mathsf{in}\ \ldots go \ldots \\
&\Longrightarrow_{Float\ out} \\
&\mathsf{let}\ t = expensive\ x\ \mathsf{in}\ \mathsf{letrec}\ go\ y = \mathsf{case}\ x\ \mathsf{of}\ z\ \{(a,b) \to \ldots t \ldots\}\ \mathsf{in}\ \ldots go \ldots
\end{aligned}$$

In this example, $expensive\ x$ is now computed once, instead of once per loop iteration.

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 optimised Core, the program is translated down to the Spineless Tagless G-Machine (STG) language [29]. 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.

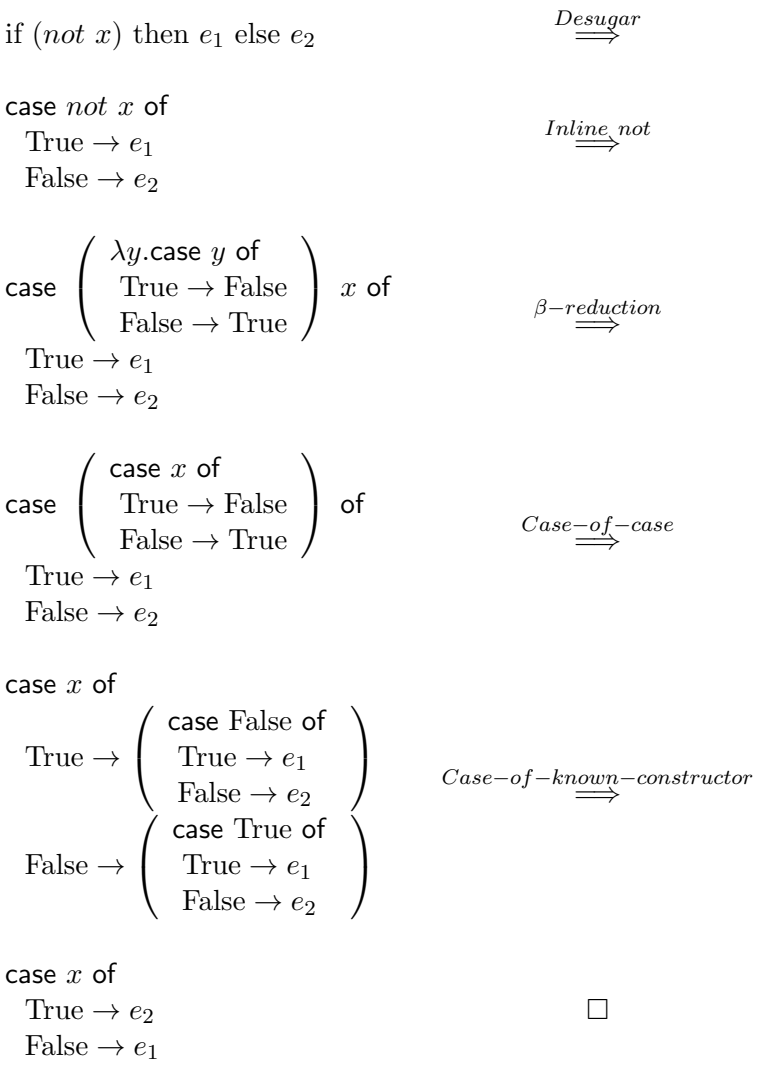

Figure 2.5: Example sequence of transformations

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.

CHAPTER 3

A Type System for Semantic Linearity in Core

A linear type system statically guarantees that linear resources are consumed *exactly once*. Even though linear types exist Linear Haskell brings the promises of linear types to the Haskell. With linear types, programmers can create powerful abstractions that enforce certain resources are used linearly, such as file handles or allocated memory.

- Linear types are unique in Haskell because of laziness! (amongst some other things)
- 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
- Somewhere add a stronger case for controlling program optimisations (see 7.1 of Linear Haskell) with linearity
- Somewhere say we build on linear haskell?

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.

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 ...

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 Haskell-like 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 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 semantics, 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 semantics, 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 *call-by-need*, but also because the distinction between semantically and syntactically consuming a resource is only exposed under non-strict (i.e. lazy) 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 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 or light orange background mark programs that are semantically linear, but are not seen as linear by Core's (naive wrt laziness) 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). 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 semantically linearly 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.

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 3.2.3.

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:

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

[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

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.

Summarising, 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 it 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:

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

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 [5]:

- 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 **??** 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:

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

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:

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

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:

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

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:

$$f14 :: a \multimap b \multimap (a \multimap b \multimap c) \to c$$
$$f14\ x\ y\ use = \mathbf{case}\ (x, y)\ \mathbf{of}\ z\ \{(a, b) \to uncurry\ use\ z\}$$

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

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

A second example of an unrestricted pattern, where *K2* has no fields and *K1* has one linear field, seems as though it shouldn't typecheck since the resource x must have been fully consumed to take the *K2* branch, but because the scrutinee is known to be *K1*, the *K2* branch is *absurd*, so, in reality any resource could be freely used in that branch, and the example is *semantically* linear (despite not being seen as so by our system):

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

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 optimising compiler, many programs such as this naturally occur (e.g. if the definition

of a function is inlined in the scrutinee).

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)
```

Before last, 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 linearity is seen as before but without fully consuming the scrutinee linear resources (in non-linear patterns) nor binding new linear resources (in linear patterns). In short, 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.

Finally, we discuss the special case of a case expression scrutinizing a variable x:

$$\lambda x.\ \textbf{case}\ x\ \textbf{of}\ \_ \to x$$

It might seem as though the program is linear:

- If x is in WHNF, then scrutinizing it is a no-op, and returning x just returns the resource intact.
- If x is not in WHNF, then scrutinizing it evaluates it to WHNF, and returning x returns the result of evaluating x that still had to be consumed exactly once.

However, in practice, it depends on the evaluation strategy. If linear function applications are β-reduced *call-by-name* (a common practice, as linear functions use their argument exactly once), and the above function is considered linear, then an application might duplicate linear resources during evaluation. For example:

$$\begin{array}{l}(\lambda x.\ \textbf{case}\ x\ \textbf{of}\ \_ \to x)\ (free\ x)\\ \Longrightarrow_{CBN\ \beta-reduction}\\ \textbf{case}\ free\ x\ \textbf{of}\ \_ \to free\ x\end{array}$$

Therefore, the type system we present in the next section, which evaluates linear applications call-by-name, does not accept the above program as linear. Foreshadowing, this particular interaction between evaluation and linearity comes up in the type preservation

proof of our program, and is again explored with the reverse binder swap transformation in Section 3.3.4.

In 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 need to be used, in mutual exclusion. 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.

3.2 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 [5] 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 crystallizing 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. Besides type safety, we prove that multiple optimising Core-to-Core transformations preserve linearity in Linear Core. These same transformations don't preserve linearity under Core's current type system. As far as we know, we are the first to prove optimisations preserve types in a non-strict linear language.

The first key idea for typing linearity semantically is to delay *consuming a resource* to when a computation that depends on that resource is effectively evaluated or returned, by annotating relevant binders with *usage environments* (§ 3.2.3). The second key idea is to have two distinct rules for case expressions, branching on whether the scrutinee is in Weak Head Normal Form, and using "proof irrelevance" to track resources that are no longer available but haven't yet been fully consumed (§ 3.2.4). Additionally, we introduce tagged resources to split the resources between the pattern-bound variables as usage environments, encoding that pattern variables jointly "finish consuming" the scrutinee resources. 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 non-strict languages.

We present Linear Core's syntax and type system iteratively, starting with the judgements and base linear calculi rules for multiplicity and term lambdas plus the variable rules (§ 3.2.2). Then usage environments, the rule for Δ-bound variables, and rules for (recursive) let bindings (§ 3.2.3). Finally, we introduce the rules to typecheck case expressions and alternatives, along with the key insights to do so, namely branching on WHNF-ness of scrutinee, proof irrelevant resources, and tagged variables (§ 3.2.4).

[4] We explain a main avenue of future work, multiplicity coercions, in Section 4.2

3.2.1 Language Syntax and Operational Semantics

The complete syntax of Linear Core is given by Figure 3.1. The types of Linear Core are algebraic datatypes, function types, and multiplicity schemes to support multiplicity polymorphism: datatypes ($T\ \overline{p}$) are parametrised by multiplicities, function types ($\varphi \to_\pi \sigma$) are also annotated with a multiplicity, and a multiplicity can be 1, ω (read *many*), or a multiplicity variable p introduced by a multiplicity universal scheme ($\forall p.\ \varphi$).

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

The terms are variables x, y, z, data constructors K, multiplicity abstractions $\Lambda p.\ e$ and applications $e\ \pi$, term abstractions $\lambda x{:}_\pi\sigma.\ e$ and applications $e\ e'$, where lambda binders are annotated with a multiplicity π and a type σ. Then, there are non-recursive let bindings **let** $x{:}_\Delta\sigma = e$ **in** e', recursive let bindings **let rec** $\overline{x{:}_\Delta\sigma = e}$ **in** e', where the overline denotes a set of distinct bindings $x_1{:}_{\Delta_1}\sigma_1 \ldots x_n{:}_{\Delta_n}\sigma_n$ and associated expressions $e_1 \ldots e_n$, and case expressions **case** e **of** $z{:}_\Delta\sigma\ \{\overline{\rho \to e'}\}$, where z is the case binder and the overline denotes a set of distinct patterns $\rho_1 \ldots \rho_n$ and corresponding right hand sides $e'_1 \ldots e'_n$. Notably, (recursive) let-bound binders and case-bound binders are annotated with a so-called *usage environment* Δ – a fundamental construct for type-checking semantic linearity in the presence of laziness we present in Section 3.2.3. Case patterns ρ can be either the *default* or *wildcard* pattern _, which matches any expression, or a constructor K and a set of variables that bind its arguments, where each field of the constructor has an associated multiplicity denoting whether the pattern-bound variables must consumed linearly (ultimately, in order to consume the scrutinee linearly). Additionally, the set of patterns in a case expression is guaranteed to be exhaustive, i.e. there is always at least one pattern which matches the scrutinized expression.

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

Linear Core takes the idea of annotating lets with usage environments from the unpublished Linear Mini-Core document by Arnaud Spiwack et. all [6], which first tentatively tackled Core's linearity issues. We discuss this work in more detail in Section 4.1.2.

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

The (small-step) operational semantics of Linear Core are given by Figure 3.2. We use call-by-name evaluation for Linear Core as it captures the non-strict semantics in which our type system understands linearity, while being simpler to reason about than call-by-need

Types

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

Terms

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

Environments

Γ	$::=$	$\cdot \mid \Gamma, x{:}_\omega \sigma \mid \Gamma, K{:}\sigma \mid \Gamma, p \mid z{:}_\Delta \sigma$	Unrestricted (delta-)variables
Δ	$::=$	$\cdot \mid \Delta, x{:}_\pi \sigma \mid \Delta, [x{:}_\pi \sigma]$	Linear (and irrelevant) resources

Multiplicities

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

Usage Environments

$\Delta \quad ::= \cdot \mid \Delta_1 + \Delta_2 \mid \pi\Delta$

Declarations

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

Figure 3.1: Linear Core Syntax

operational semantics which is traditionally modelled with a mutable heap to store *thunks* and the values they are overwritten with. Furthermore, linear function applications, even in a *call-by-need* system, are usually reduced *call-by-name* as the function argument is guaranteed to be used exactly once (thus avoiding unnecessarily allocating memory on the heap for a redundant *thunk*). Specifically, function applications are reduced by the standard *call-by-name* β-reduction, substituting the argument whole by occurrences of the lambda binder in its body, case expressions evaluate their scrutinee to WHNF, substituting the result by the case binder and constructor arguments for pattern-bound bound variables matching on that same constructor.

Values

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

Evaluation Contexts

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

Expression Reductions

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

Figure 3.2: Linear Core Operational Semantics (call-by-name)

3.2.2 Typing Foundations

Linear Core (λ^π_Δ) is a linear lambda calculus akin to Linear Haskell's $\lambda^q_\to$ in that both have multiplicity polymorphism, (recursive) let bindings, case expressions, and algebraic data types. λ^π_Δ diverges from $\lambda^q_\to$ primarily when typing lets, case expressions, and alternatives (in its purpose to type semantic linearity). Otherwise, the base rules of the calculus for, multiplicity and term, abstraction and application are quite similar. In this subsection, we present the linear calculi's typing rules that share much in common with $\lambda^q_\to$, and, in the subsequent subsections, the rules encoding the novel insights from Linear Core in typing semantic linearity, that were explored by example in Section 3.1. We note, however, that we handle multiplicity polymorphism differently from Linear Haskell in ignoring the multiplicities semiring and instead conservatively treating all multiplicity polymorphic functions as linear. The full type system is given by Figure 3.3, with auxiliary judgements given by Figure 3.4.

We start with the main typing judgement. As is customary for linear type systems, we use two typing environments, an *unrestricted* Γ and a *linear* Δ environment. Variables in Γ can be freely discarded (*weakened*) and duplicated (*contracted*), while resources in Δ must be used exactly once (hence can't be discarded nor duplicated). Despite not having explicit weakening and contraction rules in our system, they are available as admissible rules for Γ (but not for Δ), since, equivalently (via [18]), resources from Γ are duplicated for sub-derivations and may unrestrictedly exist in the variable rules. The main typing judgement reads "expression e has type $\sigma \to \varphi$ under the unrestricted environment Γ and linear environment Δ":

$$\Gamma; \Delta \vdash e : \sigma \to \varphi$$

Occurrences of unrestricted variables from Γ are well-typed as long as the linear environment is empty, while occurrences of linear variables are only well-typed when the variable being typed is the only resource available in the linear context.

$$\frac{}{\Gamma, x{:}_\omega\sigma; \cdot \vdash x : \sigma}\ (Var_\omega) \qquad \frac{}{\Gamma; x{:}_1\sigma \vdash x : \sigma}\ (Var_1)$$

In both cases, the linear context must contain exactly what is required to prove the proposition, whereas the unrestricted context may contain arbitrary variables. Variables in contexts are annotated with their type and multiplicity, so $x{:}_\pi\sigma$ is a variable named x of type σ and multiplicity π.

Linear functions are introduced via the function type $(\sigma \to_\pi \varphi)$ with $\pi = 1$, i.e. a function of type $\sigma \to_1 \varphi$ (or, equivalently, $\sigma \multimap \varphi$) introduces a linear resource of type σ in the linear environment Δ to then type an expression of type φ. Unrestricted functions are introduced via the function type $(\sigma \to_\pi \varphi)$ with $\pi = \omega$, and the λ-bound variable is introduced in Γ:

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

A linear function application is well-typed if there exists a disjoint split of the linear resources into Δ, Δ' s.t. the function and argument, each under a distinct split, are both well-typed and the argument type matches the function's expected argument type. Conversely, unrestricted resources are duplicated and available whole to both sub-derivations.

$$\frac{\Gamma; \Delta \vdash e : \sigma \to_1 \varphi \qquad \Gamma; \Delta' \vdash e' : \sigma}{\Gamma; \Delta, \Delta' \vdash e\ e' : \varphi}\ (\lambda E_1)$$

An unrestricted function, unlike a linear one, consumes its argument unrestrictedly (zero or more times). Therefore, in an unrestricted function application, allowing any linear resources to occur in the argument expression entails consuming those resources unrestrictedly, since the variable binding the argument expression could be discarded or used more than once in the function body. Thus, argument expressions to unrestricted functions must also be unrestricted, i.e. no linear variables can be used to type them.

$$\frac{\Gamma; \Delta \vdash e : \sigma \to_\omega \varphi \qquad \Gamma; \cdot \vdash e' : \sigma}{\Gamma; \Delta \vdash e\ e' : \varphi}\ (\lambda E_\omega)$$

Typing linear and unrestricted function applications separately is less general than typing applications of functions of any multiplicity π by scaling (per the multiplicity semiring) the multiplicities of the resources used to type the argument by π, however, our objective of typing semantic linearity does not benefit much from doing so, and by keeping the simple approach we can have the linear and unrestricted environments be separate.

Multiplicity abstractions ($\Lambda p.\ e$) introduce a multiplicity variable p (in the unrestricted context), and construct expressions of type $\forall p.\ \ldots$, i.e. a type universally quantified over a multiplicity variable p. We note that, in the body of the abstraction, function types annotated with a p variable and datatype fields with multiplicity p are typed as thought they are linear functions and linear fields, because p can be instantiated at both ω and 1 (so types using p must be well-typed at both instantiations).

$$\frac{\Gamma, p; \Delta \vdash e : \sigma \qquad p \notin \Gamma}{\Gamma; \Delta \vdash \Lambda p.\ e : \forall p.\ \sigma}\ (\Lambda I)$$

A multiplicity application instantiates a multiplicity-polymorphic type $\forall p.\ \sigma$ at a particular (argument) multiplicity π, resulting in an expression of type σ where occurrences of p are substituted by π, i.e. $\sigma[\pi/p]$.

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

The rule additionally requires that π be *well-formed* in order for the expression to be well-typed, using the judgement $\Gamma \vdash_{mult} \pi$, where well-formedness is given by π either being 1, ω, or an in-scope multiplicity variable in Γ.

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

These rules conclude the foundations of our linear calculi. In subsequent subsections we type (recursive) let bindings and case expressions, accounting for semantic linearity as per the insights construed in Section 3.1, effectively distilling them into the key ideas of our work – encoded as rules.

3.2.3 Usage environments

A *usage environment* Δ is the means to encode the idea that lazy bindings don't consume the resources required by the bound expression when defined, but rather when the bindings themselves are fully consumed. Specifically, we annotate so-called Δ-bound variables with a *usage environment* to denote that consuming these variables equates to consuming the resources in the usage environment Δ they are annotated with, where a usage environment is essentially a multiset of linear resources. Δ-bound variables are introduced by a handful of constructs, namely, (recursive) let binders, case binders, and case pattern variables. In the following example, as per the insights into semantic linearity developed in Section 3.1.1, the resources required to typecheck the body of the binder u, x and y, are only used if the let-var u is consumed in the let-body e. Accordingly, the usage environment of the let-bound u is $\{x, y\}$:

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

Furthermore, usage environments guarantee that using a Δ-bound variable is mutually exclusive with directly using the resources it is annotated with – using the Δ-bound variable consumes all linear resources listed in its usage environment, meaning they are no longer available for direct usage. Dually, using the linear resources directly means they are no longer available to consume through the usage environment of the Δ-bound variable.

Finally, we note that usage environments bear a strong resemblance to the linear typing environments to the right of the semicolon in the main typing judgement, i.e. the environment with the linear resources required to type an expression. In fact, usage environments and linear typing contexts differ only in that the former are used to annotate variables, while the latter used to type expressions. Yet, this distinction is slightly blurred after introducing how typing environments can be moved to usage environments, or otherwise occurs in rules relating the two.

Δ-bound variables

A Δ-bound variable u is a variable annotated with a usage environment Δ. Crucially, for any Δ-bound variable u:

1. Using u is equivalent to using all the linear resources in Δ
2. Using u is mutually exclusive with using the Δ resources it depends on elsehow
3. u can be safely discarded as long as the resources in Δ are consumed elsehow

Fortunately, since linear resources must be linearly split across sub-derivations, (2) follows from (1): consuming the linear resources in Δ to type u makes them unavailable in the context of any other derivation. Therefore, expressions using these resources a second time, directly, or indirectly through the same (or other) usage environment, is ill-typed, as the resources are already allocated to the derivation of u. Similarly, (3) also follows from (1), because if the linear resources aren't consumed in the Δ-var derivation, they must be consumed in an alternative derivation (or otherwise the expression is ill-typed).

These observations all boil down to one typing rule for Δ-bound variables, which fundamentally encodes (1), implying the other two bullets:

$$\frac{}{\Gamma, x{:}_{\Delta}\sigma; \Delta \vdash x : \sigma}\ (Var_{\Delta})$$

The rule reads that an occurrence of a Δ-bound variable is well-typed if the linear environment is exactly the resources in the usage environment of that variable.

Δ-variables are always introduced in Γ since they can be discarded and duplicated, despite multiple occurrences of the same Δ-variable not possibly being well-typed as, ultimately, it would imply non-linear usage of linear resources.

Lazy let bindings

In Section 3.1.1, we discussed how linear resources used in let-bound expressions are only consumed when the same let-bound expressions are fully evaluated, i.e. linear resources required by let-bound expressions are consumed lazily. Moreover, resources from a let-bound expression cannot be used *together* with the variable binding, since said resources would end up being consumed more than once, violating (semantic) linearity – the binder has to be used in mutual exclusion with the linear resources required to type the expression it binds, and either *must* be used, or we'd be discarding resources.

Indeed, usage environments allow us to encode mutual exclusivity between alternative ways of consuming linear resources (between Δ-vars and direct resource usage). Let-bound variables are the canonical example of a Δ-bound variable, that is, let-variables bind expressions in which the resources required to type them are consumed lazily rather than eagerly. Effectively, annotating let-bound variables with a usage environment Δ *delays* the consumption of resources to when the variables themselves are used.

Summarily, let-bindings introduce Δ-variables whose usage environments are the linear typing environments of the bindings' bodies:

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

The rule for non-recursive let bindings splits the linear environment in Δ and Δ'. Δ is used to type the body e of the let binding x. Perhaps surprisingly, the resources Δ used to type e are still available in the environment to type the let body e', alongside the unrestricted x binding annotated with the usage environment Δ. Ultimately, the resources being available in e' reflects the fact that typing a lazily bound expression doesn't consume resources, and the binding x being Δ-bound reflects that its usage entails consuming the resources Δ the expression e it binds depends on.

Recursive let bindings

Recursive let bindings are very similar to non-recursive ones, the main exception being that the recursive bindings may be defined in terms of themselves, and we may have more

than one binding. In our system, groups of recursive let bindings are always assumed to be strongly connected, that is, all the bindings in a recursive let group are mutually recursive in the sense that they all (transitively) depend on one another.

As before, recursive let bindings bind expressions *lazily*, so they similarly introduce a Δ-variable for each binding, and resources required to type the let-bindings are still available in the body of the let, to later be consumed via Δ-variables or directly, if the let-bindings are unused. However, as shown by example in Section 3.1.1, we must consider how recursive uses of a binder in its own definition entails consuming all resources otherwise required to type the binder's body. Extrapolating to a strongly-connected group of recursive bindings, (mutually) recursive uses of binders entail consuming all resources required to type those binders. By definition, those binders in turn recursively use binders that used them, and thus all the resources otherwise required to type them. Ultimately, all binders in a strongly-connected group of mutually recursive let bindings have to be typed with the same linear resources (which are the least upper bound of resources needed to type the bodies of all binders, accounting for uses of mutually-recursive binders).

The typing rule for recursive groups of bindings leverages our assumption that all recursive let bindings are strongly connected and exactly the observation that every binder in a strongly connected group of recursive bindings is typed with the same linear context. Consequently, all bindings of the recursive group are introduced as Δ-vars with the same Δ environment – using any one of the bindings in a recursive group entails consuming all resources required to type that same group, which is also why we can use the same linear resources to type each binder:

$$\frac{\overline{\Gamma, \overline{x_i{:}_\Delta \sigma_i}; \Delta \vdash e_i : \sigma} \qquad \Gamma, \overline{x_i{:}_\Delta \sigma_i}; \Delta, \Delta' \vdash e' : \varphi}{\Gamma; \Delta, \Delta' \vdash \textbf{let rec } \overline{x_i{:}_\Delta \sigma_i = e_i} \textbf{ in } e' : \varphi} \text{ (LetRec)}$$

Unfortunately, this formulation is ill-suited for a syntax-directed system (from which an implementation is direct) because determining a particular Δ to type and annotate all binders is difficult. We present our system and metatheory agnostically to the challenge of inferring this linear typing environment by assuming recursive let expressions are annotated with the correct typing environment.

In practice, determining this typing environment Δ amounts to finding a least upper bound of the resources needed to type each mutually-recursive binding that (transitively) uses all binders in the recursive group. We propose a naive algorithm for inferring usage environments of recursive bindings in Section 3.4 orthogonally to the theory developed in this section. The algorithm is a $O(n^2)$ traversal over the so-called *naive usage environments* used to type each binding. Inference of usage environments for recursive binding groups bears some resemblance to the inference of principle types for recursive bindings traditionally achieved through the Hindley–Milner inference algorithm [16], there might be an opportunity to develop a better algorithm leveraging existing inference techniques. Despite being a seemingly useful observation, we leave exploring a potential connection between inference of usage environments and type inference algorithms as future work.

3.2.4 Case Expressions

Case expressions *drive evaluation* – a case expression *evaluates its scrutinee* to weak head normal form (WHNF), *then* selects the case alternative corresponding to the pattern matching the weak head normal form of the scrutinee[5]. An expression in weak head

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

normal form can either be:

- A lambda expression $\lambda x.\ e$,
- or a datatype constructor application $K\ \overline{e}$

In both cases, the sub-expressions e or $\overline{e}$ occurring in the lambda body or as constructor arguments needn't be evaluated for the lambda or constructor application to be in weak head normal form (if otherwise all sub-expressions were fully evaluated the whole expression would also be in *normal form*). Accordingly, these sub-expressions might still depend on linear resources to be well-typed (these resources will be consumed when the expression is evaluated). As will be made clear in later sections, we need to devise a specialized typing judgement for scrutinees that is able to distinguish between terms in WHNF and terms that are not in WHNF. Following the discussion on expressions in weak head normal form, we present a typing judgement $\Gamma; \Delta \Vdash e : \sigma \gg \overline{\Delta_i}$ for expressions in WHNF, and a rule for each of the forms given above:

$$\frac{(\mathrm{WHNF}_K)}{\dfrac{\overline{\Gamma; \cdot \vdash e_\omega : \sigma} \qquad \overline{\Gamma; \Delta_i \vdash e_i : \sigma} \qquad \overline{\Delta_i} = \Delta}{\Gamma; \Delta \Vdash K\ \overline{e_\omega e_i} : \sigma \gg \overline{\Delta_i}}} \qquad \frac{(\mathrm{WHNF}_\lambda)}{\dfrac{\Gamma; \Delta \vdash \lambda x.\ e : \sigma}{\Gamma; \Delta \Vdash \lambda x.\ e : \sigma \gg \Delta}}$$

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

Recall that the operational semantics encode the evaluation of case expressions as:

$$\begin{array}{ll}
\textbf{case}\ e\ \textbf{of}\ z\ \{\overline{\rho_i \to e_i}\} \longrightarrow^* \textbf{case}\ e'\ \textbf{of}\ z\ \{\overline{\rho_i \to e_i}\} & \text{where } e' \text{ is in WHNF and } e \text{ is not} \\
\textbf{case}\ K\ \overline{e}\ \textbf{of}\ z\ \{K\ \overline{x} \to e_i\} \longrightarrow e_i\overline{[e/x]}[K\ \overline{e}/z] & \\
\textbf{case}\ e\ \textbf{of}\ z\ \{_ \to e_i\} \longrightarrow e_i[e/z] & \text{where } e \text{ is in WHNF}
\end{array}$$

When a scrutinee $K\ \overline{e}$ matches a constructor pattern $K\ \overline{x{:}_\pi\sigma}$, evaluation proceeds in the case alternative corresponding to the matching pattern, with occurrences of $\overline{x}$ being substituted by $\overline{e}$, and occurrences of the case binder z substituted by the whole scrutinee $K\ \overline{e}$. Constructors and lambda expressions otherwise match the wildcard pattern whose alternative body is evaluated only substituting the case binder by the scrutinee.

We highlight that when evaluating a case expression, computation only effectively happens when a scrutinee not in WHNF is evaluated to WHNF. When the scrutinee is already in WHNF, evaluation continues in the alternative by substituting in the appropriate scrutinee expressions, but without having performed any computation. In terms of linearity, resources are consumed only if evaluation happens. Therefore, resources used to type a scrutinee not in WHNF will be consumed when the case is evaluated, making said resources unavailable in the case alternatives. Conversely, when the scrutinee is already in WHNF, linear resources required to type the scrutinee are still available in the alternatives. The linear resources used by an expression in WHNF are exactly those which occur

to the right of $\succcurlyeq$ in the WHNF judgement shown above (corresponding to the resources required to typecheck the lambda body or the constructor arguments).

Branching on WHNF-ness

The dichotomy between evaluation (hence resource usage) of a case expression whose scrutinee is in weak head normal form, or otherwise, leads to one of our key insights: we must *branch on weak head normal formed-ness* to type case expressions. When the scrutinee is already in weak head normal form, the resources are unused upon evaluation and thus available in the alternatives. When it is not, resources will be consumed and cannot be used in the alternative. To illustrate, consider a case expression with a scrutinee in weak head normal form and another whose scrutinee is not:

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

The first function uses x linearly, while the second does not. Alternatives may also use the case binder or pattern variables, referring to, respectively, the whole scrutinee (and all resources used to type the scrutinee) or constructor arguments (and the resources to type each argument).

Linear resources must be used exactly once, but there are three *competing* ways to use the resources from a scrutinee in WHNF in a case alternative: directly, via the case binder, or by using *all* the pattern-bound variables. Recall how Δ-variables can encode mutual exclusivity between alternative ways of consuming resources – it follows that case binders and pattern-bound variables are another instance of Δ-bound variables. Intuitively, resources in a scrutinee that is already in WHNF are only properly consumed when all (linear) fields of the pattern are used, satisfying the definition of consuming resources given in Linear Haskell. This suggests the following rule:

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

which captures the key intuitions of typing case expressions whose scrutinees are in WHNF. However, as will be made clear when we discuss matching on alternatives ..., our treatment of such a case expression is slightly more involved.

First, we assert this rule is only applicable to expressions in weak head normal form. Second, we use the typing judgement for expressions in WHNF previously introduced to determine the split of resources amongst the scrutinee sub-expressions. Finally, we type all case alternatives with the same context, using the $\vdash_{alt}$ judgement. Specifically:

- We introduce the case binder z in the environment as a Δ-bound variable whose usage environment is the linear resources used to type the scrutinee
- We make all the resources $\overline{\Delta_i}$ used to type the scrutinee available in the linear typing environment.
- We annotate the *alt* judgement with the disjoint set of linear resources $\overline{\Delta_i}$ used to typecheck the scrutinee sub-expressions
- We annotate the judgement with the name of the case binder z and use the $\Rightarrow$ arrow in the judgement – this is of most importance when typing the alternative itself, and will be motivated together with the alternative judgement below

The alternative judgement $\Gamma; \Delta \vdash_{alt} \rho \to e :^{z}_{\Delta} \sigma \Rightarrow \varphi$ is used to type case alternatives, but it encompasses three "sub-judgements", distinguished by the arrow that is used: for alternatives of case expressions whose scrutinee is in WHNF ($\Mapsto$), for case expressions in which the scrutinee is not in WHNF ($\Rrightarrow$), and for alternatives agnostic to the WHNF-ness of the scrutinee ($\Rightarrow$), with $\Rightarrow$ also generalizing the other two. Following the $Case_{\text{WHNF}}$ rule in which we use the $\Mapsto$ alternative judgement, the rule for typing a case alternative whose pattern is a constructor with $n > 0$ linear components is:

$$\frac{\Gamma, \overline{x :_{\omega} \sigma}, \overline{y_i :_{\Delta_i} \sigma_i}^{n}; \Delta \vdash e : \varphi}{\Gamma; \Delta \vdash_{alt} K\ \overline{x :_{\omega} \sigma}, \overline{y_i :_{1} \sigma_i}^{n} \to e :^{z}_{\overline{\Delta_i}^{n}} \sigma \Mapsto \varphi}\ (\textsc{AltN}_{\text{WHNF}})$$

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

Secondly, the rule for alternatives that match on the wildcard pattern:

$$\frac{\Gamma; \Delta \vdash e : \varphi}{\Gamma; \Delta \vdash_{alt} \_ \to e :^{z}_{\Delta_s} \sigma \Rightarrow \varphi}\ (\textsc{Alt}\_)$$

To type a wildcard alternative we simply type the expression with the main judgement, ignoring all annotations on the judgement; recalling that the case binder was already introduced in the environment with the appropriate usage environment by the case expression, rather than in the case alternative rule.

Finally, consider an alternative matching on a case constructor without any linear components. According to the definition of consuming a resource from Linear Haskell, the linear resources of a scrutinee matching such a pattern are fully consumed in the body of the corresponding alternative, since the scrutinee must have been evaluated to a form that does not have any linear components. This definition agrees with the intuition we have developed by example in the previous section, and with the typing rule we devised for alternatives matching constructors without linear components.

Taking into account that case expressions introduce the linear resources of the scrutinee in the typing environment of all alternatives, and in the usage environment of the case binder, we must reactively update the typing environments after matching on such a pattern. The *Alt*0 rule essentially encodes this insight, and is applicable regardless of the WHNF-ness of the scrutinee (hence the $\Rightarrow$ arrow), as long as the constructor pattern has no linear fields:

$$\frac{\Gamma\,[\cdot/\Delta_s]_z\,, \overline{x :_{\omega} \sigma}; \Delta\,[\cdot/\Delta_s] \vdash e : \varphi}{\Gamma; \Delta \vdash_{alt} K\ \overline{x :_{\omega} \sigma} \to e :^{z}_{\Delta_s} \sigma \Rightarrow \varphi}\ (\textsc{Alt0})$$

The rule deletes the annotated scrutinee environment Δ_s from two select environments:

- The linear typing environment, effectively deleting the resources from the scrutinee made available here by the case expressions (written $\Delta[\cdot/\Delta_s]$, a substitution of the scrutinee typing environment by the empty linear environment $\cdot$).
- The usage environment of the case binder z, written $\Gamma[\cdot/\Delta_s]_z$ to denote replacing the usage environment of the variable z in Γ, which is necessarily Δ_s (since we

always annotate the judgement with the environment of the scrutinee), by the empty environment.

The rule faithfully encodes the notion that an expression matching such a pattern is unrestricted when evaluated to WHNF, implying that all linear resources have been consumed to produce it, and the result is something that can be freely discarded or duplicated. It ensures that when we match on an unrestricted pattern we no longer need to consume the scrutinee resources. Otherwise, for example, **case** $K_1\ x$ **of** $\{K_2 \to K_2, K_1\ y \to K_1\ y\}$ would not be well-typed since the resource x is not consumed in the first branch. Furthermore, since the case binder in such an alternative refers to the unrestricted expression, the case binder too may be used unrestrictedly, which we allow by making its usage environment empty.

It might seem as though deleting the resources from the environment in this rule is necessary to guarantee a resource is not used after it is consumed. However, let us consider two discrete situations – pattern matches in a case expression whose scrutinee is in WHNF, and matches on a case expression whose scrutinee is *not* in WHNF:

- When the scrutinee is in WHNF, it is either an unrestricted expression against which any match will only introduce unrestricted variables, or an expression that depends on linear resources. The first case trivially allows any resource from the scrutinee in the alternatives as well. The second is further divided:

  1. The pattern is unrestricted while the scrutinee is not, so entering this branch is impossible as long as the case expression is well-typed; by contradiction, the linear resources from the scrutinee could occur unrestrictedly in that branch, since from falsity anything follows (*ex falso quodlibet*). For uniformity, we type such alternatives as those for scrutinees that are not in WHNF.
  2. The pattern is linear and matches the scrutinee, in which case the $AltN_{\text{WHNF}}$ is applicable instead of $Alt0$.
  3. The pattern is linear but does not match the scrutinee and so, the same reasoning as (1) above applies: any resource could theoretically be used in such alternatives, however, for uniformity, it is also typed as though the scrutinee were not in WHNF.

- However, if the scrutinee is not in WHNF, the resources occurring in the scrutinee will be consumed when evaluation occurs. Therefore, the resources used in the scrutinee cannot occur in the alternative body (e.g. x cannot occur in the alternative in **case** *close* x **of** $\{K_1 \to x\}$) – regardless of the pattern. We guarantee resources from a scrutinee that is not in weak head normal form cannot occur/directly be used in any case alternative, in our rule for typing cases not in WHNF, which we introduce below.

Proof irrelevant resources

Resources used in a scrutinee that is not in weak head normal form must definitely not be used in the case alternatives However, it is not sufficient to evaluate the scrutinee to weak head normal form to *fully* consume all resources used in the scrutinee, since subexpressions such as constructor arguments will be left unevaluated. To *fully* consume all resources occurring in the scrutinee, the scrutinee must be evaluated to normal form or s.t. all linear components of the scrutinee are fully evaluated, as witnessed by the $Alt0$ rule. In short, for a case expression whose scrutinee is not in WHNF:

- The scrutinee resources must *not* be used directly in the case alternatives;
- But the result of evaluating the scrutinee to WHNF must still be consumed, as all sub-expressions of the scrutinee remain unevaluated and must be consumed.
- Since the scrutinee resources cannot be consumed directly, they must be consumed indirectly through Δ-variables, namely, either the case binder, or the linear pattern-bound variables introduced in the alternative.

We introduce *proof irrelevant* resources, denoted as linear resources within square brackets $[\Delta]$, to encode linear resources that cannot be directly used (the Var rule is not applicable). Proof irrelevant resources are linear resources in all other senses, meaning they must be used *exactly once*. However, since proof irrelevant resources cannot be forgotten neither used directly, they have to be consumed *indirectly* – by Δ-bound variables.

To type a case expression whose scrutinee is in weak head normal form, we type the scrutinee with linear resources Δ and type the case alternatives by introducing the case binder with a usage environment $[\Delta]$, having the same proof irrelevant linear context $[\Delta]$ in the typing environment, annotating the judgement with the proof irrelevant resources, and using the $\Rrightarrow$ judgement:

$$\frac{\begin{array}{l}(\textsc{Case}_{\textsc{Not whnf}})\\ \text{e is not in } \mathit{WHNF} \qquad \Gamma;\Delta \vdash e : \sigma \qquad \overline{\Gamma, z{:}_{[\Delta]}\sigma;\, [\Delta]\,, \Delta' \vdash_{alt} \rho \to e' :^{z}_{[\Delta]} \sigma \Rrightarrow \varphi}\end{array}}{\Gamma;\Delta,\Delta' \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_{[\Delta]}\sigma\ \{\overline{\rho \to e'}\} : \varphi}$$

Note how the rule is quite similar to the one for scrutinees in WHNF, only diverging in that the resources in the case binder, typing environment, and judgement annotation, are made irrelevant.

Finally, ...

- Em WHNF dá para determinar qual dos padrões faz match
- Esse tem de ser tratado de forma especial
- Os restantes trato de forma especial também pq nunca vão ser aplicados
- Note that unusal to type WHNF specifically, this happens in the language of an optimising compiler

$$\frac{\begin{array}{l}(\textsc{Case}_{\textsc{WHNF}})\\ \text{e is in } \mathit{WHNF} \qquad \Gamma;\Delta \Vdash e : \sigma \gg \overline{\Delta_i} \qquad e \ \mathsf{matches}\ \rho_j\\ \Gamma, z{:}_{\overline{\Delta_i}}\sigma;\, \overline{\Delta_i}, \Delta' \vdash_{alt} \rho_j \to e' :^{z}_{\overline{\Delta_i}} \sigma \mapsto \varphi \qquad \overline{\Gamma, z{:}_{[\Delta]}\sigma;\, [\Delta]\,, \Delta' \vdash_{alt} \rho \to e' :^{z}_{[\Delta]} \sigma \Rrightarrow \varphi}\end{array}}{\Gamma;\Delta,\Delta' \vdash \mathbf{case}\ e\ \mathbf{of}\ z{:}_{\overline{\Delta_i}}\sigma\ \{\overline{\rho \to e'}\} : \varphi}$$

Splitting and tagging fragments

Intuitively, in case alternatives whose scrutinee is not in weak head normal form, the proof-irrelevant resources introduced by the case expression must be fully consumed, either via the case binder z, or by using all linear pattern-bound variables (for uniformity, we also treat alternatives that do not match a scrutinee in WHNF this way).

However, unlike with scrutinees in WHNF, the resources used by a scrutinee not in WHNF do not necessarily match those used by each sub-expression of the expression evaluated to WHNF. Therefore, there is no direct mapping between the usage environments of the linear pattern-bound variables and the resources used in the scrutinee.

We introduce *tagged resources* to guarantee all linearly-bound pattern variables are jointly used to consume all resources occurring in the environment (in alternative to the case binder), or not at all. Given linear resources $[\Delta_s]$ used to type a scrutinee, and a pattern $K\ \overline{x_\omega}, \overline{y_i}$ with i linear components, we assign a usage environment Δ_i to each linear pattern variable where, Δ_i is obtained from the scrutinee environment tagged with the constructor name and linear-variable index $\Delta_s \# K_i$, and $y{:}_{\Delta_i}\sigma$ is introduced in Γ.

$$\frac{\begin{array}{l}(\textsc{AltN}_{\textsc{Not whnf}}) \\ \Gamma, \overline{x{:}_\omega\sigma}, \overline{y_i{:}_{\Delta_i}\sigma_i}; \Delta \vdash e : \varphi \qquad \overline{\Delta_i} = \overline{\Delta_s \# K_i}^n\end{array}}{\Gamma; \Delta \vdash_{alt} K\ \overline{x{:}_\omega\sigma}, \overline{y_i{:}_1\sigma_i}^n \to e :^z_{\Delta_s} \sigma \Rrightarrow \varphi}$$

The tag consists of a constructor name K and an index i identifying the position of the pattern variable among all bound variables in that pattern. The key idea is that a linear resource x can be split into n resources at a given constructor, where n is the number of positional linear arguments of the constructor. This is given by the rule:

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

By assigning to each linear pattern variable a fragment of the scrutinee resources with a tag, we guarantee that all linear pattern variables are simultaneously used to consume all the scrutinee resources, since for any of scrutinee resources to be used by a linear pattern-bound var be used, the resources must be *Split* for the fragments corresponding to that Δ-var to be consumed, and, consequently, the remaining fragments have to be consumed through the other linear pattern-bound variables.

Exemplo; consider X, z e introd na alternaveice como Y e pat vars como Zs...

3.2.5 Linear Core Examples

Linear Mini-Core [6] lists examples of Core programs where semantic linearity must be understood in order for them to be well-typed. In this section, we show those examples in Linear Core (λ^π_Δ), briefly explaining why they are indeed well-typed.

Equations

The Linear Haskell function is compiled in Linear Core as

```
data C = Red | Green | Blue
f :: C ⊸ C ⊸ C
f Red q = q
f p Green = p
f Blue q = q
```

$$\begin{array}{l}\lambda p{:}_1 C\ q{:}_1 C.\ \textbf{case}\ p\ \textbf{of}\ p2{:}_{\{p\}}C \\ \{Red \to q \\ ;\_ \to \textbf{case}\ q\ \textbf{of}\ q2{:}_{\{q\}}C \\ \{Green \to p2 \\ ;\_ \to \textbf{case}\ p2\ \textbf{of}\ p3{:}_{\{p\}}C \\ \{Blue \to q2\}\}\}\end{array}$$

Unrestricted Fields

The following is well-typed: Let **data** K **where** $K : A \multimap B \to C$, and f:

$$\lambda x{:}_1\sigma.\ \textbf{case}\ x\ \textbf{of}\ z{:}_x\sigma\ \{K\ a\ b \to (z, b)\}$$

Wildcard

The following is ill-typed:

$$f = \lambda x \to \textbf{case}\ x\ \textbf{of}\ z\ \{\_ \to \mathit{True}\}$$

Duplication

The following is ill-typed:

$$\textbf{data}\ \mathit{Foo} = \mathit{Foo}\ A$$
$$f = \lambda x \to \textbf{case}\ x\ \textbf{of}\ z\ \{\mathit{Foo}\ a \to (z, a)\}$$

3.3 Metatheory

The λ^{π}_{Δ} system is sound: well-typed programs in Linear Core do not get *stuck*. Besides type safety (§ 3.3.3), we prove multiple optimising transformations preserve linearity (§ 3.3.4), and prove an auxiliary result regarding proof irrelevant resources, stating that a case alternative well-typed in a proof irrelevant context is also well-typed if proof irrelevant resources are substituted by an arbitrary environment of relevant resources. Additionally, we state our assumptions that outline an isomorphism between using a linear variable $x{:}_1\sigma$ and a Δ-variable $x{:}_\Delta\sigma$ that consumes existing resources Δ, for any Δ.

3.3.1 Assumptions

We use two main assumptions in our proofs, which are dual. First, a program well-typed with a linear variable ($x{:}_1\sigma$) is equivalently well-typed if that same linear variable were instead Δ-bound ($x{:}_\Delta\sigma$) with usage environment Δ, Δ were available in the linear context instead of the linear variable.

Assumption 1 ($1 \Rightarrow \Delta$)**.** *A linear variable can be moved to the unrestricted context as a Δ-var with usage environment Δ by introducing Δ in the linear resources*
If $\Gamma; \Delta', x{:}_1\sigma \vdash e : \sigma$ *then* $\Gamma[\Delta/x], x{:}_\Delta\sigma; \Delta, \Delta' \vdash e : \sigma$.

Second, a program well-typed with resources Δ and Δ-bound variable ($x{:}_\Delta\sigma$) is equivalently well-typed, as long as Δ is consumed through the use of x, if x is moved to the linear context, resources Δ are removed from the linear context, and occurrences of Δ whole in usage environments are substituted by x (occurrences of fragments of Δ in usage environments are unimportant since Δ was consumed whole by x, not by any of the fragment-using Δ-vars).

Assumption 2 ($\Delta \Rightarrow 1$)**.** *A Δ-variable can replace its usage environment Δ as a linear variable if Δ is decidedly consumed through it*
If $\Gamma, x{:}_\Delta\sigma; \Delta, \Delta' \vdash e : \sigma$ *and* Δ *is consumed through* $x{:}_\Delta\sigma$ *in* e *then* $\Gamma[x/\Delta]; \Delta', x{:}_1\sigma \vdash e : \sigma$.

We additionally state that unrestricted resources are equivalent to Δ-bound variables with an empty $(\cdot)$ usage environment:

Assumption 3 ($x{:}_\omega\sigma = x{:}_\cdot\sigma$)**.** *An unrestricted variable is equivalent to a Δ-var with an empty usage environment.*
$\Gamma, x{:}_\omega\sigma; \Delta \vdash e : \sigma$ *iff* $\Gamma, x{:}_\cdot\sigma; \Delta \vdash e : \sigma$

3.3.2 Irrelevance

As discussed above, proof irrelevant resources are resources that can only be consumed indirectly, and are used to type case expressions whose scrutinee is not in WHNF, essentially encoding that the scrutinee resources must be consumed through the case binder or the linear pattern-bound variables. As a case expression is evaluated, the scrutinee will eventually be in WHNF, which must then be typed with rule $Case_{\text{WHNF}}$. Crucially, these rules must "work together" in the system, in the sense that case expressions typed using the $Case_{\text{Not WHNF}}$ rule must also be well-typed after the scrutinee is evaluated to WHNF, which is then typed using the $Case_{\text{WHNF}}$ rule.

The *Irrelevance* lemma is required to prove preservation for that evaluation case. We need to prove that the alternatives of a case expression typed with proof irrelevant resources are still well-typed when the proof irrelevant resource is substituted by the scrutinee resources as it is evaluated to WHNF. In this sense, the *Irrelevance* lemma witnesses the soundness of typing a case alternative with proof irrelevant resources in a certain context with respect to typing the same expression with arbitrary resources (we note, however, typing an alternative with proof irrelevant resources is not complete wrt using arbitrary resources – a counter example needs only to use a resource directly).

Lemma 1 (Irrelevance)**.** *If* $\Gamma, z{:}_{[\Delta]}\sigma; [\Delta], \Delta' \vdash_{alt} \rho \to e :^{z}_{[\Delta]} \sigma \Rrightarrow \varphi$
then $\Gamma, z{:}_{\Delta^\dagger}\sigma; \Delta^\dagger, \Delta' \vdash_{alt} \rho \to e :^{z}_{\Delta^\dagger} \sigma \Rightarrow \varphi$*, for* any $\Delta^\dagger$

Intuitively, the lemma holds since proof irrelevant resources must be used through the case binder or pattern-bound variables. If we consistently replace the proof irrelevant resources both in the typing environment and in the usage environments containing them, the expression remains well-typed.

3.3.3 Type safety

We prove type safety of the Linear Core system via the standard type preservation and progress results. As is customary, we make use of multiple substitution lemmas, one for each kind of variable: unrestricted variables $x{:}_{\omega}\sigma$, linear variables $x{:}_{1}\sigma$, and Δ-bound variables $x{:}_{\Delta}\sigma$.

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

Type preservation states that a well-typed expression e that evaluates to e' remains well-typed under the same context: The proof is done by structural induction on the reductions $e \longrightarrow e'$ from the operational semantics. Most cases are straightforward and usually appeal to one or more of the substitution lemmas described below. The most interesting case is that of case expressions whose scrutinee can be further evaluated – we branch on whether the scrutinee becomes in WHNF, and invoke the *Irrelevance* lemma if so. This case guarantees that the separation of rules for treating scrutinees is consistent, in the sense that a well-typed case expression with a scrutinee not in WHNF remains well-typed after the scrutinee is evaluated to WHNF.

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* $e \longrightarrow e'$*.*

Progress states that the evaluation of a well-typed term does not block: Similarly, progress is proved by induction on typing.

Substitution Lemmas

The preservation and progress theorems depend on multiple substitution lemmas, one for each kind of variable, as is standard.

The linear substitution lemma states that a well-typed expression e with a linear variable x of type σ remains well-typed if occurrences of x in the e are replaced by an expression e' of the same type σ, and occurrences of x in the linear context and in usage environments of Δ-bound variables are replaced by the linear context Δ' used to type e':

Lemma 2 (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 Δ-variables in Γ by the linear variables in Δ'.

The substitution of the resource in the usage environments is illustrated by the following example. Consider the term **let** $y = use\ x$ **in** y where use and x are free variables: if we replace occurrences of x by e' (where $\Gamma; \Delta \vdash e : \sigma$), then the "real" usage environment of y goes from $\{x\}$ to Δ. If we don't update the usage environment of y accordingly, we'll ultimately be typing $y{:}_{\{x\}}\varphi$ with Δ instead of x, which is not valid.

The linear substitution lemma extends to case alternatives as well. The lemma for substitution of linear variables in case alternatives is similar to the linear substitution lemma, applied to the case alternative judgement.

Lemma 2.1 (Substitution of linear variables on case alternatives preserves typing)**.**
If $\Gamma; \Delta, x{:}_1\sigma \vdash_{alt} \rho \to e :^z_{\Delta_s} \sigma \Rightarrow \varphi$ *and* $\Gamma; \Delta' \vdash e' : \sigma$ *and* $\Delta_s \subseteq \Delta, x$
then $\Gamma\,[\Delta'/x]\,; \Delta, \Delta' \vdash_{alt} \rho \to e[e'/x] :^z_{\Delta_s[\Delta'/x]} \sigma \Rightarrow \varphi$

We further require that the environment annotated in the case alternative judgement, Δ_s, is a subset of the environment used to type the whole alternative $\Delta_s \subseteq \Delta$. In all occurrences of the alternative judgement (in $Case_{\text{WHNF}}$ and $Case_{\text{Not WHNF}}$), the environment annotating the alternative judgement is *always* a subset of the alternative environment.

The substitution lemma for unrestricted variables follows the usual formulation, with the added restriction (common to linear type systems) that the expression e' that is going to substitute the unrestricted variable x is typed on an empty linear environment:

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

Similarly, we also prove the substitution of unrestricted variables preserves types on an alternative case expression:

Lemma 3.1 (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* $\Gamma; \Delta \vdash_{alt} \rho \to e[e'/x] :^z_{\Delta_s} \sigma' \Rightarrow \varphi$

Finally, we introduce the lemma stating that substitution of Δ-bound variables by expressions of the same type preserves the type of the original expression. What distinguishes this lemma from traditional substitution lemmas is that the usage environment Δ of the variable x being substituted by expression e' must match exactly the typing environment Δ of e' and the environment of the original expression doesn't change with the substitution:

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

Intuitively, if x is well-typed with Δ in e, substituting x by an expression e' which is typed in the same environment Δ allows the distribution of resources Δ, Δ' used to type e across sub-derivations to remain unchanged. To prove the theorems, we don't need a "stronger" substitution of Δ-vars lemma (allowing arbitrary resources Δ'' to type e', as in other substitution lemmas), as we only ever substitute Δ-variables by expressions whose typing environment matches the variables usage environment. However, it is not obvious whether such a lemma is possible to prove for Δ-variables (e.g. let $\Gamma; \Delta \vdash e : \sigma$ and $\Gamma; \Delta' \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ x$, if we substitute e for x the resources Δ' are no longer consumed).

The Δ-substitution lemma on case alternatives reflects again that the typing environment of the expression substitution the variable must match its usage environment. We recall that $\Delta_s \subseteq \Delta, \Delta'$ states that the annotated environment is always contained in the typing environment, which is true of all occurrences of this judgement. An alternative formulation of this lemma could instead explicitly list Δ_s as part of the typing environment for the same effect:

Lemma 4.1 (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* $\Gamma; \Delta, \Delta' \vdash_{alt} \rho \to e[e'/x] :^{z}_{\Delta_s} \sigma' \Rightarrow \varphi$

The proofs for preservation, progress, irrelevance, and for the substitution lemmas are available in full in the appendix.

3.3.4 Optimisations preserve linearity

One of the primary goals of the Linear Core type system is being suitable for intermediate representations of optimising compilers for lazy languages with linear types. In light of this goal, we prove that *multiple optimising transformations* are type preserving in Linear Core, and thus preserve linearity.

The optimising transformations proved sound wrt Linear Core in this section have been previously explained and motivated in Section 2.6.2. Transformations are described by an arbitrary well-typed expression with a certain shape, on the left hand side (lhs) of the arrow $\Longrightarrow$, resulting in an expression on the right hand side (rhs) that we prove to be well-typed. For each transformation, we describe the intuition behind the transformation preserving linearity in our system.

Inlining

Inlining substitutes occurrences of a let-bound Δ-variable x with the expression e it is bound to, which determines its usage environment Δ. Intuitively, in the let body e', x can occur once or not at all: if x occurs, then the linear resources Δ used indirectly through x are used via e instead; if x does not occur, then the resources Δ are already used linearly in e' and the substitution is a no-op.

Theorem 3 (Inlining preserves types)**.**
If $\Gamma; \Delta, \Delta' \vdash \mathbf{let}\ x{:}_{\Delta}\sigma = e\ \mathbf{in}\ e' : \varphi$ *then* $\Gamma; \Delta, \Delta' \vdash \mathbf{let}\ x{:}_{\Delta}\sigma = e\ \mathbf{in}\ e'[e/x] : \varphi$

Proof.

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

(5) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash e'[e/x] : \varphi$ by (admissible) $Weaken_{\Delta}$
(6) $\Gamma; \Delta, \Delta' \vdash \mathbf{let}\ x{:}_{\Delta}\sigma = e\ \mathbf{in}\ e'[e/x] : \varphi$ by (let) (2,5)

□

β-reduction

β-reduction evaluates a λ-abstraction application by substituting the λ-bound variable x with the argument e' in the body of the λ-abstraction e. We consider two definitions of β-reductions, one that substitutes all occurrences of a variable by an expression, as in call-by-name, and other which creates a lazy let binding to share the result of computing the argument expression amongst uses of the variable, as in call-by-need.

The first kind of β-reduction on term abstractions can be seen to preserve linearity by case analysis. When the function is linear, the binding x is used exactly once in the body of the lambda, thus can be substituted by an expression typed with linear resources, since the expression is guaranteed to be used exactly once in place of x. The proof is direct by type preservation.

Theorem 4 (β-reduction preserves types)**.**
If $\Gamma; \Delta \vdash (\lambda x{:}_{\pi}\sigma.\ e)\ e' : \varphi$ *then* $\Gamma; \Delta \vdash e[e'/x] : \varphi$

Proof.

(1) $\Gamma; \Delta \vdash (\lambda x{:}_{\pi}\sigma.\ e)\ e' : \varphi$
(2) $(\lambda x{:}_{\pi}\sigma.\ e)e' \longrightarrow e[e'/x]$
(3) $\Gamma; \Delta \vdash e[e'/x] : \varphi$ by type preservation theorem (1,2)

□

We assume the β-reduction with sharing (i.e. the one that creates a let binding) is only applicable when the λ-abstraction has an unrestricted function type. Otherwise, the call-by-name β-reduction is always favourable, as we know the resource to be used exactly once and hence sharing would be counterproductive, and result in an unnecessary heap allocation. Consequently, the argument to the function must be unrestricted (hence use no linear resources) for the term to be well-typed, and so it vacuously follows that linearity is preserved by this transformation.

Theorem 5 (β-reduction with sharing preserves types)**.**
If $\Gamma; \Delta \vdash (\lambda x{:}_{\omega}\sigma.\ e)\ e' : \varphi$ *then* $\Gamma; \Delta \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e : \varphi$ *NB: We only apply this transformation when x is unrestricted, otherwise beta-reduction without sharing is the optimization applied.*

Proof.

(1) $\Gamma; \Delta \vdash (\lambda x{:}_{\omega}\sigma.\ e)\ e' : \varphi$
(2) $\Gamma; \Delta \vdash (\lambda x{:}_{\omega}\sigma.\ e) : \sigma \to_{\omega} \varphi$ by inv. on λE_{ω}
(3) $\Gamma; \cdot \vdash e' : \sigma$ by inv. on λE_{ω}
(4) $\Gamma, x{:}_{\omega}\sigma; \Delta \vdash e : \varphi$ by inv. on λI
(5) $\Gamma, x{:}_{\cdot}\sigma; \Delta \vdash e : \varphi$ by $x{:}_{\omega}\sigma = x{:}_{\cdot}\sigma$ lemma
(6) $\Gamma, \Gamma' \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e : \varphi$ by let (5,3)

□

Finally, β-reduction on multiplicity abstractions is also type preserving. The argument of the application is a multiplicity rather than an expression, so no resources are needed to type it, and since the body of the lambda must treat the multiplicity p as though it were a linear, the body uses the argument linearly regardless of the instantiation of p at π.

Theorem 6 (β-reduction on multiplicity abstractions preserves types)**.**
If $\Gamma; \Delta \vdash (\Lambda p.\ e)\ \pi : \varphi$ *then* $\Gamma; \Delta \vdash e[\pi/p] : \varphi$

Proof. Trivial by invoking preservation using the Λ application reduction and the assumption □

Case of known constructor

Case-of-known constructor is a transformation that essentially evaluates a case expression of a known constructor at compile time, by substituting in the matching alternative the case binder by the scrutinee and pattern variables by constructor arguments. Intuitively, either the case binder is used exactly once to consume the resources of the scrutinee, or the pattern components whose matching constructor argument uses linear resources are used exactly once, so one of the substitutions is a no-op. If the pattern variables are substituted by the matching scrutinee expressions, the expressions are still only used once, and if the case binder is substituted by the scrutinee, it is still used exactly once. The proof follows trivially from the preservation theorem.

Theorem 7 (Case-of-known-constructor preserves types)**.**
If $\Gamma; \Delta, \Delta' \vdash \mathbf{case}\ K\ \overline{e}\ \mathbf{of}\ z{:}_\Delta\sigma\ \{..., K\ \overline{x} \to e_i\} : \varphi$ *then* $\Gamma; \Delta, \Delta' \vdash e_i\overline{[e/x]}[K\ \overline{e}/z] : \varphi$

Proof.

(1) $\Gamma; \Delta, \Delta' \vdash \mathbf{case}\ K\ \overline{e}\ \mathbf{of}\ z{:}_\Delta\sigma\ \{..., K\ \overline{x} \to e_i\} : \varphi$
(2) $\mathbf{case}\ K\ \overline{e}\ \mathbf{of}\ z{:}_\Delta\sigma\ \{..., K\ \overline{x} \to e_i\} \longrightarrow e_i\overline{[e/x]}[K\ \overline{e}/z]$
(3) $\Gamma; \Delta, \Delta' \vdash e_i\overline{[e/x]}[K\ \overline{e}/z] : \varphi$ by preservation theorem (1,2)

□

Let floating

The let floating transformation move lazy let constructs in and out of other constructs, in order to further unblock other optimisations. In essence, since let bindings consume resources lazily (by introducing a Δ-variable with usage environment Δ, where Δ is the typing environment of the bound expression), we can intuitively move them around without violating linearity. We prove let floating transformations *full-laziness* and three *local-transformations* preserve types and linearity.

Theorem 8 (Full-laziness preserves types)**.**
If $\Gamma; \Delta, \Delta' \vdash \lambda y{:}_\pi\sigma.\ \mathbf{let}\ x{:}_\Delta\sigma = e\ \mathbf{in}\ e' : \varphi$ *and* y *does not occur in* e
then $\Gamma; \Delta, \Delta' \vdash \mathbf{let}\ x{:}_\Delta\sigma = e\ \mathbf{in}\ \lambda y{:}_\pi\sigma.\ e'$

Proof.

(1) $\Gamma; \Delta, \Delta' \vdash \lambda y{:}_{\pi}\sigma.\ \mathbf{let}\ x{:}_{\Delta}\sigma = e\ \mathbf{in}\ e' : \sigma' \to \varphi$
Subcase $\pi = 1$
(2) $\Gamma; \Delta, \Delta', y{:}_{1}\sigma \vdash \mathbf{let}\ x{:}_{\Delta}\sigma = e\ \mathbf{in}\ e' : \varphi$ by inv. on λI
(3) $\Gamma; \Delta \vdash e : \sigma$ by inv. on (let)
(4) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta', y{:}_{1}\sigma \vdash e' : \varphi$ by inv. on (let)
(5) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash \lambda y{:}_{1}\sigma.\ e' : \sigma' \to \varphi$ by (λI) (4)
(6) $\Gamma; \Delta, \Delta' \vdash \mathbf{let}\ x{:}_{\Delta}\sigma = e\ \mathbf{in}\ \lambda y{:}_{1}\sigma.\ e' : \sigma' \to \varphi$ by (let) $(3, 5)$
Subcase $\pi = \omega$
As above but x is put in the unrestricted context Γ

□

Theorem 9 (Local-transformations preserve types)**.** *There are three lemmas for local transformations:*

1. $\Gamma; \Delta \vdash (\mathbf{let}\ v = e\ \mathbf{in}\ b)\ a : \varphi \quad \Rightarrow \quad \Gamma; \Delta \vdash \mathbf{let}\ v = e\ \mathbf{in}\ b\ a : \varphi$
2. $\Gamma; \Delta \vdash \mathbf{case}\ (\mathbf{let}\ v = e\ \mathbf{in}\ b)\ \mathbf{of}\ \ldots : \varphi \quad \Rightarrow \quad \Gamma; \Delta \vdash \mathbf{let}\ v = e\ \mathbf{in}\ \mathbf{case}\ b\ \mathbf{of}\ \ldots : \varphi$
3. $\Gamma; \Delta \vdash \mathbf{let}\ x = (\mathbf{let}\ v = e\ \mathbf{in}\ b)\ \mathbf{in}\ c : \varphi \quad \Rightarrow \quad \Gamma; \Delta \vdash \mathbf{let}\ v = e\ \mathbf{in}\ \mathbf{let}\ x = b\ \mathbf{in}\ c : \varphi$

1. First local-transformation

Proof.

(1) $\Gamma; \Delta, \Delta', \Delta'' \vdash (\mathbf{let}\ x{:}_{\Delta}\sigma = e_1\ \mathbf{in}\ e_2)\ e_3 : \varphi$
(2) $\Gamma; \Delta, \Delta' \vdash \mathbf{let}\ x{:}_{\Delta}\sigma = e_1\ \mathbf{in}\ e_2 : \sigma' \to_{\pi} \varphi$ by inv. on 1
(3) $\Gamma; \Delta'' \vdash e_3 : \sigma'$ by inv. on 1
(4) $\Gamma; \Delta \vdash e_1 : \sigma$ by inv. on 2
(5) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash e_2 : \sigma' \to_{\pi} \varphi$ by inv. on 2
(6) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta', \Delta'' \vdash e_2\ e_3 : \varphi$ by $\lambda_{\pi}E$
(7) $\Gamma; \Delta, \Delta', \Delta'' \vdash \mathbf{let}\ x{:}_{\Delta}\sigma = e_1\ \mathbf{in}\ e_2\ e_3 : \varphi$ by let (4,6)

□

2. Second local-transformation

Proof.

(1) $\Gamma; \Delta, \Delta', \Delta'' \vdash \mathbf{case}\ \mathbf{let}\ x{:}_{\Delta}\sigma = e_1\ \mathbf{in}\ e_2\ \mathbf{of}\ z{:}_{\Delta,\Delta'}\sigma'\ \{\overline{\rho \to e_3}\} : \varphi$
(2) $\Gamma; \Delta, \Delta' \vdash \mathbf{let}\ x{:}_{\Delta}\sigma = e_1\ \mathbf{in}\ e_2 : \sigma'$ by inv. on 1
(3) $\Gamma; \Delta \vdash e_1 : \sigma$ by inv. on 2
(4) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash e_2 : \sigma'$ by inv. on 2
Subcase e_2 is in WHNF
(5) $\overline{\Gamma, z{:}_{\Delta,\Delta'}\sigma'; \Delta, \Delta', \Delta'' \vdash_{alt} \rho \to e_3 :^{z}_{\Delta,\Delta'} \sigma' \Rightarrow \varphi}$ by inv. on 1
(6) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta', \Delta'' \vdash \mathbf{case}\ e_2\ \mathbf{of}\ z{:}_{\Delta,\Delta'}\sigma'\ \{\overline{\rho \to e_3}\} : \varphi$ by CaseWHNF (4,5)
(7) $\Gamma; \Delta, \Delta', \Delta'' \vdash \mathbf{let}\ x{:}_{\Delta}\sigma = e_1\ \mathbf{in}\ \mathbf{case}\ e_2\ \mathbf{of}\ z{:}_{\Delta,\Delta'}\sigma'\ \{\overline{\rho \to e_3}\} : \varphi$ by Let (3,6)
Subcase e_2 is not in WHNF
(5) $\overline{\Gamma, z{:}_{[\Delta,\Delta']}\sigma'; [\Delta, \Delta'], \Delta'' \vdash_{alt} \rho \to e_3 :^{z}_{[\Delta,\Delta']} \sigma' \Rrightarrow \varphi}$ by inv. on 1
(6) $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta', \Delta'' \vdash \mathbf{case}\ e_2\ \mathbf{of}\ z{:}_{\Delta,\Delta'}\sigma'\ \{\overline{\rho \to e_3}\} : \varphi$ by CaseWHNF (4,5)
(7) $\Gamma; \Delta, \Delta', \Delta'' \vdash \mathbf{let}\ x{:}_{\Delta}\sigma = e_1\ \mathbf{in}\ \mathbf{case}\ e_2\ \mathbf{of}\ z{:}_{\Delta,\Delta'}\sigma'\ \{\overline{\rho \to e_3}\} : \varphi$ by Let (3,6)

□

3. Third local-transformation

Proof.

(1) $\Gamma, \Delta; \Delta', \Delta'' \vdash \textbf{let } x{:}_{\Delta,\Delta'}\sigma' = (\textbf{let } y{:}_{\Delta}\sigma' = e_1 \textbf{ in } e_2) \textbf{ in } e_3 : \varphi$
(2) $\Gamma; \Delta, \Delta' \vdash \textbf{let } y{:}_{\Delta}\sigma' = e_1 \textbf{ in } e_2 : \sigma'$ by inv. on 1
(3) $\Gamma, x{:}_{\Delta,\Delta'}\sigma'; \Delta, \Delta', \Delta'' \vdash e_3$ by inv. on 1
(4) $\Gamma; \Delta \vdash e_1 : \sigma$ by inv. on 2
(5) $\Gamma, y{:}_{\Delta}\sigma'; \Delta, \Delta' \vdash e_2 : \sigma'$ by inv. on 2
(6) $\Gamma, y{:}_{\Delta}\sigma'; \Delta, \Delta', \Delta'' \vdash \textbf{let } x{:}_{\Delta,\Delta'}\sigma' = e_2 \textbf{ in } e_3 : \varphi$ by Let (3,5) and Weaken
(7) $\Gamma; \Delta, \Delta', \Delta'' \vdash \textbf{let } y{:}_{\Delta}\sigma' = e_1 \textbf{ in let } x{:}_{\Delta,\Delta'}\sigma' = e_2 \textbf{ in } e_3 : \varphi$ by Let (4,6)

□

η-conversions

The η-conversion transformations are η-expansion and η-reduction. In both transformations, linearity is preserved since the resources used to type the function f do not change neither when the lambda and its argument are removed, nor when we add a lambda and apply f to the bound argument.

Theorem 10 (η-expansion preserves types)**.**
If $\Gamma; \Delta \vdash f : \sigma \to_\pi \varphi$ *then* $\Gamma; \Delta \vdash \lambda x.\ f\ x : \sigma \to_\pi \varphi$

Proof.

Subcase f is linear
(1) $\Gamma; \Delta \vdash f : \sigma \multimap \varphi$
(2) $\Gamma; x{:}_1\sigma \vdash x : \sigma$
(3) $\Gamma; \Delta, x{:}_1\sigma \vdash f\ x : \varphi$ by λE
(4) $\Gamma; \Delta \vdash (\lambda x{:}_1\sigma.\ f\ x) : \sigma \multimap \varphi$ by λI
Subcase f is unrestricted
As above but x is introduced in Γ and functions are unrestricted

□

Theorem 11 (η-reduction preserves types)**.**
If $\Gamma; \Delta \vdash \lambda x.\ f\ x : \sigma \to_\pi \varphi$ *then* $\Gamma; \Delta \vdash f : \sigma \to_\pi \varphi$

Proof.

(1) $\Gamma; \Delta \vdash (\lambda x{:}_\pi\sigma.\ f\ x) : \sigma \to_\pi \varphi$
Subcase $\pi = 1$
(2) $\Gamma; \Delta, x{:}_1\sigma \vdash f\ x : \varphi$ by inv. on λI
(3) $\Gamma; \Delta \vdash f : \sigma \to_1 \varphi$ by inv. on λE
Subcase $\pi = \omega$
As above but x is introduced in Γ

□

Binder Swap

The binder swap transformation applies to case expressions whose scrutinee is a single variable x, and it substitutes occurrences of x in the case alternatives for the case binder z. If x is a linear resource, x cannot occur in the case alternatives (as we conservatively consider variables are not in WHNF), so the substitution preserves types vacuously. Otherwise, x can be freely substituted by z, since z is also an unrestricted resource (it's usage environment is empty because x is unrestricted).

Theorem 12 (Binder-swap preserves types)**.**
If $\Gamma; \Delta \vdash \mathbf{case}\ x\ \mathbf{of}\ z\ \{\overline{\rho_i \to e_i}\} : \varphi$ *then* $\Gamma; \Delta \vdash \mathbf{case}\ x\ \mathbf{of}\ z\ \{\overline{\rho_i \to e_i[z/x]}\} : \varphi$

Proof.

Subcase x is linear
(1) $\Gamma; \Delta, x{:}_1\sigma \vdash \mathbf{case}\ x\ \mathbf{of}\ z{:}_{[x]}\sigma\ \{\overline{\rho \to e}\} : \varphi$
(2) $\Gamma; x{:}_1\sigma \vdash x : \sigma$ by inv. on $Case_{\text{Not WHNF}}$
(3) $\overline{\Gamma, z{:}_{[x]}\sigma; \Delta, [x{:}_1\sigma] \vdash_{alt} \rho \to e :^{z}_{[x]} \sigma \Rrightarrow \varphi}$ by inv. on $Case_{\text{Not WHNF}}$
(4) $\overline{\Gamma, z{:}_{[x]}\sigma; \Delta, [x{:}_1\sigma] \vdash_{alt} \rho \to e[z/x] :^{z}_{[x]} \sigma \Rrightarrow \varphi}$
by x cannot occur in e bc it's proof irrelevant
(5) $\Gamma; \Delta, x{:}_1\sigma \vdash \mathbf{case}\ x\ \mathbf{of}\ z{:}_{[x]}\sigma\ \{\overline{\rho \to e[z/x]}\} : \varphi$ by $Case_{\text{Not WHNF}}$
Subcase x is unrestricted
(1) $\Gamma, x{:}_\omega\sigma; \Delta \vdash \mathbf{case}\ x\ \mathbf{of}\ z{:}_{.}\sigma\ \{\overline{\rho \to e}\} : \varphi$
(2) $\Gamma, x{:}_\omega\sigma; \cdot \vdash x : \sigma$ by inv. on $Case_{\text{Not WHNF}}$
(3) $\overline{\Gamma, x{:}_\omega\sigma, z{:}_{.}\sigma; \Delta \vdash_{alt} \rho \to e :^{z} \sigma \Rrightarrow \varphi}$ by inv. on $Case_{\text{Not WHNF}}$
(4) $\Gamma, z{:}_{.}\sigma; \cdot \vdash z : \sigma$ by Var_Δ
(5) $\overline{\Gamma, z{:}_{.}\sigma; \Delta \vdash_{alt} \rho \to e[z/x] :^{z} \sigma \Rrightarrow \varphi}$ by unr. subst. lemma (3,4)
(6) $\Gamma, x{:}_\omega\sigma; \Delta \vdash \mathbf{case}\ x\ \mathbf{of}\ z{:}_{.}\sigma\ \{\overline{\rho \to e[z/x]}\} : \varphi$ by $Weaken_\omega$ and $Case_{\text{Not WHNF}}$

□

Reverse Binder Swap Considered Harmful

The reverse binder swap transformation substitutes occurrences of the case binder z in case alternatives by the scrutinee, when the scrutinee is a variable x.

Proposition 1 (Reverse-binder-swap preserves types)**.**
If $\Gamma; \Delta \vdash \mathbf{case}\ x\ \mathbf{of}\ z\ \{\overline{\rho_i \to e_i}\} : \varphi$ *then* $\Gamma; \Delta \vdash \mathbf{case}\ x\ \mathbf{of}\ z\ \{\overline{\rho_i \to e_i[x/z]}\} : \varphi$

This is exactly reverse from what the binder swap transformation does in hope of eliminating multiple uses of x so as to inline it. However, by using the scrutinee x instead of the case binder, we might be able to float out expressions from the alternative using the case binder. For example, we might float an expensive computation involving z out of the case alternative, where z is out of scope but x isn't:

$\lambda x.\ \mathbf{let\ rec}\ go\ y = \mathbf{case}\ x\ \mathbf{of}\ z\ \{(a,b) \to \ldots (expensive\ z) \ldots\}\ \mathbf{in}\ \ldots$
$\Longrightarrow_{\text{Reverse binder swap}}$
$\lambda x.\ \mathbf{let\ rec}\ go\ y = \mathbf{case}\ x\ \mathbf{of}\ z\ \{(a,b) \to \ldots (expensive\ x) \ldots\}\ \mathbf{in}\ \ldots$
$\Longrightarrow_{\text{Float out}}$
$\lambda x.\ \mathbf{let}\ t = expensive\ x\ \mathbf{in\ let\ rec}\ go\ y = \mathbf{case}\ x\ \mathbf{of}\ z\ \{(a,b) \to \ldots t \ldots\}\ \mathbf{in}\ \ldots$

If go is a loop, by applying the reverse binder swap we now only compute $expensive\ x$ once instead of in every loop iteration.

Despite GHC applying the reverse binder swap transformation to core programs during Core-to-Core optimisation passes, this optimisation violates linearity when considered as a transformation on Linear Core programs. In practice, the optimisation preserves linearity in Core when applied as part of the GHC transformation pipeline only due to the occurrence analyser being naive with regard to semantic linearity. Initially, it might seem as though an expression in which a variable x occurs both in the case scrutinee and in the alternatives is linear, for example:

$$\Gamma; x{:}_1\sigma \vdash \textbf{case}\ x\ \textbf{of}\ \_ \to x : \sigma$$

The reasoning is done by branching on whether x refers to an expression in WHNF or an unevaluated thunk.

- If x refers to an unevaluated expression, then scrutinizing it results in the expression bound by x to be evaluated to WHNF. In a subsequent use of x in the alternatives, x refers to the evaluated scrutinee in WHNF, which must be consumed. Since x is just another name (like the case binder) for the scrutinee in WHNF, we may use it instead of the case binder or pattern variables.

- If x already refers to an expression evaluated to WHNF, then scrutinizing it in a case expression is a no-op, thus we may use it again (in mutual exclusion with the case binder and pattern variables)

Even though, on its own, it makes intuitive sense that this example indeed uses x linearly, when considered as part of a complete type system, allowing this expression to be linear makes the system unsound.

We recall that β-reduction reduces an application of a linear function using call-by-name – if we know the argument is used exactly once, a binding to share the result of computing the argument is unnecessary, so we instead substitute the argument expression for the linearly-bound variable in the λ-body directly.

Consider the function application

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

where y is a free linear variable. Assuming $\lambda x.\ \textbf{case}\ x\ \textbf{of}\ \_ \to x$ is a linear function by the reasoning above, β-reduction transforms the application in $(\textbf{case}\ x\ \textbf{of}\ \_ \to x)[use\ y/x]$, i.e. $\textbf{case}\ use\ y\ \textbf{of}\ \_ \to use\ y$. Now, y is a linear variable *consumed* in the scrutinee of the case expression, yet it occurs in the body of the case alternative as well – linearity is violated by using the linear resource y twice.

Essentially, Linear Core would be unsound, and even duplicate resources, if the above kind of expressions, where linear variable scrutinees occur in the alternatives body, were well-typed, because of its interaction with the call-by-name reduction of linear functions. In this sense, the reverse binder swap is an optimisation that creates ill-typed expressions from well-typed ones, so it is deemed an invalid optimisation that doesn't preserve types in our system.

The reverse binder swap is not a problem in the GHC simplifier because of the weaker notion of linearity understood by occurrence analysis. Occurrence analysis is a static analysis pass which can be used to determine whether a lambda application can be β-reduced call-by-name, and $\textbf{case}\ x\ \textbf{of}\ \_ \to x$ is *not* seen as using x linearly by the analysis. Thus, β-reduction is done with call-by-need on such an expression. If the above example

were reduced with call-by-need:

$$(\lambda x.\ \textbf{case}\ x\ \textbf{of}\ \_ \to x)\ (use\ y)$$
$$\Longrightarrow_{\text{call-by-need } \beta\text{-reduction}}$$
$$\textbf{let}\ x = use\ y\ \textbf{in case}\ y\ \textbf{of}\ \_ \to y$$

Then the computation using y would be let-bound, and y used as a scrutinee variable, which is indeed an expression semantically linear in x.

Concluding, in being able to understand more programs as linear, our type system allows more expressions to be considered linear for β-reduction without a let-allocation, however, it makes reverse binder swap an invalid transformation since its output, when considered linear, might violate linearity when further optimised.

Case of Case

The case of case transformation applies to case expressions whose scrutinee is another case expression, and returns the innermost case expression transformed by repeating the outermost case expression in each alternative of the innermost case, scrutinizing the original alternative body.

Intuitively, since the scrutinee of the outermost case is not in WHNF, no resources from it can directly occur in the outermost alternatives. By moving the outermost alternatives inwards with a different scrutinee the alternatives remain well-typed because they are typed using either the case binder or the pattern bound variables, which, by the *Irrelevance* lemma, makes it well-typed for any scrutinee consuming arbitrary resources.

Theorem 13 (Case-of-case preserves types)**.**
If $\Gamma; \Delta, \Delta', \Delta'' \vdash \textbf{case case}\ e_c\ \textbf{of}\ z{:}_{\Delta}\sigma\ \{\overline{\rho_{c_i} \to e_{c_i}}\}\ \textbf{of}\ w{:}_{[\Delta,\Delta']}\sigma'\ \{\overline{\rho_i \to e_i}\} : \varphi$
then $\Gamma; \Delta, \Delta', \Delta'' \vdash \textbf{case}\ e_c\ \textbf{of}\ z{:}_{\Delta}\sigma\ \{\overline{\rho_{c_i} \to \textbf{case}\ e_{c_i}\ \textbf{of}\ w\ \{\overline{\rho_i \to e_i}\}}\} : \varphi$

Proof.

(1)

□

3.4 Linear Core as a GHC Plugin

In this section, we discuss our prototype implementation of Linear Core as a Core plugin for the Glasgow Haskell Compiler. The Linear Core Plugin typechecks linearity in all Core programs produced both by the desugarer and after each optimisation pass. The plugin successfully validates linearity of Core throughout compilation of linearity-heavy libraries, namely `linear-base` and `linear-smc`. Additionally, we discuss the implementation of the Linear Core type system directly in the Glasgow Haskell Compiler.

A implementação existe; link para o github; validei o linear-base (excepto multiplicty coercions, e tive successo pq a implementação validou); validei os exemplos do inicio escrevendo Core à mão; Syntax-directedness

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.

Discuss a bit syntax-directedness non existent in the system and that our implementation slightly tweaks it to be more syntax directed or something

Talk about using our plugin on linear-base and other code bases... If I can get a few more case studies it would be pretty good. But then it's imperative to also use -dlinear-lint and make sure my plugin rejects a few of the examples

3.4.1 Consuming tagged resources as needed

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.

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

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

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

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

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

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

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

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

(CaseWHNF)
$$\frac{\text{e is in } \textit{WHNF} \qquad \Gamma; \Delta \Vdash e : \sigma \gg \overline{\Delta_i} \qquad e \text{ matches } \rho_j \qquad \Gamma, z{:}_{\overline{\Delta_i}}\sigma; \overline{\Delta_i}, \Delta' \vdash_{alt} \rho_j \to e' :^z_{\overline{\Delta_i}} \sigma \mapsto \varphi \qquad \overline{\Gamma, z{:}_{[\Delta]}\sigma; [\Delta], \Delta' \vdash_{alt} \rho \to e' :^z_{[\Delta]} \sigma \Rrightarrow \varphi}}{\Gamma; \Delta, \Delta' \vdash \textbf{case } e \textbf{ of } z{:}_{\overline{\Delta_i}}\sigma\ \{\overline{\rho \to e'}\} : \varphi}$$

(CaseNot WHNF)
$$\frac{\text{e is not in } \textit{WHNF} \qquad \Gamma; \Delta \vdash e : \sigma \qquad \overline{\Gamma, z{:}_{[\Delta]}\sigma; [\Delta], \Delta' \vdash_{alt} \rho \to e' :^z_{[\Delta]} \sigma \Rrightarrow \varphi}}{\Gamma; \Delta, \Delta' \vdash \textbf{case } e \textbf{ of } z{:}_{[\Delta]}\sigma\ \{\overline{\rho \to e'}\} : \varphi}$$

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

(AltNWHNF)
$$\frac{\Gamma, \overline{x{:}_\omega\sigma}, \overline{y_i{:}_{\Delta_i}\sigma_i}^{\,n}; \Delta \vdash e : \varphi}{\Gamma; \Delta \vdash_{alt} K\ \overline{x{:}_\omega\sigma}, \overline{y_i{:}_1\sigma_i}^{\,n} \to e :^z_{\overline{\Delta_i}^{\,n}} \sigma \mapsto \varphi}$$

(AltNNot WHNF)
$$\frac{\Gamma, \overline{x{:}_\omega\sigma}, \overline{y_i{:}_{\Delta_i}\sigma_i}; \Delta \vdash e : \varphi \qquad \overline{\Delta_i} = \overline{\Delta_s \# K_i}^{\,n}}{\Gamma; \Delta \vdash_{alt} K\ \overline{x{:}_\omega\sigma}, \overline{y_i{:}_1\sigma_i}^{\,n} \to e :^z_{\Delta_s} \sigma \Rrightarrow \varphi}$$

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

(Alt_)
$$\frac{\Gamma; \Delta \vdash e : \varphi}{\Gamma; \Delta \vdash_{alt} \_ \to e :^z_{\Delta_s} \sigma \Rightarrow \varphi}$$

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

(WHNF$_K$)
$$\frac{\overline{\Gamma; \cdot \vdash e_\omega : \sigma} \qquad \overline{\Gamma; \Delta_i \vdash e_i : \sigma} \qquad \overline{\Delta_i} = \Delta}{\Gamma; \Delta \Vdash K\ \overline{e_\omega e_i} : \sigma \gg \overline{\Delta_i}}$$

(WHNF$_\lambda$)
$$\frac{\Gamma; \Delta \vdash \lambda x.\ e : \sigma}{\Gamma; \Delta \Vdash \lambda x.\ e : \sigma \gg \Delta}$$

Figure 3.3: Linear Core Type System

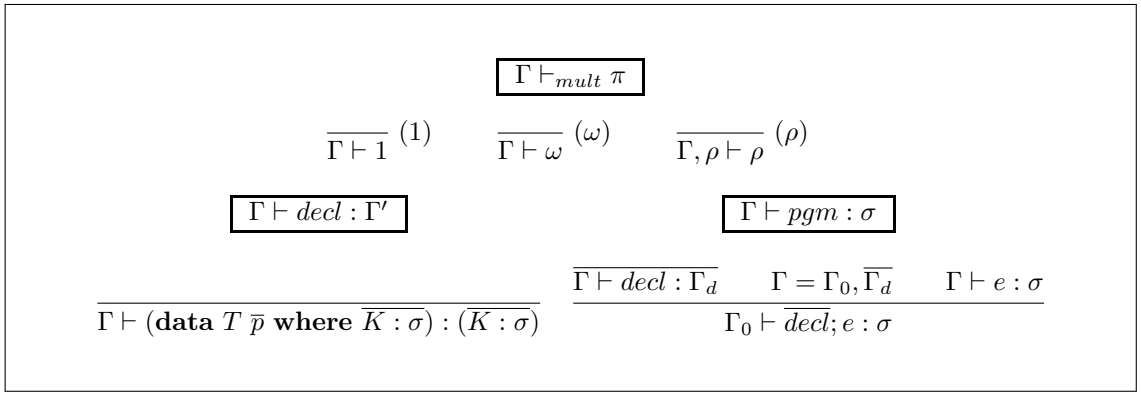

Figure 3.4: Linear Core Auxiliary Judgements

CHAPTER 4

Conclusion

Linear Core is an intermediate language with a type system system that understands (semantic) linearity in the presence of laziness, suitable for optimising compilers with these characteristics which leverage laziness and (possibly) linearity in its transformations.

In this chapter, review the literature related to our own work, highlighting Linear Core's novel contributions in light of the existing prominent works in the area, or how they otherwise compare, consider further research deemed out of the scope of our work and of the Linear Core type system (notably, we discuss so-called *multiplicity coercions* to handle the interaction between linearity and coercions, a key feature of Core which we left out our system), and conclude.

4.1 Related Work

In this section we discuss related work, namely, Linear Haskell [5], Linear Mini-Core [6], and linearity-influenced optimising transformations [39, 40, 5].

4.1.1 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 accommodate possible future features, in particular, its design is forwards compatible with affine and dependent types.

Linear Haskell [5] 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. Core's type system is unable to type *semantic* linearity of programs (in contrast to *syntactic* linearity), which result in Core rejecting linear programs resulting from optimising transformations that leverage the non-strict semantics of Core. Linear Core overcomes the limitations of Core's linear type system derived from Linear Haskell by understanding semantic linearity in the presence of laziness, and provably accepts multiple Core-to-Core passes. Linear Core, ultimately, can also be seen as a system that validates the programs written in Linear Haskell and compiled by GHC by guaranteeing (through typing) that linear resources are still used exactly once throughout the optimising transformations.

4.1.2 Linear Mini-Core

Linear Mini-Core [6] is a specification of linear types in Core as they were being implemented in GHC, and doubles as the (unpublished) precursor to our work. Linear Mini-Core first observes the incapacity of Core's type system to accept linear programs after transformations, and first introduces usage environments for let-bound variables with the same goal of Linear Core of specifying a linear type system for Core that accepts the optimising transformations.

We draw from Linear Mini-Core's the rule for non-recursive let expressions and how let-bound variables are annotated with a usage environment, however, our work further explores the interaction of laziness with linearity in depth, and diverges in rules for typing other constructs, notably, case expressions and case alternatives. Furthermore, unlike Mini-Core, we prove type safety of our system and that multiple optimising transformations, when applied to Linear Core programs, preserve linearity as understood by the system.

4.1.3 Linearity-influenced optimizations

Core-to-Core transformations appear in many works across the research literature [39, 40, 43, 37, 3, 36, 8, 44], all designed in the context of a typed language (Core) which does not have linear types. However, [39, 40, 5] 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 optimising 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 optimising transformations that currently don't take any linearity into account.

4.2 Future Work

In this section we highlight some avenues of further research. Briefly, these include *multiplicity coercions*, optimisations leveraging linearity, resource inference for usage environments, and ultimately using Linear Core in a mature optimising compiler with lazy

evaluation and linear types – the Glasgow Haskell Compiler. Lastly, we discuss the generalization of Linear Core to the surface Haskell language.

Multiplicity Coercions. Linear Core doesn't have type equality coercions, a flagship feature of GHC Core's type system. Coercions, briefly explained in Section 2.5, allow the Core intermediate language to encode a panoply of Haskell source type-level features such as GADTs, type families or newtypes. In Linear Haskell, multiplicities are introduced as annotations to function arrows which specify the linearity of the function. In practice, multiplicities are simply types of kind *Multiplicity*, where *One* and *Many* are the type constructors of the kind *Multiplicity*; multiplicity polymorphism follows from type polymorphism, where multiplicity variables are just type variables. Encoding multiplicities as types allows Haskell programs to leverage features available for types to naturally extend to multiplicities as well. Consequently, we might define, e.g., using a GADT *SBool* and a type family *If*, the function *dep* which is linear in the second argument if the first argument is *STrue* and unrestricted otherwise:

```
data SBool :: Bool → Type where
  STrue :: SBool True
  SFalse :: SBool False

type family If b t f where
  If True t _ = t
  If False _ f = f

dep :: SBool b → Int %(If b One Many) → Int
dep STrue x = x
dep SFalse _ = 0
```

This example is linear and should be accepted. However, the example is rejected by the GHC's Core type checker. Critically, Core doesn't currently understand so-called *multiplicity coercions*. Even though after matching on *STrue* we have access to a coercion from the function multiplicity m to 1 ($m \sim 1$), we cannot use this coercion to determine whether the usages of the linear resources match the multiplicity. Studying the interaction between coercions and multiplicities is a main avenue of future work for Linear Core.

Optimisations leveraging linearity. We only briefly mentioned how linearity can inform optimisations to produce more performant programs. We leave exploring optimisations unblocked by preserving linearity in the intermediate language with Linear Core as future work. Linearity influenced optimising transformations have been also discussed by Linear Haskell [5] and in [39, 40]. An obvious candidate is *inlining*, which is applied based on heuristics from information provided by the *cardinality analysis* pass that counts occurrences of bound variables. Linearity can be used to non-heuristically inform the inliner [5]. Additionally, we argue that in Linear Core accepting more programs as linear there are more chances to use linearity, in contrast to a linear type system which does not account for lazy evaluation and thus rejects more programs.

Usage environment resource inference. In Section 3.1, we explained that the linear resources used by a group of recursive bindings aren't obvious and must be consistent with each other (i.e. considering the mutually-recursive calls) as though the resources used by each binder are the solution to a set determined by the recursive bindings group. In Section 3.2, we further likened the challenge of determining usage environments for a

recursive group of bindings to a unification problem as that solved by the Hindley-Milner type inference algorithm [17] based on generating and solving constraints. Even though these are useful observations, our implementation of Linear Core uses a naive algorithm to determine the usage environments, thereby leaving as future work the design of a principled algorithm to determine the usage environments of recursive group of bindings.

Linear Core in the Glasgow Haskell Compiler. Linear Core is suitable as the intermediate language of an optimising compiler for a linear and lazy language such as Haskell Core, in that optimising transformations in Linear Core preserve types *and* linearity, since Linear Core understands (semantic) in the presence of laziness, unlike Core's current type system under which optimisations currently violate linearity. Integrating Linear Core in the Glasgow Haskell Compiler is one of the ultimate goals of our work. Core's current type system ignores linearity due to its limitation in understanding semantic linearity, and our work fills this gap and would allow Core to be linearly typed all throughout. A linearly typed Core that preserves linearity throughout the optimisation pipeline of GHC both validates the correctness of the compiler, which is already achieved to a great extent by preserving (non-linear) types, and informs optimisations, allowing the compiler to generate more performant programs.

Implementing Linear Core in GHC is a challenging endeavour, since we must account for all other Core features (e.g. strict constructor fields) and more optimisations. Despite our initiative in this direction[1], we leave this as future work.

Generalizing Linear Core to Haskell. Linear types, despite their compile-time correctness guarantees regarding resource management, impose a burden on programmers in being a restrictive typing discipline (witnessed, e.g., by Linear Constraints [46]). Linear Core eases the restrictions of linear typing by being more flexible in understanding linearity for lazily evaluated languages such as Haskell. In this sense, it is an avenue of future work to apply the ideas from Linear Core to the surface Haskell language.

4.3 Conclusion

Optimising compilers with a typed and lazy intermediate language with linear types (of which GHC is the prime example) leverage laziness to heavily transform and rewrite programs into simpler forms. However, these optimising transformations push the interaction between linearity and laziness to the limits where linearity can no longer be seen syntactically, despite being maintained semantically, in the sense that linear resources are still used exactly once when the optimised program is run.

In this work we explored linearity in the presence of laziness by example through the interactions of linear types with lazy (recursive) let bindings and case expressions that evaluate their scrutinee to Weak Head Normal. Most example programs were linear semantically, but not syntactically. We developed a linear type system, Linear Core, for an intermediate language akin to GHC Core, with laziness and linearity. In contrast to GHC Core's type system, or any other linear type system (to the best of our knowledge), our type system understands semantic linearity, and can thus correctly type a wider range of linear programs, as those explored in the semantic linearity examples. Crucially, we proved soundness of the type system, and proved multiple optimising transformations preserve linearity, despite most violating linearity in other linear type systems. Additionally,

[1] https://gitlab.haskell.org/ghc/ghc/-/issues/23218

we implemented Linear Core as a GHC plugin to further explore its suitability in the intermediate language of an optimising compiler.

Concluding, Linear Core is a suitable type system for linear, lazy, intermediate languages of optimising compilers such as GHC, as it understands linearity in the presence of laziness s.t. optimisations preserve types and linearity, and further unblocks optimisations influenced by linearity, e.g. inlining and call-by-name β-reduction for applications of (semantically) linear functions.

Bibliography

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

[2] Z. M. Ariola, J. Maraist, M. Odersky, M. Felleisen, and P. Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 233–246, New York, NY, USA, 1995. Association for Computing Machinery.

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

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

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

[6] J. Bernardy, R. Eisenberg, M. Boespflug, R. Newton, S. P. Jones, and A. Spiwack. Linear mini-core. https://gitlab.haskell.org/ghc/ghc/-/wikis/uploads/355cd9a03291a852a518b0cb42f960b4/minicore.pdf, 2020.

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

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

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

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

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

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

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

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

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

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

[17] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, page 207–212, New York, NY, USA, 1982. Association for Computing Machinery.

[18] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3):795–807, 1992.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[46] A. Spiwack, C. Kiss, J.-P. Bernardy, N. Wu, and R. A. Eisenberg. Linearly qualified types: Generic inference for capabilities and uniqueness. *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022.

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

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

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

APPENDIX A

Type Safety Proofs

A.1 Type Preservation

Theorem 14 (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: $\textbf{case } K\ \overline{e} \textbf{ of } z{:}_\Delta \sigma\ \{\ldots, K\ \overline{x{:}_\pi \sigma} \to e'\} \longrightarrow e'\overline{[e/x]}[K\ \overline{e}/z]$

(1) $\Gamma; \Delta, \Delta' \vdash \textbf{case } K\ \overline{e_\omega e_i} \textbf{ of } z{:}_\Delta \sigma\ \{\ldots, K\ \overline{w{:}_\pi \sigma} \to e'\} : \varphi$
(2) $K\ \overline{e_\omega e_i}$ is in WHNF by def. of WHNF
(3) $\Gamma; \Delta \Vdash K\ \overline{e_\omega e_i} : \sigma \gg \overline{\Delta_i}$ by inv. on $CaseWHNF$
(4) $\overline{\Gamma; \Delta_i \vdash e_i : \sigma'}$ by inv. on $WHNF_K$
(5) $\overline{\Gamma; \cdot \vdash e_\omega : \sigma'}$ by inv. on $WHNF_K$
(6) $\Delta = \overline{\Delta_i}$ by inv. on $WHNF_K$
(7) $\Gamma, z{:}_{\overline{\Delta_i}} \sigma; \overline{\Delta_i}, \Delta' \vdash_{alt} K\ \overline{w{:}_\pi \sigma} \to e' :^z_{\overline{\Delta_i}} \sigma \Mapsto \varphi$ by inv. on $CaseWHNF$
Subcase $K\ \overline{w{:}_\pi \sigma} = K\ \overline{x{:}_\omega \sigma}, \overline{y_i{:}_1 \sigma_i}$
(8) $\Gamma, z{:}_{\overline{\Delta_i}} \sigma, \overline{x{:}_\omega \sigma}, \overline{y_i{:}_{\Delta_i} \sigma_i}; \overline{\Delta_i}, \Delta' \vdash e' : \varphi$ by inv. on $AltN_{WHNF}$
(9) $\Gamma, z{:}_{\overline{\Delta_i}} \sigma, \overline{y_i{:}_{\Delta_i} \sigma_i}; \overline{\Delta_i}, \Delta' \vdash e'\overline{[e_\omega/x]} : \varphi$ by unr. subst (5,8)
(10) $\Gamma, z{:}_{\overline{\Delta_i}} \sigma; \overline{\Delta_i}, \Delta' \vdash e'\overline{[e_\omega/x][e_i/y_i]} : \varphi$ by Δ-subst (4,9)
(11) $\Gamma; \overline{\Delta_i}, \Delta' \vdash e'\overline{[e_\omega/x][e_i/y_i]}[K\ \overline{e_\omega e_i}/z] : \varphi$ by Δ-subst (3,10)
(12) $\Gamma; \Delta, \Delta' \vdash e'\overline{[e_\omega/x][e_i/y_i]}[K\ \overline{e_\omega e_i}/z] : \varphi$ by (6)
Subcase $K\ \overline{w{:}_\pi \sigma} = K\ \overline{x{:}_\omega \sigma}$
(8) $\Delta = \cdot$ by $\overline{e_i} = \cdot \Rightarrow \overline{\Delta_i} = \cdot$ and 6
(9) $\Gamma, z{:}_\cdot \sigma, x{:}_\omega \sigma; \Delta' \vdash e' : \varphi$ by inv. on $Alt0$ and def. of empty subst.
(10) $\Gamma, z{:}_\cdot \sigma; \Delta' \vdash e'\overline{[e_\omega/x]} : \varphi$ by unr. subst. (5,9)
(11) $\Gamma; \Delta' \vdash e'\overline{[e_\omega/x]}[K\ \overline{e_\omega}/z] : \varphi$ by Δ-subst (8,3,10)
(12) $\Gamma; \Delta, \Delta' \vdash e'\overline{[e_\omega/x]}[K\ \overline{e_\omega}/z] : \varphi$ by 8

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

(1) $e \longrightarrow e'$ by inversion on case reduction
(2) $\Gamma; \Delta, \Delta' \vdash$ **case** e **of** $z{:}_\Delta\sigma\ \{\rho_i \to e_i''\} : \varphi$
(3) e is not in WHNF since it evaluates further by (1) instead of a case alternative being evaluated
(4) $\Gamma; \Delta \vdash e : \sigma$
(5) $\overline{\Gamma, z{:}_{[\Delta]}\sigma; [\Delta], \Delta' \vdash_{alt} \rho \to e'' :^z_{[\Delta]} \sigma \Rrightarrow \varphi}$ by inv. on CaseNotWHNF
(6) $\Gamma; \Delta \vdash e' : \sigma'$ by i.h. (1,4)
Subcase e' is not in WHNF
(7) $\Gamma; \Delta, \Delta' \vdash$ **case** e' **of** $z{:}_\Delta\sigma\ \{\rho_i \to e_i'\} : \varphi$ by $CaseNotWHNF$
Subcase e' is in WHNF
(7) $\overline{\Gamma, z{:}_{\overline{\Delta_i}}\sigma; \overline{\Delta_i}, \Delta' \vdash_{alt} \rho \to e'' :^z_{\overline{\Delta_i}} \sigma \Mapsto \varphi}$ for any $\overline{\Delta_i}$ by (5) and by soundness of not whnf wrt whnf lemma
(8) $\Delta = \overline{\Delta_i}$ by $\overline{\Delta_i}$ can be anything
(9) $\Gamma; \Delta \Vdash e : \sigma \gg \overline{\Delta_i}$ by (8) and rhs of $\gg = \Delta$
(10) $\Gamma; \Delta, \Delta' \vdash$ **case** e' **of** $z{:}_\Delta\sigma\ \{\rho_i \to e_i'\} : \varphi$ by $CaseWHNF$ (7,9)

□

A.2 Progress

Theorem 15 (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
(6) $e_1 = K\ \overline{e}$ by canonical forms lemma
(7) e is in WHNF by (6)
(8) $\overline{\rho_i \to e_i}$ is a complete pattern by coverage checker
(9) **case** $K\ \overline{e}$ **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) **case** e **of** $z{:}.\sigma\ \{\overline{\rho_i \to e_1}\} \longrightarrow$ **case** e' **of** $z{:}.\sigma\ \{\rho_i \to e_i\}$ by ctx. case reduction

□

A.3 Irrelevance

Lemma 5 (Irrelevance)**.** *If* $\Gamma, z{:}_{[\Delta]}\sigma; [\Delta], \Delta' \vdash_{alt} \rho \to e :^{z}_{[\Delta]} \sigma \Rrightarrow \varphi$
then $\Gamma, z{:}_{\Delta^\dagger}\sigma; \Delta^\dagger, \Delta' \vdash_{alt} \rho \to e :^{z}_{\Delta^\dagger} \sigma \Rightarrow \varphi$*, for* any $\Delta^\dagger$

Proof. By structural induction on the case alternative typing derivation.

Case: $Alt\_$

(1) $\Gamma, z{:}_{[\Delta]}\sigma; [\Delta], \Delta' \vdash_{alt} \_ \to e :^{z}_{[\Delta]} \sigma \Rightarrow \varphi$
(2) $\Gamma, z{:}_{[\Delta]}\sigma; [\Delta], \Delta' \vdash e : \varphi$
(3) $[\Delta]$ is used through z since $[\Delta]$ can't otherwise be used and is introduced in this alternative uniquely (since we have multi-tier proof irrelevance, i.e. $[[\Delta]] \neq [\Delta]$)
(4) $\Gamma; z{:}_1\sigma, \Delta' \vdash e : \varphi$ by $\Delta \Rightarrow 1$ lemma (2,3)
(5) $\Gamma, z{:}_{\overline{\Delta_i}}\sigma; \overline{\Delta_i}, \Delta' \vdash e : \varphi$ by $1 \Rightarrow \Delta$ lemma (4)
(6) $\Gamma, z{:}_{\overline{\Delta_i}}\sigma; \overline{\Delta_i}, \Delta' \vdash_{alt} \_ \to e :^{z}_{\overline{\Delta_i}} \sigma \Rightarrow \varphi$ by $Alt\_$

Case: $Alt0$

(1) $\Gamma, z{:}_{[\Delta]}\sigma; [\Delta], \Delta' \vdash_{alt} K\ \overline{x{:}_\omega\sigma} \to e :^{z}_{[\Delta]} \sigma \Rightarrow \varphi$
(2) $\Gamma, z{:}.\sigma, \overline{x{:}_\omega\sigma}; \Delta' \vdash e : \varphi$ by inv. on $Alt0$ and def. of empty subst.
(3) $\Gamma, z{:}_{\overline{\Delta_i}}\sigma; \overline{\Delta_i}, \Delta' \vdash_{alt} K\ \overline{x{:}_\omega\sigma} \to e :^{z}_{\overline{\Delta_i}} \sigma \Rightarrow \varphi$ by $Alt0$ (2)

Case: $AltN_{\text{WHNF}}$

Not applicable since $\Rrightarrow$ is only generalized by $\Rightarrow$, not $\Mapsto$.

Case: $AltN_{\text{Not WHNF}}$ We prove the theorem for constructing both an $AltN_{\text{Not WHNF}}$ and a $AltN_{\text{WHNF}}$ from a proof-irrelevant $AltN_{\text{Not WHNF}}$, to prove the statement holds for any $\Rightarrow$ kind for $AltN$, rather than just $\Rrightarrow$ or $\Mapsto$.

(1) $\Gamma, z{:}_{[\Delta]}\sigma; [\Delta], \Delta' \vdash_{alt} K\ \overline{x{:}_{\omega}\sigma}, \overline{y_i{:}_1\sigma_i}^n \to e :^z_{[\Delta]} \sigma \Rrightarrow \varphi$
(2) $\Gamma, z{:}_{[\Delta]}\sigma, \overline{x{:}_{\omega}\sigma}, \overline{y_j{:}_{\Delta_j}\sigma_j}^n; [\Delta], \Delta' \vdash e : \varphi$ by inv. on $AltN_{\text{Not WHNF}}$
(3) $\overline{\Delta_j} = [\Delta] \# K_j$ by inv. on $AltN_{\text{Not WHNF}}$
Subcase $[\Delta]$ is consumed through z
(4) $\Gamma, \overline{x{:}_{\omega}\sigma}, \overline{y_j{:}_{\Delta_j}\sigma_j}^n; z{:}_1\sigma, \Delta' \vdash e : \varphi$ by $\Delta \Rightarrow 1$ lemma (2,subcase)
Subcase constructing $AltN_{\text{WHNF}}$ ($\mapsto$)
(5) $\Gamma, \overline{x{:}_{\omega}\sigma}, \overline{y_j{:}_{\Delta_i}\sigma_j}^n; z{:}_1\sigma, \Delta' \vdash e : \varphi$ by $\overline{y}$ does not consume resources, and $Weaken$
(6) $\Gamma, \overline{x{:}_{\omega}\sigma}, \overline{y_j{:}_{\Delta_i}\sigma_j}^n, z{:}_{\overline{\Delta_i}}\sigma; \overline{\Delta_i}, \Delta' \vdash e : \varphi$ by $1 \Rightarrow \Delta$ lemma (5)
(7) $\Gamma, z{:}_{\overline{\Delta_i}}\sigma; \overline{\Delta_i}, \Delta' \vdash_{alt} K\ \overline{x{:}_{\omega}\sigma}, \overline{y_i{:}_1\sigma_i}^n \to e :^z_{\overline{\Delta_i}} \sigma \mapsto \varphi$ by $AltN_{\text{WHNF}}$ (6)
Subcase constructing $AltN_{\text{Not WHNF}}$ ($\Rrightarrow$)
(5) $\Gamma, \overline{x{:}_{\omega}\sigma}, \overline{y_j{:}_{\overline{\Delta_i \# K_j}}\sigma_j}^n; z{:}_1\sigma, \Delta' \vdash e : \varphi$
by $\overline{y}$ does not consume resources, and $Weaken$
(6) $\Gamma, \overline{x{:}_{\omega}\sigma}, \overline{y_j{:}_{\overline{\Delta_i \# K_j}}\sigma_j}^n, z{:}_{\overline{\Delta_i}}\sigma; \overline{\Delta_i}, \Delta' \vdash e : \varphi$ by $1 \Rightarrow \Delta$ lemma (5)
(7) $\Gamma, z{:}_{\overline{\Delta_i}}\sigma; \overline{\Delta_i}, \Delta' \vdash_{alt} K\ \overline{x{:}_{\omega}\sigma}, \overline{y_i{:}_1\sigma_i}^n \to e :^z_{\overline{\Delta_i}} \sigma \Rrightarrow \varphi$ by $AltN_{\text{Not WHNF}}$ (6)
Subcase $[\Delta]$ is (fully) consumed by $\overline{y}$ (after splitting)
(4) $\Gamma, z{:}_{[\Delta]}\sigma, \overline{x{:}_{\omega}\sigma}; \overline{y_j{:}_1\sigma_j}^n, \Delta' \vdash e : \varphi$ by $\Delta \Rightarrow 1$ lemma (2, subcase)
(5) $\Gamma, z{:}_{\overline{\Delta_i}}\sigma, \overline{x{:}_{\omega}\sigma}; \overline{y_j{:}_1\sigma_j}^n, \Delta' \vdash e : \varphi$ by $Weaken$ and z does not consume resources
Subcase constructing $AltN_{\text{WHNF}}$ ($\mapsto$)
(6) $\Gamma, z{:}_{\overline{\Delta_i}}\sigma, \overline{x{:}_{\omega}\sigma}, \overline{y_j{:}_{\Delta_i}\sigma_j}^n; \overline{\Delta_i}, \Delta' \vdash e : \varphi$ by $1 \Rightarrow \Delta$ lemma (5)
(7) $\Gamma, z{:}_{\overline{\Delta_i}}\sigma; \overline{\Delta_i}, \Delta' \vdash K\ \overline{x{:}_{\omega}\sigma}, \overline{y_i{:}_1\sigma_i}^n \to e :^z_{\overline{\Delta_i}} \sigma \mapsto \varphi$ by $AltN_{\text{WHNF}}$
Subcase constructing $AltN_{\text{Not WHNF}}$ ($\Rrightarrow$)
(6) $\Gamma, z{:}_{\overline{\Delta_i}}\sigma, \overline{x{:}_{\omega}\sigma}, \overline{y_j{:}_{\overline{\Delta_i \# K_j}}\sigma_j}^n; \overline{\Delta_i}, \Delta' \vdash e : \varphi$ by $1 \Rightarrow \Delta$ lemma (5)
(7) $\Gamma, z{:}_{\overline{\Delta_i}}\sigma; \overline{\Delta_i}, \Delta' \vdash K\ \overline{x{:}_{\omega}\sigma}, \overline{y_i{:}_1\sigma_i}^n \to e :^z_{\overline{\Delta_i}} \sigma \Rrightarrow \varphi$ by $AltN_{\text{Not WHNF}}$

□

A.4 Substitution Lemmas

Lemma 6 (Substitution of linear variables preserves typing)**.** *If* $\Gamma; \Delta, x{:}_1\sigma \vdash e : \varphi$ *and* $\Gamma; \Delta' \vdash e : \sigma$ *then* $\Gamma\,[\Delta'/x]\,; \Delta, \Delta' \vdash e[e'/x] : \varphi$

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) $\overline{e[e'/x] = e}$ 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 6.1 (Substitution of linear variables on case alternatives preserves typing)**.**
If $\Gamma;\Delta,x{:}_1\sigma \vdash_{alt} \rho \to e :^{z}_{\Delta_s} \sigma \Rightarrow \varphi$ *and* $\Gamma;\Delta' \vdash e' : \sigma$ *and* $\Delta_s \subseteq \Delta, x$
then $\Gamma\,[\Delta'/x]\,;\Delta,\Delta' \vdash_{alt} \rho \to e[e'/x] :^{z}_{\Delta_s[\Delta'/x]} \sigma \Rightarrow \varphi$

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

Case: $AltN_{WHNF}$

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

Case: $AltN_{NotWHNF}$

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

Case: *Alt*0 This is one of the most interesting proof cases, and challenging 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.
(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 $Alt0$
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 7 (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) $\Gamma, x{:}_\omega\sigma; \Delta, \Delta' \vdash \textbf{let rec } \overline{y{:}_\Delta\sigma' = e} \textbf{ in } e'' : \varphi$
(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) $\overline{\Gamma, x{:}_\omega\sigma, \overline{y{:}_\Delta\sigma'}; \Delta \vdash e : \sigma'}$ by inversion on *LetRec*
(5) $\Gamma, \overline{y{:}_\Delta\sigma'}; \Delta, \Delta' \vdash e''[e'/x] : \varphi$ by induction hypothesis (2,3)
(6) $\overline{\Gamma, \overline{y{:}_\Delta\sigma'}; \Delta \vdash e[e'/x] : \sigma'}$ by induction hypothesis (2,4)
(7) $\Gamma; \Delta, \Delta' \vdash \textbf{let rec } \overline{y{:}_\Delta\sigma' = e[e'/x]} \textbf{ in } e''[e'/x] : \varphi$ by *LetRec* (5,6)
(8) $(\textbf{let rec } \overline{y{:}_\Delta\sigma' = e} \textbf{ in } e'')[e'/x] = (\textbf{let rec } \overline{y{:}_\Delta\sigma' = e[e'/x]} \textbf{ in } e''[e'/x])$

Case: *CaseWHNF*

(1) $\Gamma; \cdot \vdash e' : \sigma$
(2) $\Gamma, x{:}_\omega\sigma; \Delta, \Delta' \vdash \textbf{case } e \textbf{ of } z{:}_\Delta\sigma' \ \{\overline{\rho \to e''}\} : \varphi$
(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) $\Gamma; \Delta, \Delta' \vdash \textbf{case } e[e'/x] \textbf{ of } z{:}_\Delta\sigma' \ \{\overline{\rho \to e''[e'/x]}\} : \varphi$

Case: *CaseNotWHNF*

(1) $\Gamma; \cdot \vdash e' : \sigma$
(2) $\Gamma, x{:}_\omega\sigma; \Delta, \Delta' \vdash \textbf{case } e \textbf{ of } z{:}_{[\Delta]}\sigma' \ \{\overline{\rho \to e''}\} : \varphi$
(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) $\Gamma; \Delta, \Delta' \vdash \textbf{case } e[e'/x] \textbf{ of } z{:}_{[\Delta]}\sigma' \ \{\overline{\rho \to e''[e'/x]}\} : \varphi$

□

Lemma 7.1 (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: $AltN_{WHNF}$ (trivial induction)

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

Case: $AltN_{NotWHNF}$ (trivial induction)

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

Case: $Alt0$

(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 $Alt0$

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 8 (Substitution of Δ-variables 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 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 8.1 (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: $AltN_{WHNF}$ (trivial induction)

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

Case: $AltN_{NotWHNF}$ (trivial induction)

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

□

A.5 Assumptions

Assumption 4 ($\Delta \Rightarrow 1$)**.** *A Δ-variable can replace its usage environment Δ as a linear variable if Δ is decidedly consumed through it*
If $\Gamma, x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash e : \sigma$ and Δ is consumed through $x{:}_{\Delta}\sigma$ in e then $\Gamma[x/\Delta]; \Delta', x{:}_{1}\sigma \vdash e : \sigma$.

Assumption 5 ($1 \Rightarrow \Delta$)**.** *A linear variable can be moved to the unrestricted context as a Δ-var with usage environment Δ by introducing Δ in the linear resources*
If $\Gamma; \Delta', x{:}_{1}\sigma \vdash e : \sigma$ then $\Gamma[\Delta/x], x{:}_{\Delta}\sigma; \Delta, \Delta' \vdash e : \sigma$.

Assumption 6 ($x{:}_{\omega}\sigma = x{:}_{.}\sigma$)**.** *An unrestricted variable is equivalent to a Δ-var with an empty usage environment.*
$\Gamma, x{:}_{\omega}\sigma; \Delta \vdash e : \sigma$ *iff* $\Gamma, x{:}_{.}\sigma; \Delta \vdash e : \sigma$