École doctorale IAEM Lorraine

Formalization and Implementation of Safe Destination Passing in Functional Programming Languages

THÈSE

présentée et soutenue publiquement le April 2025

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Thomas Bagrel

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Résumé

TBD.

Abstract

Destination-passing style programming introduces destinations, which represent the address of a write-once memory cell. Those destinations can be passed as function parameters, and thus enable the caller of a function to keep control over memory management: the body of the called function will just be responsible of filling that memory cell. This is especially useful in functional programming languages, in which the body of a function is typically responsible for allocation of the result value.

Programming with destination in Haskell is an interesting way to improve performance of critical parts of some programs, without sacrificing memory guarantees. Indeed, thanks to a linearly-typed API I present, a write-once memory cell cannot be left uninitialized before being read, and is still disposed of by the garbage collector when it is not in use anymore, eliminating the risk of uninitialized read, memory leak, or double-free errors that can arise when memory is managed manually.

In this article, I present an implementation of destinations for Haskell, which relies on so-called compact regions. I demonstrate, in particular, a simple parser example for which the destination-based version uses 35% less memory and time than its naive counterpart for large inputs.

Destination passing —aka. out parameters— is taking a parameter to fill rather than returning a result from a function. Due to its apparent imperative nature, destination passing has struggled to find its way to pure functional programming. In this paper, we present a pure core calculus with destinations. Our calculus subsumes all the existing systems, and can be used to reason about their correctness or extension. In addition, our calculus can express programs that were previously not known to be expressible in a pure language. This is guaranteed by a modal type system where modes are used to represent both linear types and a system of ages to manage scopes. Type safety of our core calculus was largely proved formally with the Coq proof assistant.

Remerciements

Les remerciements.

Je dédie cette thèse à TBD.

Contents

List of Figures

List of Listings

Introduction

1 Memory management: a core design axis for programming languages

Over the last fifty years, programming languages have evolved into indispensable tools, profoundly shaping how we interact with computers and process information across almost every domain. Much like human languages, the vast variety of programming languages reflects the diversity of computing needs, design philosophies, and objectives that developers and researchers have pursued. This diversity is a response to specialized applications, user preferences, and the continuous search for improvements in speed, efficiency, and expressiveness. Today, hundreds of programming languages exist, each with unique features and tailored strengths, making language selection a nuanced process for developers.

At a high level, programming languages differ in how they handle core aspects of data and program execution, and they can be classified along several key dimensions. These dimensions often reveal the underlying principles of the language and its suitability for different types of applications. Some of the main characteristics that distinguish programming languages include:

- how the user communicates intent to the computer — whether through explicit step-by-step instructions in procedural languages or through a more functional/declarative style that emphasizes what to compute rather than how;
- the organization and manipulation of data — whether through object-oriented paradigms that encapsulate domain data within objects than can interact with each other or through simpler data structures that can be operated on and dissected by functions and procedures;
- the level of abstraction — a higher-level language abstracts technical details to enable more complex functionality in fewer lines of code, while lower-level languages provide more control over the environment and execution of a task;
- the management of memory — whether memory allocation and deallocation are automatic, handled by the language itself, or require explicit intervention from the programmer;
- side effects and exceptions — whether they can in represented by the language, caught, and/or manipulated.

Among these, memory management is one of the most critical, yet often less visible, aspects of programming language design. Every program requires memory to store data and instructions, and this memory must be managed efficiently to prevent errors and maintain performance. Typically, programs operate with input sizes and data structures that can vary greatly, requiring dynamic memory allocation to ensure smooth execution. Therefore, the way a language handles memory management impacts not only how programmers write and structure their code but also the scope of features the language can support.

High-level languages, such as Python or Java, often manage memory automatically, through garbage collection. With automatic garbage collection (thanks to a tracing or reference-counting garbage collector), memory allocation and deallocation happen behind the scenes, freeing developers from the complexities of manual memory control. This automatic memory management simplifies programming, allowing developers to focus on functionality and making the language more accessible for rapid development. Although garbage collection is decently fast overall, it can be slow for some specific use-cases, and is also dreaded for its unpredictable overhead or pauses in the program execution.

In contrast, low-level languages like C or Zig tend to provide developers with direct control over memory allocation and deallocation. It indeed allows for greater optimization and efficient resource usage, particularly useful in systems programming or performance-sensitive applications. This control, however, comes with increased risk; errors like memory leaks, buffer overflows, and dangling pointers can lead to instability and security vulnerabilities if not carefully detected and addressed.

Interestingly, some languages defy these typical categorizations by combining high-level features with rigorous memory management. Such languages let the user manage resource lifetimes explicitly while taking the responsibility of allocating and deallocating memory at the start and end of each resource's lifetime. The most well-known examples are smart pointers in C++ and the ownership model in Rust, whose founding principles are also knew as *Scope-Bound Resource Management* (SBRM) or *Resource Acquisition is Initialization* (RAII). Initially applicable only to stack-allocated objects in early C++, SBRM evolved with the introduction of smart pointers for heap-allocated objects in the early 2000s. These tools have since become fundamental to modern C++ and Rust, significantly improving memory safety. In particular, Rust provides safety guarantees comparable to those of garbage-collected languages but without the garbage collection overhead, at the cost of a steep learning curve. This makes Rust suitable for both high-level application programming and low-level systems development, bridging a gap that was traditionally divided by memory management style.

Ultimately, as we've seen, memory management is more than a technical detail; it is a foundational characteristic that influences not only the efficiency and reliability of the language but also the range of possible features and applications.

2 Destination passing style: taking roots in the old imperative world

In systems programming, where fine control over memory is essential, the same memory buffer is often (re)used several times, in order to save up memory space and decrease the number of costly calls to the memory allocator. As a result, we don't want intermediary functions to allocate memory for the result of their computations. Instead, we want to provide them with a memory address in which to write their result, so that we can decide to reuse an already allocated memory slot or maybe write to a memory-mapped file, or RDMA buffer...

In practice, this means that the parent scope manages not only the allocation and deallocation of a function's inputs, but also those of *the function's outputs*. Output slots are passed to functions as pointers (or mutable references in higher-level languages), allowing the function to write directly into memory managed by the caller. These pointers, called out parameters or *destinations*, specify the exact memory location where the function should store its output.

This idea forms the foundation of destination-passing style programming (DPS): instead of letting a function allocate memory for its outputs, the caller provides it with write access to a pre-allocated memory space that can hold the function's results.

More control over memory Assigning memory allocation responsibility to the caller, rather than the callee, makes functions more flexible to use. Indeed, in destination-passing style programming (DPS), memory allocation gets decoupled from the actual function logic. This holds true even in the functional interpretation of DPS that will follow: destination-passing style is strictly more general than when functions allocate memory for their results themselves.

Also, DPS offers additional performance benefits. For example, a large memory block can be allocated in advance and divided into smaller segments, each designated as an output slot for a specific function call. This minimizes the number of separate allocations required, which is often a performance bottleneck, especially in contexts where memory allocation is intensive.

3 Functional programming languages

Functional programming languages are often seen as the opposite end of the spectrum from system languages.

Functional programming lacks a single definition, but most agree that a functional language:

- supports lambda abstractions, aka. anonymous functions, as first-class values, allowing functions to be stored, passed as parameters, and to capture parts of their parent environment (closures);
- emphasizes expressions over statements, where each instruction produces a value of a specific type;
- builds complex expressions by applying and composing functions rather than chaining statements.

From these principles, functional languages tend to favor immutable data structures and a declarative style, where new data structures are defined by specifying their differences from existing ones, often relying on structural sharing and persistent data structures instead of direct mutation.

Since "everything is an expression" in functional languages, they are particularly amenable to mathematical formalization. This is no accident: many core concepts in functional programming originate in mathematics, such as lambda calculus and its extensions. Lambda calculus, a minimal yet fully expressive model of computation, is as powerful as a Turing machine (which is a rather empirical model of computation) but also connects closely with formal proofs through the Curry-Howard isomorphism.

Despite this, many functional languages still permit expressions with side effects. Side effects encompass all the observable actions a program can take on its environment, e.g. writing to a file, or to the console, or altering memory. Side effects are hard to reason about, especially if any expression of the language is likely to emit one, but they are in fact a very crucial piece of any programming language: without them, there is no way for a program to communicate its results to the "outside".

Pure functional languages In response, some languages enforce a stricter approach by disallowing side effects until the boundaries of the main program, a concept known as *pure functional programming*. Here, all programmer-defined expressions compute and return values without side effects. Instead, the *intention* of performing side effects (like printing to the console) can be represented as a value, which only the language runtime evaluates to trigger the intended side effect.

This restriction provides substantial benefits. Programs can be wholly modeled as mathematical functions, making reasoning about behavior easier and allowing for more powerful analysis of a program's behavior.

A key property of pure functional languages is *referential transparency*: any function call can be substituted with its result without altering the program's behavior. This property is obviously absent in languages like C; for example, replacing `write(myfile, string, 12)` with the value `12` (its return if the write is successful) would not produce the intended behavior: the latter would not produce any write effect at all.

This ability to reason about programs as pure functions greatly improves the predictability of programs, especially across library boundaries, and overall improves software safety. Pure functional languages also support easier and more efficient application of formal methods, enabling stronger guarantees with less effort.

Memory management in functional languages Explicit memory management is inherently *impure*, as modifying memory in a way that can later be inspected *is* a side effect. Consequently, most functional languages, even those that aren't fully *pure* like OCaml or Scala, tend to rely on a garbage collector to abstract memory management and access away from the user.

However, this doesn't mean that functional languages must entirely forgo memory control; it simply means that memory control cannot involve explicit instructions like `malloc` or `free`. Instead, memory management must remain orthogonal to the "business logic", in other terms, expressions that carries the real meaning of the program.

In practice, it requires that memory management should not affect the value of an expression being computed within the language. One way to achieve this is by using annotations attached to specific functions or blocks of code, indicating how or where memory should be allocated. Another approach is to employ a type system with *modes* that specify how values are managed in memory (as in the recent modal development for Ocaml [Lor+24a]). Additionally, some languages use special functions, like `oneShot` in Haskell, which do not affect the result of the expression they wrap around, but carry special significance for the compiler in managing memory.

There is also another possible solution. We previously mentioned that side effects — prohibited in pure functional languages — are any modifications of the environment or memory that are *observable* by the user. What if we allow explicit memory management expressions while ensuring that these (temporary) changes to memory or the environment remain unobservable?

This is the approach we will adopt, allowing for finer-grained memory control while upholding the principles of purity.

4 Functional structures with holes: pioneered by Minamide and quite active since

In most contexts, functional languages with garbage collectors efficiently manage memory without requiring programmer intervention. However, a major limitation arises from the *immutability* characteristic common to most functional languages, which restricts us to constructing data structures directly in their final form. That means the programmer has to give an initial value to every field of the structure, even if no meaningful value for them has been computed yet. And later, any update beyond simply expanding the original structure requires creating a (partial) copy of that structure. This incur load on the garbage collector.

As a result, algorithms that generate large structures — whether by reading from an external source or transforming an existing structure — might create many intermediate structures, each representing a temporary processing state. While immutability has advantages, in this case, it can become an obstacle to optimizing performance.

To lift this limitation, Minamide's work[Min98] introduces an extension for functional languages that allows to represent *yet incomplete* data structures, that can be extended and completed later. Incomplete structures are not allowed to be read until they are completed — this is enforced by the type system — so that the underlying mutations that occur while updating the structure from its incomplete state to completion are hidden from the user, who can only observe the final result. That way, it preserves the feel of an immutable functional language and doesn't break purity per se. This method of making imperative or impure computations opaque to the user by making sure that their effects remain unobservable to the user thanks to type-level guarantees, is central to the approach developed in this document.

In Minamide's work, incomplete structures, or *structures with a hole*, are represented by hole abstractions — essentially pure functions that take the missing component of the structure as an argument and return the completed structure. In other terms, it represents the pending construction of a structure, than can theoretically only be carried out when all its missing pieces have been supplied. Hole abstractions can also be composed, like functions: $(\boldsymbol{\lambda} x \mapsto 1 :: x) \circ (\boldsymbol{\lambda} y \mapsto 2 :: y) \rightsquigarrow \boldsymbol{\lambda} y \mapsto 1 :: 2 :: y$. Behind the scenes however, each hole abstraction is not represented in memory by a function object, but rather by a pair of pointers, one read pointer to the root of the data structure that the hole abstraction describes, and one write pointer to the yet unspecified field in the data structure. Composition of two holes abstractions can be carried out immediately, with no need to wait for the missing value of the second hole abstraction, and results, memory-wise, in the mutation of the previously missing field of the first one, to point to the root of the second one.

In Minamide's system, the (pointer to the) hole and the structure containing it are inseparable, which limits an incomplete structure to having only a single hole. For these reasons, I would argue that Minamide's approach does not yet qualify as destination-passing style programming. Nonetheless, it is one of the earliest examples of a pure functional language allowing data structures to be passed with write permissions while preserving key language guarantees such as memory safety and purity.

This idea has been refreshed and extended more recently with [LL23] and [Lor+24b].

To reach true destination-passing style programming however, we need a way to refer to the hole(s) of the incomplete structure without having to pass around the structure that has the hole(s). This in fact hasn't been explored much in functional settings so far. [BCS21] is probably the closest existing work on this matter, but destination-passing style is only used at the intermediary language level, in the compiler, to do optimizations, so the user can't make use of DPS in their code.

5 Unifying and formalizing Functional DPS frameworks

What I'll develop on in this thesis is a functional language in which *structure with holes* are a first-class type in the language, as in [Min98], but that also integrate first-class *pointer to these holes*, aka. *destinations*, as part of the user-facing language.

We'll see that combining these two features give us performance improvement of some functional algorithms like in [BCS21] or [LL23], but we also get some extra expressiveness that is usually a privilege of imperative languages only. It should be indeed the first instance of a functional language that supports user-facing write pointers to make flexible and efficient data structure building!

In fact, I'll design a formal language in which *structures with holes* are the basis component for building any data structure, breaking from the traditional way of building structures in functional languages using data constructor. I'll also demonstrate how these ideas can be implemented in a practical functional language and that predicted benefits of the approach are mostly preserved in this implementation.

The formal language will also aim at providing a framework that encompass most existing work on functional DPS, and can be used to prove that earlier work is indeed sound.

6 Linear types: a tool to reconcile write pointers and immutability

We cannot simply introduce imperative write pointers into a functional setting and hope for the best. Instead, we need tools to ensure that only controlled mutations occur. The goal is to allow a structure to remain partially uninitialized temporarily, with write pointers referencing these uninitialized portions. However, once the structure has been fully populated, with a value assigned to each field, it should become immutable to maintain the integrity of the functional paradigm. Put simply, we need a write-once discipline for fields within a structure, which can be mutated through their associated write pointers aka. *destinations.* While enforcing a single-use discipline for destinations at runtime would be unwieldy, we can instead deploy static guarantees through the type system to ensure that every destination will be used exactly once.

In programming language theory, when new guarantees are required, it's common to design a type system to enforce them, ensuring the program remains well-typed. This is precisely the approach I'll take here, facilitated by an established type system — *linear types* — which can monitor resource usage, especially the number of times a resource is used.

Linear Logic, introduced by Jean-Yves Girard in the 1980s, refines classical and intuitionistic logic by restricting certain rules of deduction, specifically *weakening* and *contraction*. In sequent calculus, these rules allow assumptions to be duplicated or discarded freely. Linear Logic, on the other hand, eliminates these rules to enforce a stricter form of logical reasoning in which each assumption must be used exactly once.

Through the Curry-Howard isomorphism, intuitionistic linear logic becomes the linear simply typed λ-calculus. It inherits the same property has linear logic: there is no weakening and contraction allowed for normal types, in other terms, by default, each variable in a typing context must be used exactly once to form a valid program.

Leveraging a linear type system, we can design a language in which structures with holes are first-class citizens, and write pointers to these holes are only usable once. This ensures that each hole is filled exactly once, preserving immutability and making sure that structure are indeed completed before being read.

Chapter 1

Linear λ-calculus

1.1 From λ-calculus to linear λ-calculus

At the end of the 30s, Church introduced the untyped λ-calculus as a formal mathematical model of computation. Untyped λ-calculus is based on the concept of function abstraction and application, and is Turing-complete: in other terms, it has the same expressive power as the empirical model of computation that Turing machines represent.

In 1940, Church defined a typed variant of its original calculus, the simply typed λ-calculus, or STLC, that give up Turing-completeness but become strongly-normalizing: every well-typed term eventually reduces to a normal form. STLC assign types to terms, and restricts the application of functions to terms of the right type. This restriction is enforced by the typing rules of the calculus. In the following we assume that the reader is familiar with the simply typed lambda calculus, its typing rules, and usual semantics. We also assume some degree of familiarity with natural deduction.

It has been observed by Howard in 1969 that the intuitionistic variant of natural deduction proof system is isomorphic to the simply typed λ-calculus. This observation relates to prior work by Curry where the former observed that the typed fragment of combinatory logic, another model of computation with similar power, is isomorphic to proof systems like implicational logic and Hilbert deduction systems. These observations led to the Curry-Howard isomorphism, which states that types in a typed λ-calculus correspond to formulas in a proof system, and terms correspond to proofs of these formulas. The Curry-Howard isomorphism has been later extended to other logics and calculi, and has been a fruitful source of inspiration for research on both the logical and computational side.

In that sense, Girard first introduced Linear Logic, in 1987, and only later studied the corresponding calculus, aka. linear λ-calculus. Linear logic follows from the observation that in sequent calculus[1], hypotheses are duplicated or discarded using explicit rules of the system, named *contraction* and *weakening*, in contrast to natural deduction where all that happens implicitly, as it is part of the meta-theory. As a result, it is possible to track the number of times a formula is used by counting the use of these structural rules. Linear logic take this idea further, and deliberately restrict contraction and weakening, so by default, every hypothesis must be used exactly once. Consequently, logical implication $\mathtt{T} \rightarrow \mathtt{U}$ is not part of linear logic, but is

[1]a very popular deduction system that is an alternative to natural deduction and that has also been introduced by Gentzen in the 30s

replaced by linear implication $\mathsf{T} \multimap \mathsf{U}$, where T must be used exactly once to prove U. Linear logic also introduces a modality $!$, pronounced *of course* or *bang*, to allow weakening and contraction on specific formulas: $!\mathsf{T}$ denotes that T can be used an arbitrary number of times (we say it is *unrestricted*). We say that linear logic is a *substructural* logic because it restricts the use of structural rules of usual logic.

We present in Figure 1.1 the natural deduction formulation of intuitionistic linear logic (ILL), as it lends itself well to a computational interpretation as a linear λ-calculus with usual syntax. We borrow the *sequent style* notation of sequent calculus for easier transition into typing rules of terms later. However, as we are in an intuitionistic setting, rules only derive a single conclusion from a multiset of formulas.

All the rules of ILL, except the ones related to the $!$ modality, are directly taken (and slightly adapted) from natural deduction. $\oplus$ denotes (additive) disjunction, and $\otimes$ denotes (multiplicative) conjunction. Hypotheses are represented by multisets Γ. These multisets keep track of how many times each formula appear in them. The comma operator in $\Gamma_1,\ \Gamma_2$ is multiset union, so it sums the number of occurrences of each formula in Γ_1 and Γ_2.

Let's focus on the four rules for the $!$ modality now. The promotion rule ILL/!Pstates that a formula T can become an unrestricted formula $!\mathsf{T}$ if it only depends on formulas that are themselves unrestricted. This is denoted by the (potentially empty) multiset $!\Gamma$. The dereliction rule ILL/!Dstates that an unrestricted formula $!\mathsf{T}$ can be used in a place expecting a normal i.e. linear formula T. The contraction rule ILL/!Cand weakening rule ILL/!Wstates respectively that an unrestricted formula $!\mathsf{T}$ can be cloned or discarded at any time.

We might need other connectives for that, such as *with* &

The linear logic system might appear very restrictive, but we can always simulate the usual non-linear natural deduction in ILL. Girard gives precise rules for such a translation in Section 2.2.6 of [Gir95], whose main idea is to prefix most formulas with $!$ and encode the non-linear implication $\mathsf{T} \to \mathsf{U}$ as $!\mathsf{T} \multimap \mathsf{U}$.

1.2 Linear λ-calculus: a computational interpretation of linear logic

There exists several possible interpretations of linear logic as a linear λ-calculus. The first one, named *monadic* presentation of linear λ-calculus in [AND92], and denoted λ_{L1}in this document, is a direct term assignment of the natural deduction rules of ILL given in Figure 1.1. The syntax and typing rules of this presentation, inspired greatly from the work of [Bie94], are given in Figures 1.2 and 1.3.

In λ_{L1}, Γ is now a finite map from variables to types, that can be represented as a set of variable bindings $x : \mathsf{T}$. As usual, duplicated variable names are not allowed in a context Γ. The comma operator denotes disjoint union for finite maps.

Term grammar for λ_{L1} borrows most of simply typed lambda calculus grammar. In addition, the language has a data constructor/wrapper for unrestricted terms and values, denoted by $\mathsf{Many}\ t$ and $\mathsf{Many}\ v$. Elimination of unit type $\mathbf{1}$ is made with $\fatsemi$ operator. Pattern-matching on sum and product types is made with the **case** keyword. Finally, we have new operators **dup**, **drop** and **derelict** for respective contraction, weakening and dereliction of unrestricted terms of type $!\mathsf{T}$. Promotion of a term to an unrestricted form is made by direct application of constructor Mod . For easier presentation, we also introduce syntactic sugar **let** $x \coloneqq t$ **in** u and encode it as $(\boldsymbol{\lambda} x \mapsto u)\ (t)$.

$\boxed{\Gamma \vdash \mathsf{T}}$ *(Deduction rules)*

$$\frac{}{\mathsf{T} \vdash \mathsf{T}}\ \text{ILL/ID} \qquad \frac{\Gamma,\ \mathsf{T} \vdash \mathsf{U}}{\Gamma \vdash \mathsf{T} \multimap \mathsf{U}}\ \text{ILL/}{\multimap}\text{I} \qquad \frac{}{\bullet \vdash \mathsf{1}}\ \text{ILL/1I} \qquad \frac{\Gamma \vdash \mathsf{T}_1}{\Gamma \vdash \mathsf{T}_1 \oplus \mathsf{T}_2}\ \text{ILL/}{\oplus}\text{I}_1$$

$$\frac{\Gamma \vdash \mathsf{T}_2}{\Gamma \vdash \mathsf{T}_1 \oplus \mathsf{T}_2}\ \text{ILL/}{\oplus}\text{I}_2 \qquad \frac{\Gamma_1 \vdash \mathsf{T}_1 \qquad \Gamma_2 \vdash \mathsf{T}_2}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{T}_1 \otimes \mathsf{T}_2}\ \text{ILL/}{\otimes}\text{I} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash \mathsf{T} \\ \Gamma_2 \vdash \mathsf{T} \multimap \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{U}}\ \text{ILL/}{\multimap}\text{E}$$

$$\frac{\Gamma_1 \vdash \mathsf{1} \qquad \Gamma_2 \vdash \mathsf{U}}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{U}}\ \text{ILL/1E} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash \mathsf{T}_1 \oplus \mathsf{T}_2 \\ \Gamma_2,\ \mathsf{T}_1 \vdash \mathsf{U} \\ \Gamma_2,\ \mathsf{T}_2 \vdash \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{U}}\ \text{ILL/}{\oplus}\text{E} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash \mathsf{T}_1 \otimes \mathsf{T}_2 \\ \Gamma_2,\ \mathsf{T}_1,\ \mathsf{T}_2 \vdash \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{U}}\ \text{ILL/}{\otimes}\text{E}$$

$$\frac{!\Gamma \vdash \mathsf{T}}{!\Gamma \vdash\ !\mathsf{T}}\ \text{ILL/!P} \qquad \frac{\Gamma \vdash\ !\mathsf{T}}{\Gamma \vdash \mathsf{T}}\ \text{ILL/!D} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash\ !\mathsf{T} \\ \Gamma_2,\ !\mathsf{T},\ !\mathsf{T} \vdash \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{U}}\ \text{ILL/!C} \qquad \frac{\Gamma_1 \vdash\ !\mathsf{T} \qquad \Gamma_2 \vdash \mathsf{U}}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{U}}\ \text{ILL/!W}$$

Figure 1.1: Natural deduction formulation of intuitionistic linear logic (sequent-style)

$$\begin{array}{rcl}
v & ::= & \lambda x \mapsto u \ \mid\ () \ \mid\ \mathsf{Inl}\ v \ \mid\ \mathsf{Inr}\ v \ \mid\ (v_1\ ,\ v_2) \ \mid\ \mathsf{Many}\ v \\
t, u & ::= & v \ \mid\ x \ \mid\ \mathsf{Inl}\ t \ \mid\ \mathsf{Inr}\ t \ \mid\ (t_1\ ,\ t_2) \ \mid\ \mathsf{Many}\ t \ \mid\ t\ t' \ \mid\ t\ ⨾\ t' \\
 & \mid & \mathbf{case}\ t\ \mathbf{of}\ \{\mathsf{Inl}\ x_1 \mapsto u_1\ ,\ \mathsf{Inr}\ x_2 \mapsto u_2\} \ \mid\ \mathbf{case}\ t\ \mathbf{of}\ (x_1\ ,\ x_2) \mapsto u \\
 & \mid & \mathbf{dup}\ t\ \mathbf{as}\ x_1\ ,\ x_2\ \mathbf{in}\ u \ \mid\ \mathbf{drop}\ t\ \mathbf{in}\ u \ \mid\ \mathbf{derelict}\ t \\
 & & \\
\mathsf{T}, \mathsf{U} & ::= & \mathsf{T} \multimap \mathsf{U} \ \mid\ \mathsf{1} \ \mid\ \mathsf{T}_1 \oplus \mathsf{T}_2 \ \mid\ \mathsf{T}_1 \otimes \mathsf{T}_2 \ \mid\ !\mathsf{T} \\
 & & \\
\Gamma & ::= & \bullet \ \mid\ x : \mathsf{T} \ \mid\ \Gamma_1,\ \Gamma_2
\end{array}$$

Figure 1.2: Grammar of linear λ-calculus in monadic presentation (λ_{L1})

$\boxed{\Gamma \vdash t : \mathtt{T}}$ *(Typing judgment for terms)*

$$\frac{}{x : \mathtt{T} \vdash x : \mathtt{T}}\ \lambda_{L1}\text{–TY/ID} \qquad \frac{\Gamma,\ x : \mathtt{T} \vdash u : \mathtt{U}}{\Gamma \vdash \boldsymbol{\lambda} x \mapsto u : \mathtt{T} \multimap \mathtt{U}}\ \lambda_{L1}\text{–TY/}{\multimap}\text{I} \qquad \frac{}{\bullet \vdash () : \mathtt{1}}\ \lambda_{L1}\text{–TY/1I}$$

$$\frac{\Gamma \vdash t_1 : \mathtt{T}_1}{\Gamma \vdash \mathsf{Inl}\, t_1 : \mathtt{T}_1 \oplus \mathtt{T}_2}\ \lambda_{L1}\text{–TY/}{\oplus}\text{I}_1 \qquad \frac{\Gamma \vdash t_2 : \mathtt{T}_2}{\Gamma \vdash \mathsf{Inr}\, t_2 : \mathtt{T}_1 \oplus \mathtt{T}_2}\ \lambda_{L1}\text{–TY/}{\oplus}\text{I}_2$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t_1 : \mathtt{T}_1 \\ \Gamma_2 \vdash t_2 : \mathtt{T}_2\end{array}}{\Gamma_1,\ \Gamma_2 \vdash (t_1\ ,\ t_2) : \mathtt{T}_1 \otimes \mathtt{T}_2}\ \lambda_{L1}\text{–TY/}{\otimes}\text{I} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash t : \mathtt{T} \\ \Gamma_2 \vdash t' : \mathtt{T} \multimap \mathtt{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash t'\ t : \mathtt{U}}\ \lambda_{L1}\text{–TY/}{\multimap}\text{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathtt{1} \\ \Gamma_2 \vdash u : \mathtt{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash t \,\mathbin{;}\, u : \mathtt{U}}\ \lambda_{L1}\text{–TY/1E} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash t : \mathtt{T}_1 \oplus \mathtt{T}_2 \\ \Gamma_2,\ x_1 : \mathtt{T}_1 \vdash u_1 : \mathtt{U} \\ \Gamma_2,\ x_2 : \mathtt{T}_2 \vdash u_2 : \mathtt{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathbf{case}\ t\ \mathbf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1\ ,\ \mathsf{Inr}\, x_2 \mapsto u_2\} : \mathtt{U}}\ \lambda_{L1}\text{–TY/}{\oplus}\text{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathtt{T}_1 \otimes \mathtt{T}_2 \\ \Gamma_2,\ x_1 : \mathtt{T}_1,\ x_2 : \mathtt{T}_2 \vdash u : \mathtt{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathbf{case}\ t\ \mathbf{of}\ (x_1\ ,\ x_2) \mapsto u : \mathtt{U}}\ \lambda_{L1}\text{–TY/}{\otimes}\text{E} \qquad \frac{!\Gamma \vdash t : \mathtt{T}}{!\Gamma \vdash \mathsf{Many}\ t : !\mathtt{T}}\ \lambda_{L1}\text{–TY/!P}$$

$$\frac{\Gamma \vdash t : !\mathtt{T}}{\Gamma \vdash \mathbf{derelict}\ t : \mathtt{T}}\ \lambda_{L1}\text{–TY/!D} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash t : !\mathtt{T} \\ \Gamma_2,\ x_1 : !\mathtt{T},\ x_2 : !\mathtt{T} \vdash u : \mathtt{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathbf{dup}\ t\ \mathbf{as}\ x_1\ ,\ x_2\ \mathbf{in}\ u : \mathtt{U}}\ \lambda_{L1}\text{–TY/!C}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : !\mathtt{T} \\ \Gamma_2 \vdash u : \mathtt{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathbf{drop}\ t\ \mathbf{in}\ u : \mathtt{U}}\ \lambda_{L1}\text{–TY/!W}$$

Figure 1.3: Typing rules for linear λ-calculus in monadic presentation (λ_{L1})

$$
\begin{array}{rcl}
v & ::= & \lambda x \mapsto u \;\mid\; () \;\mid\; \mathsf{Inl}\, v \;\mid\; \mathsf{Inr}\, v \;\mid\; (v_1\,,\,v_2) \;\mid\; \mathsf{Many}\, v \\
t, u & ::= & v \;\mid\; x \;\mid\; \mathsf{Inl}\, t \;\mid\; \mathsf{Inr}\, t \;\mid\; (t_1\,,\,t_2) \;\mid\; \mathsf{Many}\, t \;\mid\; t\ t' \;\mid\; t \fatsemi t' \\
 & \mid & \mathbf{case}\ t\ \mathbf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1\,,\ \mathsf{Inr}\, x_2 \mapsto u_2\} \;\mid\; \mathbf{case}\ t\ \mathbf{of}\ (x_1\,,\,x_2) \mapsto u \;\mid\; \mathbf{case}\ t\ \mathbf{of}\ \mathsf{Many}\, x \mapsto u \\
 & & \\
\mathsf{T}, \mathsf{U} & ::= & \mathsf{T} \multimap \mathsf{U} \;\mid\; \mathsf{1} \;\mid\; \mathsf{T}_1 \oplus \mathsf{T}_2 \;\mid\; \mathsf{T}_1 \otimes \mathsf{T}_2 \;\mid\; !\mathsf{T} \\
 & & \\
\Gamma & ::= & \bullet \;\mid\; x : \mathsf{T} \;\mid\; \Gamma_1,\ \Gamma_2 \\
\mho & ::= & \bullet \;\mid\; x : \mathsf{T} \;\mid\; \mho_1,\ \mho_2
\end{array}
$$

Figure 1.4: Grammar of linear λ-calculus in dyadic presentation (λ_{L2})

1.3 Implicit structural rules for unrestricted resources

In the monadic presentation λ_{L1}, the use of unrestricted terms can become very verbose and unhandy because of the need for explicit contraction, weakening and dereliction. A second and equivalent presentation of our linear λ-calculus, named *dyadic* presentation or λ_{L2}, tend to alleviate this issue by using two typing contexts on each judgment, one for linear variables and one for unrestricted variables. The syntax and typing rules of λ_{L2} are given in Figures 1.4 and 1.5.

In λ_{L2}there is no longer rules for contraction, weakening, and dereliction of unrestricted resources. Instead, each judgment is equipped with a second context $\mho$ that holds variable bindings that can be used in an unrestricted fashion. The $\mathbf{case}\ t\ \mathbf{of}\ \mathsf{Many}\, x \mapsto u$ construct is used to bind a term of type $!\mathsf{T}$ as a variable binding $x : \mathsf{T}$ in the unrestricted context $\mho$. It's important to note that $\mathbf{case}\ t\ \mathbf{of}\ \mathsf{Many}\, x \mapsto u$ is not dereliction: one can still use x several times within body u, or recreate t by wrapping x back as $\mathsf{Many}\, x$; while that wouldn't be possible in $\mathbf{let}\ x := \mathbf{derelict}\ t\ \mathbf{in}\ u$ of λ_{L1}. Morally, we can view the pair of contexts $\Gamma; \mho$ of λ_{L2} as a single context $\Gamma,\ !\mho$ of λ_{L1}, where $!\mho$ is the context with the same variable bindings as $\mho$, except that all types are prefixed by $!$.

In λ_{L2}, contraction for unrestricted resources happens implicitly every time a rule has two subterms as premises. Indeed, the unrestricted context $\mho$ is duplicated in both premises, unlike the linear context Γ that must be split into two disjoint parts. All unrestricted variable bindings are thus propagated to the leaves of the typing tree, that is, the rules with no premises λ_{L2}–TY/ID$_{\text{LIN}}$, λ_{L2}–TY/ID$_{\text{UR}}$, and λ_{L2}–TY/$\mathsf{1}$I. These three rules discard all bindings of the unrestricted context $\mho$ that aren't used, performing several implicit weakening steps. Finally, this system has two identity rules. The first one, λ_{L2}–TY/ID$_{\text{LIN}}$, is the usual linear identity: it asks for the variable x to be in the linear typing context, and later x cannot be reused in another subterm. The second one, λ_{L2}–TY/ID$_{\text{UR}}$, is the unrestricted identity: it lets us use the variable x from the unrestricted typing context in a place where a linear variable is expected, performing a sort of implicit dereliction.

[AND92] has a detailed proof that both λ_{L1} and λ_{L2} are equivalent, in other terms, that a program t types in the pair of contexts $\Gamma; \mho$ in λ_{L2} if and only if it types in context $\Gamma,\ !\mho$ in λ_{L1}.

$\boxed{\Gamma \mathbin{;} \mho \vdash t : \mathsf{T}}$ *(Typing judgment for terms)*

$$\frac{}{x : \mathsf{T} \mathbin{;} \mho \vdash x : \mathsf{T}}\ \lambda_{L2}\text{–TY}/\text{ID}_{\text{LIN}} \qquad \frac{}{\bullet \mathbin{;} \mho,\ x : \mathsf{T} \vdash x : \mathsf{T}}\ \lambda_{L2}\text{–TY}/\text{ID}_{\text{UR}}$$

$$\frac{\Gamma,\ x : \mathsf{T} \mathbin{;} \mho \vdash u : \mathsf{U}}{\Gamma \mathbin{;} \mho \vdash \boldsymbol{\lambda} x \mapsto u : \mathsf{T} \multimap \mathsf{U}}\ \lambda_{L2}\text{–TY}/{\multimap}\text{I} \qquad \frac{}{\bullet \mathbin{;} \mho \vdash () : \mathbf{1}}\ \lambda_{L2}\text{–TY}/\mathbf{1}\text{I}$$

$$\frac{\Gamma \mathbin{;} \mho \vdash t_1 : \mathsf{T}_1}{\Gamma \mathbin{;} \mho \vdash \mathsf{Inl}\, t_1 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \lambda_{L2}\text{–TY}/{\oplus}\text{I}_1 \qquad \frac{\Gamma \mathbin{;} \mho \vdash t_2 : \mathsf{T}_2}{\Gamma \mathbin{;} \mho \vdash \mathsf{Inr}\, t_2 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \lambda_{L2}\text{–TY}/{\oplus}\text{I}_2$$

$$\frac{\begin{array}{c}\Gamma_1 \mathbin{;} \mho \vdash t_1 : \mathsf{T}_1 \\ \Gamma_2 \mathbin{;} \mho \vdash t_2 : \mathsf{T}_2\end{array}}{\Gamma_1,\ \Gamma_2 \mathbin{;} \mho \vdash (t_1\,,\ t_2) : \mathsf{T}_1 \otimes \mathsf{T}_2}\ \lambda_{L2}\text{–TY}/{\otimes}\text{I} \qquad \frac{\bullet \mathbin{;} \mho \vdash t : \mathsf{T}}{\bullet \mathbin{;} \mho \vdash \mathsf{Many}\, t : {!}\mathsf{T}}\ \lambda_{L2}\text{–TY}/{!}\text{I}$$

$$\frac{\begin{array}{c}\Gamma_1 \mathbin{;} \mho \vdash t : \mathsf{T} \\ \Gamma_2 \mathbin{;} \mho \vdash t' : \mathsf{T} \multimap \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \mathbin{;} \mho \vdash t'\ t : \mathsf{U}}\ \lambda_{L2}\text{–TY}/{\multimap}\text{E} \qquad \frac{\begin{array}{c}\Gamma_1 \mathbin{;} \mho \vdash t : \mathbf{1} \\ \Gamma_2 \mathbin{;} \mho \vdash u : \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \mathbin{;} \mho \vdash t \fatsemi u : \mathsf{U}}\ \lambda_{L2}\text{–TY}/\mathbf{1}\text{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \mathbin{;} \mho \vdash t : \mathsf{T}_1 \oplus \mathsf{T}_2 \\ \Gamma_2,\ x_1 : \mathsf{T}_1 \mathbin{;} \mho \vdash u_1 : \mathsf{U} \\ \Gamma_2,\ x_2 : \mathsf{T}_2 \mathbin{;} \mho \vdash u_2 : \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \mathbin{;} \mho \vdash \mathbf{case}\ t\ \mathbf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1\,,\ \mathsf{Inr}\, x_2 \mapsto u_2\} : \mathsf{U}}\ \lambda_{L2}\text{–TY}/{\oplus}\text{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \mathbin{;} \mho \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \\ \Gamma_2,\ x_1 : \mathsf{T}_1,\ x_2 : \mathsf{T}_2 \mathbin{;} \mho \vdash u : \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \mathbin{;} \mho \vdash \mathbf{case}\ t\ \mathbf{of}\ (x_1\,,\ x_2) \mapsto u : \mathsf{U}}\ \lambda_{L2}\text{–TY}/{\otimes}\text{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \mathbin{;} \mho \vdash t : {!}\mathsf{T} \\ \Gamma_2 \mathbin{;} \mho,\ x : \mathsf{T} \vdash u : \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \mathbin{;} \mho \vdash \mathbf{case}\ t\ \mathbf{of}\ \mathsf{Many}\, x \mapsto u : \mathsf{U}}\ \lambda_{L2}\text{–TY}/{!}\text{E}$$

Figure 1.5: Typing rules for linear λ-calculus in dyadic presentation (λ_{L2})

1.4 Back to a single context with the graded modal approach

So far, we only considered type systems that are linear, but that are otherwise fairly standard with relation to usual simply-typed λ-calculus. Anticipating on our future needs, further in this document we'll need to encode more information and ensure more invariants throughout the type system than just linearity alone.

Naively, we could just multiply the typing contexts, for each new modality or control axis that we need (linearity being one of them, with two values, either *linear* or *unrestricted*). The problem is, that approach is not really scalable: as we increase the number of axes, in the end, we need one context per combination of value on each axis, so it grows really quickly.

Even without multiple axis, we already have a problem if we want a finer control over linearity. What if we want to allow variables to be used a fixed number of times that isn't just 1 or unlimited? In [GSS92], Girard considers that *bounded* extension of linear logic, where the number of uses of hypotheses can be restricted to any value aka. multiplicity m instead of being just linear or unrestricted. For that he extends the $!$ modality with an index m that specifies how many times an hypothesis should be used. We say that $!_{\mathsf{m}}$ is a *graded modality*.

Having a family of modalities $(!_{\mathsf{m}})_{\mathsf{m}\in\mathsf{M}}$ with an arbitrary number of elements instead of the single $!$ modality of original linear logic means that we cannot really have a distinct context for each of them as in the dyadic presentation. One solution is to go back to the monadic presentation, where variables carry their modality on their type until the very end where they get used. We would also need a new operator to go from $!_{\mathsf{m}}\mathsf{T}$ to a pair $!_{\mathsf{m}-1}\mathsf{T}, \mathsf{T}$ so that we can extract a single use from a value wrapped in a modality allowing several uses. Actually, it gets hairy and unpractical very fast.

Fortunately, there's a way out of this. Instead of having m be part of the modality and thus of the type of terms, we can bake it in as an annotation on variable bindings. In place of $x : \mathsf{S}, y : !_{\mathsf{m}}\mathsf{T}, z : !_{\mathsf{n}}\mathsf{U} \vdash \ldots$, we can have $x :_{1} \mathsf{S}, y :_{\mathsf{m}} \mathsf{T}, z :_{\mathsf{n}} \mathsf{U} \vdash \ldots$ without adding or losing any information. We'll call the new annotations on bindings *modes*. Note that every binding is equipped with a mode m, even linear bindings that previously didn't have modalities on their types[2]. Now, we can completely encode the restrictions and rules of our system by defining operations on modes and by extension, on typing contexts, and use these operations in typing rules, instead of needing extra operators such as **`derelict`** or **`dup`** that need their own typing rules and have to be used explicitly by the user. With that modal approach, we recover a system, like the dyadic one, in which contraction, weakening, and dereliction can be made conveniently implicit, without loosing any control power over resource use, and with no explosion of the number of contexts!

Moreover, we build on the key insight, which seem to originate with [GS14], that equipping the set of modes with a semiring structure is sufficient to express, algebraically, all the typing context manipulation that we need. The idea is to have two operations on modes: *times* $\cdot$ that represents what happens to modes when there is composition (e.g. for linearity, if function f uses its argument 2 times and g too, then $f\ (g\ x)$ uses x $2 \cdot 2$ times), while *plus* $+$ describes what happens to modes when a same variable is used in two (or more) subterms (if subterm t uses x 2 times and u uses x 3 times, then $t \fatsemi u$ uses x $2 + 3$ times).

[2]This uniform approach where every binding receive a mode seems to originate from [GS14] and [POM14].

$$
\begin{array}{rcl}
v & ::= & \lambda x \,{}_{\mathtt{m}}\!\mapsto u \;\mid\; () \;\mid\; \mathtt{Inl}\, v \;\mid\; \mathtt{Inr}\, v \;\mid\; (v_1\,,\, v_2) \;\mid\; \mathtt{Mod}_{\mathtt{n}}\, v \\
t, u & ::= & v \;\mid\; x \;\mid\; \mathtt{Inl}\, t \;\mid\; \mathtt{Inr}\, t \;\mid\; (t_1\,,\, t_2) \;\mid\; \mathtt{Mod}_{\mathtt{n}}\, t \;\mid\; t\ t' \;\mid\; t \fatsemi t' \\
 & \mid & \mathtt{case}\ t\ \mathtt{of}\ \{\mathtt{Inl}\, x_1 \mapsto u_1\,,\ \mathtt{Inr}\, x_2 \mapsto u_2\} \;\mid\; \mathtt{case}\ t\ \mathtt{of}\ (x_1\,,\, x_2) \mapsto u \;\mid\; \mathtt{case}\ t\ \mathtt{of}\ \mathtt{Mod}_{\mathtt{n}}\, x \mapsto u \\
 & & \\
\mathtt{T}, \mathtt{U} & ::= & \mathtt{T}\ {}_{\mathtt{m}}\!\multimap \mathtt{U} \;\mid\; \mathtt{1} \;\mid\; \mathtt{T}_1 \oplus \mathtt{T}_2 \;\mid\; \mathtt{T}_1 \otimes \mathtt{T}_2 \;\mid\; !_{\mathtt{n}}\mathtt{T} \\
\mathtt{m}, \mathtt{n} & ::= & 1 \;\mid\; \omega \\
 & & \\
\Gamma & ::= & \bullet \;\mid\; x :_{\mathtt{m}} \mathtt{T} \;\mid\; \Gamma_1,\ \Gamma_2
\end{array}
$$

Figure 1.6: Grammar of linear λ-calculus in modal presentation (λ_{Lm})

We then lift these operations to variable bindings, and then to typing contexts themselves in the following way. We lift *times* so that it can scale a variable binding by a mode $\mathtt{n}$: we pose $\mathtt{n}\cdot(x :_{\mathtt{m}} \mathtt{T}) \triangleq x :_{\mathtt{n}\cdot\mathtt{m}} \mathtt{T}$. We then extend this operator point-wise to act on whole typing contexts as in $\mathtt{n}\cdot\Gamma$, not just single bindings.

We also define typing context *plus* as a partial operation where $(x :_{\mathtt{m}} \mathtt{T}) + (x :_{\mathtt{m}'} \mathtt{T}) = x :_{\mathtt{m}+\mathtt{m}'} \mathtt{T}$ and $(x :_{\mathtt{m}} \mathtt{T}) + \Gamma = x :_{\mathtt{m}} \mathtt{T},\ \Gamma$ if $x \notin \Gamma$.

We'll now show what the concrete *modal* presentation for intuitionistic λ-calculus or λ_{Lm}looks like. We will go back to a very simple ringoid[3] for modes, that just models linearity equivalently as the previous presentations $\lambda_{L1}$and λ_{L2}: 1 for variables that must be managed in strict linear fashion, and ω for unrestricted ones. We give the following operation tables:

$+$	1	ω
1	ω	ω
ω	ω	ω

$\cdot$	1	ω
1	1	ω
ω	ω	ω

Equipped with those, we can move to the grammar and typing rules of λ_{Lm}, given in Figures 1.6 and 1.7. Constructor for $!_{\mathtt{m}}\mathtt{T}$ is now $\mathtt{Mod}_{\mathtt{n}}\ t$ (instead of $\mathtt{Many}\ t$ for $!\mathtt{T}$).

In λ_{Lm} we're back to a single identity rule λ_{Lm}–TY/ID, that asks for x to be in the typing context with a mode $\mathtt{m}$ that must be *compatible with a single linear use* (we note $1 \lessapprox \mathtt{m}$). In our ringoid, with only two elements, we have both $1 \lessapprox 1$ and $1 \lessapprox \omega$, so x can actually have any mode (either linear or unrestricted, so encompassing both λ_{L2}–TY/ID$_{\text{LIN}}$ and λ_{L2}–TY/ID$_{\text{UR}}$), but in more complex modal systems, not all modes have to be compatible with the unit of the ringoid[4]. Rules λ_{Lm}–TY/ID and λ_{Lm}–TY/1I also allows to discard any context composed only of unrestricted bindings, denoted by $\omega\cdot\Gamma$ (equivalent to notation $!\Gamma$ in λ_{L1}), so they are performing implicit weakening as in λ_{L2}.

Every rule of λ_{Lm} that mentions two subterms uses the newly defined $+$ operator on typing contexts in the conclusion of the rule. If a same variable x is required in both Γ_1 and Γ_2 (either with mode 1 or ω, it doesn't matter), then $\Gamma_1 + \Gamma_2$ will contain binding $x :_{\omega} \mathtt{T}$. Said differently, the parent term will automatically deduce wether x needs to be linear or unrestricted based on how (many) subterms use x, thanks to the $+$ operator. It's a vastly different approach than the

[3]We still get all the algebraic benefits of [GS14] even without a zero for our structure.

[4]Actually, in the type system for λ_d detailed in **??**, mode $1\uparrow$ is not compatible with unit 1ν.

$\boxed{\Gamma \vdash t : \mathsf{T}}$ *(Typing judgment for terms)*

$$\frac{1 \leqslant \mathsf{m}}{\omega\cdot\Gamma,\ x :_{\mathsf{m}} \mathsf{T} \vdash x : \mathsf{T}}\ \lambda_{Lm}\text{–TY/ID} \qquad \frac{\Gamma,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}}{\Gamma \vdash \boldsymbol{\lambda} x\ {}_{\mathsf{m}}\!\mapsto u : \mathsf{T}\ {}_{\mathsf{m}}\!\multimap \mathsf{U}}\ \lambda_{Lm}\text{–TY/}\!\multimap\!\text{I}$$

$$\frac{}{\omega\cdot\Gamma \vdash () : \mathbf{1}}\ \lambda_{Lm}\text{–TY/1I} \qquad \frac{\Gamma \vdash t_1 : \mathsf{T}_1}{\Gamma \vdash \mathsf{Inl}\, t_1 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \lambda_{Lm}\text{–TY/}\oplus\text{I}_1 \qquad \frac{\Gamma \vdash t_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{Inr}\, t_2 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \lambda_{Lm}\text{–TY/}\oplus\text{I}_2$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t_1 : \mathsf{T}_1\\ \Gamma_2 \vdash t_2 : \mathsf{T}_2\end{array}}{\Gamma_1 + \Gamma_2 \vdash (t_1\,,\ t_2) : \mathsf{T}_1 \otimes \mathsf{T}_2}\ \lambda_{Lm}\text{–TY/}\otimes\text{I} \qquad \frac{\Gamma \vdash t : \mathsf{T}}{\mathsf{n}\cdot\Gamma \vdash \mathsf{Mod}_{\mathsf{n}}\, t : !_{\mathsf{n}}\mathsf{T}}\ \lambda_{Lm}\text{–TY/!I}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}\\ \Gamma_2 \vdash t' : \mathsf{T}\ {}_{\mathsf{m}}\!\multimap \mathsf{U}\end{array}}{\mathsf{m}\cdot\Gamma_1 + \Gamma_2 \vdash t'\ t : \mathsf{U}}\ \lambda_{Lm}\text{–TY/}\!\multimap\!\text{E} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash t : \mathbf{1}\\ \Gamma_2 \vdash u : \mathsf{U}\end{array}}{\Gamma_1 + \Gamma_2 \vdash t \ ⨟\ u : \mathsf{U}}\ \lambda_{Lm}\text{–TY/1E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \oplus \mathsf{T}_2\\ \Gamma_2,\ x_1 :_{1} \mathsf{T}_1 \vdash u_1 : \mathsf{U}\\ \Gamma_2,\ x_2 :_{1} \mathsf{T}_2 \vdash u_2 : \mathsf{U}\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{case}\ t\ \mathbf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1\,,\ \mathsf{Inr}\, x_2 \mapsto u_2\} : \mathsf{U}}\ \lambda_{Lm}\text{–TY/}\oplus\text{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2\\ \Gamma_2,\ x_1 :_{1} \mathsf{T}_1,\ x_2 :_{1} \mathsf{T}_2 \vdash u : \mathsf{U}\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{case}\ t\ \mathbf{of}\ (x_1\,,\ x_2) \mapsto u : \mathsf{U}}\ \lambda_{Lm}\text{–TY/}\otimes\text{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : !_{\mathsf{n}}\mathsf{T}\\ \Gamma_2,\ x :_{\mathsf{n}} \mathsf{T} \vdash u : \mathsf{U}\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{case}\ t\ \mathbf{of}\ \mathsf{Mod}_{\mathsf{n}}\, x \mapsto u : \mathsf{U}}\ \lambda_{Lm}\text{–TY/!E}$$

Figure 1.7: Typing rules for linear λ-calculus in modal presentation (λ_{Lm})

linear context split for subterms in $\lambda_{L1,2}$ and unrestricted context duplication for subterms in λ_{L2}. I would argue that it makes the system smoother, and allows for easy extension without changing the typing rules much (compared to the monadic / dyadic presentation that are very specialized to handle linearity/unrestrictedness, and that alone).

Much as in λ_{L2}, the exponential modality $!_n$ is eliminated by a $\mathtt{case}\ t\ \mathtt{of}\ \mathtt{Mod}_n\, x \mapsto u$ expression, that binds the payload to a new variable x with modality n. The original boxed value can be recreated if needed with $\mathtt{Mod}_n\, x$; indeed, as in λ_{L2}, elimination for $!_n$ is *not* dereliction.

The last specificity of this system is that the function arrow $\multimap$ now has a mode m to which it consumes its argument. Actually that doesn't change the expressivity of the system compared to previous presentations; we would have been just fine with only a purely linear arrow, but because we have to assign a mode to any variable binding in this presentation, then it makes sense to allow this mode m to be whatever the programmer wants, and not just default to 1. In application rule λ_{Lm}–TY/$\multimap$E, the typing context Γ_1 required to type the argument is scaled in the conclusion of the rule by the mode m that represent the "number" of use of the argument by the function.

1.5 Deep modes: projecting modes through fields of data structures

In original linear logic from Girard (see ILL and presentation λ_{L1} above), there is only a one-way morphism between $!T \otimes !U$ and $!(T \otimes U)$; we cannot go from $!(T \otimes U)$ to $!T \otimes !U$. In other terms, an unrestricted pair $!(T \otimes U)$ doesn't allow for unrestricted use of its components. The pair can be duplicated or discarded at will, but to be used, it needs to be derelicted first, to become $T \otimes U$, that no longer allow to duplicate or discard any of T or U. As a result, T and U will have to be used exactly the same number of times, even though they are part of an unrestricted pair!

Situation is no different in λ_{L2} (resp. λ_{Lm}): although the pair of type $!(T \otimes U)$ can be bound in an unrestricted binding $x : T \otimes U \in \mho$ (resp. $x :_{\omega} T \otimes U$) and be used as this without need for dereliction, when it will be pattern-matched on with $\lambda_{L2,m}$–TY/$\otimes$E, implicit dereliction will still happen, and linear-only bindings will be made for its components: $x_1 : T$, $x_2 : U$ in the linear typing context (resp. $x_1 :_1 T$, $x_2 :_1 U$).

It's in fact useful to lift this limitation. One motivation for *deep modes* (in the sense that they propagate throughout a data structure) is that they make it more convenient to write non-linear programs in a linear language. Indeed, in a modal linear λ-calculus with deep modes (λ_{Ldm}), functions of type $T\ {}_{\omega}\!\!\multimap U$ have exactly the same expressive power has ones with type $T \to U$ in STLC. In other terms, any program that is valid in STLC but doesn't abide by linearity can still type in λ_{Ldm} without any restructuration needed!

Let's take the `fst` function as an example, that extracts the first component of a pair. With deep modes in λ_{Ldm} we are able to write

$$\mathtt{fst_{dm}} \ :\ T \otimes U\ {}_{\omega}\!\!\multimap T$$
$$\mathtt{fst_{dm}} \ \triangleq\ \lambda x\ {}_{\omega}\!\!\mapsto \mathtt{case}_{\omega}\ x\ \mathtt{of}\ (x_1\,,\,x_2) \mapsto x_1$$

as we would do in a non-linear language, like STLC, in which `fst` would have type $T \otimes U \to T$. On the other hand, in λ_{Lm} as presented in Figures 1.6 and 1.7, we need to indicate individually for each component of the pair that we want them unrestricted (at least for the second one that we intent on discarding), and add extra elimination for the $!$ modality:

$\boxed{\Gamma \vdash t : \mathsf{T}}$ *(Typing judgment for terms)*

$$\dfrac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T_1}\oplus\mathsf{T_2} \\ \Gamma_2,\ x_1 :_{\mathsf{m}} \mathsf{T_1} \vdash u_1 : \mathsf{U} \\ \Gamma_2,\ x_2 :_{\mathsf{m}} \mathsf{T_2} \vdash u_2 : \mathsf{U}\end{array}}{\mathsf{m}\cdot\Gamma_1 + \Gamma_2 \vdash \mathbf{case}_{\mathsf{m}}\ t\ \mathbf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1\ ,\ \mathsf{Inr}\, x_2 \mapsto u_2\} : \mathsf{U}}\ \lambda_{Ldm}\text{–TY}/\oplus\text{E}$$

$$\dfrac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T_1}\otimes\mathsf{T_2} \\ \Gamma_2,\ x_1 :_{\mathsf{m}} \mathsf{T_1},\ x_2 :_{\mathsf{m}} \mathsf{T_2} \vdash u : \mathsf{U}\end{array}}{\mathsf{m}\cdot\Gamma_1 + \Gamma_2 \vdash \mathbf{case}_{\mathsf{m}}\ t\ \mathbf{of}\ (x_1\ ,\ x_2) \mapsto u : \mathsf{U}}\ \lambda_{Ldm}\text{–TY}/\otimes\text{E}$$

$$\dfrac{\begin{array}{c}\Gamma_1 \vdash t : !_{\mathsf{n}}\mathsf{T} \\ \Gamma_2,\ x :_{\mathsf{m}\cdot\mathsf{n}} \mathsf{T} \vdash u : \mathsf{U}\end{array}}{\mathsf{m}\cdot\Gamma_1 + \Gamma_2 \vdash \mathbf{case}_{\mathsf{m}}\ t\ \mathbf{of}\ \mathsf{Mod}_{\mathsf{n}}\, x \mapsto u : \mathsf{U}}\ \lambda_{Ldm}\text{–TY}/!\text{E}$$

Figure 1.8: Altered typing rules for deep modes in linear λ-calculus in modal presentation (λ_{Ldm})

$$\begin{array}{lcl}\mathbf{fst}_{\mathsf{m}} & : & \mathsf{T}\otimes(!_{\omega}\mathsf{U})\ {}_{1}\!\multimap \mathsf{T} \\ \mathbf{fst}_{\mathsf{m}} & \triangleq & \lambda x\ {}_{1}\!\mapsto \mathbf{case}\ x\ \mathbf{of}\ (x_1\ ,\ ux_2) \mapsto \\ & & \quad \mathbf{case}\ ux_2\ \mathbf{of}\ \mathsf{Mod}_{\omega}\ x_2 \mapsto x_1\end{array}$$

Adding deep modes to a practical linear language is not a very original take; it has been done in Linear Haskell [Ber+18] and [Lor+24a]. With deep modes, we get the following equivalences:

$$\begin{array}{rcl} !_{\mathsf{m}}(\mathsf{T}\otimes\mathsf{U}) & \simeq & (!_{\mathsf{m}}\mathsf{T})\otimes(!_{\mathsf{m}}\mathsf{U}) \\ !_{\mathsf{m}}(\mathsf{T}\oplus\mathsf{U}) & \simeq & (!_{\mathsf{m}}\mathsf{T})\oplus(!_{\mathsf{m}}\mathsf{U}) \\ !_{\mathsf{m}}(!_{\mathsf{n}}\mathsf{T}) & \simeq & !_{(\mathsf{m}\cdot\mathsf{n})}\mathsf{T}\end{array}$$

The only change needed on the grammar is that **case** constructs now take a mode m to which they consume the scrutinee, that is propagated to the field(s) of the scrutinee in the body of the **case**. The new typing rules for **case** are presented in Figure 1.8, all the other rules of linear λ-calculus with deep modes, λ_{Ldm}, are supposed identical to Figure 1.7.

We observe that the typing context Γ_1 in which the scrutinee types is scaled by m in the conclusion of all the **case** rules. This is very analog to the application rule λ_{Lm}–TY/$\multimap$E : it makes sure that the resources required to type t are consumed m times if we want to use t at mode m. Given that $x_2 :_{1} \mathsf{T_2} \vdash ((),\ x_2) : \mathsf{1}\otimes\mathsf{T_2}$ (the pair uses x_2 exactly once), if we want to extract the pair components with mode ω to drop x_2 (as in $\mathbf{fst}_{\mathbf{dm}}$), then $\mathbf{case}_{\omega}\ ((),\ x_2)\ \mathbf{of}\ (x_1\ ,\ x_2) \mapsto x_2$ will require context $\omega\cdot(x_2 :_{1} \mathsf{T_2})$ i.e. $x_2 :_{\omega} \mathsf{T_2}$. In other terms, we cannot use parts of a structure made using linear resources in an unrestricted way (as that would break linearity).

The linear λ-calculus with deep modes, λ_{Ldm}, will be the basis for our core contribution that follows in next chapter: the destination calculus λ_d.

1.6 Semantics of linear λ-calculus (with deep modes)

Many semantics presentations exist for lambda calculus. In Figure 1.9 I present a small-step reduction system for λ_{Ldm}, in *reduction semantics* style inspired from [Fel87] and subsequent [DN04; BD07], that is, a semantics in which the evaluation context E is manipulated explicitly and syntactically as a stack.

Small-step evaluation rules are of three kinds:

- focusing rules (F) that split the current term under focus in two parts: an evaluation context component that is pushed on the stack E for later use, and a subterm that is put under focus;
- unfocusing rules (U) that recreate a larger term once the term under focus is a value, by popping the most recent evaluation component from the stack and merging it with the value;
- contraction rules (C) that do not operate on the stack E but just transform the term under focus when it's a redex.

To have a fully deterministic and and straightforward reduction system, focusing rules can only trigger when the subterm in question is not already a value (denoted by $\star$ in Figure 1.9).

Data constructors only have focusing and unfocusing rules, as they do not describe computations that can be reduced. In that regard, $\mathtt{Mod}_{\mathsf{n}}\ t$ is treated as an unary data constructor. Once a data constructor is focused, it is evaluated fully to a value form.

In most cases, unfocusing and contraction rules could be merged into a single step. For example, a contraction could be triggered as soon as we have $\big(E \circ \mathtt{case}_{\mathsf{m}}\ [\,]\ \mathtt{of}\ (x_1\ ,\ x_2) \mapsto u\big)\big[(v_1\ ,\ v_2)\big]$, without needing recreate the term $\mathtt{case}_{\mathsf{m}}\ (v_1\ ,\ v_2)\ \mathtt{of}\ (x_1\ ,\ x_2) \mapsto u$. However, I prefer the presentation in three distinct steps, that despite being more verbose, clear shows that contraction rules do not modify the evaluation context.

The rest of the system is very standard. In particular, contraction rules of λ_{Ldm} — that capture the essence of the calculus — are very similar to those of strict lambda-calculi with sum and product types.

$\boxed{E\left[\,t\,\right] \;\longrightarrow\; E'\left[\,t'\,\right]}$ *(Small-step evaluation)*

$E\left[\mathtt{Inl}\,t\right] \longrightarrow \left(E \circ \mathtt{Inl}\,[\,]\right)\left[t\right]$	$\star$	λ_{Ldm}–SEM/$\oplus\mathrm{I}_1\mathrm{F}$
$\left(E \circ \mathtt{Inl}\,[\,]\right)\left[v\right] \longrightarrow E\left[\mathtt{Inl}\,v\right]$		λ_{Ldm}–SEM/$\oplus\mathrm{I}_1\mathrm{U}$
$E\left[\mathtt{Inr}\,t\right] \longrightarrow \left(E \circ \mathtt{Inr}\,[\,]\right)\left[t\right]$	$\star$	λ_{Ldm}–SEM/$\oplus\mathrm{I}_2\mathrm{F}$
$\left(E \circ \mathtt{Inr}\,[\,]\right)\left[v\right] \longrightarrow E\left[\mathtt{Inr}\,v\right]$		λ_{Ldm}–SEM/$\oplus\mathrm{I}_2\mathrm{U}$
$E\left[(t_1\,,\,t_2)\right] \longrightarrow \left(E \circ ([\,]\,,\,t_2)\right)\left[t_1\right]$	$\star$	λ_{Ldm}–SEM/$\otimes\mathrm{IF}_1$
$\left(E \circ ([\,]\,,\,t_2)\right)\left[v_1\right] \longrightarrow E\left[(v_1\,,\,t_2)\right]$		λ_{Ldm}–SEM/$\otimes\mathrm{IU}_1$
$E\left[(v_1\,,\,t_2)\right] \longrightarrow \left(E \circ (v_1\,,\,[\,])\right)\left[t_2\right]$	$\star$	λ_{Ldm}–SEM/$\otimes\mathrm{IF}_2$
$\left(E \circ (v_1\,,\,[\,])\right)\left[v_2\right] \longrightarrow E\left[(v_1\,,\,v_2)\right]$		λ_{Ldm}–SEM/$\otimes\mathrm{IU}_2$
$E\left[\mathtt{Mod}_{n}\,t\right] \longrightarrow \left(E \circ \mathtt{Mod}_{n}\,[\,]\right)\left[t\right]$	$\star$	λ_{Ldm}–SEM/$!\mathrm{IF}$
$\left(E \circ \mathtt{Mod}_{n}\,[\,]\right)\left[v\right] \longrightarrow E\left[\mathtt{Mod}_{n}\,v\right]$		λ_{Ldm}–SEM/$!\mathrm{IU}$
$E\left[t'\;t\right] \longrightarrow \left(E \circ t'\,[\,]\right)\left[t\right]$	$\star$	λ_{Ldm}–SEM/$\multimap\mathrm{EF}_1$
$\left(E \circ t'\,[\,]\right)\left[v\right] \longrightarrow E\left[t'\;v\right]$		λ_{Ldm}–SEM/$\multimap\mathrm{EU}_1$
$E\left[t'\;v\right] \longrightarrow \left(E \circ [\,]\;v\right)\left[t'\right]$	$\star$	λ_{Ldm}–SEM/$\multimap\mathrm{EF}_2$
$\left(E \circ [\,]\;v\right)\left[v'\right] \longrightarrow E\left[v'\;v\right]$		λ_{Ldm}–SEM/$\multimap\mathrm{EU}_2$
$E\left[(\boldsymbol{\lambda} x\ {}_{m}\!\mapsto u)\;v\right] \longrightarrow E\left[u[x \coloneqq v]\right]$		λ_{Ldm}–SEM/$\multimap\mathrm{EC}$
$E\left[t \,⨟\, u\right] \longrightarrow \left(E \circ [\,] \,⨟\, u\right)\left[t\right]$	$\star$	λ_{Ldm}–SEM/$1\mathrm{EF}$
$\left(E \circ [\,] \,⨟\, u\right)\left[v\right] \longrightarrow E\left[v \,⨟\, u\right]$		λ_{Ldm}–SEM/$1\mathrm{EU}$
$E\left[() \,⨟\, u\right] \longrightarrow E\left[u\right]$		λ_{Ldm}–SEM/$1\mathrm{EC}$
$E\left[\mathtt{case}_{m}\,t\,\mathtt{of}\,\{\mathtt{Inl}\,x_1 \mapsto u_1\,,\ \mathtt{Inr}\,x_2 \mapsto u_2\}\right] \longrightarrow \left(E \circ \mathtt{case}_{m}\,[\,]\,\mathtt{of}\,\{\mathtt{Inl}\,x_1 \mapsto u_1\,,\ \mathtt{Inr}\,x_2 \mapsto u_2\}\right)\left[t\right]$	$\star$	λ_{Ldm}–SEM/$\oplus\mathrm{EF}$
$\left(E \circ \mathtt{case}_{m}\,[\,]\,\mathtt{of}\,\{\mathtt{Inl}\,x_1 \mapsto u_1\,,\ \mathtt{Inr}\,x_2 \mapsto u_2\}\right)\left[v\right] \longrightarrow E\left[\mathtt{case}_{m}\,v\,\mathtt{of}\,\{\mathtt{Inl}\,x_1 \mapsto u_1\,,\ \mathtt{Inr}\,x_2 \mapsto u_2\}\right]$		λ_{Ldm}–SEM/$\oplus\mathrm{EU}$
$E\left[\mathtt{case}_{m}\,(\mathtt{Inl}\,v_1)\,\mathtt{of}\,\{\mathtt{Inl}\,x_1 \mapsto u_1\,,\ \mathtt{Inr}\,x_2 \mapsto u_2\}\right] \longrightarrow E\left[u_1[x_1 \coloneqq v_1]\right]$		λ_{Ldm}–SEM/$\oplus\mathrm{EC}_1$
$E\left[\mathtt{case}_{m}\,(\mathtt{Inr}\,v_2)\,\mathtt{of}\,\{\mathtt{Inl}\,x_1 \mapsto u_1\,,\ \mathtt{Inr}\,x_2 \mapsto u_2\}\right] \longrightarrow E\left[u_2[x_2 \coloneqq v_2]\right]$		λ_{Ldm}–SEM/$\oplus\mathrm{EC}_2$
$E\left[\mathtt{case}_{m}\,t\,\mathtt{of}\,(x_1\,,\,x_2) \mapsto u\right] \longrightarrow \left(E \circ \mathtt{case}_{m}\,[\,]\,\mathtt{of}\,(x_1\,,\,x_2) \mapsto u\right)\left[t\right]$	$\star$	λ_{Ldm}–SEM/$\otimes\mathrm{EF}$
$\left(E \circ \mathtt{case}_{m}\,[\,]\,\mathtt{of}\,(x_1\,,\,x_2) \mapsto u\right)\left[v\right] \longrightarrow E\left[\mathtt{case}_{m}\,v\,\mathtt{of}\,(x_1\,,\,x_2) \mapsto u\right]$		λ_{Ldm}–SEM/$\otimes\mathrm{EU}$
$E\left[\mathtt{case}_{m}\,(v_1\,,\,v_2)\,\mathtt{of}\,(x_1\,,\,x_2) \mapsto u\right] \longrightarrow E\left[u[x_1 \coloneqq v_1][x_2 \coloneqq v_2]\right]$		λ_{Ldm}–SEM/$\otimes\mathrm{EC}$
$E\left[\mathtt{case}_{m}\,t\,\mathtt{of}\,\mathtt{Mod}_{n}\,x \mapsto u\right] \longrightarrow \left(E \circ \mathtt{case}_{m}\,[\,]\,\mathtt{of}\,\mathtt{Mod}_{n}\,x \mapsto u\right)\left[t\right]$	$\star$	λ_{Ldm}–SEM/$!\mathrm{EF}$
$\left(E \circ \mathtt{case}_{m}\,[\,]\,\mathtt{of}\,\mathtt{Mod}_{n}\,x \mapsto u\right)\left[v\right] \longrightarrow E\left[\mathtt{case}_{m}\,v\,\mathtt{of}\,\mathtt{Mod}_{n}\,x \mapsto u\right]$		λ_{Ldm}–SEM/$!\mathrm{EU}$
$E\left[\mathtt{case}_{m}\,\mathtt{Mod}_{n}\,v\,\mathtt{of}\,\mathtt{Mod}_{n}\,x \mapsto u\right] \longrightarrow E\left[u[x \coloneqq v]\right]$		λ_{Ldm}–SEM/$!\mathrm{EC}$

$\star$: only allowed if the term under focus is not already a value

Figure 1.9: Small-step semantics for λ_{Ldm}

Chapter 2

A formal functional language based on first-class destinations: λ_d

In the previous chapter we laid out important building blocks for the functional language that we'll develop here, namely λ_d. But first, let's focus again on destination-passing style programming, and why it matters in a functional setting.

2.1 Destination-passing style programming

As brushed out in the introduction, in destination-passing style, a function doesn't return a value: it takes as an argument a location where the output of the function ought to be written. In this chapter, we will denote destinations by $\lfloor \mathtt{T} \rfloor$ where $\mathtt{T}$ is the type of what can be stored in the destination. A function of type $\mathtt{T} \to \mathtt{U}$ would have signature $\mathtt{T} \to \lfloor \mathtt{U} \rfloor \to \mathtt{1}$ instead when transformed into destination-passing style, where instead of returning a $\mathtt{U}$, it consumes a destination of type $\lfloor \mathtt{U} \rfloor$ instead and returns just *unit* (type $\mathtt{1}$). This style is common in system programming, where destinations $\lfloor \mathtt{U} \rfloor$ are more commonly known as *out parameters* (in C, $\lfloor \mathtt{U} \rfloor$ would typically be a pointer of type $\mathtt{U}*$).

The reason why system programs rely on destinations so much is that using destinations can save calls to the memory allocator. If a function returns a $\mathtt{U}$, it has to allocate the space for a $\mathtt{U}$. But with destinations, the caller is responsible for finding space for a $\mathtt{U}$. The caller may simply ask for the space to the memory allocator, in which case we've saved nothing; but it can also reuse the space of an existing $\mathtt{U}$ which it doesn't need anymore, or space in an array, or even space in a region of memory that the allocator doesn't have access to, like a memory-mapped file. In fact, as we'll see in Section 2.5, destination-passing style is always more general than its direct style counterpart; there is no drawbacks to use it as we can always recover regular direct style functions in a systematic way from the destination-passing ones.

So far we mostly considered destination passing in imperative contexts, such as systems programming, but we argue that destination passing also presents many benefits for functional programming languages, even pure ones. Where destinations truly shine in functional programming is that they let us borrow imperative-like programming techniques to get a more expressive language, with more control over processing, and with careful attention, we can still preserve the nice properties of functional languages such as purity and (apparent) immutability.

Thus, the goal here will be to extend a rather-standard functional programming language just enough to be able to build immutable structures by destination passing without endangering purity and memory safety. Destinations in that particular setting become *write-once-only* references into a structure that contains holes and that cannot be read yet. Quite more restrictive than C pointers (that can be used in read and write fashion without restrictions), but still powerful enough to bring new programming techniques into the functional world.

More precisely, there are two key elements that make the extra expressiveness of destination passing in functional contexts:

- structures can be built in any order. Not only from the leaves to the root, like in ordinary functional programming, but also from the root to the leaves, or any combination thereof. This can be done in ordinary functional programming using function composition in a form of continuation-passing; however destinations act as a very direct optimization of this scheme. This line of work was pioneered by Minamide [Min98];
- when destinations are first-class values, they can be passed and stored like ordinary values. The consequence is that not only the order in which a structure is built is arbitrary, this order can be determined dynamically during the runtime of the program. Here is the true novelty of our approach.

In this chapter, we introduce λ_d, as a pure functional calculus that is based on the very concept of destinations at its core. We intend λ_d to serve as a foundational, theoretical calculus to reason about safe destinations in a functional setting; thus, we will only cover very briefly the implementation concerns in this chapter, as λ_d is not really meant to be implemented as a real programming language. Actual implementation of destination passing in an existing functional language will be the focus of the next chapters.

λ_d is supposed to subsume all the functional systems with destinations from the literature, from [Min98] to [Lor+24b]. As such we expect that these systems or their extensions can be justified simply by giving them a translation into λ_d, in order to get all the safety results and metatheory of λ_d for free. In particular, we proved type safety theorems for λ_d with the Coq proof assistant, as described in Section 2.7.

2.2 Working with destinations

Let's introduce and get familiar with λ_d, our modal, linear, simply typed λ-calculus with destinations λ-calculus with destination. We borrow most of the syntax of the previous chapter, especially from λ_{Ldm}. We still use linear logic's $\mathsf{T}\oplus\mathsf{U}$ and $\mathsf{T}\otimes\mathsf{U}$ for sums and products, and linear function arrow $\multimap$, since λ_d is linearly typed. The main difference with previous calculi being that λ_d doesn't have first-class data constructors like $\mathsf{Inl}\ t$ or $(t_1\ ,\ t_2)$, as they are replaced with the more general destination-filling operators that we'll discover in the next paragraphs.

2.2.1 Building up a vocabulary

In its simplest form, destination passing, much like continuation passing, is using a location, received as an argument, to write a the output value instead of returning it proper. Instead of a linear function with signature $\mathsf{T} \multimap \mathsf{U}$, in λ_d you would have $\mathsf{T} \multimap \lfloor\mathsf{U}\rfloor \multimap \mathbf{1}$, where $\lfloor\mathsf{U}\rfloor$ is read "destination for type U". For instance, here is a destination-passing version of the identity function:

$$\begin{aligned}&\textbf{dId} : \mathsf{T} \multimap \lfloor \mathsf{T} \rfloor \multimap \mathbf{1}\\ &\textbf{dId}\ x\ d \ \triangleq\ d \blacktriangleleft x\end{aligned}$$

We think of a destination as a reference to an uninitialized memory location, and $d \blacktriangleleft x$ (read "fill d with x") as writing x to the memory location pointed to by d.

The form $d \blacktriangleleft x$ is the simplest way to use a destination: with that we fill it with a whole, complete value. But we don't have to fill a destination with a complete value in a single step; instead, we can fill a destination piecemeal by specifying just the outermost constructor that we want to fill it with:

$$\begin{aligned}&\textbf{fillWithInl} : \lfloor \mathsf{T} \oplus \mathsf{U} \rfloor \multimap \lfloor \mathsf{T} \rfloor\\ &\textbf{fillWithInl}\ d \ \triangleq\ d \triangleleft \mathsf{Inl}\end{aligned}$$

In this example, we're filling a destination for type $\mathsf{T} \oplus \mathsf{U}$ by setting the outermost constructor to left variant Inl. We think of $d \triangleleft \mathsf{Inl}$ (read "fill d with Inl") as allocating memory to store a block of the form $\mathsf{Inl}\ \boxed{}$, write the address of that block to the location that d points to, and return a new destination of type $\lfloor \mathsf{T} \rfloor$ pointing to the uninitialized argument of Inl. Uninitialized memory, when part of a structure or value, like $\boxed{}$ in $\mathsf{Inl}\ \boxed{}$, is called a *hole*.

Notice that with **fillWithInl** we are constructing the structure from the outermost constructor inward: we've written a value of the form $\mathsf{Inl}\ \boxed{}$ into a hole, but we have yet to describe what goes in the new hole $\boxed{}$. Such data constructors with uninitialized arguments are called *hollow constructors*[5]. This is opposite to how functional programming usually works, where values are built from the innermost constructors outward: first we make a value v and only then can we use Inl to make $\mathsf{Inl}\ v$. This is one of the two key ingredients in the expressiveness of destination passing that we mentioned earlier.

Yet, everything we've shown so far could have been done with continuations. So it's worth asking: how are destinations different from continuations? Part of the answer lies in our intention to effectively implement destinations as pointers to uninitialized memory (see Section 3.5). But where destinations really differ from continuations is when one has several destinations at hand. Then they have to fill *all* the destinations; whereas when one has multiple continuations, they can only return to one of them. Multiple destination arises when a destination for a pair gets filled with a hollow pair constructor:

$$\begin{aligned}&\textbf{fillWithPair} : \lfloor \mathsf{T} \otimes \mathsf{U} \rfloor \multimap \lfloor \mathsf{T} \rfloor \otimes \lfloor \mathsf{U} \rfloor\\ &\textbf{fillWithPair}\ d \ \triangleq\ d \triangleleft (,)\end{aligned}$$

After using **fillWithPair**, both the first field *and* the second field must be filled, using the destinations of type $\lfloor \mathsf{T} \rfloor$ and $\lfloor \mathsf{U} \rfloor$ respectively. The key remark here is that **fillWithPair** couldn't exist if we replaced destinations by continuations, as we couldn't use both returned continuations easily.

We can already note there that there is a sort of duality between destination-filling operators and associated constructors. Usual Inl constructor has signature $\mathsf{T} \multimap \mathsf{T} \oplus \mathsf{U}$, while destination-filling $\triangleleft$Inl has signature $\lfloor \mathsf{T} \oplus \mathsf{U} \rfloor \multimap \lfloor \mathsf{T} \rfloor$. Similarly, pair constructor (,) has signature $\mathsf{T} \multimap \mathsf{U} \multimap \mathsf{T} \otimes \mathsf{U}$, while destination-filling $\triangleleft$(,) has signature $\lfloor \mathsf{T} \otimes \mathsf{U} \rfloor \multimap \lfloor \mathsf{T} \rfloor \otimes \lfloor \mathsf{U} \rfloor$. Assuming some flexibility with

[5]The full triangle $\blacktriangleleft$ is used to fill a destination with a fully-formed value, while the *hollow* triangle $\triangleleft$ is used to fill a destination with a *hollow constructor*.

currying, we see that the types of arguments and results switch sides around the arrow, and get wrapped/unwrapped from $\lfloor\cdot\rfloor$ when we go from the constructor to the destination-filling operator and vice-versa. This observation will generalize to all destination-filling operators and the corresponding constructors.

Structures with holes It is crucial to note that while a destination is used to build a structure, the destination refers only to a specific part of the structure that hasn't been defined yet; not the structure as a whole. Consequently, the (root) type of the structure being built will often be different from the type of the destination at hand. A destination of type $\lfloor T\rfloor$ only indicates that some bigger structure has at least a hole of type T somewhere in it. The type of the structure itself never appears in the signature of destination-filling functions (for instance, using **fillWithPair** only indicates that the structure being operated on has a hole of type $T\otimes U$ that is being written to).

Thus, we still need a type to tie the structure under construction — left implicit by destination-filling primitives — with the destinations representing its holes. To represent this, λ_d introduces a type $S \ltimes \lfloor T\rfloor$ for a structure of type S missing a value of type T to be complete. There can be several holes in S — resulting in several destinations on the right hand side — and as long as there remains holes in S, it cannot be read. For instance, $S \ltimes (\lfloor T\rfloor \otimes \lfloor U\rfloor)$ represents a S that misses both a T and a U to be complete (thus to be readable). The right hand side of $\ltimes$ is not restricted to pair and destination types only; it can be of arbitrarily complex type.

The general form $S \ltimes T$ is read "S ampar T". The name "ampar" stands for "asymmetric memory par". The "par" $⅋$ operator originally comes from (classical) linear logic, and is an associative and commutative operator that can be used in place of linear implication and is slightly more flexible: $T \multimap S$ is equivalent to $T^{\perp} ⅋ S$ or $S ⅋ T^{\perp}$. Similarly, $T_1 \multimap T_2 \multimap S$ is equivalent to $T1^{\perp} ⅋ T2^{\perp} ⅋ S$. Function input's types —which are in negative position— are wrapped in the dualizing operator $\cdot^{\perp}$ when a function is put into "par" form. Here we take sort of the same approach: Minamide [Min98] first observed that structures with holes are akin to linear functions $T \multimap S$, where T represents the missing part; so we decide to represent them in a more flexible fashion under the form $S \ltimes \lfloor T\rfloor$, so that destinations are made into a first-class type $\lfloor T\rfloor$. The asymmetric nature of our memory par is a bit disappointing, but it comes from limitations that we are in an intuitionistic setting, not a classical one like $⅋$ needs.

Destinations, albeit being first-class, always exist within the context of a structure with holes. A destination is both a witness of a hole present in the structure, and a handle to write to it. Crucially, destinations are otherwise ordinary values. To access the destinations of an ampar, λ_d provides a $\mathbf{upd}_{\ltimes}$ construction, which lets us apply a function to the right-hand side of the ampar. It is in the body of $\mathbf{upd}_{\ltimes}$ that functions operating on destinations can be called to update the structure:

$$\mathtt{fillWithInl'} : S \ltimes \lfloor T\oplus U\rfloor \multimap S \ltimes \lfloor T\rfloor$$
$$\mathtt{fillWithInl'}\ x \triangleq \mathbf{upd}_{\ltimes}\ x\ \mathtt{with}\ d \mapsto \mathtt{fillWithInl}\ d$$
$$\mathtt{fillWithPair'} : S \ltimes \lfloor T\otimes U\rfloor \multimap S \ltimes (\lfloor T\rfloor \otimes \lfloor U\rfloor)$$
$$\mathtt{fillWithPair'}\ x \triangleq \mathbf{upd}_{\ltimes}\ x\ \mathtt{with}\ d \mapsto \mathtt{fillWithPair}\ d$$

To tie this up, we need a way to introduce and to eliminate structures with holes. Structures with holes are introduced with $\mathtt{new}_{\ltimes}$ which creates a value of type $\mathtt{T} \ltimes \lfloor \mathtt{T} \rfloor$. $\mathtt{new}_{\ltimes}$ is a bit like the identity function: it is a hole (of type $\mathtt{T}$) that needs a value of type $\mathtt{T}$ to be a complete value of type $\mathtt{T}$. Memory-wise, it is an uninitialized block large enough to host a value of type $\mathtt{T}$, and a destination pointing to it. Conversely, structures with holes are eliminated with[6] $\mathtt{from}'_{\ltimes} : \mathtt{S} \ltimes \mathtt{1} \multimap \mathtt{S}$: if all the destinations have been consumed and only unit remains on the right side, then $\mathtt{S}$ no longer has holes and thus is just a normal, complete structure.

Equipped with these, we can, for instance, derive traditional constructors from piecemeal filling. Indeed, as we said earlier, λ_d doesn't have primitive constructor forms, constructors in λ_d are syntactic sugar. We show here the definition of $\mathtt{Inl}$ and $(,)$, but the other constructors are derived similarly. Operator $\fatsemi$, present in second example, is used to chain operations returning unit type $\mathtt{1}$.

$$
\begin{array}{l}
\mathtt{Inl} \;:\; \mathtt{T} \multimap \mathtt{T} \oplus \mathtt{U} \\
\mathtt{Inl}\, x \;\triangleq\; \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes}\,(\mathtt{new}_{\ltimes} : (\mathtt{T}\oplus\mathtt{U}) \ltimes \lfloor \mathtt{T}\oplus\mathtt{U} \rfloor)\ \mathtt{with}\ d \mapsto d \triangleleft \mathtt{Inl} \blacktriangleleft x) \\
(,) \;:\; \mathtt{T} \multimap \mathtt{U} \multimap \mathtt{T} \otimes \mathtt{U} \\
(x\,,\,y) \;\triangleq\; \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes}\,(\mathtt{new}_{\ltimes} : (\mathtt{T}\otimes\mathtt{U}) \ltimes \lfloor \mathtt{T}\otimes\mathtt{U} \rfloor)\ \mathtt{with}\ d \mapsto \mathtt{case}\ (d \triangleleft (,))\ \mathtt{of}\ (d_1\,,\,d_2) \mapsto d_1 \blacktriangleleft x \fatsemi d_2 \blacktriangleleft y)
\end{array}
$$

Memory safety and purity We must reassure the reader here. Of course, using destinations in an unrestricted fashion is not memory safe. We need a linear discipline on destinations for them to be safe. Otherwise, we can encounter two sorts of issues:

- if destinations are not written at least once, as in:

  $$
  \begin{array}{l}
  \mathtt{forget} \;:\; \mathtt{T} \\
  \mathtt{forget} \;\triangleq\; \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes}\,(\mathtt{new}_{\ltimes} : \mathtt{T} \ltimes \lfloor \mathtt{T} \rfloor)\ \mathtt{with}\ d \mapsto ())
  \end{array}
  $$

  then the result of $\mathtt{forget}$ would result in reading the location pointed to by a destination that we never used, in other words, reading uninitialized memory. That's clearly the biggest issue we must prevent;
- if destinations are written several times, as in:

  $$
  \begin{array}{l}
  \mathtt{ambiguous1} \;:\; \mathtt{Bool} \\
  \mathtt{ambiguous1} \;\triangleq\; \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes}\,(\mathtt{new}_{\ltimes} : \mathtt{Bool} \ltimes \lfloor \mathtt{Bool} \rfloor)\ \mathtt{with}\ d \mapsto d \blacktriangleleft \mathtt{true} \fatsemi d \blacktriangleleft \mathtt{false}) \\
  \mathtt{ambiguous2} \;:\; \mathtt{Bool} \\
  \mathtt{ambiguous2} \;\triangleq\; \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes}\,(\mathtt{new}_{\ltimes} : \mathtt{Bool} \ltimes \lfloor \mathtt{Bool} \rfloor)\ \mathtt{with}\ d \mapsto \mathtt{let}\ x := (d \blacktriangleleft \mathtt{false})\ \mathtt{in}\ d \blacktriangleleft \mathtt{true} \fatsemi x)
  \end{array}
  $$

  then we have $\mathtt{ambiguous1}$ that returns $\mathtt{false}$ and $\mathtt{ambiguous2}$ that returns $\mathtt{true}$ due to evaluation order, so we break *let expansion* that is supposed to be valid in a functional programming language. We ought to keep that property true for λ_d.

Actually, we'll see in Section 2.3 that a linear discipline is not even enough to ensure fully safe use of destinations in λ_d. But before dealing with that, let's get more familiar with λ_d keywords and operators through more complex examples.

[6]As the name suggest, there is a more general elimination $\mathtt{from}_{\ltimes}$. It will be discussed in Section 2.5.

2.2.2 Tail-recursive map

Let's see how destinations can be used to build usual data structures. For these examples, we suppose that λ_d has equirecursive types and a fixed-point operator. These aren't part of the formal system of Section 2.5 but don't add any complication.

Linked lists We define lists as a fixpoint, as usual: $\mathsf{List}\ \mathsf{T} \triangleq 1 \oplus (\mathsf{T} \otimes (\mathsf{List}\ \mathsf{T}))$. For convenience, we also define filling operators $\triangleleft[]$ and $\triangleleft(::)$, as macros that use the primitive destination-filling operators for sum, product and unit types:

$$\begin{array}{ll|ll} \triangleleft[]\ : & \lfloor \mathsf{List}\ \mathsf{T} \rfloor \multimap 1 & \triangleleft(::)\ : & \lfloor \mathsf{List}\ \mathsf{T} \rfloor \multimap \lfloor \mathsf{T} \rfloor \otimes \lfloor \mathsf{List}\ \mathsf{T} \rfloor \\ d \triangleleft [] \triangleq & d \triangleleft \mathsf{Inl} \triangleleft () & d \triangleleft (::) \triangleq & d \triangleleft \mathsf{Inr} \triangleleft (,) \end{array}$$

Just like we did in Section 2.2.1 we can recover traditional constructors systematically from destination-filling operators, using $\mathbf{new}_{\ltimes}$, $\mathbf{upd}_{\ltimes}$ and $\mathbf{from}'_{\ltimes}$:

$$\begin{array}{l} (::)\ :\ \mathsf{T} \otimes (\mathsf{List}\ \mathsf{T}) \multimap \mathsf{List}\ \mathsf{T} \\ x :: xs\ \triangleq\ \mathbf{from}'_{\ltimes}(\mathbf{upd}_{\ltimes}\,(\mathbf{new}_{\ltimes} : (\mathsf{List}\ \mathsf{T}) \ltimes \lfloor \mathsf{List}\ \mathsf{T} \rfloor)\ \mathbf{with}\ d \mapsto \\ \qquad\qquad\qquad \mathbf{case}\ (d \triangleleft (::))\ \mathbf{of}\ (dx\,,\, dxs) \mapsto dx \blacktriangleleft x\ ⨾\ dxs \blacktriangleleft xs) \end{array}$$

A tail-recursive map function List being ubiquitous in functional programming, the fact that the most natural way to write a **map** function on lists isn't tail recursive (hence consumes unbounded stack space), is unpleasant. Map can be made tail-recursive in two passes: first build the result list in reverse, then reverse it. But thanks to destinations, we'll be able to avoid this two-pass process altogether, as they let us extend the tail of the result list directly. We give the complete implementation in Figure 2.1.

The tail-recursive function is $\mathbf{map}'$, it has type $(\mathsf{T} \multimap \mathsf{U})\ {}_{\omega\infty}\!\multimap \mathsf{List}\ \mathsf{T} \multimap \lfloor \mathsf{List}\ \mathsf{U} \rfloor \multimap 1$. That is, instead of returning a resulting list, it takes a destination as an input and fills it with the result. At each recursive call, $\mathbf{map}'$ creates a new hollow cons cell to fill the destination. A destination pointing to the tail of the new cons cell is also created, on which $\mathbf{map}'$ is called (tail) recursively. This is really the same algorithm that you could write to implement map on a mutable list in an imperative language. Nevertheless λ_d is a pure language with only immutable types[7].

To obtain the regular **map** function, all is left to do is to call $\mathbf{new}_{\ltimes}$ to create an initial destination, and $\mathbf{from}'_{\ltimes}$ when all destinations have been filled to extract the completed list; much like when we make constructors out of filling operators, like (::) above.

2.2.3 Functional queues, with destinations

Implementations for a tail-recursive map are present in most previous work, from [Min98], to recent work [BCS21; LL23]. Tail-recursive map doesn't need the full power of λ_d's first-class destinations: it just needs a notion of structures with a (single) hole. In Section 2.4, we will build an example which fully uses first-class destinations, but first, we need some more material.

[7]We rely here on the fact that structure with holes cannot be scrutinized while they haven't been fully completed, so destination-related mutations are completely opaque and unobservable in λ_d.

```
List T ≜ 1⊕(T⊗(List T))
map′ : (T ⊸ U) ω∞⊸ List T ⊸ ⌊List U⌋ ⊸ 1
map′ f l dl ≜
   case l of {
      [] ↦ dl ◁ [] ,
      x :: xs ↦ case (dl ◁ (::)) of
         (dx , dxs) ↦ dx ◀ f x ⨟ map′ f xs dxs}
map : (T ⊸ U) ω∞⊸ List T ⊸ List U
map f l ≜ from′⋉ (upd⋉ (new⋉ : (List U) ⋉ ⌊List U⌋) with dl ↦ map′ f l dl)
```

Figure 2.1: Tail-recursive **map** function on lists

Difference lists Just like in any language, iterated concatenation of linked lists $((xs_1 +\!\!+ xs_2) +\!\!+ \ldots) +\!\!+ xs_n$ is quadratic in λ_d. The usual solution to improve that complexity is *difference lists*. The name *difference lists* covers many related implementations for the concept of a "linked list missing a queue". The idea is that a difference list carries the same elements as a list would, but can be easily extended by the back in constant time as we retain a way to set a value for its queue later. In pure functional languages, a difference list is usually represented as a function instead [Hug86], as we usually don't have write pointers. A singleton difference list is then $\lambda ys \mapsto x :: ys$, and concatenation of difference lists is function composition. A difference is turned into a list by setting its queue to be the empty list, or in the functional case, by applying it to the empty list. The consequence is that no matter how many compositions we have, each cons cell will be allocated a single time, making the iterated concatenation linear indeed.

The problem is that in the functional implementation, each concatenation still allocates a closure. If we're building a difference list from singletons and composition, there's roughly one composition per cons cell, so iterated composition effectively performs two traversals of the list. In λ_d, we actually have write pointers, in the form of *destinations*, so we can do better by representing a difference list as a list with a hole, much like in an imperative setting. A singleton difference list becomes $x :: \boxed{}$, and concatenation is filling the hole with another difference list, using composition operator $\triangleleft\!\circ$. The detailed implementation is given on the left of Figure 2.2. This encoding for difference lists makes no superfluous traversal: concatenation is just an $O(1)$ in-place update.

Efficient queue using difference lists In an immutable functional language, a queue can be implemented as a pair of lists $(front\,, back)$ [HM81]. $back$ stores new elements in reverse order ($O(1)$ prepend). We pop elements from $front$, except when it is empty, in which case we set the queue to (**reverse** $back$, []), and pop from the new front.

For their simple implementation, Hood-Melville queues are surprisingly efficient: the cost of the reverse operation is $O(1)$ amortized for a single-threaded use of the queue. Still, it would be better to get rid of this full traversal of the back list. Taking a step back, this $back$ list that has to be reversed before it is accessed is really merely a representation of a list that can be extended from the back. And we already know an efficient implementation for this: difference lists.

```
DList T ≜ (List T) ⋉ ⌊List T⌋

append : DList T ⊸ T ⊸ DList T
ys append y ≜
   upd⋉ ys with dys ↦ case (dys ◁ (::)) of
      (dy , dys′) ↦ dy ◀ y ⨟ dys′

concat : DList T ⊸ DList T ⊸ DList T
ys concat ys′ ≜ upd⋉ ys with d ↦ d ◁∘ ys′

toList : DList T ⊸ List T
toList ys ≜ from′⋉ (upd⋉ ys with d ↦ d ◁ [])
```

```
Queue T ≜ (List T)⊗(DList T)

singleton : T ⊸ Queue T
singleton x ≜ (Inr (x :: []) , (new⋉ : DList T))

enqueue : Queue T ⊸ T ⊸ Queue T
q enqueue y ≜
   case q of (xs , ys) ↦ (xs , ys append y)

dequeue : Queue T ⊸ 1⊕(T⊗(Queue T))
dequeue q ≜
   case q of {
      ((x :: xs) , ys) ↦ Inr (x , (xs , ys)) ,
      ([] , ys) ↦ case (toList ys) of {
         [] ↦ Inl () ,
         x :: xs ↦ Inr (x , (xs , (new⋉ : DList T)))}}
```

Figure 2.2: Difference list and queue implementation in equirecursive λ_d

So we can give an improved version of the simple functional queue using destinations. This implementation is presented on the right-hand side of Figure 2.2. Note that contrary to an imperative programming language, we can't implement the queue as a single difference list: as mentioned earlier, our type system prevents us from reading the front elements of difference lists. Just like for the simple functional queue, we need a pair of one list that we can read from, and one that we can extend. Nevertheless this implementation of queues is both pure, as guaranteed by the λ_d type system, and nearly as efficient as what an imperative programming language would afford.

2.3 Scope escape of destinations

In Section 2.2, we've been making an implicit assumption: establishing a linear discipline on destinations ensures that all destinations will eventually find their way to the left of a fill operator ◀ or ◁, so that the associated holes get written to. This turns out to be slightly incomplete.

To see why, let's consider the type ⌊⌊T⌋⌋: the type of a destination pointing to a hole where a destination is expected. Think of it as an equivalent of the pointer type T ∗∗ in the C language. Destinations are indeed ordinary values, so they can be stored in data structures, and before they get stored, holes stand in their place in the structure. For instance, if we have *d* : ⌊T⌋ and *dd* : ⌊⌊T⌋⌋, we can form *dd* ◀ *d*: *d* will be stored in the structure pointed to by *dd*.

Should we count *d* as linearly used here? The alternatives don't seem promising:

- If we count this as a non-linear use of *d*, then *dd* ◀ *d* is rejected since destinations (represented here by *d*) can only be used linearly. This choice is fairly limiting, as it would prevent us from storing destinations in structures with holes, as we do, crucially, in Section 2.4.

- If we do not count this use of *d* at all, we can write *dd* ◀ *d* ⨟ *d* ◀ v so that *d* is both stored for later use *and* filled immediately (resulting in the corresponding hole being potentially written to twice), which is unsound, as discussed in Section 2.2.1.

So linear use it is. But it creates a problem: there's no way, within our linear type system, to distinguish between "a destination has been used on the left of a triangle so its corresponding hole has been filled" and "a destination has been stored and its hole still exists at the moment". This oversight may allow us to read uninitialized memory!

Let's compare two examples. We assume a simple store semantics for now where structures with holes stay in the store until they are completed. We'll need the $\mathtt{alloc} : (\lfloor \mathsf{T} \rfloor \multimap \mathsf{1}) \multimap \mathsf{T}$ operator. The semantics of **alloc** is: allocate a structure with a single root hole in the store, call the supplied function with the destination to the root hole as an argument; when the function has consumed all destinations (so only unit remains), pop the structure from the store to obtain a complete T.

In this snippet, structures with holes are given names v and vd in the store; holes are given names too and denoted by $\boxed{h}$ and $\boxed{hd}$, and concrete destinations are denoted by $\rightarrow h$ and $\rightarrow hd$.

When the building scope of $v : \mathsf{Bool}$ is parent to the one of $vd : \lfloor \mathsf{Bool} \rfloor$, everything works well because vd, that contains destination pointing to $\boxed{h}$, has to be consumed before v can be read:

$$
\begin{array}{lr|l}
 & \{\ \} & \mathtt{alloc}\ (\lambda d \mapsto (\mathtt{alloc}\ (\lambda dd \mapsto dd \blacktriangleleft d) : \lfloor \mathsf{Bool} \rfloor) \blacktriangleleft \mathtt{true}) \\
\longrightarrow & \{v := \boxed{h}\} & (\mathtt{alloc}\ (\lambda dd \mapsto dd \blacktriangleleft \rightarrow h) : \lfloor \mathsf{Bool} \rfloor) \blacktriangleleft \mathtt{true} \fatsemi \mathtt{deref}\ v \\
\longrightarrow & \{v := \boxed{h}, vd := \boxed{hd}\} & (\rightarrow hd \blacktriangleleft \rightarrow h \fatsemi \mathtt{deref}\ vd) \blacktriangleleft \mathtt{true} \fatsemi \mathtt{deref}\ v \\
\longrightarrow & \{v := \boxed{h}, vd := \rightarrow h\} & \mathtt{deref}\ vd \blacktriangleleft \mathtt{true} \fatsemi \mathtt{deref}\ v \\
\longrightarrow & \{v := \boxed{h}\} & \rightarrow h \blacktriangleleft \mathtt{true} \fatsemi \mathtt{deref}\ v \\
\longrightarrow & \{v := \mathtt{true}\} & \mathtt{deref}\ v \\
\longrightarrow & \{\ \} & \mathtt{true}
\end{array}
$$

However, when vd's scope is parent to v's, we can write a linearly typed yet unsound program:

$$
\begin{array}{lr|l}
 & \{\ \} & \mathtt{alloc}\ (\lambda dd \mapsto \mathtt{case}\ (\mathtt{alloc}\ (\lambda d \mapsto dd \blacktriangleleft d) : \mathsf{Bool})\ \mathtt{of}\ \{\mathtt{true} \mapsto ()\,,\ \mathtt{false} \mapsto ()\}) \\
\longrightarrow & \{vd := \boxed{hd}\} & \mathtt{case}\ (\mathtt{alloc}\ (\lambda d \mapsto \rightarrow hd \blacktriangleleft d) : \mathsf{Bool})\ \mathtt{of}\ \{\mathtt{true} \mapsto ()\,,\ \mathtt{false} \mapsto ()\} \fatsemi \mathtt{deref}\ vd \\
\longrightarrow & \{vd := \boxed{hd}, v := \boxed{h}\} & \mathtt{case}\ (\rightarrow hd \blacktriangleleft \rightarrow h \fatsemi \mathtt{deref}\ v)\ \mathtt{of}\ \{\mathtt{true} \mapsto ()\,,\ \mathtt{false} \mapsto ()\} \fatsemi \mathtt{deref}\ vd \\
\longrightarrow & \{vd := \rightarrow h, v := \boxed{h}\} & \mathtt{case}\ (\mathtt{deref}\ v)\ \mathtt{of}\ \{\mathtt{true} \mapsto ()\,,\ \mathtt{false} \mapsto ()\} \fatsemi \mathtt{deref}\ vd \\
\longrightarrow & \{vd := \rightarrow h\} & \mathtt{case}\ \boxed{h}\ \mathtt{of}\ \{\mathtt{true} \mapsto ()\,,\ \mathtt{false} \mapsto ()\} \fatsemi \mathtt{deref}\ vd \qquad 💣💣💣
\end{array}
$$

Here the expression $dd \blacktriangleleft d$ results in d escaping its scope for the parent one, so v is just uninitialized memory (the hole $\boxed{h}$) when we dereference it. This example must be rejected by our type system.

Again, the problem is, using purely a linear type system, we can only reject this example if we also reject the first, sound example. In this case, the type $\lfloor \lfloor \mathsf{T} \rfloor \rfloor$ would become practically useless: such destinations can never be filled. This isn't the direction we want to take: we really want to be able to store destinations in data structures with holes. So we want t in $d \blacktriangleleft t$ to be allowed to be linear. Without further restrictions, it wouldn't be sound, so to address this, we need an extra control system to prevent scope escape; it can't be just linearity. For λ_d, we decided to use a system of *ages* to represent which scope a resource originates from. Ages are described in detail in Section 2.5; but first, let's see an exemple where storing destinations in data structures really matters.

2.4 Breadth-first tree traversal

As a full-fledged example, which uses the full expressive power of λ_d, we focus breadth-first tree relabeling: “ *Given a tree, create a new one of the same shape, but with the values at the nodes replaced by the numbers* $1 \ldots |T|$ *in breadth-first order.* ”

This isn’t a very natural problem for functional programming, as breadth-first traversal implies that the order in which the structure must be built (left-to-right, top-to-bottom) is not the same as the structural order of a functional tree — building the leaves first and going up to the root. So it usually requires fancy functional workarounds [Oka00; Gib93; Gib+23].

It’s very tempting to solve this exercise in an efficient imperative-like fashion, where a queue drives the processing order. That’s the standard algorithm taught at university, where the next node to process is dequeued from the front of the queue, and its children nodes are enqueued at the back of the queue for later processing, achieving breadth-first traversal. For that, Minamide [Min98]’s system where structures with holes are represented as linear functions cannot help. We really need destinations as first-class values to borrow from this imperative power.

Figure 2.3 presents how we can implement this breadth-first tree traversal in λ_d, thanks to first-class destinations. We assume the data type `Tree T` as been defined, unsurprisingly, as `Tree T` $\triangleq$ `1`$\oplus$`(T`$\otimes$`((Tree T)`$\otimes$`(Tree T)))`; and we refer to the constructors of `Tree T` as `Nil` and `Node`, defined as syntactic sugar as we did for the other constructors before. We also assume some encoding of the type `Nat` of natural number. Remember that `Queue T` is the efficient queue type from Section 2.2.3.

The core idea of our algorithm is that we hold a queue of pairs, storing each an input subtree and (a destination to) its corresponding output subtree. When the element (*tree* , *dtree*) at the front of the queue has been processed, the children nodes of *tree* and children destinations of *dtree* are enqueued to be processed later, much as the original imperative algorithm.

We implement the actual breadth-first relabeling **relabelDPS** as an instance of a more general breadth-first traversal function **mapAccumBFS**, which applies any state-passing style transformation of labels in breadth-first order. In **mapAccumBFS**, we create a new destination *dtree* into which we will write the result of the traversal, then call **go**. The **go** function is in destination-passing style, but what’s remarkable is that **go** takes an unbounded number of destinations as arguments, since there are as many destinations as items in the queue. This is where we use the fact that destinations are ordinary values.

This freshly gained expressivity has a cost though: we need a type system that combines linearity control *and* age control to make the system sound, as exemplified in the previous section. We’ll combine both linearity and the aforementioned ages in the same *mode* system[8]. You already know the linearity annotations 1 and ω; here we also introduce the new age annotation ∞, that indicates that the associated argument cannot carry destinations.Arguments with no modes displayed on them, or function arrows with no modes, default to the unit of the semiring/ringoid; in particular they are linear, and can capture destinations.

```
go : (S ω∞⊸ T₁ ⊸ (!ω∞S)⊗T₂) ω∞⊸ S ω∞⊸ Queue (Tree T₁⊗⌊Tree T₂⌋) ⊸ 1
         rec
go f st q  ≜  case (dequeue q) of {
                  Inl () ↦ () ,
                  Inr ((tree , dtree) , q′) ↦ case tree of {
                      Nil ↦ dtree ◁ Nil ⨾ go f st q′ ,
                      Node x tl tr ↦ case (dtree ◁ Node) of
                          (dy , (dtl , dtr)) ↦ case (f st x) of
                              (Modω∞ st′ , y) ↦
                                  dy ◀ y ⨾
                                  go f st′ (q′ enqueue (tl , dtl) enqueue (tr , dtr))}}
mapAccumBFS : (S ω∞⊸ T₁ ⊸ (!ω∞S)⊗T₂) ω∞⊸ S ω∞⊸ Tree T₁ 1∞⊸ Tree T₂
mapAccumBFS f st tree  ≜  from′⋉ (upd⋉ (new⋉ : (Tree T₂) ⋉ ⌊Tree T₂⌋) with dtree ↦
                                    go f st (singleton (tree , dtree)))
relabelDPS : Tree 1 1∞⊸ Tree Nat
relabelDPS tree  ≜  mapAccumBFS (λst ω∞↦ λun ↦ un ⨾ (Modω∞ (succ st) , st)) 1 tree
```

Figure 2.3: Breadth-first tree traversal in destination-passing style

2.5 Type system

λ_d is a simply typed λ-calculus with unit ($\mathbf{1}$), product ($\otimes$) and sum ($\oplus$) types, inspired from λ_{Ldm} from Chapter 1. Its most distinctive features are the destination $\lfloor_{m}\mathsf{T}\rfloor$ and ampar $\mathsf{S} \ltimes \mathsf{T}$ types which we've introduced in Sections 2.2 to 2.4.

To ensure that destinations are used soundly, we need both to enforce the linearity of destination but also to prevent destinations from escaping their scope, as discussed in Section 2.3. To that effect, λ_d tracks the *age* of destinations, that is how many nested scope have been open between the current expression and the scope from which a destination originates. We'll see in Section 2.5.2 that scopes are introduced by the $\mathbf{upd}_\ltimes\ t\ \mathbf{with}\ x \mapsto t'$ construct. For instance, we have a term $\mathbf{upd}_\ltimes\ t_1\ \mathbf{with}\ x_1 \mapsto \mathbf{upd}_\ltimes\ t_2\ \mathbf{with}\ x_2 \mapsto \mathbf{upd}_\ltimes\ t_3\ \mathbf{with}\ x_3 \mapsto x_1$, then wil will say that the innermost occurrence of x_1 has age $\uparrow^2$ because two nested $\mathbf{upd}_\ltimes$ separate the definition and use site of x_1.

For λ_d, we take the same modal approach as for λ_{Ldm}, but we enrich our mode ringoid to have an *age* axis. Thanks to the algebraic nature of the modal approach, for most typing rules, we'll be able to reuse those of λ_{Ldm} without any modification as just the elements of the ringoid change, not the properties nor the structure of the ringoid.

The syntax of λ_d terms is presented in Figure 2.4, including the syntactic sugar that we've been using in Sections 2.2 to 2.4.

[8]We said earlier that the modal approach would allow us to combine several control axes efficiently in the same mode system.

Core grammar of terms:

$$
\begin{aligned}
t, u \ ::= \ & x \ \mid \ t' \ t \ \mid \ t \ ⨟ \ t' \\
\mid \ & \mathtt{case}_{m}\ t\ \mathtt{of}\ \{\mathtt{Inl}\, x_1 \mapsto u_1\,,\ \mathtt{Inr}\, x_2 \mapsto u_2\} \ \mid \ \mathtt{case}_{m}\ t\ \mathtt{of}\ (x_1\,,\, x_2) \mapsto u \ \mid \ \mathtt{case}_{m}\ t\ \mathtt{of}\ \mathtt{Mod}_{n}\ x \mapsto u \\
\mid \ & \mathtt{upd}_{\ltimes}\ t\ \mathtt{with}\ x \mapsto t' \ \mid \ \mathtt{to}_{\ltimes}\ t \ \mid \ \mathtt{from}_{\ltimes}\ t \ \mid \ \mathtt{new}_{\ltimes} \\
\mid \ & t \triangleleft () \ \mid \ t \triangleleft \mathtt{Inl} \ \mid \ t \triangleleft \mathtt{Inr} \ \mid \ t \triangleleft (,) \ \mid \ t \triangleleft \mathtt{Mod}_{m} \ \mid \ t \triangleleft (\boldsymbol{\lambda} x \ {}_{m}\!\!\mapsto u) \ \mid \ t \triangleleft\!\circ\ t' \ \mid \ t \blacktriangleleft t'
\end{aligned}
$$

Syntactic sugar for terms:

$$
\begin{aligned}
\mathtt{Inl}\ t &\triangleq \mathtt{from}'_{\ltimes}\,(\mathtt{upd}_{\ltimes}\ \mathtt{new}_{\ltimes}\ \mathtt{with}\ d \mapsto \\
&\qquad d \triangleleft \mathtt{Inl} \blacktriangleleft t) \\
\mathtt{Inr}\ t &\triangleq \mathtt{from}'_{\ltimes}\,(\mathtt{upd}_{\ltimes}\ \mathtt{new}_{\ltimes}\ \mathtt{with}\ d \mapsto \\
&\qquad d \triangleleft \mathtt{Inr} \blacktriangleleft t) \\
\mathtt{Mod}_{m}\ t &\triangleq \mathtt{from}'_{\ltimes}\,(\mathtt{upd}_{\ltimes}\ \mathtt{new}_{\ltimes}\ \mathtt{with}\ d \mapsto \\
&\qquad d \triangleleft \mathtt{Mod}_{m} \blacktriangleleft t) \\
\boldsymbol{\lambda} x \ {}_{m}\!\!\mapsto u &\triangleq \mathtt{from}'_{\ltimes}\,(\mathtt{upd}_{\ltimes}\ \mathtt{new}_{\ltimes}\ \mathtt{with}\ d \mapsto \\
&\qquad d \triangleleft (\boldsymbol{\lambda} x \ {}_{m}\!\!\mapsto u))
\end{aligned}
$$

$$
\begin{aligned}
&\mathtt{from}'_{\ltimes}\ t \triangleq \\
&\quad \mathtt{case}\,(\mathtt{from}_{\ltimes}\,(\mathtt{upd}_{\ltimes}\ t\ \mathtt{with}\ un \mapsto un \ ⨟\ \mathtt{Mod}_{1\infty}\ ()))\ \mathtt{of} \\
&\qquad (st\,,\, ex) \mapsto \mathtt{case}\ ex\ \mathtt{of} \\
&\qquad\quad \mathtt{Mod}_{1\infty}\ un \mapsto un \ ⨟\ st \\
&() \triangleq \mathtt{from}'_{\ltimes}\,(\mathtt{upd}_{\ltimes}\ \mathtt{new}_{\ltimes}\ \mathtt{with}\ d \mapsto d \triangleleft ()) \\
&(t_1\,,\, t_2) \triangleq \mathtt{from}'_{\ltimes}\,(\mathtt{upd}_{\ltimes}\ \mathtt{new}_{\ltimes}\ \mathtt{with}\ d \mapsto \mathtt{case}\,(d \triangleleft (,))\ \mathtt{of} \\
&\qquad\qquad (d_1\,,\, d_2) \mapsto d_1 \blacktriangleleft t_1 \ ⨟\ d_2 \blacktriangleleft t_2)
\end{aligned}
$$

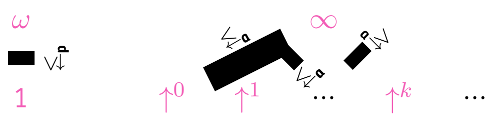
Grammar of types, modes and contexts:

$$
\begin{aligned}
\mathsf{T}, \mathsf{U}, \mathsf{S} \ ::= \ & \lfloor_{n} \mathsf{T} \rfloor \qquad \textit{(destination)} \\
\mid \ & \mathsf{S} \ltimes \mathsf{T} \qquad \textit{(ampar)} \\
\mid \ & \mathsf{1} \ \mid \ \mathsf{T}_1 \oplus \mathsf{T}_2 \ \mid \ \mathsf{T}_1 \otimes \mathsf{T}_2 \ \mid \ !_{m}\mathsf{T} \ \mid \ \mathsf{T}\ {}_{m}\!\!\multimap \mathsf{U} \\
\Gamma \ ::= \ & \bullet \ \mid \ x :_{m} \mathsf{T} \ \mid \ \Gamma_1,\ \Gamma_2
\end{aligned}
$$

$$
\begin{aligned}
m, n \ &::= \ pa \qquad \textit{(pair of multiplicity and age)} \\
p \ &::= \ 1 \ \mid \ \omega \\
a \ &::= \ \uparrow^{k} \ \mid \ \infty \\
\nu \triangleq \uparrow^{0} &\qquad \uparrow \triangleq \uparrow^{1}
\end{aligned}
$$

Ordering on modes:

$$pa \lesssim p'a' \iff p \lesssim_{p} p' \ \land \ a \lesssim_{a} a'$$



Operations on modes:

$+$	1	ω
1	ω	ω
ω	ω	ω

$\cdot$	1	ω
1	1	ω
ω	ω	ω

$$(pa)\cdot(p'a') \triangleq (p\cdot p')(a\cdot a')$$

$+$	$\uparrow^{k}$	∞
$\uparrow^{j}$	if $k = j$ then $\uparrow^{k}$ else ∞	∞
∞	∞	∞

$\cdot$	$\uparrow^{k}$	∞
$\uparrow^{j}$	$\uparrow^{k+j}$	∞
∞	∞	∞

$$(pa) + (p'a') \triangleq (p + p')(a + a')$$

Operations on typing contexts:

$$
\begin{aligned}
n \cdot \bullet &\triangleq \bullet \\
n \cdot (x :_{m} \mathsf{T},\ \Gamma) &\triangleq (x :_{n\cdot m} \mathsf{T}),\ n \cdot \Gamma
\end{aligned}
$$

$$
\begin{aligned}
\bullet + \Gamma &\triangleq \Gamma \\
(x :_{m} \mathsf{T},\ \Gamma_1) + \Gamma_2 &\triangleq x :_{m} \mathsf{T},\ (\Gamma_1 + \Gamma_2) \qquad \text{if } x \notin \Gamma_2 \\
(x :_{m} \mathsf{T},\ \Gamma_1) + (x :_{m'} \mathsf{T},\ \Gamma_2) &\triangleq x :_{m+m'} \mathsf{T},\ (\Gamma_1 + \Gamma_2)
\end{aligned}
$$

Figure 2.4: Terms, types and modes of λ_d

2.5.1 Modes and the age semiring

Our mode ringoid (more precisely, a commutative additive semigroup $+$ and a multiplicative monoid $(\cdot, 1)$, with the usual distributivity law $\mathtt{n}\cdot(\mathtt{m}_1 + \mathtt{m}_2) = \mathtt{n}\cdot\mathtt{m}_1 + \mathtt{n}\cdot\mathtt{m}_2$, and equipped with a partial order $\lesssim$, such that $+$ and $\cdot$ are order-preserving) is, as promised, the product of a multiplicity ringoid, to track linearity, and an age ringoid, to prevent scope escape. The resulting compound structure is also a ringoid.

The multiplicity ringoid has elements 1 (linear) and ω (unrestricted), it's the same ringoid as in [Atk18] or [Ber+18], and the same as described in detail in Chapter 1. The full description of the multiplicity ringoid is given again in Figure 2.4.

Ages are more interesting. We write ages as $\uparrow^k$ (with k a natural number), for "defined k scopes ago". We also have an age ∞ for variables that don't originate from a $\mathtt{upd}_\ltimes\ t\ \mathtt{with}\ x \mapsto t'$ construct i.e. that aren't destinations, and can be freely used in and returned by any scope. The main role of age ∞ is thus to act as a guarantee that a value doesn't contain destinations. Finally, we will write $\nu \triangleq \uparrow^0$ ("now") for the age of destinations that originate from the current scope; and $\uparrow \triangleq \uparrow^1$.

The operations or order on ages aren't the usual ones on natural numbers though. It is crucial that λ_d tracks the precise age of variables. Variables from two scopes ago cannot be used as if they were from one scope ago, or vice-versa. The ordering reflects this with finite ages being arranged in a flat order, with ∞ being bigger than all of them. Multiplication of ages reflects nesting of scope, as such, (finite) ages are multiplied by adding their numerical exponents $\uparrow^k\cdot\uparrow^j = \uparrow^{k+j}$. In the typing rules, the most common form of scope nesting is opening a scope, which is represented by multiplying the age of existing bindings by $\uparrow$ (that is, adding 1 to the ages seen as a natural numbers). When a same variable is shared between two subterms, $+$ takes the least upper bound for the age order above, in other terms, the variable must be at the same age in both subterms, or have age ∞ otherwise, which let it assume whichever age it needs in each subterm.

The unit of the new mode ringoid is the pair of the units from the multiplicity and age ringoids, ie. 1ν. We will usually omit the mode annotations when the mode is that unit.

Finally, as in λ_{Lm} and λ_{Ldm} from Chapter 1, operations and modes are lifted to typing contexts, still following the insights from [GS14] (see Figure 2.4).

2.5.2 Typing rules

Typing rules of λ_d are given in Figure 2.5. Rules for elimination of $\multimap$, 1, $\oplus$, $\otimes$ and $!_\mathtt{m}$ are the same as in λ_{Ldm} of Chapter 1, so we won't cover them again here.

One notable difference with λ_{Ldm} though, is, as announced, that introduction rules for data structures, aka. data constructors for types 1, $\oplus$, $\otimes$ and $!_\mathtt{m}$, are derived from elimination rules of destinations, and thus are not part of the core language. They are presented on the distinct bottom part of Figure 2.5. The introduction form for functions λ_d–$\text{TY}_\text{S}/\multimap\text{I}$ is also derived from elimination form λ_d–$\text{TY}/\lfloor\multimap\rfloor\text{E}$ for the corresponding destinations of function, although functions are not purely "data".

Rules λ_d–TY/IDV, λ_d–$\text{TY}/\ltimes\text{NEW}$ and λ_d–$\text{TY}_\text{S}/1\text{I}$, that are the potential leaves of the typing tree, is where weakening for unrestricted variables is allowed to happen. That's why they allow to discard any typing context of the form $\omega\nu\cdot\Gamma$. It's very similar to what happened in λ_{Ldm}–TY/ID or λ_{Ldm}–$\text{TY}/1\text{I}$, except that Γ is scaled by $\omega\nu$ instead of ω now.

$\boxed{\Gamma \vdash t : \mathsf{T}}$ *(Typing judgment for terms)*

$$\lambda_d\text{–TY/IDV}\quad \frac{1\nu \lesssim \mathsf{m}}{\omega\nu \cdot \Gamma,\ x :_{\mathsf{m}} \mathsf{T} \vdash x : \mathsf{T}}$$

$$\lambda_d\text{–TY/}\multimap\text{E}\quad \frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}\\ \Gamma_2 \vdash t' : \mathsf{T}\ {}_{\mathsf{m}}\!\!\multimap \mathsf{U}\end{array}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash t'\ t : \mathsf{U}}$$

$$\lambda_d\text{–TY/1E}\quad \frac{\Gamma_1 \vdash t : \mathsf{1} \qquad \Gamma_2 \vdash u : \mathsf{U}}{\Gamma_1 + \Gamma_2 \vdash t \mathbin{;}\ u : \mathsf{U}}$$

$$\lambda_d\text{–TY/}\oplus\text{E}\quad \frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \oplus \mathsf{T}_2\\ \Gamma_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1 \vdash u_1 : \mathsf{U}\\ \Gamma_2,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u_2 : \mathsf{U}\end{array}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{case}_{\mathsf{m}}\ t\ \mathbf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1\,,\ \mathsf{Inr}\, x_2 \mapsto u_2\} : \mathsf{U}}$$

$$\lambda_d\text{–TY/}\otimes\text{E}\quad \frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2\\ \Gamma_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u : \mathsf{U}\end{array}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{case}_{\mathsf{m}}\ t\ \mathbf{of}\ (x_1\,,\ x_2) \mapsto u : \mathsf{U}}$$

$$\lambda_d\text{–TY/!E}\quad \frac{\begin{array}{c}\Gamma_1 \vdash t : !_{\mathsf{n}} \mathsf{T}\\ \Gamma_2,\ x :_{\mathsf{m}\cdot\mathsf{n}} \mathsf{T} \vdash u : \mathsf{U}\end{array}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{case}_{\mathsf{m}}\ t\ \mathbf{of}\ \mathsf{Mod}_{\mathsf{n}}\ x \mapsto u : \mathsf{U}}$$

$$\lambda_d\text{–TY/}\ltimes\text{UPD}\quad \frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{U} \ltimes \mathsf{T}\\ 1\!\uparrow \cdot \Gamma_2,\ x :_{1\nu} \mathsf{T} \vdash t' : \mathsf{T}'\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{upd}_{\ltimes}\ t\ \mathbf{with}\ x \mapsto t' : \mathsf{U} \ltimes \mathsf{T}'}$$

$$\lambda_d\text{–TY/}\ltimes\text{TO}\quad \frac{\Gamma \vdash u : \mathsf{U}}{\Gamma \vdash \mathbf{to}_{\ltimes}\ u : \mathsf{U} \ltimes \mathsf{1}}$$

$$\lambda_d\text{–TY/}\ltimes\text{FROM}\quad \frac{\Gamma \vdash t : \mathsf{U} \ltimes (!_{1\infty} \mathsf{T})}{\Gamma \vdash \mathbf{from}_{\ltimes}\ t : \mathsf{U} \otimes (!_{1\infty} \mathsf{T})}$$

$$\lambda_d\text{–TY/}\ltimes\text{NEW}\quad \frac{}{\omega\nu \cdot \Gamma \vdash \mathbf{new}_{\ltimes} : \mathsf{T} \ltimes \lfloor \mathsf{T} \rfloor}$$

$$\lambda_d\text{–TY/}\lfloor\mathsf{1}\rfloor\text{E}\quad \frac{\Gamma \vdash t : \lfloor_{\mathsf{n}} \mathsf{1} \rfloor}{\Gamma \vdash t \triangleleft () : \mathsf{1}}$$

$$\lambda_d\text{–TY/}\lfloor\oplus\rfloor\text{E}_1\quad \frac{\Gamma \vdash t : \lfloor_{\mathsf{n}} \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor}{\Gamma \vdash t \triangleleft \mathsf{Inl} : \lfloor_{\mathsf{n}} \mathsf{T}_1 \rfloor}$$

$$\lambda_d\text{–TY/}\lfloor\oplus\rfloor\text{E}_2\quad \frac{\Gamma \vdash t : \lfloor_{\mathsf{n}} \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor}{\Gamma \vdash t \triangleleft \mathsf{Inr} : \lfloor_{\mathsf{n}} \mathsf{T}_2 \rfloor}$$

$$\lambda_d\text{–TY/}\lfloor\otimes\rfloor\text{E}\quad \frac{\Gamma \vdash t : \lfloor_{\mathsf{n}} \mathsf{T}_1 \otimes \mathsf{T}_2 \rfloor}{\Gamma \vdash t \triangleleft (,) : \lfloor_{\mathsf{n}} \mathsf{T}_1 \rfloor \otimes \lfloor_{\mathsf{n}} \mathsf{T}_2 \rfloor}$$

$$\lambda_d\text{–TY/}\lfloor!\rfloor\text{E}\quad \frac{\Gamma \vdash t : \lfloor_{\mathsf{n}} !_{\mathsf{n}'} \mathsf{T} \rfloor}{\Gamma \vdash t \triangleleft \mathsf{Mod}_{\mathsf{n}'} : \lfloor_{\mathsf{n}'\cdot\mathsf{n}} \mathsf{T} \rfloor}$$

$$\lambda_d\text{–TY/}\lfloor\multimap\rfloor\text{E}\quad \frac{\begin{array}{c}\Gamma_1 \vdash t : \lfloor_{\mathsf{n}} \mathsf{T}\ {}_{\mathsf{m}}\!\!\multimap \mathsf{U} \rfloor\\ \Gamma_2,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}\end{array}}{\Gamma_1 + (1\!\uparrow\!\cdot\mathsf{n}) \cdot \Gamma_2 \vdash t \triangleleft (\boldsymbol{\lambda} x\ {}_{\mathsf{m}}\!\!\mapsto u) : \mathsf{1}}$$

$$\lambda_d\text{–TY/}\lfloor\text{C}\rfloor\text{E}\quad \frac{\begin{array}{c}\Gamma_1 \vdash t : \lfloor \mathsf{U} \rfloor\\ \Gamma_2 \vdash t' : \mathsf{U} \ltimes \mathsf{T}\end{array}}{\Gamma_1 + 1\!\uparrow \cdot \Gamma_2 \vdash t \mathbin{\triangleleft\!\circ} t' : \mathsf{T}}$$

$$\lambda_d\text{–TY/}\lfloor\text{L}\rfloor\text{E}\quad \frac{\Gamma_1 \vdash t : \lfloor_{\mathsf{n}} \mathsf{T} \rfloor \qquad \Gamma_2 \vdash t' : \mathsf{T}}{\Gamma_1 + (1\!\uparrow\!\cdot\mathsf{n}) \cdot \Gamma_2 \vdash t \blacktriangleleft t' : \mathsf{1}}$$

$\boxed{\Gamma \vdash t : \mathsf{T}}$ *(Derived typing judgment for syntactic sugar forms)*

$$\lambda_d\text{–TY}_{\text{S}}\text{/}\ltimes\text{FROM'}\quad \frac{\Gamma \vdash t : \mathsf{T} \ltimes \mathsf{1}}{\Gamma \vdash \mathbf{from}'_{\ltimes}\ t : \mathsf{T}}$$

$$\lambda_d\text{–TY}_{\text{S}}\text{/1I}\quad \frac{}{\omega\nu \cdot \Gamma \vdash () : \mathsf{1}}$$

$$\lambda_d\text{–TY}_{\text{S}}\text{/}\multimap\text{I}\quad \frac{\Gamma_2,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}}{\Gamma_2 \vdash \boldsymbol{\lambda} x\ {}_{\mathsf{m}}\!\!\mapsto u : \mathsf{T}\ {}_{\mathsf{m}}\!\!\multimap \mathsf{U}}$$

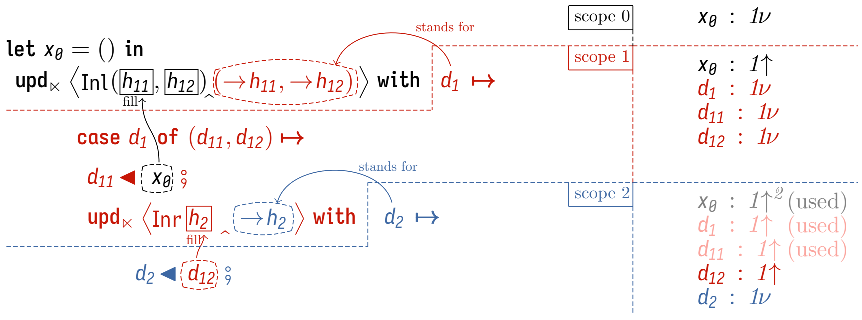
$$\lambda_d\text{–TY}_{\text{S}}\text{/}\oplus\text{I}_1\quad \frac{\Gamma_2 \vdash t : \mathsf{T}_1}{\Gamma_2 \vdash \mathsf{Inl}\, t : \mathsf{T}_1 \oplus \mathsf{T}_2}$$

$$\lambda_d\text{–TY}_{\text{S}}\text{/}\oplus\text{I}_2\quad \frac{\Gamma_2 \vdash t : \mathsf{T}_2}{\Gamma_2 \vdash \mathsf{Inr}\, t : \mathsf{T}_1 \oplus \mathsf{T}_2}$$

$$\lambda_d\text{–TY}_{\text{S}}\text{/!I}\quad \frac{\Gamma_2 \vdash t : \mathsf{T}}{\mathsf{m} \cdot \Gamma_2 \vdash \mathsf{Mod}_{\mathsf{m}}\ t : !_{\mathsf{m}} \mathsf{T}}$$

$$\lambda_d\text{–TY}_{\text{S}}\text{/}\otimes\text{I}\quad \frac{\Gamma_{21} \vdash t_1 : \mathsf{T}_1 \qquad \Gamma_{22} \vdash t_2 : \mathsf{T}_2}{\Gamma_{21} + \Gamma_{22} \vdash (t_1\,,\ t_2) : \mathsf{T}_1 \otimes \mathsf{T}_2}$$

Figure 2.5: Typing rules of λ_d



let $x_0 = ()$ in
$\mathbf{upd}_\ltimes \langle \mathrm{Inl}(\boxed{h_{11}}, \boxed{h_{12}}) \,{}_\wedge\, (\rightarrow h_{11}, \rightarrow h_{12}) \rangle$ with $d_1 \mapsto$
fill
stands for
case d_1 of $(d_{11}, d_{12}) \mapsto$
$d_{11} \blacktriangleleft x_0$;
$\mathbf{upd}_\ltimes \langle \mathrm{Inr}\,\boxed{h_2} \,{}_\wedge\, \rightarrow h_2 \rangle$ with $d_2 \mapsto$
fill
stands for
$d_2 \blacktriangleleft d_{12}$;
scope 0
scope 1
scope 2
$x_0 : 1\nu$
$x_0 : 1\uparrow$
$d_1 : 1\nu$
$d_{11} : 1\nu$
$d_{12} : 1\nu$
$x_0 : 1\uparrow^2$ (used)
$d_1 : 1\uparrow$ (used)
$d_{11} : 1\uparrow$ (used)
$d_{12} : 1\uparrow$
$d_2 : 1\nu$

Figure 2.6: Scope rules for $\mathbf{upd}_\ltimes$ in λ_d

Rule λ_d–TY/IDV, in addition to weakening, allows for coercion of the mode $\mathtt{m}$ of the variable used, with ordering constraint $1\nu \lessapprox \mathtt{m}$ as defined in Figure 2.4. Unlike in λ_{Ldm}, here not all modes $\mathtt{m}$ are compatible with 1ν. Notably, mode coercion still doesn't allow for a finite age to be changed to another, as $\uparrow^j$ and $\uparrow^k$ are not comparable w.r.t. $\lessapprox_{\mathtt{a}}$ when $j \neq k$. So for example we cannot use a variable with mode $1\uparrow$ in most contexts.

For pattern-matching, with rules λ_d–TY/$\oplus$E, λ_d–TY/$\otimes$E and λ_d–TY/!E, we keep the same *deep mode* approach as in λ_{Ldm}: **case** expressions are parametrized by a mode $\mathtt{m}$ by which the typing context Γ_1 of the scrutinee is multiplied. The variables which bind the subcomponents of the scrutinee then inherit this mode.

Let's focus now on the real novelties of the typing rules of λ_d.

Rules for scoping As destinations always exist in the context of a structure with holes, and must stay in that context, we need a formal notion of *scope*. Scopes are created by λ_d–TY/$\ltimes$UPD, as destinations are only ever accessed through $\mathbf{upd}_\ltimes$. More precisely, $\mathbf{upd}_\ltimes\ t$ **with** $x \mapsto t'$ creates a new scope which spans over t'. In that scope, x has age ν (now), and the ages of the existing bindings in Γ_2 are multiplied by $\uparrow$ (i.e. we add 1 to ages seen as a numbers). That is represented by t' typing in $1\uparrow \cdot \Gamma_2,\ x :_{1\nu} \mathsf{T}$ while the parent term $\mathbf{upd}_\ltimes\ t$ **with** $x \mapsto t'$ types in unscaled contexts $\Gamma_1 + \Gamma_2$. This difference of age between x — introduced by $\mathbf{upd}_\ltimes$, containing destinations — and Γ_2 lets us see what originates from older scopes. Specifically, distinguishing the age of destinations is crucial when typing filling primitives to avoid the pitfalls of Section 2.3. Figure 2.6 illustrates scopes introduced by $\mathbf{upd}_\ltimes$, and how the typing rules for $\mathbf{upd}_\ltimes$ and $\blacktriangleleft$ interact.

Anticipating Section 2.6.1, ampar values are pairs with a structure with holes on the left, and destinations on the right. With $\mathbf{upd}_\ltimes$ we enter a new scope where the destinations are accessible, but the structure with holes remains in the outer scope. As a result, when filling a destination with λ_d–TY/$\lfloor$L$\rfloor$E, for instance $d_{11} \blacktriangleleft x_0$ in Figure 2.6, we type d_{11} in the new scope, while we type x_0 in the outer scope, as it's being moved to the structure with holes on the left of the ampar, which lives in the outer scope too. This is the opposite of the scaling that $\mathbf{upd}_\ltimes$ does: while $\mathbf{upd}_\ltimes$ creates a new scope for its body, operator $\blacktriangleleft$, and similarly, $\triangleleft\!\circ$ and $\triangleleft(\boldsymbol{\lambda} x\ {}_{\mathtt{m}}\!\mapsto u)$, transfer their right operand to the outer scope. In other words, the right-hand side of $\blacktriangleleft$ or $\triangleleft$ is an enclave for the parent scope.

When using $\mathtt{from}'_{\ltimes}$ (rule λ_d–TY$_S$/$\ltimes$FROM'), the left of an ampar is extracted to the current scope (as seen at the bottom of Figure 2.6 with x_{22}): this is the fundamental reason why the left of an ampar has to "take place" in the current scope. We know the structure is complete and can be extracted because the right-hand side is of type unit (1), and thus no destination on the right-hand side means no hole can remain on the left. $\mathtt{from}'_{\ltimes}$ is implemented in terms of $\mathtt{from}_{\ltimes}$ in Figure 2.4 to keep the core calculus tidier (and limit the number of typing rules, evaluation contexts, etc), but it can be implemented much more efficiently in a real-world implementation.

When an ampar is eliminated with the more general $\mathtt{from}_{\ltimes}$ in rule λ_d–TY/$\ltimes$FROM however, we extract both sides of the ampar to the current scope, even though the right-hand side is normally in a different scope. This is only safe to do because the right-hand side is required to have type $!_{1\infty}\mathtt{T}$, which means it is scope-insensitive: it can't contain any scope-controlled resource. This also ensures that the right-hand side cannot contain destinations, so the structure has to be complete and thus ready to be read.

In λ_d–TY/$\ltimes$TO, on the other hand, there is no need to bother with scopes: the operator $\mathtt{to}_{\ltimes}$ embeds an already completed structure in an ampar whose left side is the structure (that continues to type in the current scope), and right-hand side is unit.

The remaining operators $\triangleleft()$, $\triangleleft\mathtt{Inl}$, $\triangleleft\mathtt{Inr}$, $\triangleleft\,\mathtt{Mod}_{\mathtt{m}}$, $\triangleleft(,)$ from rules of the form λ_d–TY/$\lfloor\ \rfloor$E are the other destination-filling primitives. They write a hollow constructor to the hole pointed by the destination operand, and return the potential new destinations that are created for new holes in the hollow constructor (or unit if there is none).

2.6 Operational semantics

We give the operational semantics of λ_d in the *reduction semantics* style that we used in Chapter 1, that's to say with explicit syntactic manipulation of evaluation contexts. However in this section we'll be more thorough. In particular, we'll give grammar of evaluation contexts properly, and we'll also define typing rules for evaluation contexts, so as to prove that typing is preserved throughout the reduction.

For that, we'll need a few ingredients. Commands $E[t]$ are used to represent a running program; they're described in Section 2.6.2. We'll also need a class of runtime values (we'll often just say *values*), as λ_d currently lacks any way to represent destinations or holes, or really any kind of value (for instance $\mathtt{Inl}\,()$ has been, so far, just syntactic sugar for a term $\mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes}\ \mathtt{new}_{\ltimes}\ \mathtt{with}\ d\ \mapsto \ldots)$). It's a peculiarity of λ_d that values (in particular, data constructors) only exist during the reduction; usually they are part of the term syntax of functional languages as in λ_{Ldm}. Our runtime values are described in described in Section 2.6.1. As announced, we'll also extend the type system to cover evaluation contexts, values and commands, so as to be able to state and prove type safety theorems.

2.6.1 Runtime values and new typing context forms

The syntax of runtime values is given in Figure 2.7. It features constructors for all of our basic types, as well as functions (note that in $\lambda_{\mathtt{v}} x\ {}_{\mathtt{m}}\!\mapsto u$, u is a term, not a value). The more interesting values are holes $\boxed{h}$, destinations $\rightarrow h$, and ampars ${}_{H}\langle v_2 \,{}_{\wedge}\, v_1\rangle$, which we'll describe in the rest of the section. In order for the operational semantics to use substitution, which requires substituting variables with values, we also extend the syntax of terms to include values through rule λ_d–TY/FROMVAL.

Grammar extended with values:

$$t, u \;::=\; \dots \;\mid\; v$$

$$\begin{array}{rll}
v \;::= & \boxed{h} & \textit{(hole)} \\
\mid & \to h & \textit{(destination)} \\
\mid & {}_{H}\langle v_2 \,^\smallfrown\, v_1 \rangle & \textit{(ampar value)} \\
\mid & () \;\mid\; \lambda_{v} x \,{}_{m}\!\mapsto u \;\mid\; \mathsf{Inl}\, v & \\
\mid & \mathsf{Inr}\, v \;\mid\; \mathsf{Mod}_{m}\, v \;\mid\; (v_1 \,,\, v_2) &
\end{array}$$

Typing values as terms:

λ_d–TY/FROMVAL
$$\frac{\Delta \vdash_{v} v : T}{\omega\nu \cdot \Gamma,\; \Delta \vdash v : T}$$

Extended grammar of typing contexts:

$$\begin{array}{lllllll}
\Delta & ::= & \bullet & \mid\; \to h :_{m} \lfloor{}_{n} T \rfloor & & \mid & \Delta_1,\; \Delta_2 \\
\Gamma & ::= & \bullet & \mid\; \to h :_{m} \lfloor{}_{n} T \rfloor & \mid\; x :_{m} T & \mid & \Gamma_1,\; \Gamma_2 \\
\Theta & ::= & \bullet & \mid\; \to h :_{m} \lfloor{}_{n} T \rfloor & \mid\; \boxed{h} :_{n} T & \mid & \Theta_1,\; \Theta_2
\end{array}$$

Operations extended to new typing context forms:

$$\begin{array}{lcll}
n' \cdot (\boxed{h} :_{n} T,\; \Theta) & \triangleq & (\boxed{h} :_{n' \cdot n} T),\; n' \cdot \Theta & \\
n' \cdot (\to h :_{m} \lfloor{}_{n} T \rfloor,\; \Gamma) & \triangleq & (\to h :_{n' \cdot m} \lfloor{}_{n} T \rfloor),\; n' \cdot \Gamma & \dagger
\end{array}$$

$$\begin{array}{lcll}
(\boxed{h} :_{n} T,\; \Theta_1) + \Theta_2 & \triangleq & \boxed{h} :_{n} T,\; (\Theta_1 + \Theta_2) & \text{if } h \notin \Theta_2 \\
(\boxed{h} :_{n} T,\; \Theta_1) + (\boxed{h} :_{n'} T,\; \Theta_2) & \triangleq & \boxed{h} :_{n+n'} T,\; (\Theta_1 + \Theta_2) & \\
(\to h :_{m} \lfloor{}_{n} T \rfloor,\; \Gamma_1) + \Gamma_2 & \triangleq & \to h :_{m} \lfloor{}_{n} T \rfloor,\; (\Gamma_1 + \Gamma_2) & \text{if } h \notin \Gamma_2 \quad \dagger \\
(\to h :_{m} \lfloor{}_{n} T \rfloor,\; \Gamma_1) + (\to h :_{m'} \lfloor{}_{n} T \rfloor,\; \Gamma_2) & \triangleq & \to h :_{m+m'} \lfloor{}_{n} T \rfloor,\; (\Gamma_1 + \Gamma_2) & \dagger
\end{array}$$

$$\begin{array}{rcl}
\to^{-1}(\bullet) & \triangleq & \bullet \\
\to^{-1}(\to h :_{1\nu} \lfloor{}_{n} T \rfloor,\; \Delta) & \triangleq & (\boxed{h} :_{n} T),\; \to^{-1}(\Delta)
\end{array}$$

$\dagger$: *same rule is also true for* Θ *or* Δ *replacing* Γ

$\boxed{\Theta \vdash_{v} v : T}$ *(Typing judgment for values)*

λ_d–TY$_V$/IDH
$$\frac{}{\boxed{h} :_{1\nu} T \vdash_{v} \boxed{h} : T}$$

λ_d–TY$_V$/IDD
$$\frac{1\nu \lessapprox m}{\to h :_{m} \lfloor{}_{n} T \rfloor \vdash_{v} \to h : \lfloor{}_{n} T \rfloor}$$

λ_d–TY$_V$/1I
$$\frac{}{\bullet \vdash_{v} () : 1}$$

λ_d–TY$_V$/$\multimap$I
$$\frac{\Delta,\; x :_{m} T \vdash u : U}{\Delta \vdash_{v} \lambda_{v} x \,{}_{m}\!\mapsto u : T \,{}_{m}\!\multimap U}$$

λ_d–TY$_V$/$\oplus$I$_1$
$$\frac{\Theta \vdash_{v} v_1 : T_1}{\Theta \vdash_{v} \mathsf{Inl}\, v_1 : T_1 \oplus T_2}$$

λ_d–TY$_V$/$\oplus$I$_2$
$$\frac{\Theta \vdash_{v} v_2 : T_2}{\Theta \vdash_{v} \mathsf{Inr}\, v_2 : T_1 \oplus T_2}$$

λ_d–TY$_V$/$\otimes$I
$$\frac{\Theta_1 \vdash_{v} v_1 : T_1 \qquad \Theta_2 \vdash_{v} v_2 : T_2}{\Theta_1 + \Theta_2 \vdash_{v} (v_1 \,,\, v_2) : T_1 \otimes T_2}$$

λ_d–TY$_V$/!I
$$\frac{\Theta \vdash_{v} v' : T}{n \cdot \Theta \vdash_{v} \mathsf{Mod}_{n}\, v' : !_{n} T}$$

λ_d–TY$_V$/$\ltimes$I
$$\frac{1\uparrow \cdot \Delta_1,\; \Delta_3 \vdash_{v} v_1 : T \qquad \Delta_2,\; \to^{-1}\Delta_3 \vdash_{v} v_2 : U}{\Delta_1,\; \Delta_2 \vdash_{v} {}_{\mathsf{hnames}(\Delta_3)}\langle v_2 \,^\smallfrown\, v_1 \rangle : U \ltimes T}$$

Figure 2.7: Runtime values and new typing context forms

Destinations and holes are two faces of the same coin, as seen in Section 2.2.1, and we must ensure that throughout the reduction, a destination always points to a hole, and a hole is always the target of exactly one destination. Thus, the new idea of our system is to feature *hole bindings* $\boxed{h} :_{\mathsf{n}} \mathsf{T}$ and *destination bindings* $\rightarrow h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor$ in typing contexts in addition to the usual variable bindings $x :_{\mathsf{m}} \mathsf{T}$. In both cases, we call h a *hole name*. By definition, a context Θ can contain both destination bindings and hole bindings, but *not a destination binding and a hole binding for the same hole name.*

We extend our previous context operations $+$ and $\cdot$ to act on the new binding forms, as described in Figure 2.7. Context addition is still very partial; for instance, $(\boxed{h} :_{\mathsf{n}} \mathsf{T}) + (\rightarrow h :_{\mathsf{m}} \lfloor_{\mathsf{n}'} \mathsf{T} \rfloor)$ is not defined, as h is present on both sides but with different binding forms.

One of the main goals of λ_d is to ensure that a hole value is never read. The type system maintains this invariant by simply not allowing any hole bindings in the context when typing terms (see Figure 2.7 for the different type of contexts used in the typing judgment). In fact, the only place where holes are introduced, is the left-hand side v_2 in an ampar ${}_H\langle v_2 \,{}^{\wedge}\, v_1 \rangle$, in λ_d–TY$_{\mathrm{V}}$/$\ltimes$I.

Specifically, holes come from the operator $\rightarrow^{-1}$, which represents the matching hole bindings for a set of destination bindings. It's a partial, pointwise operation on typing contexts Δ, as defined in Figure 2.7. Note that $\rightarrow^{-1}\Delta$ is undefined if any destination binding in Δ has a mode other than 1ν.

Furthermore, in λ_d–TY$_{\mathrm{V}}$/$\ltimes$I, the holes $\rightarrow^{-1}\Delta_3$ and the corresponding destinations Δ_3 are bound together and consequently removed from the ampar's typing context: this is how we ensure that, indeed, there's one destination per hole and one hole per destination. That being said, both sides of the ampar may also contain stored destinations from other scopes, represented by $1\uparrow \cdot \Delta_1$ and Δ_2 in the respective typing contexts of v_1 and v_2.

Rule λ_d–TY$_{\mathrm{V}}$/IDH indicates that a hole must have mode 1ν in typing context to be well-typed; in particular mode coercion is not allowed here, and neither is weakening. Only when a hole is behind an exponential, that mode can change to some arbitrary mode n. The mode of a hole constrains which values can be written to it, e.g. in $\boxed{h} :_{\mathsf{n}} \mathsf{T} \vdash_{\mathsf{v}} \mathsf{Mod}_{\mathsf{n}} \boxed{h} : !_{\mathsf{n}} \mathsf{T}$, only a value with mode n (more precisely, a value typed in a context of the form $\mathsf{n} \cdot \Theta$) can be written to $\boxed{h}$.

Surprisingly, in λ_d–TY$_{\mathrm{V}}$/IDD, we see that a destination can be typed with any mode m coercible to 1ν. We did this to mimic the rule λ_d–TY/IDV and make the general modal substitution lemma expressible for λ_d[9]. We formally proved however that throughout a well-typed closed program, m will never be of multiplicity ω or age ∞ — a destination is always linear and of finite age — so mode coercion is never actually used; and we used this result during the formal proof of the substitution lemma to make it quite easier. The other mode n, appearing in λ_d–TY$_{\mathrm{V}}$/IDD, is not the mode of the destination binding; instead it is part of the type $\lfloor_{\mathsf{n}} \mathsf{T} \rfloor$ and corresponds to the mode of values that we can write to the corresponding $\boxed{h}$; so for it no coercion can take place.

Other salient points We don't distinguish values with holes from fully-defined values at the syntactic level: instead types prevent holes from being read. In particular, while values are typed in contexts Θ allowing both destination and hole bindings, when using a value as a term in λ_d–TY/FROMVAL, it's only allowed to have free destinations, but no free holes.

[9]Generally, in modal systems, if $x :_{\mathsf{m}} \mathsf{T},\ \Gamma \vdash u : \mathsf{U}$ and $\Delta \vdash v : \mathsf{T}$ then $\mathsf{m} \cdot \Delta,\ \Gamma \vdash u[x := v] : \mathsf{U}$ [AB20]. We have $x :_{\omega\infty} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor \vdash () : \mathbf{1}$ and $\rightarrow h :_{1\nu} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor \vdash \rightarrow h : \lfloor_{\mathsf{n}} \mathsf{T} \rfloor$ so $\omega\infty \cdot (\rightarrow h :_{1\nu} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor) \vdash ()[x := \rightarrow h] : \mathbf{1}$ should be valid.

Notice, also, that values can't have free variables, since contexts Θ only contain hole and destination bindings, no variable binding. That values are closed is a standard feature of denotational semantics or abstract machine semantics. This is true even for function values (λ_d–$\text{TY}_{\text{V}}/\multimap\text{I}$), which, also is prevented from containing free holes. Since a function's body is unevaluated, it's unclear what it'd mean for a function to contain holes; at the very least it'd complicate our system a lot, and we are unaware of any benefit supporting free holes in functions could bring.

One might wonder how we can represent a curried function $\boldsymbol{\lambda} x \mapsto \boldsymbol{\lambda} y \mapsto x\ \mathtt{concat}\ y$ at the value level, as the inner abstraction captures the free variable x. The answer is that such a function, at value level, is encoded as $\boldsymbol{\lambda}_{\mathsf{v}} x \mapsto \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes}\ \mathtt{new}_{\ltimes}\ \mathtt{with}\ d \mapsto d \triangleleft (\boldsymbol{\lambda} y \mapsto x\ \mathtt{concat}\ y))$, where the inner closure is not yet in value form. As the form $d \triangleleft (\boldsymbol{\lambda} y \mapsto x\ \mathtt{concat}\ y)$ is part of term syntax, it's allowed to have free variable x.

2.6.2 Evaluation contexts and commands

A running program is represented by a pair $E\big[t\big]$ of an evaluation context E, and an (extended) term t under focus. We call such a pair $E\big[t\big]$ a *command*, borrowing the terminology from Curien and Herbelin [CH00].

The grammar of evaluation contexts is given in Figure 2.8. As in λ_{Ldm}, an evaluation context E is the composition of an arbitrary number of focusing components e. It might seem surprising that we don't need a notion of store or state in our semantics to represent the mutation induced by filling destinations. In fact, destination filling only require a very tame notion of state — so tame that we can simply represent writing to a hole by a substitution *on the evaluation context*, instead of using more heavy store semantics.

TODO: Find bib refs (Feilleisen et al.) about linear subsitutions in evaluation contexts

It's important to remember that the command $E\big[t\big]$ is formally a pair, so it won't always have a corresponding term. The reason is that focusing components are all directly derived from the term syntax, except for the *open ampar* focusing component ${}^{\text{op}}_{H}\langle v_2 \,{}_{\wedge}\, [\,]\rangle$ that doesn't have a corresponding term construct. This focusing component indicates that an ampar is currently being processed by $\mathtt{upd}_{\ltimes}$, with its left-hand side v_2 (the structure being built) being attached to the open ampar focusing component, while its right-hand side (containing destinations) is either in subsequent focusing components, or in the term under focus. Ampars being open during the evaluation of $\mathtt{upd}_{\ltimes}$'s body and closed back afterwards is counterpart to the notion of scopes in typing rules.

Evaluation contexts are typed in a context Δ that can only contain destination bindings. As we will later see in rule λ_d–TY/CMD of Figure 2.9, Δ is exactly the typing context that the term t has to use to form a valid $E\big[t\big]$. In other words, while $\Gamma \vdash t : \mathtt{T}$ *requires* the bindings of Γ, judgment $\Delta \dashv E : \mathtt{T} \rightarrowtail \mathtt{U}_0$ *provides* the bindings of Δ. Typing rules for evaluation contexts are given in Figure 2.8.

An evaluation context has a context type $\mathtt{T} \rightarrowtail \mathtt{U}_0$. The meaning of $E : \mathtt{T} \rightarrowtail \mathtt{U}_0$ is that given $t : \mathtt{T}$, $E\big[t\big]$ returns a value of type $\mathtt{U}_0$. Composing an evaluation context $E : \mathtt{T} \rightarrowtail \mathtt{U}_0$ with a new focusing component never affects the type $\mathtt{U}_0$ of the future command; only the type $\mathtt{T}$ of the focus is altered.

All typing rules for evaluation contexts can be derived systematically from the ones for the corresponding term (except for the rule λ_d–$\text{TY}_{\text{E}}/\ltimes\text{OP}$ that is a truly new form). Let's take the rule λ_d–$\text{TY}_{\text{E}}/\otimes\text{E}$ as an example:

Grammar of evaluation contexts:

$$
\begin{array}{rcl}
e & ::= & t'\ [\,] \ \mid\ [\,]\ v \ \mid\ [\,] \fatsemi u \\
 & \mid & \mathtt{case}_{\mathtt{m}}\ [\,]\ \mathtt{of}\ \{\mathtt{Inl}\,x_1 \mapsto u_1\ ,\ \mathtt{Inr}\,x_2 \mapsto u_2\} \ \mid\ \mathtt{case}_{\mathtt{m}}\ [\,]\ \mathtt{of}\ (x_1\ ,\ x_2) \mapsto u \ \mid\ \mathtt{case}_{\mathtt{m}}\ [\,]\ \mathtt{of}\ \mathtt{Mod}_{\mathtt{n}}\ x \mapsto u \\
 & \mid & \mathtt{upd}_{\ltimes}\ [\,]\ \mathtt{with}\ x \mapsto t' \ \mid\ \mathtt{to}_{\ltimes}\ [\,] \ \mid\ \mathtt{from}_{\ltimes}\ [\,] \ \mid\ [\,] \triangleleft\!\circ\, t' \ \mid\ v \triangleleft\!\circ\, [\,] \ \mid\ [\,] \blacktriangleleft t' \ \mid\ v \blacktriangleleft [\,] \\
 & \mid & [\,] \triangleleft () \ \mid\ [\,] \triangleleft \mathtt{Inl} \ \mid\ [\,] \triangleleft \mathtt{Inr} \ \mid\ [\,] \triangleleft (,) \ \mid\ [\,] \triangleleft \mathtt{Mod}_{\mathtt{m}} \ \mid\ [\,] \triangleleft (\boldsymbol{\lambda} x\ {}_{\mathtt{m}}\!\mapsto u) \\
 & \mid & {}^{\mathrm{op}}_{H}\langle v_2 \,{}_{\frown}\, [\,] \rangle \qquad \textit{(open ampar focusing component)} \\
E & ::= & [\,] \ \mid\ E \circ e
\end{array}
$$

$\boxed{\Delta \dashv E : \mathtt{T} \rightarrowtail \mathtt{U}_0}$ *(Typing judgment for evaluation contexts)*

$$\lambda_d\text{–TY}_{\text{E}}/\text{ID}\qquad \frac{}{\cdot \dashv [\,] : \mathtt{U}_0 \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/{\multimap}\text{E}_1\qquad \frac{\mathtt{m} \cdot \Delta_1,\ \Delta_2 \dashv E : \mathtt{U} \rightarrowtail \mathtt{U}_0 \qquad \Delta_2 \vdash t' : \mathtt{T}\ {}_{\mathtt{m}}\!\multimap \mathtt{U}}{\Delta_1 \dashv E \circ t'\ [\,] : \mathtt{T} \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/{\multimap}\text{E}_2\qquad \frac{\mathtt{m} \cdot \Delta_1,\ \Delta_2 \dashv E : \mathtt{U} \rightarrowtail \mathtt{U}_0 \qquad \Delta_1 \vdash v : \mathtt{T}}{\Delta_2 \dashv E \circ [\,]\ v : (\mathtt{T}\ {}_{\mathtt{m}}\!\multimap \mathtt{U}) \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/\mathtt{1}\text{E}\qquad \frac{\Delta_1,\ \Delta_2 \dashv E : \mathtt{U} \rightarrowtail \mathtt{U}_0 \qquad \Delta_2 \vdash u : \mathtt{U}}{\Delta_1 \dashv E \circ [\,] \fatsemi u : \mathtt{1} \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/{\oplus}\text{E}\qquad \frac{\mathtt{m} \cdot \Delta_1,\ \Delta_2 \dashv E : \mathtt{U} \rightarrowtail \mathtt{U}_0 \qquad \Delta_2,\ x_1 :_{\mathtt{m}} \mathtt{T}_1 \vdash u_1 : \mathtt{U} \qquad \Delta_2,\ x_2 :_{\mathtt{m}} \mathtt{T}_2 \vdash u_2 : \mathtt{U}}{\Delta_1 \dashv E \circ \mathtt{case}_{\mathtt{m}}\ [\,]\ \mathtt{of}\ \{\mathtt{Inl}\,x_1 \mapsto u_1\ ,\ \mathtt{Inr}\,x_2 \mapsto u_2\} : (\mathtt{T}_1 \oplus \mathtt{T}_2) \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/{\otimes}\text{E}\qquad \frac{\mathtt{m} \cdot \Delta_1,\ \Delta_2 \dashv E : \mathtt{U} \rightarrowtail \mathtt{U}_0 \qquad \Delta_2,\ x_1 :_{\mathtt{m}} \mathtt{T}_1,\ x_2 :_{\mathtt{m}} \mathtt{T}_2 \vdash u : \mathtt{U}}{\Delta_1 \dashv E \circ \mathtt{case}_{\mathtt{m}}\ [\,]\ \mathtt{of}\ (x_1\ ,\ x_2) \mapsto u : (\mathtt{T}_1 \otimes \mathtt{T}_2) \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/!\text{E}\qquad \frac{\mathtt{m} \cdot \Delta_1,\ \Delta_2 \dashv E : \mathtt{U} \rightarrowtail \mathtt{U}_0 \qquad \Delta_2,\ x :_{\mathtt{m}\cdot\mathtt{m}'} \mathtt{T} \vdash u : \mathtt{U}}{\Delta_1 \dashv E \circ \mathtt{case}_{\mathtt{m}}\ [\,]\ \mathtt{of}\ \mathtt{Mod}_{\mathtt{m}'}\ x \mapsto u : !_{\mathtt{m}'}\mathtt{T} \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/{\ltimes}\text{UPD}\qquad \frac{\Delta_1,\ \Delta_2 \dashv E : \mathtt{U} \ltimes \mathtt{T}' \rightarrowtail \mathtt{U}_0 \qquad \mathtt{1}{\uparrow} \cdot \Delta_2,\ x :_{\mathtt{1}\nu} \mathtt{T} \vdash t' : \mathtt{T}'}{\Delta_1 \dashv E \circ \mathtt{upd}_{\ltimes}\ [\,]\ \mathtt{with}\ x \mapsto t' : (\mathtt{U} \ltimes \mathtt{T}) \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/{\ltimes}\text{TO}\qquad \frac{\Delta \dashv E : (\mathtt{U} \ltimes \mathtt{1}) \rightarrowtail \mathtt{U}_0}{\Delta \dashv E \circ \mathtt{to}_{\ltimes}\ [\,] : \mathtt{U} \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/{\ltimes}\text{FROM}\qquad \frac{\Delta \dashv E : (\mathtt{U} \otimes (!_{\mathtt{1}\infty}\mathtt{T})) \rightarrowtail \mathtt{U}_0}{\Delta \dashv E \circ \mathtt{from}_{\ltimes}\ [\,] : (\mathtt{U} \ltimes (!_{\mathtt{1}\infty}\mathtt{T})) \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/\lfloor\mathtt{1}\rfloor\text{E}\qquad \frac{\Delta \dashv E : \mathtt{1} \rightarrowtail \mathtt{U}_0}{\Delta \dashv E \circ [\,] \triangleleft () : \lfloor_{\mathtt{n}} \mathtt{1} \rfloor \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/\lfloor\oplus\rfloor\text{E}_1\qquad \frac{\Delta \dashv E : \lfloor_{\mathtt{n}} \mathtt{T}_1 \rfloor \rightarrowtail \mathtt{U}_0}{\Delta \dashv E \circ [\,] \triangleleft \mathtt{Inl} : \lfloor_{\mathtt{n}} \mathtt{T}_1 \oplus \mathtt{T}_2 \rfloor \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/\lfloor\oplus\rfloor\text{E}_2\qquad \frac{\Delta \dashv E : \lfloor_{\mathtt{n}} \mathtt{T}_2 \rfloor \rightarrowtail \mathtt{U}_0}{\Delta \dashv E \circ [\,] \triangleleft \mathtt{Inr} : \lfloor_{\mathtt{n}} \mathtt{T}_1 \oplus \mathtt{T}_2 \rfloor \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/\lfloor\otimes\rfloor\text{E}\qquad \frac{\Delta \dashv E : (\lfloor_{\mathtt{n}} \mathtt{T}_1 \rfloor \otimes \lfloor_{\mathtt{n}} \mathtt{T}_2 \rfloor) \rightarrowtail \mathtt{U}_0}{\Delta \dashv E \circ [\,] \triangleleft (,) : \lfloor_{\mathtt{n}} \mathtt{T}_1 \otimes \mathtt{T}_2 \rfloor \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/\lfloor!\rfloor\text{E}\qquad \frac{\Delta \dashv E : \lfloor_{\mathtt{m}\cdot\mathtt{n}} \mathtt{T} \rfloor \rightarrowtail \mathtt{U}_0}{\Delta \dashv E \circ [\,] \triangleleft \mathtt{Mod}_{\mathtt{m}} : \lfloor_{\mathtt{n}} !_{\mathtt{m}} \mathtt{T} \rfloor \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/\lfloor{\multimap}\rfloor\text{E}\qquad \frac{\Delta_1,\ (\mathtt{1}{\uparrow}\cdot\mathtt{n}) \cdot \Delta_2 \dashv E : \mathtt{1} \rightarrowtail \mathtt{U}_0 \qquad \Delta_2,\ x :_{\mathtt{m}} \mathtt{T} \vdash u : \mathtt{U}}{\Delta_1 \dashv E \circ [\,] \triangleleft (\boldsymbol{\lambda} x\ {}_{\mathtt{m}}\!\mapsto u) : \lfloor_{\mathtt{n}} \mathtt{T}\ {}_{\mathtt{m}}\!\multimap \mathtt{U} \rfloor \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/\lfloor\text{C}\rfloor\text{E}_1\qquad \frac{\Delta_1,\ \mathtt{1}{\uparrow} \cdot \Delta_2 \dashv E : \mathtt{T} \rightarrowtail \mathtt{U}_0 \qquad \Delta_2 \vdash t' : \mathtt{U} \ltimes \mathtt{T}}{\Delta_1 \dashv E \circ [\,] \triangleleft\!\circ\, t' : \lfloor \mathtt{U} \rfloor \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/\lfloor\text{C}\rfloor\text{E}_2\qquad \frac{\Delta_1,\ \mathtt{1}{\uparrow} \cdot \Delta_2 \dashv E : \mathtt{T} \rightarrowtail \mathtt{U}_0 \qquad \Delta_1 \vdash v : \lfloor \mathtt{U} \rfloor}{\Delta_2 \dashv E \circ v \triangleleft\!\circ\, [\,] : \mathtt{U} \ltimes \mathtt{T} \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/\lfloor\text{L}\rfloor\text{E}_1\qquad \frac{\Delta_1,\ (\mathtt{1}{\uparrow}\cdot\mathtt{n}) \cdot \Delta_2 \dashv E : \mathtt{1} \rightarrowtail \mathtt{U}_0 \qquad \Delta_2 \vdash t' : \mathtt{T}}{\Delta_1 \dashv E \circ [\,] \blacktriangleleft t' : \lfloor_{\mathtt{n}} \mathtt{T} \rfloor \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/\lfloor\text{L}\rfloor\text{E}_2\qquad \frac{\Delta_1,\ (\mathtt{1}{\uparrow}\cdot\mathtt{n}) \cdot \Delta_2 \dashv E : \mathtt{1} \rightarrowtail \mathtt{U}_0 \qquad \Delta_1 \vdash v : \lfloor_{\mathtt{n}} \mathtt{T} \rfloor}{\Delta_2 \dashv E \circ v \blacktriangleleft [\,] : \mathtt{T} \rightarrowtail \mathtt{U}_0}$$

$$\lambda_d\text{–TY}_{\text{E}}/{\ltimes}\text{OP}\qquad \frac{\mathtt{hnames}(E)\ \#\#\ \mathtt{hnames}(\Delta_3) \qquad \Delta_1,\ \Delta_2 \dashv E : (\mathtt{U} \ltimes \mathtt{T}') \rightarrowtail \mathtt{U}_0 \qquad \Delta_2,\ \rightarrow^{-1}\Delta_3 \vdash_{\mathsf{v}} v_2 : \mathtt{U}}{\mathtt{1}{\uparrow} \cdot \Delta_1,\ \Delta_3 \dashv E \circ {}^{\mathrm{op}}_{\mathtt{hnames}(\Delta_3)}\langle v_2 \,{}_{\frown}\, [\,] \rangle : \mathtt{T}' \rightarrowtail \mathtt{U}_0}$$

Figure 2.8: Evaluation contexts and their typing rules

λ_d–TY$_\text{E}/\otimes$E

$$\frac{\mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv E : \mathsf{U}\rightarrowtail\mathsf{U}_0 \qquad \Delta_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u : \mathsf{U}}{\Delta_1 \dashv E \ \circ\ \mathbf{case}_{\mathsf{m}}\ [\,]\ \mathbf{of}\ (x_1\ ,\ x_2) \mapsto u : (\mathsf{T}_1\otimes\mathsf{T}_2)\rightarrowtail\mathsf{U}_0}$$

λ_d–TY$/\otimes$E

$$\frac{\Gamma_1 \vdash t : \mathsf{T}_1\otimes\mathsf{T}_2 \qquad \Gamma_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u : \mathsf{U}}{\mathsf{m}\cdot\Gamma_1 + \Gamma_2 \vdash \mathbf{case}_{\mathsf{m}}\ t\ \mathbf{of}\ (x_1\ ,\ x_2) \mapsto u : \mathsf{U}}$$

- the typing context $\mathsf{m}\cdot\Delta_1,\ \Delta_2$ in the premise for E in λ_d–TY$_\text{E}/\otimes$E corresponds to $\mathsf{m}\cdot\Gamma_1 + \Gamma_2$ in the conclusion of λ_d–TY$/\otimes$E;
- the typing context $\Delta_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2$ in the premise for term u in λ_d–TY$_\text{E}/\otimes$E corresponds to the typing context $\Gamma_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2$ for the same term in λ_d–TY$/\otimes$E;
- the typing context Δ_1 in the conclusion for $E\ \circ\ \mathbf{case}_{\mathsf{m}}\ [\,]\ \mathbf{of}\ (x_1\ ,\ x_2) \mapsto u$ in λ_d–TY$_\text{E}/\otimes$E corresponds to the typing context Γ_1 in the premise for t in λ_d–TY$/\otimes$E (the term t is located where the focus $[\,]$ is in λ_d–TY$_\text{E}/\otimes$E).

We think of the typing rule for an evaluation context as a rotation of the typing rule for the associated term, where the typing contexts of one subterm and the conclusion are swapped, and the typing contexts of the other potential subterms are kept unchanged (with the difference that typing contexts for evaluation contexts are of shape Δ instead of Γ).

2.6.3 Reduction semantics

Now that every piece is in place, let's focus on reduction rules of λ_d. As in λ_{Ldm}, focus, unfocus and contraction rules of λ_d (see Figures 2.9 and 2.10) are triggered in a purely deterministic fashion. Once a subterm is a value, it cannot be focused on again.

Figure 2.9 presents all the rules that don't incur any substitution on the evaluation context.

Reduction rules for function application and pattern-matching are the same as in λ_{Ldm}. Reduction rules for $\mathbf{to}_{\ltimes}$ are pretty straightforward: once we have a value at hand, we embed it in a trivial ampar with just unit on the right. Rules for $\mathbf{from}_{\ltimes}$ are fairly symmetric: once we have an ampar with a value of the shape $\mathsf{Mod}_{1\infty}\ v_1$ on the right, we can extract both the left and right side out of the ampar shell, into a normal pair.

Finally, $\mathbf{new}_{\ltimes}$ has only a single rule that transforms it into the "identity" ampar object with just one hole on the left, and the corresponding destination on the right.

Rules of Figure 2.10 are all related to $\mathbf{upd}_{\ltimes}$ and destination-filling operators, whose contraction rules modify the evaluation context deeply, instead of just pushing or popping a focusing composant. For that, we need to introduce the special substitution $E(\!|h :=_H v|\!)$ that is used to update structures under construction, that are attached to open ampar focusing components in the stack. Such a substitution is triggered when a destination $\rightarrow\! h$ is filled in the term under focus, typically in destination-filling primitives reductions, and results in the value v being written to hole $\boxed{h}$. The value v may contain holes itself (e.g. when the hollow constructor $\mathsf{Inl}\ \boxed{h'+1}$ is being written to the hole $\boxed{h}$ in λ_d–SEM$/\lfloor\oplus\rfloor$E$_1$C), hence the set H tracks the potential hole names introduced by value v, and is used to update the hole name set of the corresponding (open) ampar. Proper definition of $E(\!|h :=_H v|\!)$ is given at the top of Figure 2.9.

λ_d–SEM$/\lfloor 1\rfloor$EC and λ_d–SEM$/\lfloor\multimap\rfloor$EC of Figure 2.10 do not create any new hole; they only write a value to an existing one. On the other hand, rules λ_d–SEM$/\lfloor\oplus\rfloor$E$_1$C, λ_d–SEM$/\lfloor\oplus\rfloor$E$_2$C, λ_d–SEM$/\lfloor !\rfloor$EC and λ_d–SEM$/\lfloor\otimes\rfloor$EC all write a hollow constructor to the hole $\boxed{h}$ that contains new holes. Thus, we need to generate fresh names for these new holes, and also return a destination for each new hole with a matching name.

$\boxed{\vdash E[\,t\,] : \mathtt{T}}$ *(Typing judgment for commands)*

$$\frac{\Delta \dashv E : \mathtt{T} \rightarrowtail \mathtt{U}_0 \qquad \Delta \vdash t : \mathtt{T}}{\vdash E[\,t\,] : \mathtt{U}_0}\ \lambda_d\text{–TY/CMD}$$

Name set shift and conditional name shift:

$$H \pm h' \triangleq \{h + h' \mid h \in H\}$$

$$h[H \pm h'] \triangleq \begin{cases} h + h' & \text{if } h \in H \\ h & \text{otherwise} \end{cases}$$

Special substitution for open ampars:

$$\left(E \circ {}^{\text{op}}_{\{h\} \sqcup H}\langle v_2 \,{}^\wedge\, [\,]\rangle\right) \llparenthesis h :=_{H'} v' \rrparenthesis = E \circ {}^{\text{op}}_{H \sqcup H'}\langle v_2 \llparenthesis h :=_{H'} v' \rrparenthesis \,{}^\wedge\, [\,]\rangle$$

$$\left(E \circ e\right) \llparenthesis h :=_{H'} v' \rrparenthesis = E \llparenthesis h :=_{H'} v' \rrparenthesis \circ e \qquad \text{if } h \notin e$$

$\boxed{E[\,t\,] \longrightarrow E'[\,t'\,]}$ *(Small-step evaluation of commands)*

$$E[\,t'\ t\,] \longrightarrow (E \circ t'\ [\,])[\,t\,] \qquad \star \quad \lambda_d\text{–SEM}/\multimap\text{EF}_1$$

$$(E \circ t'\ [\,])[\,v\,] \longrightarrow E[\,t'\ v\,] \qquad \lambda_d\text{–SEM}/\multimap\text{EU}_1$$

$$E[\,t'\ v\,] \longrightarrow (E \circ [\,]\ v)[\,t'\,] \qquad \star \quad \lambda_d\text{–SEM}/\multimap\text{EF}_2$$

$$(E \circ [\,]\ v)[\,v'\,] \longrightarrow E[\,v'\ v\,] \qquad \lambda_d\text{–SEM}/\multimap\text{EU}_2$$

$$E[\,(\lambda_{\mathtt{v}} x\ {}_{\mathtt{m}}\!\mapsto u)\ v\,] \longrightarrow E[\,u[x := v]\,] \qquad \lambda_d\text{–SEM}/\multimap\text{EC}$$

$$E[\,t \fatsemi u\,] \longrightarrow (E \circ [\,] \fatsemi u)[\,t\,] \qquad \star \quad \lambda_d\text{–SEM}/1\text{EF}$$

$$(E \circ [\,] \fatsemi u)[\,v\,] \longrightarrow E[\,v \fatsemi u\,] \qquad \lambda_d\text{–SEM}/1\text{EU}$$

$$E[\,() \fatsemi u\,] \longrightarrow E[\,u\,] \qquad \lambda_d\text{–SEM}/1\text{EC}$$

$$E[\,\mathtt{case}_{\mathtt{m}}\ t\ \mathtt{of}\ \{\mathtt{Inl}\,x_1 \mapsto u_1,\ \mathtt{Inr}\,x_2 \mapsto u_2\}\,] \longrightarrow (E \circ \mathtt{case}_{\mathtt{m}}\ [\,]\ \mathtt{of}\ \{\mathtt{Inl}\,x_1 \mapsto u_1,\ \mathtt{Inr}\,x_2 \mapsto u_2\})[\,t\,] \qquad \star \quad \lambda_d\text{–SEM}/\oplus\text{EF}$$

$$(E \circ \mathtt{case}_{\mathtt{m}}\ [\,]\ \mathtt{of}\ \{\mathtt{Inl}\,x_1 \mapsto u_1,\ \mathtt{Inr}\,x_2 \mapsto u_2\})[\,v\,] \longrightarrow E[\,\mathtt{case}_{\mathtt{m}}\ v\ \mathtt{of}\ \{\mathtt{Inl}\,x_1 \mapsto u_1,\ \mathtt{Inr}\,x_2 \mapsto u_2\}\,] \qquad \lambda_d\text{–SEM}/\oplus\text{EU}$$

$$E[\,\mathtt{case}_{\mathtt{m}}\ (\mathtt{Inl}\ v_1)\ \mathtt{of}\ \{\mathtt{Inl}\,x_1 \mapsto u_1,\ \mathtt{Inr}\,x_2 \mapsto u_2\}\,] \longrightarrow E[\,u_1[x_1 := v_1]\,] \qquad \lambda_d\text{–SEM}/\oplus\text{EC}_1$$

$$E[\,\mathtt{case}_{\mathtt{m}}\ (\mathtt{Inr}\ v_2)\ \mathtt{of}\ \{\mathtt{Inl}\,x_1 \mapsto u_1,\ \mathtt{Inr}\,x_2 \mapsto u_2\}\,] \longrightarrow E[\,u_2[x_2 := v_2]\,] \qquad \lambda_d\text{–SEM}/\oplus\text{EC}_2$$

$$E[\,\mathtt{case}_{\mathtt{m}}\ t\ \mathtt{of}\ (x_1, x_2) \mapsto u\,] \longrightarrow (E \circ \mathtt{case}_{\mathtt{m}}\ [\,]\ \mathtt{of}\ (x_1, x_2) \mapsto u)[\,t\,] \qquad \star \quad \lambda_d\text{–SEM}/\otimes\text{EF}$$

$$(E \circ \mathtt{case}_{\mathtt{m}}\ [\,]\ \mathtt{of}\ (x_1, x_2) \mapsto u)[\,v\,] \longrightarrow E[\,\mathtt{case}_{\mathtt{m}}\ v\ \mathtt{of}\ (x_1, x_2) \mapsto u\,] \qquad \lambda_d\text{–SEM}/\otimes\text{EU}$$

$$E[\,\mathtt{case}_{\mathtt{m}}\ (v_1, v_2)\ \mathtt{of}\ (x_1, x_2) \mapsto u\,] \longrightarrow E[\,u[x_1 := v_1][x_2 := v_2]\,] \qquad \lambda_d\text{–SEM}/\otimes\text{EC}$$

$$E[\,\mathtt{case}_{\mathtt{m}}\ t\ \mathtt{of}\ \mathtt{Mod}_{\mathtt{n}}\,x \mapsto u\,] \longrightarrow (E \circ \mathtt{case}_{\mathtt{m}}\ [\,]\ \mathtt{of}\ \mathtt{Mod}_{\mathtt{n}}\,x \mapsto u)[\,t\,] \qquad \star \quad \lambda_d\text{–SEM}/!\text{EF}$$

$$(E \circ \mathtt{case}_{\mathtt{m}}\ [\,]\ \mathtt{of}\ \mathtt{Mod}_{\mathtt{n}}\,x \mapsto u)[\,v\,] \longrightarrow E[\,\mathtt{case}_{\mathtt{m}}\ v\ \mathtt{of}\ \mathtt{Mod}_{\mathtt{n}}\,x \mapsto u\,] \qquad \lambda_d\text{–SEM}/!\text{EU}$$

$$E[\,\mathtt{case}_{\mathtt{m}}\ \mathtt{Mod}_{\mathtt{n}}\ v'\ \mathtt{of}\ \mathtt{Mod}_{\mathtt{n}}\,x \mapsto u\,] \longrightarrow E[\,u[x := v']\,] \qquad \lambda_d\text{–SEM}/!\text{EC}$$

$$E[\,\mathtt{to}_{\ltimes}\ u\,] \longrightarrow (E \circ \mathtt{to}_{\ltimes}\ [\,])[\,u\,] \qquad \star \quad \lambda_d\text{–SEM}/\ltimes\text{TOF}$$

$$(E \circ \mathtt{to}_{\ltimes}\ [\,])[\,v_2\,] \longrightarrow E[\,\mathtt{to}_{\ltimes}\ v_2\,] \qquad \lambda_d\text{–SEM}/\ltimes\text{TOU}$$

$$E[\,\mathtt{to}_{\ltimes}\ v_2\,] \longrightarrow E[\,{}_{\{\}}\langle v_2 \,{}^\wedge\, ()\rangle\,] \qquad \lambda_d\text{–SEM}/\ltimes\text{TOC}$$

$$E[\,\mathtt{from}_{\ltimes}\ t\,] \longrightarrow (E \circ \mathtt{from}_{\ltimes}\ [\,])[\,t\,] \qquad \star \quad \lambda_d\text{–SEM}/\ltimes\text{FROMF}$$

$$(E \circ \mathtt{from}_{\ltimes}\ [\,])[\,v\,] \longrightarrow E[\,\mathtt{from}_{\ltimes}\ v\,] \qquad \lambda_d\text{–SEM}/\ltimes\text{FROMU}$$

$$E[\,\mathtt{from}_{\ltimes}\ {}_{\{\}}\langle v_2 \,{}^\wedge\, \mathtt{Mod}_{1\infty}\ v_1\rangle\,] \longrightarrow E[\,(v_2, \mathtt{Mod}_{1\infty}\ v_1)\,] \qquad \lambda_d\text{–SEM}/\ltimes\text{FROMC}$$

$$E[\,\mathtt{new}_{\ltimes}\,] \longrightarrow E[\,{}_{\{1\}}\langle \boxed{1} \,{}^\wedge\, \to\!1\rangle\,] \qquad \lambda_d\text{–SEM}/\ltimes\text{NEWC}$$

$\star$: only allowed if the term under focus is not already a value

Figure 2.9: Small-step reduction of commands for λ_d (part 1)

$$
\begin{array}{lcll}
E\left[\mathsf{upd}_{\ltimes}\ t\ \mathsf{with}\ x \mapsto t'\right] & \longrightarrow & \left(E \circ \mathsf{upd}_{\ltimes}\ [\,]\ \mathsf{with}\ x \mapsto t'\right)\left[t\right] & \star\ \ \lambda_d\text{–SEM}/\ltimes\text{UPDF} \\
\left(E \circ \mathsf{upd}_{\ltimes}\ [\,]\ \mathsf{with}\ x \mapsto t'\right)\left[v\right] & \longrightarrow & E\left[\mathsf{upd}_{\ltimes}\ v\ \mathsf{with}\ x \mapsto t'\right] & \ \ \lambda_d\text{–SEM}/\ltimes\text{UPDU} \\
E\left[\mathsf{upd}_{\ltimes}\ {}_{H}\langle v_2 \mathbin{^\frown} v_1\rangle\ \mathsf{with}\ x \mapsto t'\right] & \longrightarrow & \left(E \circ \overset{\text{op}}{\underset{H\pm h'''}{}}\langle v_2[H{\pm}h'''] \mathbin{^\frown} [\,]\rangle\right)\left[t'[x := v_1[H{\pm}h''']]\right] & \star\ \ \lambda_d\text{–SEM}/\ltimes\text{OP} \\
\left(E \circ \overset{\text{op}}{\underset{H}{}}\langle v_2 \mathbin{^\frown} [\,]\rangle\right)\left[v_1\right] & \longrightarrow & E\left[{}_{H}\langle v_2 \mathbin{^\frown} v_1\rangle\right] & \ \ \lambda_d\text{–SEM}/\ltimes\text{CL} \\
E\left[t \lhd ()\right] & \longrightarrow & \left(E \circ [\,] \lhd ()\right)\left[t\right] & \star\ \ \lambda_d\text{–SEM}/\lfloor 1\rfloor\text{EF} \\
\left(E \circ [\,] \lhd ()\right)\left[v\right] & \longrightarrow & E\left[v \lhd ()\right] & \ \ \lambda_d\text{–SEM}/\lfloor 1\rfloor\text{EU} \\
E\left[\rightarrow h \lhd ()\right] & \longrightarrow & E(\!|h :=_{\{\}}\ ()|\!)\left[()\right] & \ \ \lambda_d\text{–SEM}/\lfloor 1\rfloor\text{EC} \\
E\left[t \lhd \mathsf{Inl}\right] & \longrightarrow & \left(E \circ [\,] \lhd \mathsf{Inl}\right)\left[t\right] & \star\ \ \lambda_d\text{–SEM}/\lfloor\oplus\rfloor\text{E}_1\text{F} \\
\left(E \circ [\,] \lhd \mathsf{Inl}\right)\left[v\right] & \longrightarrow & E\left[v \lhd \mathsf{Inl}\right] & \ \ \lambda_d\text{–SEM}/\lfloor\oplus\rfloor\text{E}_1\text{U} \\
E\left[\rightarrow h \lhd \mathsf{Inl}\right] & \longrightarrow & E(\!|h :=_{\{h'+1\}}\ \mathsf{Inl}\ \boxed{h'+1}|\!)\left[\rightarrow h'+1\right] & \star\ \ \lambda_d\text{–SEM}/\lfloor\oplus\rfloor\text{E}_1\text{C} \\
E\left[t \lhd \mathsf{Inr}\right] & \longrightarrow & \left(E \circ [\,] \lhd \mathsf{Inr}\right)\left[t\right] & \star\ \ \lambda_d\text{–SEM}/\lfloor\oplus\rfloor\text{E}_2\text{F} \\
\left(E \circ [\,] \lhd \mathsf{Inr}\right)\left[v\right] & \longrightarrow & E\left[v \lhd \mathsf{Inr}\right] & \ \ \lambda_d\text{–SEM}/\lfloor\oplus\rfloor\text{E}_2\text{U} \\
E\left[\rightarrow h \lhd \mathsf{Inr}\right] & \longrightarrow & E(\!|h :=_{\{h'+1\}}\ \mathsf{Inr}\ \boxed{h'+1}|\!)\left[\rightarrow h'+1\right] & \star\ \ \lambda_d\text{–SEM}/\lfloor\oplus\rfloor\text{E}_2\text{C} \\
E\left[t \lhd \mathsf{Mod}_{\mathrm{m}}\right] & \longrightarrow & \left(E \circ [\,] \lhd \mathsf{Mod}_{\mathrm{m}}\right)\left[t\right] & \star\ \ \lambda_d\text{–SEM}/\lfloor !\rfloor\text{EF} \\
\left(E \circ [\,] \lhd \mathsf{Mod}_{\mathrm{m}}\right)\left[v\right] & \longrightarrow & E\left[v \lhd \mathsf{Mod}_{\mathrm{m}}\right] & \ \ \lambda_d\text{–SEM}/\lfloor !\rfloor\text{EU} \\
E\left[\rightarrow h \lhd \mathsf{Mod}_{\mathrm{m}}\right] & \longrightarrow & E(\!|h :=_{\{h'+1\}}\ \mathsf{Mod}_{\mathrm{m}}\ \boxed{h'+1}|\!)\left[\rightarrow h'+1\right] & \star\ \ \lambda_d\text{–SEM}/\lfloor !\rfloor\text{EC} \\
E\left[t \lhd (,)\right] & \longrightarrow & \left(E \circ [\,] \lhd (,)\right)\left[t\right] & \star\ \ \lambda_d\text{–SEM}/\lfloor\otimes\rfloor\text{EF} \\
\left(E \circ [\,] \lhd (,)\right)\left[v\right] & \longrightarrow & E\left[v \lhd (,)\right] & \ \ \lambda_d\text{–SEM}/\lfloor\otimes\rfloor\text{EU} \\
E\left[\rightarrow h \lhd (,)\right] & \longrightarrow & E(\!|h :=_{\{h'+1,h'+2\}}\ (\boxed{h'+1}, \boxed{h'+2})|\!)\left[(\rightarrow h'+1, \rightarrow h'+2)\right] & \star\ \ \lambda_d\text{–SEM}/\lfloor\otimes\rfloor\text{EC} \\
E\left[t \lhd (\boldsymbol{\lambda} x\ {}_{\mathrm{m}}\!\mapsto u)\right] & \longrightarrow & \left(E \circ [\,] \lhd (\boldsymbol{\lambda} x\ {}_{\mathrm{m}}\!\mapsto u)\right)\left[t\right] & \star\ \ \lambda_d\text{–SEM}/\lfloor\multimap\rfloor\text{EF} \\
\left(E \circ [\,] \lhd (\boldsymbol{\lambda} x\ {}_{\mathrm{m}}\!\mapsto u)\right)\left[v\right] & \longrightarrow & E\left[v \lhd (\boldsymbol{\lambda} x\ {}_{\mathrm{m}}\!\mapsto u)\right] & \ \ \lambda_d\text{–SEM}/\lfloor\multimap\rfloor\text{EU} \\
E\left[\rightarrow h \lhd (\boldsymbol{\lambda} x\ {}_{\mathrm{m}}\!\mapsto u)\right] & \longrightarrow & E(\!|h :=_{\{\}}\ \boldsymbol{\lambda}_{\mathsf{v}} x\ {}_{\mathrm{m}}\!\mapsto u|\!)\left[()\right] & \ \ \lambda_d\text{–SEM}/\lfloor\multimap\rfloor\text{EC} \\
E\left[t \mathbin{\lhd\!\circ} t'\right] & \longrightarrow & \left(E \circ [\,] \mathbin{\lhd\!\circ} t'\right)\left[t\right] & \star\ \ \lambda_d\text{–SEM}/\lfloor\text{C}\rfloor\text{EF}_1 \\
\left(E \circ [\,] \mathbin{\lhd\!\circ} t'\right)\left[v\right] & \longrightarrow & E\left[v \mathbin{\lhd\!\circ} t'\right] & \ \ \lambda_d\text{–SEM}/\lfloor\text{C}\rfloor\text{EU}_1 \\
E\left[v \mathbin{\lhd\!\circ} t'\right] & \longrightarrow & \left(E \circ v \mathbin{\lhd\!\circ} [\,]\right)\left[t'\right] & \star\ \ \lambda_d\text{–SEM}/\lfloor\text{C}\rfloor\text{EF}_2 \\
\left(E \circ v \mathbin{\lhd\!\circ} [\,]\right)\left[v'\right] & \longrightarrow & E\left[v \mathbin{\lhd\!\circ} v'\right] & \ \ \lambda_d\text{–SEM}/\lfloor\text{C}\rfloor\text{EU}_2 \\
E\left[\rightarrow h \mathbin{\lhd\!\circ}_{H} \langle v_2 \mathbin{^\frown} v_1\rangle\right] & \longrightarrow & E(\!|h :=_{(H\pm h'')}\ v_2[H{\pm}h'']|\!)\left[v_1[H{\pm}h'']\right] & \star\ \ \lambda_d\text{–SEM}/\lfloor\text{C}\rfloor\text{EC} \\
E\left[t \blacktriangleleft t'\right] & \longrightarrow & \left(E \circ [\,] \blacktriangleleft t'\right)\left[t\right] & \star\ \ \lambda_d\text{–SEM}/\lfloor\text{L}\rfloor\text{EF}_1 \\
\left(E \circ [\,] \blacktriangleleft t'\right)\left[v\right] & \longrightarrow & E\left[v \blacktriangleleft t'\right] & \ \ \lambda_d\text{–SEM}/\lfloor\text{L}\rfloor\text{EU}_1 \\
E\left[v \blacktriangleleft t'\right] & \longrightarrow & \left(E \circ v \blacktriangleleft [\,]\right)\left[t'\right] & \star\ \ \lambda_d\text{–SEM}/\lfloor\text{L}\rfloor\text{EF}_2 \\
\left(E \circ v \blacktriangleleft [\,]\right)\left[v'\right] & \longrightarrow & E\left[v \blacktriangleleft v'\right] & \ \ \lambda_d\text{–SEM}/\lfloor\text{L}\rfloor\text{EU}_2 \\
E\left[\rightarrow h \blacktriangleleft v\right] & \longrightarrow & E(\!|h :=_{\{\}}\ v|\!)\left[()\right] & \ \ \lambda_d\text{–SEM}/\lfloor\text{L}\rfloor\text{EC}
\end{array}
$$

$$
\text{where}\quad \left\{\begin{array}{lcl}
h' & = & \mathsf{max}(\mathsf{hnames}(E) \cup \{h\})+1 \\
h'' & = & \mathsf{max}(H \cup (\mathsf{hnames}(E) \cup \{h\}))+1 \\
h''' & = & \mathsf{max}(H \cup \mathsf{hnames}(E))+1
\end{array}\right.
$$

$\star$: only allowed if the term under focus is not already a value

Figure 2.10: Small-step reduction of commands for λ_d (part 2)

The substitution $E\,(\!|h :=_H v|\!)$ should only be performed if h is a globally unique name; otherwise we break the promise of a write-once memory model. To this effect, we allow name shadowing while an ampar is closed (which is why $\mathtt{new}_{\ltimes}$ is allowed to reduce to the same ${}_{\{1\}}\langle \boxed{1} \,{}_{\smallfrown}\, \rightarrow\! 1\rangle$ every time), but as soon as an ampar is open, it should have globally unique hole names. This restriction is enforced in rule λ_d–TY$_{\text{E}}$/$\ltimes$OP by premise $\mathtt{hnames}(E)\ \#\#\ \mathtt{hnames}(\Delta_3)$, requiring hole name sets from E and Δ_3 to be disjoint when an open ampar focusing component is created during reduction of $\mathtt{upd}_{\ltimes}$. Likewise, any hollow constructor written to a hole should have globally unique hole names. We assume that hole names are natural numbers for simplicity's sake.

To obtain globally fresh names, in the premises of the corresponding rules, we first set $h' = \mathtt{max}(\mathtt{hnames}(E) \cup \{h\}) + 1$ or similar definitions for h'' and h''' (see at the bottom of Figure 2.10) to find the next unused name. Then we use either the *shifted set* $H \pm h'$ or the *conditional shift operator* $h[H \pm h']$ as defined at top of Figure 2.9 to replace all names or just specific one with fresh unused names. We extend *conditional shift* $\bullet[H \pm h']$ to arbitrary values, terms, and typing contexts in the obvious way (keeping in mind that ${}_{H'}\langle v_2 \,{}_{\smallfrown}\, v_1\rangle$ binds the names in H').

Rules λ_d–SEM/$\ltimes$OP and λ_d–SEM/$\ltimes$CL dictate how and when a closed ampar (a value) is converted to an open ampar (a focusing component) and vice-versa, and they make use of the shifting strategy we've just introduced. With λ_d–SEM/$\ltimes$OP, the hole names bound by the ampar gets renamed to fresh ones, and the left-hand side gets attached to the focusing component ${}^{\text{op}}_{H \pm h'}\langle v_2[H \pm h'''] \,{}_{\smallfrown}\, [\,]\rangle$ while the right-hand side (containing destinations) is substituted in the body of the $\mathtt{upd}_{\ltimes}$ statement (which becomes the new term under focus). The rule λ_d–SEM/$\ltimes$CL triggers when the body of a $\mathtt{upd}_{\ltimes}$ statement has reduced to a value. In that case, we can close the ampar, by popping the focusing component from the stack E and merging back with v_2 to form a closed ampar again.

In rule λ_d–SEM/$\lfloor$C$\rfloor$EC, we write the left-hand side v_2 of a closed ampar ${}_{H}\langle v_2 \,{}_{\smallfrown}\, v_1\rangle$ to a hole $\boxed{h}$ that is part of a structure with holes somewhere inside E. This results in the composition of two structures with holes. Because we dissociate v_2 and v_1 that were previously bound together by the ampar connective (v_2 is merged with another structure, while v_1 becomes the new focus), their hole names are no longer bound, so we need to make them globally unique, as we do when an ampar is opened with $\mathtt{upd}_{\ltimes}$. This renaming is carried out by the conditional shift $v_2[H \pm h'']$ and $v_1[H \pm h'']$.

Type safety With the semantics now defined, we can state the usual type safety theorems:

Theorem 1 (Type preservation). *If* $\vdash E\,[\,t\,] : \mathtt{T}$ *and* $E\,[\,t\,] \longrightarrow E'\,[\,t'\,]$ *then* $\vdash E'\,[\,t'\,] : \mathtt{T}$.

Theorem 2 (Progress). *If* $\vdash E\,[\,t\,] : \mathtt{T}$ *and* $\forall v, E[t] \neq [\,]\,[v]$ *then* $\exists E', t'.\ E\,[\,t\,] \longrightarrow E'\,[\,t'\,]$.

A command of the form $[\,]\,[v]$ cannot be reduced further, as it only contains a fully determined value, and no pending computation. This it is the stopping point of the reduction, and any well-typed command eventually reaches this form.

2.7 Formal proof of type safety

We've proved type preservation and progress theorems with the Coq proof assistant. The artifact containing the formalization of λ_d and the machine-verified proofs of type-safety (Theorems 1 and 2) is available at `https://doi.org/10.5281/zenodo.14534423`.

Turning to a proof assistant was a pragmatic choice: typing context handling in λ_d can be quite finicky, and it was hard, without computer assistance, to make sure that we hadn't made mistakes in our proofs. The version of λ_d that we've proved is written in Ott [Sew+07], the same Ott file is used as a source for this article, making sure that we've proved the same system as we're presenting; though some visual simplification is applied by a script to produce the version in the article.

Most of the proof was done by myself with little prior experience with Coq. This goes to show that Coq is reasonably approachable even for non-trivial development. However the development sped up dramatically thanks to the help of my industrial advisor, who was able to use his long prior experience with Coq to introduce the good core lemmas upon which we could easily base many others.

In total the proof is about 7000 lines long, and contains nearly 500 lemmas. Many of the cases of the type preservation and progress lemmas are similar. To handle such repetitive cases, the use of a large-language-model based autocompletion system has proven quite effective. Also the proofs aren't particularly elegant. For instance, we don't have any abstract formalization of semirings: it was more expedient to brute-force the properties we needed by hand.

There are nonetheless a few points of interest in our Coq development. First, we represent contexts as finite-domain functions, rather than as syntactic lists. This works much better when defining sums of context. There are a handful of finite-function libraries in the ecosystem, but we needed finite dependent functions (because the type of binders depend on whether we're binding a variable name or a hole name). This didn't exist, but for our limited purpose, it ended up not being too costly rolling our own (about 1000 lines of proofs). The underlying data type is actual functions: this was simpler to develop, but in exchange equality gets more complex than with a bespoke data type.

Secondly, addition of context is partial since we can only add two binding of the same name if they also have the same type. Instead of representing addition as a binary function to an optional context, we represent addition as a total function to contexts, but we change contexts to allow faulty bindings on some names. This works well better for our Ott-written rules, at the cost of needing well-formedness preconditions in the premises of typing rules as well as some lemmas.

Finally, we decided to assume a few axioms. The issue we encountered was that the inference rules produced by Ott weren't conducive to using setoid equality, which turned out to be problematic with our type for finite function:

```
Record T A B := {
   underlying :> forall x:A, option (B x);
   supported : exists l : list A, Support l underlying;
  }.
```

where `Support l f` means that `l` contains the domain of `f`. To make the equality of finite function be strict equality `eq`, we assumed functional extensionality and proof irrelevance. In some circumstances, we've also needed to list the finite functions' domains. But in the definition, the domain is sealed behind a proposition, so we also assumed classical logic as well as indefinite description:

```
Axiom constructive_indefinite_description :
   forall (A : Type) (P : A->Prop), (exists x, P x) -> { x : A | P x }.
```

Together, they let us extract the domain from the proposition. Again this isn't particularly elegant, we could have avoided some of these axioms at the price of more complex development. But we decided to favor expediency over elegance, given how much time the formal development had already consumed from the three years allowed to complete the PhD work.

2.8 Implementation of λ_d using in-place memory mutations

The formal language presented in Sections 2.5 and 2.6 is not meant to be implemented as-is. Practical implementation of most λ_d's ideas will be the focus of Chapters 3 and 4.

First, λ_d doesn't have recursion, this would have obscured the formal presentation of the system. However, adding a standard form of recursion doesn't create any complication.

Secondly, ampars are not managed linearly in λ_d; only destinations are. That is to say that an ampar can be wrapped in an exponential, e.g. $\texttt{Mod}_{\omega\nu}\ {}_{\{h\}}\langle 0 :: \boxed{h} \,{}_{\wedge} {\to}h \rangle$ (representing a difference list 0::$\boxed{}$ that can be used non-linearly), and then used twice, each time in a different way:

$$\begin{array}{ll}
\texttt{case}\ \texttt{Mod}_{\omega\nu}\ {}_{\{h\}}\langle 0 :: \boxed{h} \,{}_{\wedge} {\to}h \rangle\ \texttt{of}\ \texttt{Mod}_{\omega\nu}\ x \mapsto & \\
\quad \texttt{let}\ x_1 \coloneqq x\ \texttt{append}\ 1\ \texttt{in} & \\
\quad \texttt{let}\ x_2 \coloneqq x\ \texttt{append}\ 2\ \texttt{in} & \longrightarrow^* \quad 0 :: 1 :: 0 :: 2 :: [] \\
\qquad \texttt{toList}\ (x_1\ \texttt{concat}\ x_2) &
\end{array}$$

It may seem counter-intuitive at first, but this program is valid and safe in λ_d. Thanks to the renaming discipline we detailed in Section 2.6.3, every time an ampar is operated over with $\texttt{upd}_\ltimes$, its hole names are renamed to fresh ones. One way we can support this is to allocate a fresh copy of x every time we call **append** (which is implemented in terms of $\texttt{upd}_\ltimes$), in a copy-on-write fashion. This way filling destinations is still implemented as mutation.

However, this is a long way from the efficient implementation promised in Section 2.2. Copy-on-write can be optimized using fully-in-place functional programming [LLS23], where, thanks to reference counting, we don't need to perform a copy when the difference list isn't aliased; but that won't be the direction we will follow in the following development, as we don't want to deal explicitly with reference counting.

An alternative is to refine the linear type system further in order to guarantee that ampars are unique and avoid copy-on-write altogether. We held back from doing that in the formalization of λ_d as, again, it obfuscates the presentation of the system without adding much in return.

To make ampars linear, we follow a recipe proposed by Spiwack et al. [Spi+22] and introduce a new type **Token**, together with primitives **dup** and **drop**. We also switch $\texttt{new}_\ltimes$ for $\texttt{new}_{\ltimes\texttt{IP}}$:

$$\begin{array}{lll}
\texttt{dup} & : & \texttt{Token} \multimap \texttt{Token} \otimes \texttt{Token} \\
\texttt{drop} & : & \texttt{Token} \multimap 1 \\
\texttt{new}_{\ltimes\texttt{IP}} & : & \texttt{Token} \multimap \texttt{T} \ltimes \lfloor \texttt{T} \rfloor
\end{array}$$

For the in-place system to work, one solution is to consider that a linear root token variable, tok_0, is available to a program. "Closed" programs can now typecheck in the non-empty context $\{tok_0 :_{1\infty} \texttt{Token}\}$. tok_0 can be used to create new tokens tok_k via **dup**, but each of these tokens still has to be used linearly.

Ampar produced by $\mathtt{new}_{\ltimes\mathtt{IP}}$ have a linear dependency on a variable tok_k. If an ampar produced by $\mathtt{new}_{\ltimes\mathtt{IP}}\ \mathit{tok}_k$ were to be used twice in a block t, then t would require a typing context $\{\mathit{tok}_k : {}_{\omega\nu}\mathtt{Token}\}$, that itself would require tok_0 to have multiplicity ω too. Thus the program would be rejected.

Instead of providing a root token to any closed program, which is a rather exotic modification of the type system, we can alternatively add a primitive function
$\mathtt{withToken} : (\mathtt{Token}\ {}_{1\infty}\!\multimap {!}_{\omega\infty}\mathtt{T}) \multimap {!}_{\omega\infty}\mathtt{T}$ that let the user use a token in a delimited scope, and the token cannot leak outside of it. This approach is similarly powerful and safe, while also being easier to bake in an existing functional language without modyfing the type system too much.

Now that ampars are managed linearly, we can change the allocation and renaming mechanisms:

- the hole name for a new ampar is chosen fresh right from the start (this corresponds to a new heap allocation);
- adding a new hollow constructor still require freshness for its hole names (this corresponds to a new heap allocation too);
- Using $\mathtt{upd}_{\ltimes}$ over an ampar and filling destinations or composing two ampars using $\triangleleft\!\circ$ no longer require any renaming: we have the guarantee that the all the names involved are globally fresh, and can only be used once, so it actually corresponds to an in-place memory update.

TODO: Cite new lorenzen paper here about non-linear first-class contexts

In implementation concerns of Chapters 3 and 4, we will focus only on λ_d extended with $\mathtt{Token}$s and $\mathtt{new}_{\ltimes\mathtt{IP}}$, in which ampars are linear resources.

From purely linked structures to more efficient memory forms In λ_d we only have binary product in sum types. However, it's very straightforward to extend the language and implement destination-based building for n-ary sums of n-ary products, with constructors for each variant having multiple fields directly, instead of each field needing an extra indirection as in the binary sum of products $\mathtt{1}\oplus(\mathtt{S}\otimes(\mathtt{T}\otimes\mathtt{U}))$. We will do that as soon as next chapter.

However, it's better for field's values to still be represented by pointers. Indeed, composition of incomplete structures relies on the idea that destinations pointing to holes of a structure v will still be valid if v get assigned to a field f of a bigger structure v'. That's true indeed if just the address of v is written to $v'.f$. However, if v is moved into v' completly (i.e. if f is an in-place/unpacked field), then the pointers representing destinations of v are now invalid. Our early experiments around DPS support for unpacked fields seem to indicate that we would need two classes of destinations, one supporting composition (for indirected fields) and one disallowing it (for unpacked fields). That won't be covered here.

2.9 λ_d: from theory to practice

Using a system of ages in addition to linearity, λ_d is a purely functional calculus which supports destinations in an extremely flexible way. It subsumes existing calculi from the literature for destination passing, allowing both composition of data structures with holes and storing destinations in data structures. Data structures are allowed to have multiple holes, and destinations can be stored in data structures that, themselves, have holes. The latter is the main reason to introduce ages and is key to λ_d's flexibility.

We don't anticipate that a system of ages like the one of λ_d will actually be used in a programming language: it's unlikely that destinations are so central to the design of a programming language that it's worth baking them so deeply in the type system. Perhaps a compiler that makes heavy use of destinations in its optimizer could use λ_d as a typed intermediate representation. But, more realistically, our expectation is that λ_d can be used as a theoretical framework to analyze destination-passing systems: if an API can be defined in λ_d then it's sound.

Our path forward, in next chapters, is to see which small restrictions we can impose on λ_d's flexiblity to make it possible to port most of its core ideas in a real-world functional programming language, namely, Haskell.

Chapter 3

Destination-passing style : a Haskell implementation

3.1 Introduction

Destination-passing style (DPS) programming takes its source in the early days of imperative languages with manual memory management. In the C programming language, it's quite common for a function not to allocate memory itself for its result, but rather to receive a reference to a memory location where to write its result (often named *out parameter*). In that scheme, the caller of the function has control over allocation and disposal of memory for the function result, and thus gets to choose where the latter will be written.

DPS programming is an adaptation of this idea for functional languages, based on two core concepts: having arbitrary data structures with *holes* — that is to say, memory cells that haven't been filled yet — and *destinations*, which are pointers to those holes. A destination can be passed around, as a first-class object of the language (unlike holes), and it allows remote action on its associated hole: when one *fills* the destination with a value, that value is in fact written in the hole. As structures are allowed to have holes, they can be built from the root down, rather than from the leaves up. Indeed, children of a parent node no longer have to be specified when the parent node is created; they can be left empty (which leaves holes in the parent node), and added later through destinations to those holes. It is thus possible to write very natural solutions to problems for which the usual functional bottom-up building approach is ill-fitting. On top of better expressiveness, DPS programming can lead to better time or space performance for critical parts of a program, allowing for example tail-recursive map, or efficient difference lists.

That being said, DPS programming is not about giving unlimited manual control over memory or using mutations without restrictions. The existence of a destination is directly linked to the existence of an accompanying hole: we say that a destination is *consumed* when it has already been used to write something in its associated hole. It must not be reused after that point, to ensure immutability and prevent a range of memory errors.

In this paper, I design a destination API whose memory safety (write-once model) is ensured through a linear type discipline. Linear type systems are based on Girard's Linear logic [Gir95], and introduce the concept of *linearity*: one can express through types that a function will *consume* its argument exactly once given the function result is *consumed* exactly once. Linearity helps to manage resources — such as destinations — that one should not forget to *consume* (e.g. forgetting to fill a hole before reading a structure), but also that shouldn't be reused several times.

The Haskell programming language is equipped with support for linear types through its main compiler, *GHC*, since version 9.0.1 [Ber+18]. But Haskell is also a *pure* functional language, which means that side effects are usually not safe to produce outside of monadic contexts. This led me to set a slightly more refined goal: I wanted to hide impure memory effects related to destinations behind a *pure* Haskell API, and make the whole safe through the linear type discipline. Although the proofs of type safety haven't been made yet, the early practical results seem to indicate that my API is safe, and its purity makes it more convenient to adopt DPS in a codebase compared to a monadic approach that would be more "contaminating".

The main contributions of this paper are

- a linearly-typed API for destinations that let us build and manipulate data structures with holes while exposing a pure interface (Section 3.4);
- a first implementation of destinations for Haskell relying on so-called compact regions (Section 3.5), together with a performance evaluation (Section 3.6). Implementation code is available in [Bag23a] and [Bag23b] (specifically `src/Compact/Pure/Internal.hs` and `bench/Bench`).

3.2 A short primer on linear types

Linear Haskell [Ber+18] introduces the linear function arrow, `a ⊸ b`, that guarantees that the argument of the function will be consumed exactly once when the result of the function is consumed exactly once. On the other hand, the regular function arrow `a → b` doesn't guarantee how many times its argument will be consumed when its result is consumed once.

A value is said to be *consumed once* (or *consumed linearly*) when it is pattern-matched on and its sub-components are consumed once; or when it is passed as an argument to a linear function whose result is consumed once. A function is said to be *consumed once* when it is applied to an argument and when the result is consumed exactly once. We say that a variable `x` is *used linearly* in an expression `u` when consuming `u` once implies consuming `x` exactly once. Linearity on function arrows thus creates a chain of requirements about consumption of values, which is usually bootstrapped by using the *scope function* trick, as detailed in Section 3.4.2.

Unrestricted values Linear Haskell introduces a wrapper named `Ur` which is used to indicate that a value in a linear context doesn't have to be used linearly. `Ur a` is equivalent to $!a$ in linear logic, and there is an equivalence between `Ur a ⊸ b` and `a → b`.

The value `(x, y)` is said to be consumed linearly only when both `x` and `y` are consumed exactly once; whereas `Ur x` is considered to be consumed once as long as one pattern-matches on it, even if `x` is not consumed exactly once after (it can be consumed several times or not at all). Conversely, both `x` and `y` are used linearly in `(x, y)`, whereas `x` is not used linearly in `Ur x`. As a result, only values already wrapped in `Ur` or coming from the left of a non-linear arrow can be put in another `Ur` without breaking linearity. The only exceptions are values of types that implement the `Movable` typeclass such as `Int` or `()`. `Movable` provides `move :: a ⊸ Ur a` so a value can escape linearity restrictions.

Operators Some Haskell operators are often used in the rest of this article:

`(<&>) :: Functor f ⇒ f a ⊸ (a ⊸ b) ⊸ f b` is the same as `fmap` with the order of the arguments flipped: `x <&> f = fmap f x`;

`(⨾) :: () ⊸ b ⊸ b` is used to chain a linear operation returning `()` with one returning a value of type `b` without breaking linearity;

`Class ⇒ ...` is notation for typeclass constraints (resolved implicitly by the compiler).

3.3 Motivating examples for DPS programming

The following subsections present three typical settings in which DPS programming brings expressiveness or performance benefits over a more traditional functional implementation.

3.3.1 Efficient difference lists

Linked lists are a staple of functional programming, but they aren't efficient for concatenation, especially when the concatenation calls are nested to the left.

In an imperative context, it would be quite easy to concatenate linked lists efficiently. One just has to keep both a pointer to the root and to the last *cons* cell of each list. Then, to concatenate two lists, one just has to mutate the last *cons* cell of the first one to point to the root of the second list.

It isn't possible to do so in an immutable functional context though. Instead, *difference lists* can be used: they are very fast to concatenate, and then to convert back into a list. They tend to emulate the idea of having a mutable (here, write-once) last *cons* cell. Usually, a difference list `x1 : ... : xn : □` is encoded by function `\ys → x1 : ... : xn : ys` taking a last element `ys :: [a]` and returning a value of type `[a]` too.

With such a representation, concatenation is function composition: `f1 <> f2 = f1 . f2`, and we have `mempty = id`[10], `toList f = f []` and `fromList xs = \ys → xs ++ ys`.

In DPS, instead of encoding the concept of a write-once hole with a function, we can represent the hole of type `[a]` as a first-class object with a *destination* of type `Dest [a]`. A difference list now becomes an actual data structure in memory — not just a pending computation — that has two handles: one to the root of the list of type `[a]`, and one to the yet-to-be-filled hole in the last cons cell, represented by the destination of type `Dest [a]`.

With the function encoding, it isn't possible to read the list until a last element of type `[a]` has been supplied to complete it. With the destination representation, this constraint must persist: the actual list `[a]` shouldn't be readable until the accompanying destination is filled, as pattern-matching on the hole would lead to a dreaded *segmentation fault*. This constraint

[10]`mempty` and `<>` are the usual notations for neutral element and binary operation of a monoid in Haskell.

```
data [a] = {- nil constructor -} [] | {- cons constructor -} (:) a [a]

type DList a = Incomplete [a] (Dest [a])

alloc ∷ DList a  -- API primitive (simplified signature w.r.t. Section 4)

append ∷ DList a ⊸ a → DList a
append i x =
  i <&> \d → case fill @'(:) d of
    (dh, dt) → fillLeaf x dh ⨟ dt

concat ∷ DList a ⊸ DList a ⊸ DList a
concat i1 i2 = i1 <&> \dt1 → fillComp i2 dt1

toList ∷ DList a ⊸ [a]
toList i = fromIncomplete_' (i <&> \dt → fill @'[] dt)
```

Listing 1: Implementation of difference lists with destinations

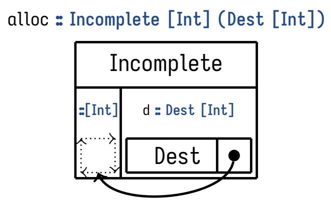


Figure 3.1: alloc

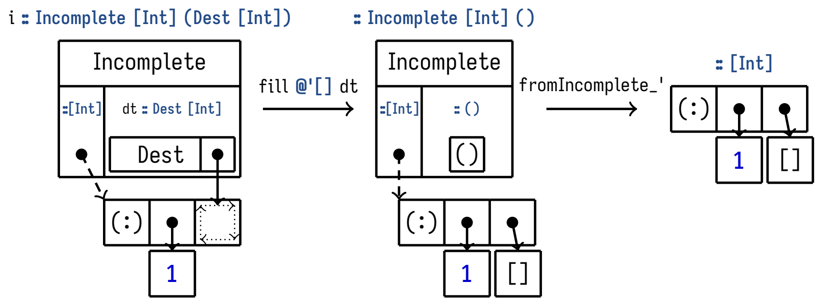


Figure 3.2: Memory behavior of toList i

is embodied by the Incomplete a b type of our destination API: b is what needs to be linearly consumed to make the a readable. The b side often carries the destinations of a structure. A difference list is then type DList a = Incomplete [a] (Dest [a]): the Dest [a] must be filled (with a [a]) to get a readable [a].

The implementation of destination-backed difference lists is presented in Listing 1. More details about the API primitives used by this implementation are given in Section 3.4. For now, it's important to note that fill is a function taking a constructor as a type parameter (often used with @ for type parameter application, and ' to lift a constructor to a type).

- alloc (Figure 3.1) returns a DList a which is exactly an Incomplete [a] (Dest [a]) structure. There is no data there yet and the list that will be fed in Dest [a] is exactly the list that the resulting Incomplete will hold. This is similar to the function encoding where \x → x represents the empty difference list;

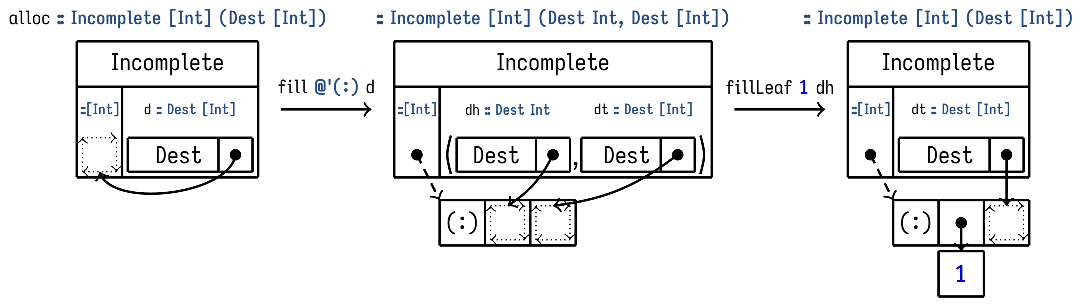

alloc :: Incomplete [Int] (Dest [Int])
:: Incomplete [Int] (Dest Int, Dest [Int])
:: Incomplete [Int] (Dest [Int])
Incomplete
::[Int]
d :: Dest [Int]
Dest
fill @'(:) d
dh :: Dest Int
dt :: Dest [Int]
(:)
fillLeaf 1 dh
1

Figure 3.3: Memory behavior of append alloc 1

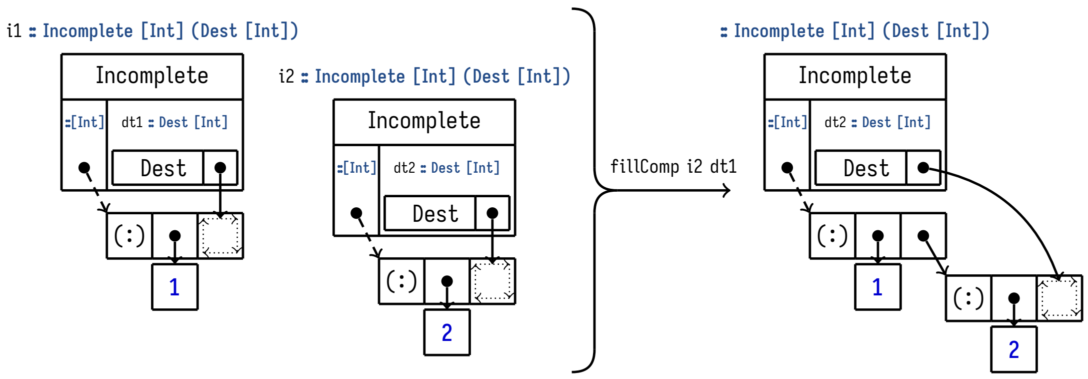

i1 :: Incomplete [Int] (Dest [Int])
i2 :: Incomplete [Int] (Dest [Int])
:: Incomplete [Int] (Dest [Int])
Incomplete
::[Int]
dt1 :: Dest [Int]
dt2 :: Dest [Int]
Dest
(:)
1
2
fillComp i2 dt1

Figure 3.4: Memory behavior of concat i1 i2 (based on fillComp)

- append (Figure 3.3) adds an element at the tail position of a difference list. For this, it first uses fill @'(:) to fill the hole at the end of the list represented by d :: Dest [a] with a hollow *cons* cell with two new holes pointed by dh :: Dest a and dt :: Dest [a]. Then, fillLeaf fills the hole represented by dh with the value of type a to append. The hole of the resulting difference list is the one pointed by dt :: Dest [a] which hasn't been filled yet.

- concat (Figure 3.4) concatenates two difference lists, i1 and i2. It uses fillComp to fill the destination dt1 of the first difference list with the root of the second difference list i2. The resulting Incomplete object hence has the same root as the first list, holds the elements of both lists, and inherits the hole of the second list. Memory-wise, concat just writes an address into a memory cell; no move is required.

- toList (Figure 3.2) completes the incomplete structure by plugging *nil* into its hole with fill @'[] (whose individual behavior is presented in Figure 3.6) and removes the Incomplete wrapper as the structure is now complete, using fromIncomplete_'.

To use this API safely, it is imperative that values of type Incomplete are used linearly. Otherwise we could first complete a difference list with l = toList i, then add a new cons cell with a hole to i with append i x (actually reusing the destination inside i for the second time). Doing that creates a hole inside l, although it is of type [a] so we are allowed to pattern-match on it (so we might get a segfault)! The simplified API of Listing 1 doesn't actually enforce the required linearity properties, I'll address that in Section 3.4.2.

```haskell
data Tree a = Nil | Node a (Tree a) (Tree a)

relabelDPS ∷ Tree a → Tree Int
relabelDPS tree = fst (mapAccumBFS (\st _ → (st + 1, st)) 1 tree)

mapAccumBFS ∷ ∀ a b s. (s → a → (s, b)) → s → Tree a → (Tree b, s)
mapAccumBFS f s0 tree =
  fromIncomplete' (    -- simplified alloc signature w.r.t. Section 4
    alloc <&> \dtree → go s0 (singleton (Ur tree, dtree)))
  where
    go ∷ s → Queue (Ur (Tree a), Dest (Tree b)) ⊸ Ur s
    go st q = case dequeue q of
      Nothing → Ur st
      Just ((utree, dtree), q') → case utree of
        Ur Nil → fill @'Nil dtree ⨾ go st q'
        Ur (Node x tl tr) → case fill @'Node dtree of
          (dy, dtl, dtr) →
            let q'' = q' `enqueue` (Ur tl, dtl) `enqueue` (Ur tr, dtr)
                (st', y) = f st x
              in fillLeaf y dy ⨾ go st' q''
```

Listing 2: Implementation of breadth-first tree traversal with destinations

This implementation of difference list matches closely the intended memory behaviour, we can expect it to be more efficient than the functional encoding. We'll see in Section 3.6 that the prototype implementation presented in Section 3.5 cannot yet demonstrate these performance improvements.

3.3.2 Breadth-first tree traversal

Consider the problem, which Okasaki attributes to Launchbury [Oka00]

> Given a tree T , create a new tree of the same shape, but with the values at the nodes replaced by the numbers $1 \ldots |T|$ in breadth-first order.

This problem admits a straightforward implementation if we're allowed to mutate trees. Nevertheless, a pure implementation is quite tricky [Oka00; Gib93] and a very elegant, albeit very clever, solution was proposed recently [Gib+23].

With destinations as first-class objects in our toolbelt, we can implement a solution that is both easy to come up with and efficient, doing only a single breadth-first traversal pass on the original tree. The main idea is to keep a queue of pairs of a tree to be relabeled and of the destination where the relabeled result is expected (as destinations can be stored in arbitrary containers!) and process each of them when their turn comes. The implementation provided in Listing 2, implements the slightly more general `mapAccumBFS` which applies on each node of the tree a relabeling function that can depend on a state.

Note that the signatures of `mapAccumBFS` and `relabelDPS` don't involve linear types. Linear types only appear in the inner loop `go`, which manipulates destinations. Linearity enforces the fact that every destination ever put in the queue is eventually filled at some point, which guarantees that the output tree is complete after the function has run.

As the state-transforming function `s → a → (s, b)` is non-linear, the nodes of the original tree won't be used in a linear fashion. However, we want to store these nodes together with their accompanying destinations in a queue of pairs, and destinations used to construct the queue have to be used linearly (because of `<&>` signature, which initially gives the root destination). That forces the queue to be used linearly, so the pairs too, so pairs' components too. Thus, we wrap the nodes of the input tree in the `Ur` wrapper, whose linear consumption allows for unrestricted use of its inner value, as detailed in Section 3.2.

This example shows how destinations can be used even in a non-linear setting in order to improve the expressiveness of the language. This more natural and less convoluted implementation of breadth-first traversal also presents great performance gains compared to the fancy functional implementation from [Gib+23], as detailed in Section 3.6.

3.3.3 Deserializing, lifetime, and garbage collection

In client-server applications, the following pattern is very frequent: the server receives a request from a client with a serialized payload, the server then deserializes the payload, runs some code, and respond to the request. Most often, the deserialized payload is kept alive for the entirety of the request handling. In a garbage collected language, there's a real cost to this: the garbage collector (GC) will traverse the deserialized payload again and again, although we know that all its internal pointers are live for the duration of the request.

Instead, we'd rather consider the deserialized payload as a single heap object, which doesn't need to be traversed, and is freed as a block. GHC supports this use-case with a feature named *compact regions* [Yan+15]. Compact regions contain normal heap objects, but the GC never follows pointers into a compact region. The flipside is that a compact region can only be collected when all of the objects it contains are dead.

With compact regions, we would first deserialize the payload normally, in the GC heap, then copy it into a compact region and only keep a reference to the copy. That way, internal pointers of the region copy will never be followed by the GC, and that copy will be collected as a whole later on, whereas the original in the GC heap will be collected immediately.

However, we are still allocating two copies of the deserialized payload. This is wasteful, it would be much better to allocate directly in the region, but this isn't part of the original compact region API. Destinations are one way to accomplish this. In fact, as I'll explain in Section 3.5, my implementation of DPS for Haskell is backed by compact regions because they provide more freedom to do low-level memory operations without interfering with GC.

Given a payload serialized as S-expressions, let's see how using destinations and compact regions for the parser can lead to greater performance. S-expressions are parenthesized lists whose elements are separated by spaces. These elements can be of several types: int, string, symbol (a textual token with no quotes around it), or a list of other S-expressions.

Parsing an S-expression can be done naively with mutually recursive functions:

```
1  parseSList :: ByteString → Int → [SExpr] → Either Error SExpr
2  parseSList bs i acc = case bs !? i of
3    Nothing → Left (UnexpectedEOFSList i)
4    Just x → if
5      | x == ')' → Right (SList i (reverse acc))
6      | isSpace x → parseSList bs (i + 1) acc
7      | otherwise → case parseSExpr bs i of
8          Left err → Left err
9          Right child → parseSList bs (endPos child + 1) (child : acc)
```

Listing 3: Implementation of the S-expression parser without destinations

- `parseSExpr` scans the next character, and either dispatches to `parseSList` if it encounters an opening parenthesis, or to `parseSString` if it encounters an opening quote, or eventually parses the string into a number or symbol;
- `parseSList` calls `parseSExpr` to parse the next token, and then calls itself again until reaching a closing parenthesis, accumulating the parsed elements along the way.

Only the implementation of `parseSList` will be presented here as it is enough for our purpose, but the full implementation of both the naive and destination-based versions of the whole parser can be found in `src/Compact/Pure/SExpr.hs` of [Bag23b] .

The implementation presented in Listing 3 is quite standard: the accumulator `acc` collects the nodes that are returned by `parseSExpr` in the reverse order (because it's the natural building order for a linked list without destinations). When the end of the list is reached (line 5), the accumulator is reversed, wrapped in the `SList` constructor, and returned.

We will see that destinations can bring very significative performance gains with only very little stylistic changes in the code. Accumulators of tail-recursive functions just have to be changed into destinations. Instead of writing elements into a list that will be reversed at the end as we did before, the program in the destination style will directly write the elements into their final location.

```
1  parseSListDPS :: ByteString → Int → Dest [SExpr] ⊸ Either Error Int
2  parseSListDPS bs i d = case bs !? i of
3    Nothing → fill @'[] d ⨾ Left (UnexpectedEOFSList i)
4    Just x → if
5      | x == ')' → fill @'[] d ⨾ Right i
6      | isSpace x → parseSListDPS bs (i + 1) d
7      | otherwise →
8          case fill @'(:) d of
9            (dh, dt) → case parseSExprDPS bs i dh of
10              Left err → fill @'[] dt ⨾ Left err
11              Right endPos → parseSListDPS bs (endPos + 1) dt
```

Listing 4: Implementation of the S-expression parser with destinations

```
data Token
consume   ::    Token ⊸ ()
dup2      ::    Token ⊸ (Token, Token)
withToken :: ∀ a. (Token ⊸ Ur a) ⊸ Ur a

data Incomplete a b
fmap              :: ∀ a b c. (b ⊸ c) ⊸ Incomplete a b ⊸ Incomplete b c
alloc             :: ∀ a.     Token ⊸ Incomplete a (Dest a)
intoIncomplete    :: ∀ a.     Token ⊸ a → Incomplete a ()
fromIncomplete_   :: ∀ a.     Incomplete a () ⊸ Ur a
fromIncomplete    :: ∀ a b.   Incomplete a (Ur c) ⊸ Ur (a, c)

data Dest a
type family DestsOf lCtor a -- returns dests associated to fields of constructor
fill     :: ∀ lCtor a. Dest a ⊸ DestsOf lCtor a
fillComp :: ∀ a b.     Incomplete a b ⊸ Dest a ⊸ b
fillLeaf :: ∀ a.       a → Dest a ⊸ ()
```

Listing 5: Destination API for Haskell

Code for `parseSListDPS` is presented in Listing 4. Let's see what changed compared to the naive implementation:

- even for error cases, we are forced to consume the destination that we receive as an argument (to stay linear), hence we write some sensible default data to it (see line 3);
- the `SExpr` value resulting from `parseSExprDPS` is not collected by `parseSListDPS` but instead written directly into its final location by `parseSExprDPS` through the passing and filling of destination `dh` (see line 9);
- adding an element of type `SExpr` to the accumulator `[SExpr]` is replaced with adding a new cons cell with `fill @'(:)` into the hole represented by `Dest [SExpr]`, writing an element to the *head* destination, and then doing a recursive call with the *tail* destination passed as an argument (which has type `Dest [SExpr]` again);
- instead of reversing and returning the accumulator at the end of the processing, it is enough to complete the list by writing a nil element to the tail destination (with `fill @'[]`, see line 5), as the list has been built in a top-down approach;
- DPS functions return the offset of the next character to read instead of a parsed value.

Thanks to that new implementation which is barely longer (in terms of lines of code) than the naive one, the program runs almost twice as fast, mostly because garbage-collection time goes to almost zero. The detailed benchmark is available in Section 3.6.

3.4 API Design

Table 5 presents my pure API for functional DPS programming. This API is sufficient to implement all the examples of Section 3.3. This section explains its various parts in detail.

3.4.1 The `Incomplete` type

The main design principle behind DPS structure building is that no structure can be read before all its destinations have been filled. That way, incomplete data structures can be freely passed around and stored, but need to be completed before any pattern-matching can be made on them.

Hence we introduce a new data type `Incomplete a b` where `a` stands for the type of the structure being built, and `b` is the type of what needs to be linearly consumed before the structure can be read. The idea is that one can map over the `b` side, which will contain destinations or containers with destinations inside, until there is no destination left but just a non-linear value that can safely escape (e.g. `()`, `Int`, or something wrapped in `Ur`). When destinations from the `b` side are consumed, the structure on the `a` side is built little by little in a top-down fashion, as we showed in Figures 3.3 and 3.4. And when no destination remains on the `b` side, the value of type `a` no longer has holes, thus is ready to be released/read.

It can be released in two ways: with `fromIncomplete_`, the value on the `b` side must be unit (`()`), and just the complete `a` is returned, wrapped in `Ur`. With `fromIncomplete`, the type on the `b` side must be of the form `Ur c`, and then a pair `Ur (a, c)` is returned.

It is actually safe to wrap the structure that has been built in `Ur` because its leaves either come from non-linear sources (as `fillLeaf :: a → Dest a ⊸ ()` consumes its first argument non-linearly) or are made of 0-ary constructors added with `fill`, both of which can be used in an unrestricted fashion safely. Variants `fromIncomplete_'` and `fromIncomplete'` from the beginning of this article just drop the `Ur` wrapper.

Conversely, the function `intoIncomplete` takes a non-linear argument of type `a` and wraps it into an `Incomplete` with no destinations left to be consumed.

3.4.2 Ensuring write-once model for holes with linear types

Types aren't linear by themselves in Linear Haskell. Instead, functions can be made to use their arguments linearly or not. So in direct style, where the consumer of a resource isn't tied to the resource creation site, there is no way to state that the resource must be used exactly once:

```haskell
createR          :: Resource -- no way to force the result to be used exactly once
consumeR          :: Resource ⊸ ()
exampleShouldFail :: () =
  let x = createR in consumeR x ⨾ consumeR x -- valid even if x is consumed twice
```

The solution is to force the consumer of a resource to become explicit at the creation site of the resource, and to check through its signature that it is indeed a linear continuation:

```haskell
withR      :: (Resource ⊸ a) ⊸ a
consumeR    :: Resource ⊸ ()
exampleFail :: () = withR (\x → consumeR x ⨾ consumeR x) -- not linear
```

The `Resource` type is in positive position in the signature of `withR`, so that the function should somehow know how to produce a `Resource`, but this is opaque for the user. What matters is that a resource can only be accessed by providing a linear continuation to `withR`.

Still, this is not enough; because `\x → x` is indeed a linear continuation, one could use `withR (\x → x)` to leak a `Resource`, and then use it in a non-linear fashion in the outside world. Hence we must forbid the resource from appearing anywhere in the return type of the continuation. To do that, we ask the return type to be wrapped in `Ur`: because the resource comes from the left of a linear arrow, and doesn't implement `Movable`, it cannot be wrapped in `Ur` without breaking linearity (see Section 3.2). On the other hand, a `Movable` value of type `()` or `Int` can be returned:

```
withR'       :: (Resource ⊸ Ur a) ⊸ Ur a
consumeR     :: Resource ⊸ ()
exampleOk'   :: Ur ()       = withR' (\x → let u :: () = consumeR x' in move u)
exampleFail' :: Ur Resource = withR' (\x → Ur x) -- not linear
```

This explicit *scope function* trick will no longer be necessary when linear constraints will land in GHC (see [Spi+22]). In the meantime, this principle has been used to ensure safety of the DPS implementation in Haskell.

Ensuring linear use of `Incomplete` objects If an `Incomplete` object is used linearly, then its destinations will be written to exactly once; this is ensured by the signature of `fmap` for `Incomplete`s. So we need to ensure that `Incomplete` objects are used linearly. For that, we introduce a new type `Token`. A token can be linearly exchanged one-for-one with an `Incomplete` of any type with `alloc`, linearly duplicated with `dup2`, or linearly deleted with `consume`. However, it cannot be linearly stored in `Ur` as it doesn't implement `Movable`.

As in the example above, we just ensure that `withToken :: (Token ⊸ Ur a) ⊸ Ur a` is the only source of `Token`s around. Now, to produce an `Incomplete`, one must get a token first, so has to be in the scope of a continuation passed to `withToken`. Putting either a `Token` or `Incomplete` in `Ur` inside the continuation would make it non-linear. So none of them can escape the scope as is, but a structure built from an `Incomplete` and finalized with `fromIncomplete` would be automatically wrapped in `Ur`, thus could safely escape[11].

3.4.3 Filling functions for destinations

The last part of the API is the one in charge of actually building the structures in a top-down fashion. To fill a hole represented by `Dest a`, three functions are available:

`fillLeaf :: ∀ a. a → Dest a ⊸ ()` uses a value of type `a` to fill the hole represented by the destination. The destination is consumed linearly, but the value to fill the hole isn't (as indicated by the first non-linear arrow). Memory-wise, the address of the object `a` is written into the memory cell pointed to by the destination (see Figure 3.7).

`fillComp :: ∀ a b. Incomplete a b ⊸ Dest a ⊸ b` is used to plug two `Incomplete` objects together. The target `Incomplete` isn't represented in the signature of the function. Instead, only the target hole that will receive the address of the child is represented by `Dest a`; and `Incomplete a b` in the signature refers to the child object. A call to `fillComp` always takes place in the scope of `fmap`/`<&>` over the parent object:

[11]This is why the `fromIncomplete'` and `fromIncomplete_'` variants aren't that useful in the actual memory-safe API (which differs slightly from the simplified examples of Section 3.3): here the built structure would be stuck in the scope function without its `Ur` escape pass.

```
parent :: Incomplete BigStruct (Dest SmallStruct, Dest OtherStruct)
child :: Incomplete SmallStruct (Dest Int)
comp = parent <&> \(ds, extra) → fillComp child ds
    :: Incomplete BigStruct (Dest Int, Dest OtherStruct)
```

The resulting structure `comp` is morally a `BigStruct` like `parent`, that inherited the hole from the child structure (`Dest Int`) and still has its other hole (`Dest OtherStruct`) to be filled. An example of memory behavior of `fillComp` in action can be seen in Figure 3.4.

`fill :: ∀ lCtor a. Dest a ⊸ DestsOf lCtor a` lets us build structures using layers of hollow constructors. It takes a constructor as a type parameter (`lCtor`) and allocates a hollow heap object that has the same header/tag as the specified constructor but unspecified fields. The address of the allocated hollow constructor is written in the destination that is passed to `fill`. As a result, one hole is now filled, but there is one new hole in the structure for each field left unspecified in the hollow constructor that is now part of the bigger structure. So `fill` returns one destination of matching type for each of the fields of the constructor. An example of the memory behavior of `fill @'(:) :: Dest [a] ⊸ (Dest a, Dest [a])` is given in Figure 3.5 and the one of `fill @'[] :: Dest [a] ⊸ ()` is given in Figure 3.6.

`DestsOf` is a type family (i.e. a function operating on types) whose role is to map a constructor to the type of destinations for its fields. For example, `DestsOf '[] [a] = ()` and `DestsOf '(:) [a] = (Dest a, Dest [a])`. More generally, there is a duality between the type of a constructor `Ctor :: (f_1 ... f_n) → a` and of the associated destination-filling functions `fill @'Ctor :: Dest a ⊸ (Dest f_1 ... De`
Destination-based data building can be seen as more general than the usual bottom-up constructor approach, as we can recover `Ctor` from the associated function `fill @'Ctor`, but not the reverse:

```
Ctor :: (f_1 ... f_n) → a
Ctor (x_1 ... x_n) = fromIncomplete_' (
  alloc <&> \(d :: Dest a) → case fill @'Ctor d of
    (dx_1 ... dx_n) → fillLeaf x_1 dx_1 ⨟ ... ⨟ fillLeaf x_n dx_n)
```

3.5 Implementing destinations in Haskell

Having incomplete structures in the memory inherently introduces a lot of tension with both the garbage collector and compiler. Indeed, the GC assumes that every heap object it traverses is well-formed, whereas incomplete structures are absolutely ill-formed: they contain uninitialized pointers, which the GC should absolutely not follow.

The tension with the compiler is of lesser extent. The compiler can make some optimizations because it assumes that every object is immutable, while DPS programming breaks that guarantee by mutating constructors after they have been allocated (albeit only one update can happen). Fortunately, these errors are easily detected when implementing the API, and fixed by asking GHC not to inline specific parts of the code (with pragmas).

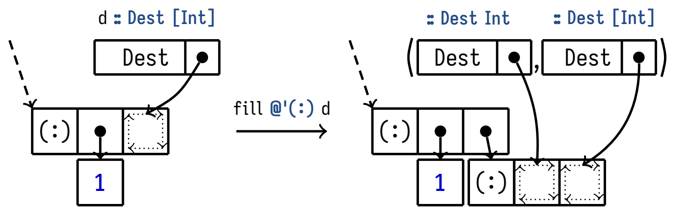


Figure 3.5: Memory behavior of fill `@'(:) :: Dest [a] ⊸ (Dest a, Dest [a])`

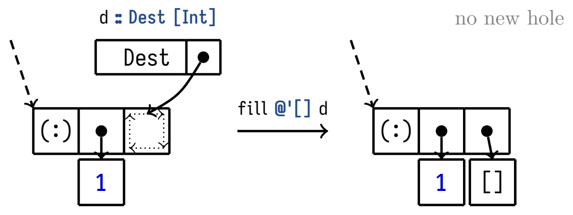


Figure 3.6: Memory behavior of fill `@'[] :: Dest [a] ⊸ ()`

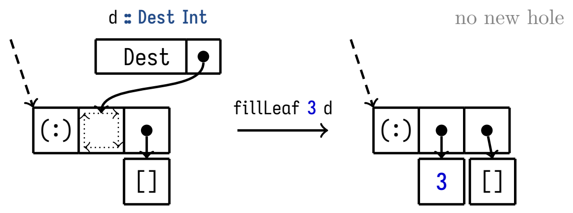


Figure 3.7: Memory behavior of `fillLeaf :: a → Dest [a] ⊸ ()`

3.5.1 Compact Regions

As I teased in Section 3.3.3, *compact regions* from [Yan+15] make it very convenient to implement DPS programming in Haskell. A compact region represents a memory area in the Haskell heap that is almost fully independent from the GC and the rest of the garbage-collected heap. For the GC, each compact region is seen as a single heap object with a single lifetime. The GC can efficiently check whether there is at least one pointer in the garbage-collected heap that points into the region, and while this is the case, the region is kept alive. When this condition is no longer matched, the whole region is discarded. The result is that the GC won't traverse any node from the region: it is treated as one opaque block (even though it is actually implemented as a chain of blocks of the same size, that doesn't change the principle). Also, compact regions are immobile in memory; the GC won't move them, so a destination can just be implemented as a raw pointer (type `Addr#` in Haskell): `data Dest r a = Dest Addr#`

By using compact regions to implement DPS programming, we completely elude the concerns of tension between the garbage collector and incomplete structures we want to build. Instead, we get two extra restrictions. First, every structure in a region must be in a fully-evaluated form. Regions are strict, and a heap object that is copied to a region is first forced into normal form. This might not always be a win; sometimes laziness, which is the default *modus operandi* of the garbage-collected heap, might be preferable.

```
type Region r ∷ Constraint
withRegion ∷ ∀ a. (∀ r. Region r ⇒ Token ⊸ Ur a) ⊸ Ur a

data Incomplete r a b
fmap            ∷ ∀ r a b c. (b ⊸ c) ⊸ Incomplete r a b ⊸ Incomplete r b c
alloc          ∷ ∀ r a.    Region r ⇒ Token ⊸ Incomplete r a (Dest r a)
intoIncomplete  ∷ ∀ r a.    Region r ⇒ Token ⊸ a → Incomplete r a ()
fromIncomplete_ ∷ ∀ r a.    Region r ⇒ Incomplete r a () ⊸ Ur a
fromIncomplete  ∷ ∀ r a b.  Region r ⇒ Incomplete r a (Ur c) ⊸ Ur (a, c)

data Dest r a
type family DestsOf lCtor r a
fill    ∷ ∀ lCtor r a. Region r ⇒ Dest r a ⊸ DestsOf lCtor r a
fillComp ∷ ∀ r a b.     Region r ⇒ Incomplete r a b ⊸ Dest r a ⊸ b
fillLeaf ∷ ∀ r a.       Region r ⇒ a → Dest r a ⊸ ()
```

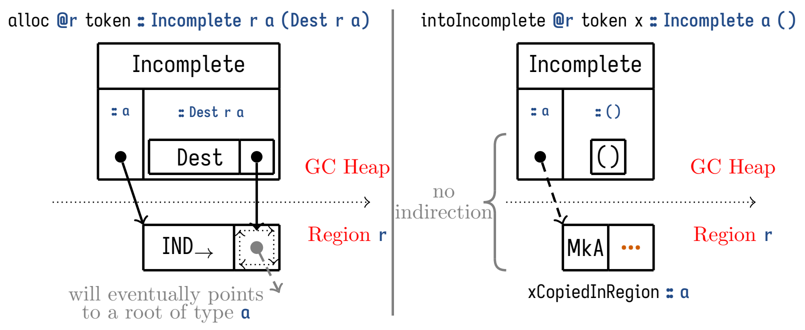

alloc @r token ∷ Incomplete r a (Dest r a)
Incomplete
∷ a
∷ Dest r a
Dest
GC Heap
IND→
Region r
will eventually points to a root of type a
no indirection
intoIncomplete @r token x ∷ Incomplete a ()
Incomplete
∷ a
∷ ()
()
GC Heap
MkA
···
Region r
xCopiedInRegion ∷ a

Figure 3.8: Memory behaviour of alloc and intoIncomplete in the region implementation

Secondly, data in a region cannot contain pointers to the garbage-collected heap, or pointers to other regions: it must be self-contained. That forces us to slightly modify the API, to add a phantom type parameter r which tags each object with the identifier of the region it belongs to. There are two related consequences: fillLeaf has to copy each *leaf* value from the garbage-collected heap into the region in which it will be used as a leaf; and fillComp can only plug together two Incompletes that come from the same region.

A typeclass Region r is also needed to carry around the details about a region that are required for the implementation. This typeclass has a single method reflect, not available to the user, that returns the RegionInfo structure associated to identifier r.

The withRegion function is the new addition to the modified API presented in Listing 6 (the Token type and its associated functions dup2 and consume are unchanged). withRegion is mostly a refinement over the withToken function from Listing 5. It receives a continuation in which r must be a free type variable. It then spawns both a new compact region and a fresh type r̲ (not a variable), and uses the reflection library to provide an instance of Region r̲ on-the-fly that links r̲ and the RegionInfo for the new region, and calls the continuation at type r̲. This is fairly standard practice since [LP94].

3.5.2 Representation of `Incomplete` objects

Ideally, as we detailed in the API, we want `Incomplete r a b` to contains an `a` and a `b`, and let the `a` free when the `b` is fully consumed (or linearly transformed into `Ur c`). So the most straightforward implementation for `Incomplete` would be a pair `(a, b)`, where `a` in the pair is only partially complete.

It is also natural for `alloc` to return an `Incomplete r a (Dest a)`: there is nothing more here than an empty memory cell (named *root receiver*) of type `a` which the associated destination of type `Dest a` points to, as presented in Figure 3.1. A bit like the identity function, whatever goes in the hole is exactly what will be retrieved in the `a` side.

If `Incomplete r a b` is represented by a pair `(a, b)`, then the root receiver should be the first field of the pair. However, the root receiver must be in the region, otherwise the GC might follow the garbage pointer that lives inside; whereas the `Incomplete` wrapper must be in the garbage-collected heap so that it can sometimes be optimized away by the compiler, and always deallocated as soon as possible.

One potential solution is to represent `Incomplete r a b` by a pair `(Ur a, b)` where `Ur` is allocated inside the region and its field `a` serves as the root receiver. With this approach, the issue of `alloc` representation is solved, but every `Incomplete` will now allocate a few words in the region (to host the `Ur` constructor) that won't be collected by the GC for a long time even if the parent `Incomplete` is collected. This makes `intoIncomplete` quite inefficient memory-wise too, as the `Ur` wrapper is useless for already complete structures.

The desired outcome is to only allocate a root receiver in the region for actual incomplete structures, and skip that allocation for already complete structures that are turned into an `Incomplete` object, while preserving a same type for both use-cases. This is made possible by replacing the `Ur` wrapper inside the `Incomplete` by an indirection object (`stg_IND` label) for the actually-incomplete case. `Incomplete r a b` will be represented by a pair `(a, b)` allocated in the garbage-collected heap, with slight variations as illustrated in Figure 3.8:

- in the pair `(a, b)` returned by `alloc`, the `a` side points to an indirection object (a sort of constructor with one field, whose resulting type `a` is the same as the field type `a`), that is allocated in the region, and serves as the root receiver;
- in the pair `(a, b)` returned by `intoIncomplete`, the `a` side directly points to the object of type `a` that has been copied to the region.

The implementation of `fromIncomplete_` is then relatively straightforward. It allocates a hollow `Ur` ☐ in the region, writes the address of the complete structure into it, and returns the `Ur` (an alternative would have been to use a regular `Ur` allocated in the GC heap).

3.5.3 Deriving `fill` for all constructors with `Generics`

The `fill @lCtor @r @a` function should plug a new hollow constructor `Ctor` ☐ `:: a` into the hole of an existing incomplete structure, and return one destination object per new hole in the structure (corresponding to the unspecified fields of the new hollow constructor). Naively, we would need one `fill` function per constructor, but that cannot be realistically implemented. Instead, we have to generalize all `fill` functions into a typeclass `Fill lCtor a`, and derive an instance of the typeclass (i.e. implement `fill`) generically for any constructor, based only on statically-known information about that constructor.

In Section 3.5.4, we will see how to allocate a hollow heap object for a specified constructor (which is known at compile-time). The only other information we need to implement `fill` generically is the shape of the constructor, and more precisely the number and type of its fields. So we will leverage `GHC.Generics` to find the required information.

`GHC.Generics` is a built-in Haskell library that provides compile-time inspection of a type metadata through the `Generic` typeclass: list of constructors, their fields, memory representation, etc. And that typeclass can be derived automatically for any type! Here's, for example, the `Generic` representation of `Maybe a`:

```
repl> :k! Rep (Maybe a) () -- display the Generic representation of Maybe a
M1 D (MetaData "Maybe" "GHC.Maybe" "base" False) (
  M1 C (MetaCons "Nothing" PrefixI False) U1
  :+: M1 C (MetaCons "Just" PrefixI False) (M1 S [...] (K1 R a)))
```

We see that there are two different constructors (indicated by `M1 C ...` lines): `Nothing` has zero fields (indicated by `U1`) and `Just` has one field of type `a` (indicated by `K1 R a`).

With a bit of type-level programming[12], we can extract the parts of that representation which are related to the constructor `lCtor` and use them inside the instance head of `Fill lCtor a` so the implementation of `fill` can depend on them. That's how we can give the proper types to the destinations returned by that function for a specified constructor. The `DestsOf lCtor a :: Type` type family also uses the generic representation of `a` to extract what it needs to know about `lCtor` and its fields.

3.5.4 Changes to GHC internals and RTS

We will see here how to allocate a hollow heap object for a given constructor, but let's first take a detour to give more context about the internals of the compiler.

Haskell's runtime system (RTS) is written in a mix of C and C–. The RTS has many roles, among which managing threads, organizing garbage collection or managing compact regions. It also defines various primitive operations, named *external primops*, that expose the RTS capabilities as normal functions. Despite all its responsibilities, however, the RTS is not responsible for the allocation of normal constructors (built in the garbage-collected heap). One reason is that it doesn't have all the information needed to build a constructor heap object, namely, the info table associated to the constructor.

The info table is what defines both the layout and behavior of a heap object. All heap objects representing a same constructor (let's say `Just`) have the same info table, even when the associated types are different (e.g. `Maybe Int` and `Maybe Bool`). Heap objects representing this constructor point to a label `<ctor>_con_info` that will be later resolved by the linker into an actual pointer to the shared info table.

The RTS is in fact a static piece of code that is compiled once when GHC is built. So the RTS has no direct way to access the information emitted during the compilation of a program. In other words, when the RTS runs, it has no way to inspect the program that it runs and info table labels have long been replaced by actual pointers so it cannot find them itself. But it is the one which knows how to allocate space inside a compact region.

As a result, I need to add two new primitives to GHC to allocate a hollow constructor:

[12] see `src/Compact/Pure/Internal.hs:418` in [Bag23b]

- one *external primop* to allocate space inside a compact region for a hollow constructor. This primop has to be implemented inside the RTS for the aforementioned reasons;
- one *internal primop* (internal primops are macros which generates C– code) that will be resolved into a normal albeit static value representing the info table pointer of a given constructor. This value will be passed as an argument to the external primop.

All the alterations to GHC that will be showed here are available in full form in [Bag23a].

External primop: allocate a hollow constructor in a region The implementation of the external primop is presented in Listing 7. The `stg_compactAddHollowzh` function (whose equivalent on the Haskell side is `compactAddHollow#`) is mostly a glorified call to the `ALLOCATE` macro defined in the `Compact.cmm` file, which tries to do a pointer-bumping allocation in the current block of the compact region if there is enough space, and otherwise add a new block to the region.

As announced, this primop takes the info table pointer of the constructor to allocate as its second parameter (`W_ info`) because it cannot access that information itself. The info table pointer is then written to the first word of the heap object in the call to `SET_HDR`.

Internal primop: reify an info table label into a runtime value The only way, in Haskell, to pass a constructor to a primop so that the primop can inspect it, is to lift the constructor into a type-level literal. It's common practice to use a `Proxy a` (the unit type with a phantom type parameter) to pass the type `a` as an input to a function. Unfortunately, due to a quirk of the compiler, primops don't have access to the type of their arguments. They can, however, access their return type. So I'm using a phantom type `InfoPtrPlaceholder# a` as the return type, to pass the contructor as an input!

The gist of this implementation is presented in Listing 8. The primop `reifyInfoPtr#` pattern-matches on the type `resTy` of its return value. In the case it reads a string literal, it resolves the primop call into the label `stg_<name>` (this is used in particular to retrieve `stg_IND` to allocate indirection heap objects). In the case it reads a lifted data constructor, it resolves the primop call into the label which corresponds to the info table pointer of that constructor. The returned `InfoPtrPlaceholder# a` can later be converted back to an `Addr#` using the `unsafeCoerceAddr` function.

As an example, here is how to allocate a hollow `Just` constructor in a compact region:

```
hollowJust ∷ Maybe a = compactAddHollow#
  compactRegion#
  (unsafeCoerceAddr (reifyInfoPtr# (# #) ∷ InfoPtrPlaceholder# 'Just ))
```

Built-in type family to go from a lifted constructor to the associated symbol The internal primop `reifyInfoPtr#` that we introduced above takes as input a constructor lifted into a type-level literal, so this is also what `fill` will use to know which constructor it should operate with. But `DestsOf` have to find the metadata of a constructor in the `Generic` representation of a type, in which only the constructor name appears.

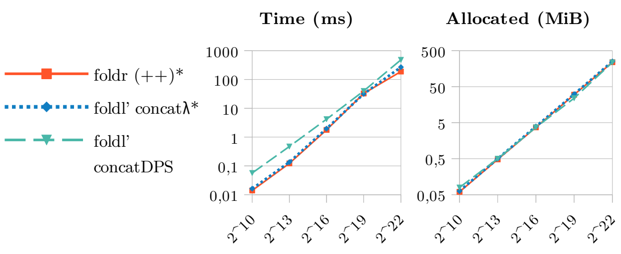
So we added a new type family `LCtorToSymbol` inside GHC that inspects its (type-level) parameter representing a constructor, fetches its associated `DataCon` structure, and returns a type-level string (kind `Symbol`) carrying the constructor name, as presented in Listing 9.

```
// compactAddHollow#
//   :: Compact# → Addr# → State# RealWorld → (# State# RealWorld, a #)
stg_compactAddHollowzh(P_ compact, W_ info) {
   W_ pp, ptrs, nptrs, size, tag, hp;
   P_ to, p; p = NULL;  // p isn't actually used by ALLOCATE macro
   again: MAYBE_GC(again); STK_CHK_GEN();

   pp = compact + SIZEOF_StgHeader + OFFSET_StgCompactNFData_result;
   ptrs  = TO_W_(%INFO_PTRS(%STD_INFO(info)));
   nptrs  = TO_W_(%INFO_NPTRS(%STD_INFO(info)));
   size = BYTES_TO_WDS(SIZEOF_StgHeader) + ptrs + nptrs;

   ALLOCATE(compact, size, p, to, tag);
   P_[pp] = to;
   SET_HDR(to, info, CCS_SYSTEM);
  #if defined(DEBUG)
   ccall verifyCompact(compact);
  #endif
   return (P_[pp]);
}
```

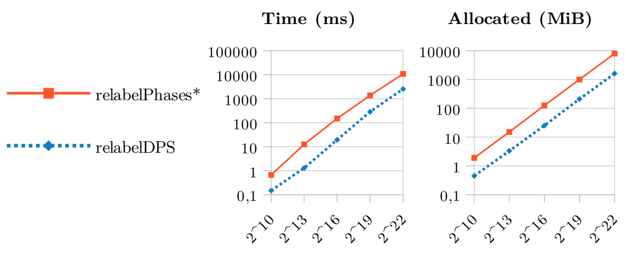
Listing 7: `compactAddHollow#` implementation in `rts/Compact.cmm`

```
case primop of
  [...]
  ReifyStgInfoPtrOp → \_ →  -- we don't care about the function argument (# #)
    opIntoRegsTy $ \[res] resTy → emitAssign (CmmLocal res) $ case resTy of
      -- when 'a' is a Symbol, and extracts the symbol value in 'sym'
      TyConApp _addrLikeTyCon [_typeParamKind, LitTy (StrTyLit sym)] →
        CmmLit (CmmLabel (
          mkCmmInfoLabel rtsUnitId (fsLit "stg_" `appendFS` sym)))
      -- when 'a' is a lifted data constructor, extracts it as a DataCon
      TyConApp _addrLikeTyCon [_typeParamKind, TyConApp tyCon _]
        | Just dataCon ← isPromotedDataCon_maybe tyCon →
        CmmLit (CmmLabel (
          mkConInfoTableLabel (dataConName dataCon) DefinitionSite))
      _ → [...] -- error when no pattern matches
```

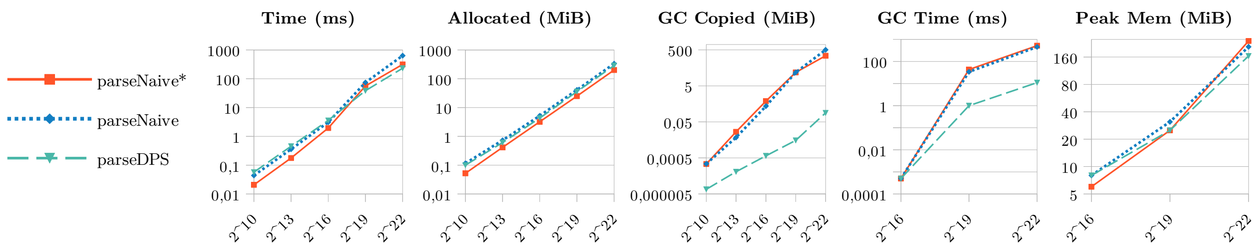
Listing 8: `reifyInfoPtr#` implementation in `compiler/GHC/StgToCmm/Prim.hs`



A. Benchmark of lists & difference lists concatenation



B. Benchmark of breadth-first tree relabeling



	Time (ms)					**Allocated (MiB)**					**GC Copied (MiB)**					**GC Time (ms)**			**Peak Mem (MiB)**		
Size	2^{10}	2^{13}	2^{16}	2^{19}	2^{22}	2^{10}	2^{13}	2^{16}	2^{19}	2^{22}	2^{10}	2^{13}	2^{16}	2^{19}	2^{22}	2^{16}	2^{19}	2^{22}	2^{16}	2^{19}	2^{22}
parseNaive*	0.0	0.2	1.9	56.0	315.0	0.1	0.4	3.2	25.0	201.0	0.0	0.0	0.7	28.0	235.0	0.0	43.0	517.0	6.0	25.0	240.0
parseNaive	0.0	0.4	3.1	74.0	633.0	0.1	0.7	5.2	41.0	334.0	0.0	0.0	0.4	28.0	503.0	0.0	34.0	455.0	8.0	31.0	207.0
parseDPS	0.1	0.5	3.6	**38.4**	**233.0**	0.1	0.6	4.3	36.0	289.0	**0.0**	**0.0**	**0.0**	**0.0**	**0.2**	**0.0**	**1.0**	**11.0**	8.0	**25.0**	**164.0**

C. Benchmark of S-expressions parser

Figure 3.9: Benchmarks performed on AMD EPYC 7401P @ 2.0 GHz (single core, `-N1 -O2`)

3.6 Evaluating the performance of DPS programming

Benchmarking methodology All over this article, I talked about programs in both naive style and DPS style. With DPS programs, the result is stored in a compact region, which also forces strictness i.e. the structure is automatically in fully evaluated form.

For naive versions, we have a choice to make on how to fully evaluate the result: either force each chunk of the result inside the GC heap (using `Control.DeepSeq.force`), or copy the result in a compact region that is strict by default (using `Data.Compact.compact`).

In programs where there is no particular long-lived piece of data, having the result of the function copied into a compact region isn't particularly desirable since it will generally inflate memory allocations. So we use `force` to benchmark the naive version of those programs (the associated benchmark names are denoted with a "*" suffix).

Concatenating lists and difference lists We compared three implementations.
`foldr (++)`* has calls to `(++)` nested to the right, giving the most optimal context for list concatenation (it should run in $\mathcal{O}(n)$ time). `foldl' concat`λ* uses function-backed difference lists, and `foldl' concatDPS` uses destination-backed ones, so both should run in $\mathcal{O}(n)$ even if calls to concat are nested to the left.

We see in part **A** of Figure 3.9 that the destination-backed difference lists have a comparable memory use as the two other linear implementations, while being quite slower (by a factor 2-4) on all datasets. We would expect better results though for a DPS implementation outside of compact regions because those cause extra copying.

Breadth-first relabeling We see in part **B** of Figure 3.9 that the destination-based tree traversal is almost one order of magnitude more efficient, both time-wise and memory-wise, compared to the implementation based on *Phases* applicatives presented in [Gib+23].

Parsing S-expressions In part **C** of Figure 3.9, we compare the naive implementation of the S-expression parser and the DPS one (see Section 3.3.3). For this particular program, where using compact regions might reduce the future GC load of the application, it is relevant to benchmark the naive version twice: once with `force` and once with `compact`.

The DPS version starts by being less efficient than the naive versions for small inputs, but gets an edge as soon as garbage collection kicks in (on datasets of size $\leq 2^{16}$, no garbage collection cycle is required as the heap size stays small).

On the largest dataset ($2^{22} \simeq$ 4MiB file), the DPS version still makes about 45% more allocations than the starred naive version, but uses 35% less memory at its peak, and more importantly, spends 47× less time in garbage collection. As a result, the DPS version only takes 0.55-0.65× the time spent by the naive versions, thanks to garbage collection savings. All of this also indicates that most of the data allocated in the GC heap by the DPS version just lasts one generation and thus can be discarded very early by the GC, without needing to be copied into the next generation, unlike most nodes allocated by the naive versions.

Finally, copying the result of the naive version to a compact region (for future GC savings) incurs a significant time and memory penalty, that the DPS version offers to avoid.

3.7 Related work

The idea of functional data structures with write-once holes is not new. Minamide already proposed in [Min98] a variant of λ-calculus with support for *hole abstractions*, which can be represented in memory by an incomplete structure with one hole and can be composed efficiently with each other (as with `fillComp` in Figure 3.4). With such a framework, it is fully possible to implement destination-backed difference lists for example.

However, in Minamide's work, there is no concept of destination: the hole in a structure can only be filled if one has the structure itself at hand. On the other hand, our approach introduces destinations, as a way to interact with a hole remotely, even when one doesn't have a handle to the associated structure. Because destinations are first-class objects, they can be passed around or stored in collections or other structure, while preserving memory safety. This is the major step forward that our paper presents.

More recently, [PP13] introduced the Mezzo programming language, in which mutable data structures can be freezed into immutable ones after having been completed. This principle is used to some extend in their list standard library module, to mimic a form of DPS programming. An earlier appearance of DPS programming as a mean to achieve better performance in a mutable language can also be seen in [Lar89].

Finally, both [Sha+17] and [BCS21] use DPS programming to make list or array processing algorithms more efficient in a functional, immutable context, by turning non tail-recursive functions into tail-recursive DPS ones. More importantly, they present an automatized way to go from a naive program to its tail-recursive version. However, holes/destinations are only supported at an intermediary language level, while both [Min98] and our present work support safe DPS programming in user-land. In a broader context, [LLS23] presents a system in which linearity is used to identify where destructive updates can be made, so as to reuse the same constructor instead of deallocating and reallocating one; but this optimization technique is still mostly invisible for the user, unlike ours which is made explicit.

3.8 Conclusion and future work

Programming with destinations definitely has a place in the realm of functional programming, as the recent adoption of *Tail Modulo Cons* [BCS21] in the OCaml compiler shows. In this paper, we have shown how destination-passing style programming can be used in user-land in Haskell safely, thanks to a linear type discipline. Adopting DPS programming opens the way for more natural and efficient programs in a variety of contexts, where the major points are being able to build structures in a top-down fashion, manipulating and composing incomplete structures, and managing holes in these structures through first-class objects (destinations). Our DPS implementation relies only on a few alterations to the compiler, thanks to *compact regions* that are already available as part of GHC. Simultaneously, it allows to build structures in those regions without copying, which wasn't possible before.

There are two limitations that we would like to lift in the future. First, DPS programming could be useful outside of compact regions: destinations could probably be used to manipulate the garbage-collected heap (with proper read barriers in place), or other forms of secluded memory areas that aren't traveled by the GC (RDMA, network serialized buffers, etc.). Secondly, at the moment, the type of `fillLeaf` implies that we can't store destinations (which are always linear) in a difference list implemented as in Section 3.3.1, whereas we can store them in a regular list or queue (like we do, for instance, in Section 3.3.2). This unwelcome restriction ensures memory safety but it's quite coarse grain. In the future we'll be trying to have a more fine-grained approach that would still ensure safety.

```haskell
matchFamLCtorToSymbol :: [Type] -> Maybe (CoAxiomRule, [Type], Type)
matchFamLCtorToSymbol [kind, ty]
  | TyConApp tyCon _ <- ty, Just dataCon <- isPromotedDataCon_maybe tyCon =
      let symbolLit = (mkStrLitTy . occNameFS . occName . getName $ dataCon)
       in Just (axLCtorToSymbolDef, [kind, ty], symbolLit)
matchFamLCtorToSymbol tys = Nothing

axLCtorToSymbolDef =
  mkBinAxiom "LCtorToSymbolDef" typeLCtorToSymbolTyCon Just
    (\case { TyConApp tyCon _ -> isPromotedDataCon_maybe tyCon ; _ -> Nothing })
    (\_ dataCon -> Just (mkStrLitTy . occNameFS . occName . getName $ dataCon))
```

Listing 9: `LCtorToSymbol` implementation in `compiler/GHC/Builtin/Types/Literal.hs`

Chapter 4

Extending DPS support for linear data structures

TBD.

Chapter 5

Related work

5.1 Destination-passing style for efficient memory management

Shaikhha et al. [Sha+17] present a destination-based intermediate language for a functional array programming language, with destination-specific optimizations, that boasts near-C performance.

This is the most comprehensive evidence to date of the benefits of destination-passing style for performance in functional languages, although their work is on array programming, while this article focuses on linked data structures. They can therefore benefit from optimizations that are perhaps less valuable for us, such as allocating one contiguous memory chunk for several arrays.

The main difference between their work and ours is that their language is solely an intermediate language: it would be unsound to program in it manually. We, on the other hand, are proposing a type system to make it sound for the programmer to program directly with destinations.

We see these two aspects as complementing each other: good compiler optimizations are important to alleviate the burden from the programmer and allow high-level abstraction; having the possibility to use destinations in code affords the programmer more control, should they need it.

5.2 Tail modulo constructor

Another example of destinations in a compiler's optimizer is [BCS21]. It's meant to address the perennial problem that the map function on linked lists isn't tail-recursive, hence consumes stack space. The observation is that there's a systematic transformation of functions where the only recursive call is under a constructor to a destination-passing tail-recursive implementation.

Here again, there's no destination in user land, only in the intermediate representation. However, there is a programmatic interface: the programmer annotates a function like

```
let[@tail_mod_cons] rec map =
```

to ask the compiler to perform the translation. The compiler will then throw an error if it can't. This way, contrary to the optimizations in [Sha+17], it is entirely predictable.

This has been available in OCaml since version 4.14. This is the one example we know of of destinations built in a production-grade compiler. Our λ_d makes it possible to express the result tail-modulo-constructor in a typed language. It can be used to write programs directly in that style, or it could serve as a typed target language for an automatic transformation. On the flip-side, tail modulo constructor is too weak to handle our difference lists or breadth-first traversal examples.

5.3 A functional representation of data structures with a hole

The idea of using linear types as a foundation of a functional calculus in which incomplete data structures can exist and be composed as first class values dates back to [Min98]. Our system is strongly inspired by theirs. In [Min98], a first-class structure with a hole is called a *hole abstraction*. Hole abstractions are represented by a special kind of linear functions with bespoke restrictions. As with any function, we can't pattern-match on their output (or pass it to another function) until they have been applied; but they also have the restriction that we cannot pattern-match on their argument —the *hole variable*— as that one can only be used directly as argument of data constructors, or of other hole abstractions. The type of hole abstractions, $(\mathsf{T},\mathsf{S})\mathtt{hfun}$ is thus a weak form of linear function type $\mathsf{T} \multimap \mathsf{S}$.

In [Min98], it's only ever possible to represent structures with a single hole. But this is a rather superficial restriction. The author doesn't comment on this, but we believe that this restriction only exists for convenience of the exposition: the language is lowered to a language without function abstraction and where composition is performed by combinators. While it's easy to write a combinator for single-argument-function composition, it's cumbersome to write combinators for functions with multiple arguments. But having multiple-hole data structures wouldn't have changed their system in any profound way.

The more important difference is that while their system is based on a type of linear functions, ours is based on the linear logic's "par" type. In classical linear logic, linear implication $\mathsf{T} \multimap \mathsf{S}$ is reinterpreted as $\mathsf{S} \parr \mathsf{T}^{\perp}$. We, likewise, reinterpret $(\mathsf{T},\mathsf{S})\mathtt{hfun}$ as $\mathsf{S} \ltimes \lfloor \mathsf{T} \rfloor$ (a sort of weak "par").

A key consequence is that destinations —as first-class representations of holes— appear naturally in λ_d, while [Min98] doesn't have them. This means that using [Min98], or the more recent but similarly expressive system from [Lor+24b], one can implement the examples with difference lists and queues from Section 2.2.3, but couldn't do our breadth-first traversal example from Section 2.4, since it requires to be able to store destinations in a structure.

Nevertheless, we still retain the main restrictions that Minamide [Min98] places on hole abstractions. For instance, we can't pattern-match on S in (unapplied) $(\mathsf{T},\mathsf{S})\mathtt{hfun}$; so in λ_d, we can't act directly on the left-hand side S of $\mathsf{S} \ltimes \mathsf{T}$, only on the right-hand side T. Similarly, hole variables can only be used as arguments of constructors or hole abstractions; it's reflected in λ_d by the fact that the only way to act on destinations is via fill operations, with either hollow constructors or another ampar.

The ability to manipulate destinations, and in particular, store them, does come at a cost though: the system needs this additional notion of ages to ensure that destinations are used soundly. On the other hand, our system is strictly more general, in Minamide [Min98]'s system can be embedded in λ_d, and if one stays in this fragment, we're never confronted with ages.

5.4 Destination-passing style programming: a Haskell implementation

Bagrel [Bag24] proposes a system much like ours: it has a destination type, and a *par*-like construct (that they call `Incomplete`), where only the right-hand side can be modified; together these elements give extra expressiveness to the language compared to [Min98].

In their system, $d \blacktriangleleft t$ requires t to be unrestricted, while in λ_d, t can be linear. The consequence is that in [Bag24], destinations can be stored in data structures but not in data structures with holes; so in a breadth-first search algorithm like in Section 2.4, they have to build the queue using normal constructors, and cannot use destination-filling primitives. Therefore both normal constructors and DPS primitives must coexist in their work, while in λ_d, only DPS primitives are required to bootstrap the system, as we later derive normal constructors from them. In exchange, they don't need a system of ages to make their system safe; just linearity is enough.

A more profound difference between their work and ours is that they describe a practical implementation of destination-passing style for an existing functional language, while we present a slightly more general theoretical framework that is meant to justify safety of DPS implementations (such as [Bag24] itself), so goals are quite different.

5.5 Semi-axiomatic sequent calculus

In [DPP20] constructors return to a destination rather than allocating memory. It is very unlike the other systems described in this section in that it's completely founded in the Curry-Howard isomorphism. Specifically it gives an interpretation of a sequent calculus which mixes Gentzen-style deduction rules and Hilbert-style axioms. As a consequence, the *par* connective is completely symmetric, and, unlike our $\lfloor \mathtt{T} \rfloor$ type, their dualization connective is involutive.

The cost of this elegance is that computations may try to pattern-match on a hole, in which case they must wait for the hole to be filled. So the semantics of holes is that of a future or a promise. In turns this requires the semantics of their calculus to be fully concurrent, which is a very different point in the design space.

5.6 Rust lifetimes

Rust uses a system of lifetimes (see e.g. [**pearce_lifetime_2021**]) to ensure that borrows don't live longer than what they reference. It plays a similar role as our system of ages.

Rust lifetimes are symbolic. Borrows and moves generate constraints (inequalities of the form $\alpha \leqslant \beta$) on the symbolic lifetimes. For instance, that a the lifetime of a reference is larger than the lifetime of any structure the reference is stored in. Without such constraints, Rust would have similar problems to those of Section 2.3. The borrow checker then checks that the constraints are solvable. This contrasts with λ_d where ages are set explicitly, with no analysis needed.

Another difference between the two systems is that λ_d's ages (and modes in general) are relative. An explicit modality $!_{1\uparrow^k}$ must be used when a part has an age different than its parent, and means that the part is k scope older than the parent. On the other hand, Rust's lifetimes are absolute, the lifetime of a part is tracked independently of the lifetime of its parent.

5.7 Oxidizing OCaml

Lorenzen et al. [Lor+24a] present an extension of the OCaml type system to support modes. Their modes are split along three different "axes", among which affinity and locality are comparable to our multiplicities and ages. Like our multiplicities, there are two modes for affinity `once` and `many`, though in [Lor+24a], `once` supports weakening, whereas λ_d's 1 multiplicity is properly linear (proper linearity matters for destination lest we end up reading uninitialized memory).

Locality tracks scope. There are two locality modes, `local` (doesn't escape the current scope) and `global` (can escape the current scope). The authors present their locality mode as a drastic simplification of Rust's lifetime system, which nevertheless fits their need.

However, such a simplified system would be a bit too weak to track the scope of destinations. The observation is that if destinations from two nested scopes are given the same mode, then we can't safely do anything with them, as it would be enough to reproduce the counterexamples of Section 2.3. So in order to type the breadth-first traversal example of Section 2.4, where destinations are stored in a structure, we need at least ν (for the current scope), $\uparrow$ (for the previous scope exactly), plus at least one extra mode for the rest of the scopes (destinations of this generic age cannot be safely used). It turns out that such systems with finitely many ages are incredibly easy to get wrong, and it was in fact much simpler to design a system with infinitely many ages.

Chapter 6

Conclusion

TBD.

Bibliography

[HM81] Robert Hood and Robert Melville. “Real-time queue operations in pure LISP”. In: *Information Processing Letters* 13.2 (1981), pp. 50–54. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(81)90030-2. URL: https://www.sciencedirect.com/science/article/pii/0020019081900302.

[Hug86] John Hughes. “A Novel Representation of Lists and its Application to the Function ”reverse”.” In: *Inf. Process. Lett.* 22 (Jan. 1986), pp. 141–144.

[Fel87] Matthias Felleisen. “The calculi of lambda-nu-cs conversion: a syntactic theory of control and state in imperative higher-order programming languages”. AAI8727494. phd. USA: Indiana University, 1987. URL: https://www2.ccs.neu.edu/racket/pubs/dissertation-felleisen.pdf.

[Lar89] James Richard Larus. “Restructuring symbolic programs for concurrent execution on multiprocessors”. AAI9006407. phd. University of California, Berkeley, 1989.

[AND92] JEAN-MARC ANDREOLI. “Logic Programming with Focusing Proofs in Linear Logic”. In: *Journal of Logic and Computation* 2.3 (June 1992), pp. 297–347. ISSN: 0955-792X. DOI: 10.1093/logcom/2.3.297. URL: https://doi.org/10.1093/logcom/2.3.297 (visited on 12/05/2024).

[GSS92] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. “Bounded linear logic: a modular approach to polynomial-time computability”. In: *Theoretical Computer Science* 97.1 (Apr. 1992), pp. 1–66. ISSN: 0304-3975. DOI: 10.1016/0304-3975(92)90386-T. URL: https://www.sciencedirect.com/science/article/pii/030439759290386T (visited on 12/05/2024).

[Gib93] Jeremy Gibbons. “Linear-time Breadth-first Tree Algorithms: An Exercise in the Arithmetic of Folds and Zips”. en-gb. In: No. 71 (1993). Number: No. 71. URL: https://www.cs.ox.ac.uk/publications/publication2363-abstract.html (visited on 10/18/2023).

[Bie94] G.M. Bierman. *On intuitionistic linear logic.* Tech. rep. UCAM-CL-TR-346. University of Cambridge, Computer Laboratory, Aug. 1994. DOI: 10.48456/tr-346. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-346.pdf.

[LP94] John Launchbury and Simon L. Peyton Jones. “Lazy functional state threads”. In: *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation.* PLDI ’94. New York, NY, USA: Association for Computing Machinery, June 1994, pp. 24–35. ISBN: 978-0-89791-662-2. DOI: 10.1145/178243.178246. URL: https://dl.acm.org/doi/10.1145/178243.178246 (visited on 12/11/2023).

[Gir95] J.-Y. Girard. “Linear Logic: its syntax and semantics”. en. In: *Advances in Linear Logic*. Ed. by Jean-Yves Girard, Yves Lafont, and Laurent Regnier. Cambridge: Cambridge University Press, 1995, pp. 1–42. ISBN: 978-0-511-62915-0. DOI: `10.1017/CBO9780511629150.002`. URL: `https://www.cambridge.org/core/product/identifier/CBO9780511629150A008/type/book_part` (visited on 03/21/2022).

[Min98] Yasuhiko Minamide. “A functional representation of data structures with a hole”. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’98. New York, NY, USA: Association for Computing Machinery, Jan. 1998, pp. 75–84. ISBN: 978-0-89791-979-1. DOI: `10.1145/268946.268953`. URL: `https://doi.org/10.1145/268946.268953` (visited on 03/15/2022).

[CH00] Pierre-Louis Curien and Hugo Herbelin. “The duality of computation”. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP ’00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 233–243. ISBN: 1581132026. DOI: `10.1145/351240.351262`. URL: `https://doi.org/10.1145/351240.351262`.

[Oka00] Chris Okasaki. “Breadth-first numbering: lessons from a small exercise in algorithm design”. In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. ICFP ’00. New York, NY, USA: Association for Computing Machinery, Sept. 2000, pp. 131–136. ISBN: 978-1-58113-202-1. DOI: `10.1145/351240.351253`. URL: `https://dl.acm.org/doi/10.1145/351240.351253` (visited on 10/12/2023).

[DN04] Olivier Danvy and Lasse R. Nielsen. “Refocusing in Reduction Semantics”. en. In: *BRICS Report Series* 11.26 (Nov. 2004). ISSN: 1601-5355, 0909-0878. DOI: `10.7146/brics.v11i26.21851`. URL: `https://tidsskrift.dk/brics/article/view/21851` (visited on 12/17/2024).

[BD07] Małgorzata Biernacka and Olivier Danvy. “A syntactic correspondence between context-sensitive calculi and abstract machines”. In: *Theoretical Computer Science*. Festschrift for John C. Reynolds’s 70th birthday 375.1 (May 2007), pp. 76–108. ISSN: 0304-3975. DOI: `10.1016/j.tcs.2006.12.028`. URL: `https://www.sciencedirect.com/science/article/pii/S0304397506009170` (visited on 12/17/2024).

[Sew+07] Peter Sewell et al. “Ott: effective tool support for the working semanticist”. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’07. Freiburg, Germany: Association for Computing Machinery, 2007, pp. 1–12. ISBN: 9781595938152. DOI: `10.1145/1291151.1291155`. URL: `https://doi.org/10.1145/1291151.1291155`.

[PP13] Jonathan Protzenko and François Pottier. “Programming with Permissions in Mezzo”. In: *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. arXiv:1311.7242 [cs]. Sept. 2013, pp. 173–184. DOI: `10.1145/2500365.2500598`. URL: `http://arxiv.org/abs/1311.7242` (visited on 10/16/2023).

[GS14] Dan R. Ghica and Alex I. Smith. “Bounded Linear Types in a Resource Semiring”. en. In: *Programming Languages and Systems*. Ed. by Zhong Shao. Berlin, Heidelberg: Springer, 2014, pp. 331–350. ISBN: 978-3-642-54833-8. DOI: `10.1007/978-3-642-54833-8_18`.

[POM14] Tomas Petricek, Dominic Orchard, and Alan Mycroft. “Coeffects: a calculus of context-dependent computation”. In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. ICFP ’14. New York, NY, USA: Association for Computing Machinery, Aug. 2014, pp. 123–135. ISBN: 978-1-4503-2873-9. DOI: `10.1145/2628136.2628160`. URL: `https://dl.acm.org/doi/10.1145/2628136.2628160` (visited on 12/05/2024).

[Yan+15] Edward Z. Yang et al. “Efficient communication and collection with compact normal forms”. en. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. Vancouver BC Canada: ACM, Aug. 2015, pp. 362–374. ISBN: 978-1-4503-3669-7. DOI: `10.1145/2784731.2784735`. URL: `https://dl.acm.org/doi/10.1145/2784731.2784735` (visited on 04/04/2022).

[Sha+17] Amir Shaikhha et al. “Destination-passing style for efficient memory management”. en. In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*. Oxford UK: ACM, Sept. 2017, pp. 12–23. ISBN: 978-1-4503-5181-2. DOI: `10.1145/3122948.3122949`. URL: `https://dl.acm.org/doi/10.1145/3122948.3122949` (visited on 03/15/2022).

[Atk18] Robert Atkey. “Syntax and Semantics of Quantitative Type Theory”. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS ’18. Oxford, United Kingdom: Association for Computing Machinery, 2018, pp. 56–65. ISBN: 9781450355834. DOI: `10.1145/3209108.3209189`. URL: `https://doi.org/10.1145/3209108.3209189`.

[Ber+18] Jean-Philippe Bernardy et al. “Linear Haskell: practical linearity in a higher-order polymorphic language”. In: *Proceedings of the ACM on Programming Languages* 2.POPL (Jan. 2018). arXiv:1710.09756 [cs], pp. 1–29. ISSN: 2475-1421. DOI: `10.1145/3158093`. URL: `http://arxiv.org/abs/1710.09756` (visited on 06/23/2022).

[AB20] Andreas Abel and Jean-Philippe Bernardy. “A unified view of modalities in type systems”. In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: `10.1145/3408972`. URL: `https://doi.org/10.1145/3408972`.

[DPP20] Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. “Semi-Axiomatic Sequent Calculus”. In: *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*. Ed. by Zena M. Ariola. Vol. 167. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, 29:1–29:22. ISBN: 978-3-95977-155-9. DOI: `10.4230/LIPIcs.FSCD.2020.29`. URL: `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2020.29`.

[BCS21] Frédéric Bour, Basile Clément, and Gabriel Scherer. “Tail Modulo Cons”. In: *arXiv:2102.09823 [cs]* (Feb. 2021). arXiv: 2102.09823. URL: `http://arxiv.org/abs/2102.09823` (visited on 03/22/2022).

[Spi+22] Arnaud Spiwack et al. “Linearly qualified types: generic inference for capabilities and uniqueness”. In: *Proceedings of the ACM on Programming Languages* 6.ICFP (Aug. 2022), 95:137–95:164. DOI: `10.1145/3547626`. URL: `https://dl.acm.org/doi/10.1145/3547626` (visited on 10/16/2023).

[Bag23a] Thomas Bagrel. *GHC with support for hollow constructor allocation.* Software Heritage, `swh:1:dir:84c7e717fd5f189c6b6222e0fc92d0a82d755e7c; origin=https://github.com/tweag/ghc; visit=swh:1:snp:141fa3c28e01574deebb6cc91693c75f49717c32; anchor=swh:1:rev:184f838b352a0d546e574bdeb83c8c190e9dfdc2`. 2023. (Visited on 10/19/2023).

[Bag23b] Thomas Bagrel. *`linear-dest`, a Haskell library that adds supports for DPS programming.* Software Heritage, `swh:1:rev:0e7db2e6b24aad348837ac78d8137712c1d8d12a; origin=https://github.com/tweag/linear-dest; visit=swh:1:snp:c0eb2661963bb176204b46788f4edd26f72ac83c`. 2023. (Visited on 10/19/2023).

[Gib+23] Jeremy Gibbons et al. "Phases in Software Architecture". en. In: *Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Software Architecture.* Seattle WA USA: ACM, Aug. 2023, pp. 29–33. ISBN: 9798400702976. DOI: `10.1145/3609025.3609479`. URL: `https://dl.acm.org/doi/10.1145/3609025.3609479` (visited on 10/02/2023).

[LL23] Daan Leijen and Anton Lorenzen. "Tail Recursion Modulo Context: An Equational Approach". en. In: *Proceedings of the ACM on Programming Languages* 7.POPL (Jan. 2023), pp. 1152–1181. ISSN: 2475-1421. DOI: `10.1145/3571233`. URL: `https://dl.acm.org/doi/10.1145/3571233` (visited on 01/25/2024).

[LLS23] Anton Lorenzen, Daan Leijen, and Wouter Swierstra. "FP^2: Fully in-Place Functional Programming". en. In: *Proceedings of the ACM on Programming Languages* 7.ICFP (Aug. 2023), pp. 275–304. ISSN: 2475-1421. DOI: `10.1145/3607840`. URL: `https://dl.acm.org/doi/10.1145/3607840` (visited on 12/11/2023).

[Bag24] Thomas Bagrel. "Destination-passing style programming: a Haskell implementation". In: *35es Journées Francophones des Langages Applicatifs (JFLA 2024).* Saint-Jacut-de-la-Mer, France, Jan. 2024. URL: `https://inria.hal.science/hal-04406360`.

[Lor+24a] Anton Lorenzen et al. "Oxidizing OCaml with Modal Memory Management". In: *Proc. ACM Program. Lang.* 8.ICFP (Aug. 2024), 253:485–253:514. DOI: `10.1145/3674642`. URL: `https://dl.acm.org/doi/10.1145/3674642` (visited on 01/06/2025).

[Lor+24b] Anton Lorenzen et al. "The Functional Essence of Imperative Binary Search Trees". In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: `10.1145/3656398`. URL: `https://doi.org/10.1145/3656398`.