UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Formalization and Implementation of Safe Destination Passing in Pure Functional Programming Settings

THÈSE

présentée et soutenue publiquement le April 2025

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Thomas Bagrel

Composition du jury

Président :	Le président	
Rapporteurs :	Le rapporteur 1	de Paris
	Le rapporteur 2	
	Le rapporteur 3	
Examinateurs :	L'examinateur 1	d'ici
	L'examinateur 2	

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

Mis en page avec la classe thesul.

Résumé

La programmation par passage de destination introduit le concept de *destination*, qui représente l'adresse d'une cellule mémoire encore vierge sur laquelle on ne peut écrire qu'une fois. Ces destinations peuvent être passées en tant que paramètres de fonction, permettant à l'appelant de garder le contrôle de la gestion mémoire : la fonction appelée se contente de remplir la cellule au lieu d'allouer de l'espace pour une valeur de retour. Bien que principalement utilisé en programmation système, le passage de destination trouve aussi des applications en programmation fonctionnelle pure, où il permet d'écrire des programmes auparavant inexpressibles avec les structures de données immuables usuelles.

Dans cette thèse, nous développons un λ-calcul avec destinations, λ_d. Ce nouveau système théorique est plus expressif que les travaux similaires existants, le passage de destination y étant conçu pour être aussi flexible que possible. Cette expressivité est rendue possible par un système de types modaux combinant types linéaires et un système d'âges pour gérer le contrôle lexical des ressources, afin de garantir la sûreté du passage de destination. Nous avons prouvé la sûreté de notre système via les théorèmes habituels de progression et de préservation des types, de façon mécanisée, avec l'assistant de preuve Coq.

Nous montrons ensuite comment le passage de destination —formalisé dans ce calcul théorique— peut être intégré à un langage fonctionnel pur existant, Haskell, dont le système de types est moins puissant que notre système théorique. Préserver la sûreté nécessite alors de restreindre la flexibilité dans la gestion des destinations. Nous affinons par la suite l'implémentation pour retrouver une grande partie de cette flexibilité, au prix d'une complexité accrue pour l'utilisateur.

L'implémentation prototype en Haskell montre des résultats encourageants pour l'adoption du passage de destination pour des parcours ou du mapping de grandes structures de données, telles que les listes ou les arbres.

Abstract

Destination-passing style programming introduces destinations, which represent the address of a write-once memory cell. These destinations can be passed as function parameters, allowing the caller to control memory management: the callee simply fills the cell instead of allocating space for a return value. While typically used in systems programming, destination passing also has applications in pure functional programming, where it enables programs that were previously unexpressible using usual immutable data structures.

In this thesis, we develop a core λ-calculus with destinations, λ_d. Our new calculus is more expressive than similar existing systems, with destination passing designed to be as flexible as possible. This is achieved through a modal type system combining linear types with a system of ages to manage scopes, in order to make destination-passing safe. Type safety of our core calculus was proved formally with the Coq proof assistant.

Then, we see how this core calculus can be adapted into an existing pure functional language, Haskell, whose type system is less powerful than our custom theoretical one. Retaining safety comes at the cost of removing some flexibility in the handling of destinations. We later refine the implementation to recover much of this flexibility, at the cost of increased user complexity.

The prototype implementation in Haskell shows encouraging results for adopting destination-passing style programming when traversing or mapping over large data structures such as lists or data trees.

Remerciements

Les remerciements.

Je dédie cette thèse à TBD.

Contents

List of Figures **xi**

List of Listings **xiii**

Introduction

1 Memory management: a core design axis for programming languages xv

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

3 Functional programming languages xvii

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

5 Unifying and formalizing Functional DPS frameworks xx

6 Reconciling write pointers and immutability xx

Chapter 1 Linear λ-calculus and Linear Haskell **1**

1.1 From λ-calculus to linear λ-calculus 1

1.2 Linear λ-calculus: a computational interpretation of linear logic 3

1.3 Implicit structural rules for unrestricted resources 5

1.4 Back to a single context with the graded modal approach 7

1.5 Deep modes: projecting modes through fields of data structures 11

1.6 Semantics of linear λ-calculus (with deep modes) 13

1.7 Linear Types in Haskell 15

1.8 Uniqueness and Linear Scopes 16

1.9 Conclusion 20

Chapter 2 A formal functional language based on first-class destinations: λ_d **21**

2.1 Destination-passing style programming, in a functional setting 21

2.2 Working with destinations in λ_d . . . 22
2.2.1 Building up a vocabulary . . . 23
2.2.2 Tail-recursive map . . . 26
2.2.3 Functional queues, with destinations . . . 27
2.3 Scope escape of destinations . . . 28
2.4 Breadth-first tree traversal . . . 30
2.5 Type system . . . 31
2.5.1 Modes and the age ringoid . . . 33
2.5.2 Typing rules . . . 34
2.6 Operational semantics . . . 38
2.6.1 Runtime values and new typing context forms . . . 39
2.6.2 Evaluation contexts and commands . . . 41
2.6.3 Reduction semantics . . . 44
2.6.4 Type safety . . . 48
2.7 Formal proof of type safety . . . 48
2.8 Translation of prior work into λ_d terms and types . . . 50
2.9 Implementation of λ_d using in-place memory mutations . . . 51
2.10 Conclusion . . . 53

Chapter 3 A first implementation of first-class destination passing in Haskell 55

3.1 Linear Haskell is our implementation target . . . 55
3.2 Restricting λ_d so that it can be made safe with just linear types . . . 56
3.3 A glimpse of DPS Haskell programming . . . 56
3.3.1 Efficient difference lists, and proper memory model . . . 56
3.3.2 Breadth-first tree traversal . . . 60
3.4 DPS Haskell API and Design concerns . . . 62
3.4.1 The `UAmpar` type . . . 62
3.4.2 Filling functions for destinations . . . 64
3.5 Compact Regions: a playground to implement the DPS Haskell API . . . 65
3.5.1 Compact Regions . . . 65

3.5.2 DPS programming in Compact regions: Deserializing, lifetime, and garbage collection . . . 66
3.5.3 Memory representation of `UAmpar` objects in Compact Regions . . . 69
3.5.4 Deriving `fill` for all constructors with `Generics` . . . 71
3.5.5 Changes to GHC internals and its RTS . . . 72
3.6 Evaluating the performance of DPS programming . . . 74
3.6.1 Mapping a function over a list . . . 77
3.7 Conclusion and future work . . . 78

Chapter 4 Extending DPS support for linear data structures **81**
4.1 Challenges of destinations for linear data . . . 81
4.2 /extend/ functions for ampars with a single hole . . . 83
4.3 Arbitrary type on the right-hand side of ampars for linear data . . . 85
4.4 Limitations of this system . . . 86
4.5 Full API and breadth-first tree traversal, updated . . . 87

Chapter 5 Related work **91**
5.1 Destination-passing style for efficient memory management . . . 91
5.2 Programming with Permissions in Mezzo . . . 91
5.3 Tail modulo constructor . . . 92
5.4 A functional representation of data structures with a hole . . . 92
5.5 Destination-passing style programming: a Haskell implementation . . . 93
5.6 Semi-axiomatic sequent calculus . . . 93
5.7 Rust lifetimes . . . 94
5.8 Oxidizing OCaml . . . 94

Chapter 6 Conclusion **95**

Bibliography **97**

List of Figures

1.1 Intuitionistic linear logic in natural deduction presentation, sequent-style (ILL) . 3
1.2 Grammar of linear λ-calculus in monadic presentation (λ_{L1}) 3
1.3 Typing rules for linear λ-calculus in monadic presentation (λ_{L1}–TY) 4
1.4 Grammar of linear λ-calculus in dyadic presentation (λ_{L2}) 5
1.5 Typing rules for linear λ-calculus in dyadic presentation (λ_{L2}–TY) 5
1.6 Grammar of linear λ-calculus in modal presentation (λ_{Lm}) 9
1.7 Typing rules for linear λ-calculus in modal presentation (λ_{Lm}–TY) 9
1.8 Altered typing rules for deep modes in linear λ-calculus in modal presentation (λ_{Ldm}–TY) 12
1.9 Small-step semantics for λ_{Ldm} (λ_{Ldm}–sem) 14

2.1 Tail-recursive `map` function on lists 27
2.2 Difference list and queue implementation in equirecursive λ_d 28
2.3 Breadth-first tree traversal in destination-passing style 31
2.4 Terms, types and modes of λ_d 32
2.5 Typing rules of λ_d (λ_d–TY) 35
2.6 Derived typing rules for syntactic sugar (λ_d–TY_S) 36
2.7 Evolution of ages through nested scopes 37
2.8 Runtime values, new typing context forms, and operators 39
2.9 Typing rules for runtime values (λ_d–TY_V) 40
2.10 Evaluation contexts 42
2.11 Typing rules of evaluation contexts (λ_d–TY_E) and commands (λ_d–TY_CMD) 43
2.12 Small-step reduction of commands for λ_d (λ_d–SEM, part 1) 45
2.13 Small-step reduction of commands for λ_d (λ_d–SEM, part 2) 46

3.1 Mappings between λ_d and DPS Haskell 57
3.2 /newUAmpar tok/ 58
3.3 Memory behavior of /toList dlist/ 58
3.4 Memory behavior of /append newUAmpar 1/ 59
3.5 Memory behavior of /concat dlist1 dlist2/ (based on /fillComp/) 59

3.6 Memory behavior of /fill @'(:) : UDest [t] ⊸ (UDest t, UDest [t])/ 65
3.7 Memory behavior of /fill @'[] : UDest [t] ⊸ ()/ 65
3.8 Memory behavior of /fillLeaf : t → UDest [t] ⊸ ()/ 65
3.9 Representation of UAmpars in the compact region implementation, using /Ur/ (not chosen) 70
3.10 Chosen representation for UAmpars in the compact region implementation, using indirections 71
3.11 Benchmarks performed on AMD EPYC 7401P @ 2.0 GHz (single core, `-N1 -O2`) . 76

List of Listings

1 Implementation of difference lists in DPS Haskell . . . 58
2 Implementation of Hood-Melville queue in Haskell . . . 61
3 Implementation of breadth-first tree traversal in DPS Haskell . . . 61
4 Destination API for Haskell . . . 62
5 Destination API using compact regions . . . 67
6 Implementation of the S-expression parser without destinations . . . 68
7 Implementation of the S-expression parser with destinations . . . 68
8 `compactAddHollow#` implementation in `rts/Compact.cmm` . . . 75
9 `reifyInfoPtr#` implementation in `compiler/GHC/StgToCmm/Prim.hs` . . . 75
10 /LCtorToSymbol/ implementation in `compiler/GHC/Builtin/Types/Literal.hs` . . . 79
11 Destination-passing API allowing linear destinations . . . 88
12 Revisit of breadth-first tree traversal, with efficient queue storing destinations, in DPS Haskell . . . 89

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 either speed or 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 representation 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 — using either object-oriented paradigms that encapsulate domain data within objects than can interact with each other or using 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 the detailed execution of a task;
- the management of memory — whether memory allocation and deallocation are handled automatically by the language, or require explicit intervention from the programmer;
- side effects and exceptions — whether they can be represented, caught, and/or manipulated with the language.

Among these, memory management is one of the most critical aspects of programming language design. Every program requires memory to store data and instructions, and this memory must be managed judiciously 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 chosen memory management scheme will shape very deeply how programmers write and structure their code but also which features can be included in the language with reasonable effort.

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, which makes it unsuited for real-time applications.

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 risks; 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 providing precise memory control in the form of safe, high-level abstractions. 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 known 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.

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 the function inputs, but also those of *the function 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. Assigning memory responsibility to the caller, rather than the callee, makes functions more flexible to use: this is the core advantage of destination-passing style programming (DPS).

In a low-level, system-programming setting, DPS also offers direct 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.

In the functional interpretation of DPS that will follow, the memory saving will be more subtle, and we will instead leverage instead the flexibility and expressivity benefits of DPS to lift some usual limitations of functional programming.

3 Functional programming languages

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

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

- supports lambda abstractions, a.k.a. functions as first-class values, that can an capture parts of their parent environment (closures), can be stored, and can passed as parameters;
- emphasizes expressions over statements, where each instruction always produces a value;
- 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 — a minimal yet fully expressive model of computation, that 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 programming languages Some functional languages enforce a stricter approach by disallowing side effects except at 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 is only turned into the intended side effect by the language runtime.

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.

Memory management in functional languages Explicit memory management is very often *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 away memory management and memory access for the user.

However, this doesn't mean that functional languages must entirely give up any aspect of memory control; it simply means that memory control cannot involve explicit effectful statements like `malloc` or `free`. 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 or pragmas attached to specific functions or blocks of code, indicating how or where memory should be allocated. Another approach is to use a custom type system with *modes*, to track at type level how values are managed in memory, as in the recent modal development for Ocaml [Lor+24a]. Additionally, some languages use special built-in 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 path we will adopt —one that others have taken before— 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 that 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 incurs a big load on the allocator and garbage collector.

As a result, algorithms that generate large structures — whether by reading from an external source or transforming an existing structure — might need many intermediate allocations, 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 [Min98] introduced 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. This 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 indissociable. One interacts with the hole in the structure by interacting with the structure itself, 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], where structures with holes are referred to as *first-class contexts*.

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 *structures with holes* are a first-class type in the language, as in [Min98], but that also integrate first-class *pointers to these holes*, aka. *destinations*, as part of the user-facing language.

We'll see that combining these two features gives us performance improvement for 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 write pointers in the frontend (and not only in intermediate representation like [BCS21; LL23]) to make flexible and efficient data structure building!

In fact, I'll design a formal language in which *structures with holes* are the core 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, namely Haskell, and that predicted benefits of the approach are mostly preserved in the prototype Haskell implementation.

The formal language will also aim at providing a theoretical framework that encompasses most existing work on functional DPS and can be used to reason about safety and correctness properties of earlier works.

6 Reconciling 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 we'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. 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 and Linear Haskell

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 the intuitionistic fragment of natural deduction. We kindly refer the reader to [Pie02] and [SU06] for a proper introduction to these notions.

1.1 From λ-calculus to linear λ-calculus

At the end of the 30s, Alonzo 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 rather imperative model of *Turing machines*, introduced in the same decade. The only available objects in the untyped λ-calculus are pure functions; but they can be used smartly to encode many other kind of data, and as a result they are enough to form a solid model of computation.

In 1940, Church defined a typed variant of its original calculus, denoted *simply-typed λ-calculus*, or STLC, that gives up Turing-completeness but becomes 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.

It has been observed by Howard in 1969[How69] that the simply-typed λ-calculus is actually isomorphic to the logical framework named *natural deduction*. This observation relates to prior work by Curry where the former also observed the same sort of symmetries between systems on the computational side and on the logical side. These observations have been formalized into the *Curry-Howard isomorphism* theorem, which states that types in a typed λ-calculus correspond to formulae in a proof system, and that terms correspond to proofs of these formulae.

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, Jean-Yves Girard first introduced Linear Logic in 1987[Gir87], and only later studied the corresponding calculus, denoted *linear λ-calculus*. Linear logic follows from the observation that in some logical presentations, notably the sequent calculus[1], hypotheses are duplicated or discarded explicitly using *structural* rules of the system, named respectively

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

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 restricts contraction and weakening, so by default, every hypothesis must be used exactly once. Consequently, logical implication $\mathtt{T} \to \mathtt{U}$ is not part of linear logic, but is replaced by linear implication $\mathtt{T} \multimap \mathtt{U}$, where $\mathtt{T}$ must be used exactly once to prove $\mathtt{U}$. Linear logic also introduces a modality $!$, pronounced *of course* or *bang*, to allow weakening and contraction on specific formulae: $!\mathtt{T}$ denotes that $\mathtt{T}$ 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 formulae.

All the rules of ILL, except the ones related to the $!$ modality, are directly taken (and slightly adapted) from natural deduction. $\mathtt{1}$ denotes the truth constant, $\oplus$ 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 Γ_1, Γ_2 is multiset union, so it sums the number of occurrences of each formula in Γ_1 and Γ_2. We denote the empty multiset by $\bullet$.

Let's focus on the four rules for the $!$ modality now. The promotion rule ILL/!P states that a formula $\mathtt{T}$ can become an unrestricted formula $!\mathtt{T}$ if it only depends on formulae that are themselves unrestricted. This is denoted by the (potentially empty) multiset $!\Gamma$. The dereliction rule ILL/!D states that an unrestricted formula $!\mathtt{T}$ can be used in a place expecting a linear formula $\mathtt{T}$. The contraction rule ILL/!C and weakening rule ILL/!W state respectively that an unrestricted formula $!\mathtt{T}$ can be cloned or discarded at any time.

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

To get more familiar with this system, let's see how we can derive the implication $!\mathtt{T} \multimap (\mathtt{T} \otimes \mathtt{T})$. In other terms, how we can get two (linear) copies of $\mathtt{T}$s out of an unrestricted $\mathtt{T}$:

$$\dfrac{\dfrac{\dfrac{}{!\mathtt{T} \vdash !\mathtt{T}}\,\mathrm{ID}}{!\mathtt{T} \vdash !\mathtt{T}}\,\mathrm{ID} \qquad \dfrac{\dfrac{\dfrac{}{!\mathtt{T} \vdash !\mathtt{T}}\,\mathrm{ID}}{!\mathtt{T} \vdash \mathtt{T}}\,!\mathrm{D} \quad \dfrac{\dfrac{}{!\mathtt{T} \vdash !\mathtt{T}}\,\mathrm{ID}}{!\mathtt{T} \vdash \mathtt{T}}\,!\mathrm{D}}{!\mathtt{T},\ !\mathtt{T} \vdash \mathtt{T} \otimes \mathtt{T}}\,\otimes\mathrm{I}}{!\mathtt{T} \vdash \mathtt{T} \otimes \mathtt{T}}\,!\mathrm{C}}{\bullet \vdash !\mathtt{T} \multimap (\mathtt{T} \otimes \mathtt{T})}\,\multimap\mathrm{I}$$

We use dereliction and identity rules in both branches of the pair, to transform the unrestricted $\mathtt{T}$ into a regular one. That way, the product $\mathtt{T} \otimes \mathtt{T}$ requires the context $!\mathtt{T}, !\mathtt{T}$ which is exactly of the right form to apply the contraction rule ILL/!C. We will see in next section how this derivation translates into a well-typed program.

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

$$\frac{}{\mathtt{T} \vdash \mathtt{T}}\ \mathrm{ID} \qquad \frac{\Gamma,\ \mathtt{T} \vdash \mathtt{U}}{\Gamma \vdash \mathtt{T} \multimap \mathtt{U}}\ \multimap\mathrm{I} \qquad \frac{}{\bullet \vdash \mathtt{1}}\ \mathtt{1}\mathrm{I} \qquad \frac{\Gamma \vdash \mathtt{T}_1}{\Gamma \vdash \mathtt{T}_1 \oplus \mathtt{T}_2}\ \oplus\mathrm{I}_1 \qquad \frac{\Gamma \vdash \mathtt{T}_2}{\Gamma \vdash \mathtt{T}_1 \oplus \mathtt{T}_2}\ \oplus\mathrm{I}_2$$

$$\frac{\Gamma_1 \vdash \mathtt{T}_1 \qquad \Gamma_2 \vdash \mathtt{T}_2}{\Gamma_1,\ \Gamma_2 \vdash \mathtt{T}_1 \otimes \mathtt{T}_2}\ \otimes\mathrm{I} \qquad \frac{\Gamma_1 \vdash \mathtt{T} \qquad \Gamma_2 \vdash \mathtt{T} \multimap \mathtt{U}}{\Gamma_1,\ \Gamma_2 \vdash \mathtt{U}}\ \multimap\mathrm{E} \qquad \frac{\Gamma_1 \vdash \mathtt{1} \qquad \Gamma_2 \vdash \mathtt{U}}{\Gamma_1,\ \Gamma_2 \vdash \mathtt{U}}\ \mathtt{1}\mathrm{E}$$

$$\frac{\begin{array}{l}\Gamma_1 \vdash \mathtt{T}_1 \oplus \mathtt{T}_2 \\ \Gamma_2,\ \mathtt{T}_1 \vdash \mathtt{U} \\ \Gamma_2,\ \mathtt{T}_2 \vdash \mathtt{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathtt{U}}\ \oplus\mathrm{E} \qquad \frac{\Gamma_1 \vdash \mathtt{T}_1 \otimes \mathtt{T}_2 \qquad \Gamma_2,\ \mathtt{T}_1,\ \mathtt{T}_2 \vdash \mathtt{U}}{\Gamma_1,\ \Gamma_2 \vdash \mathtt{U}}\ \otimes\mathrm{E} \qquad \frac{!\Gamma \vdash \mathtt{T}}{!\Gamma \vdash !\mathtt{T}}\ !\mathrm{P} \qquad \frac{\Gamma \vdash !\mathtt{T}}{\Gamma \vdash \mathtt{T}}\ !\mathrm{D} \qquad \frac{\Gamma_1 \vdash !\mathtt{T} \qquad \Gamma_2,\ !\mathtt{T},\ !\mathtt{T} \vdash \mathtt{U}}{\Gamma_1,\ \Gamma_2 \vdash \mathtt{U}}\ !\mathrm{C}$$

$$\frac{\Gamma_1 \vdash !\mathtt{T} \qquad \Gamma_2 \vdash \mathtt{U}}{\Gamma_1,\ \Gamma_2 \vdash \mathtt{U}}\ !\mathrm{W}$$

Figure 1.1: Intuitionistic linear logic in natural deduction presentation, sequent-style (ILL)

$$\begin{array}{rcl}
v & ::= & \boldsymbol{\lambda} x \mapsto u \mid () \mid \mathsf{Inl}\, v \mid \mathsf{Inr}\, v \mid (v_1\,,\, v_2) \mid \mathsf{Many}\, v \\
t, u & ::= & v \mid x \mid \mathsf{Inl}\, t \mid \mathsf{Inr}\, t \mid (t_1\,,\, t_2) \mid \mathsf{Many}\, t \mid t\ t' \mid t \fatsemi t' \\
& \mid & \mathbf{case}\ t\ \mathbf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1\,,\ \mathsf{Inr}\, x_2 \mapsto u_2\} \mid \mathbf{case}\ t\ \mathbf{of}\ (x_1\,,\, x_2) \mapsto u \\
& \mid & \mathbf{dup}\ t\ \mathbf{as}\ x_1\,,\, x_2\ \mathbf{in}\ u \mid \mathbf{drop}\ t\ \mathbf{in}\ u \mid \mathbf{derelict}\ t
\end{array}$$

$$\mathtt{T}, \mathtt{U} \ ::= \ \mathtt{T} \multimap \mathtt{U} \mid \mathtt{1} \mid \mathtt{T}_1 \oplus \mathtt{T}_2 \mid \mathtt{T}_1 \otimes \mathtt{T}_2 \mid !\mathtt{T}$$

$$\Gamma \ ::= \ \bullet \mid x : \mathtt{T} \mid \Gamma_1,\ \Gamma_2$$

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

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 $x : \mathtt{T}$. As usual, duplicated variable names are not allowed in a context Γ. The comma operator now denotes disjoint union for finite maps.

Term grammar for λ_{L1} borrows most of simply-typed lambda calculus grammar. Elimination of unit type 1 is made with the ⨾ operator. Pattern-matching on sum and product types is made with the **case** keyword. Finally, we have new operators **dup**, **drop** and **derelict** for, respectively, contraction, weakening and dereliction of unrestricted terms with type !T. In addition, the

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

$$\frac{}{x : \mathsf{T} \vdash x : \mathsf{T}}\ \mathrm{ID} \qquad \frac{\Gamma,\ x : \mathsf{T} \vdash u : \mathsf{U}}{\Gamma \vdash \boldsymbol{\lambda} x \mapsto u : \mathsf{T} \multimap \mathsf{U}}\ \multimap\mathrm{I} \qquad \frac{}{\bullet \vdash () : \mathbf{1}}\ \mathbf{1}\mathrm{I} \qquad \frac{\Gamma \vdash t_1 : \mathsf{T}_1}{\Gamma \vdash \mathsf{Inl}\, t_1 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \oplus\mathrm{I}_1$$

$$\frac{\Gamma \vdash t_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{Inr}\, t_2 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \oplus\mathrm{I}_2 \qquad \frac{\Gamma_1 \vdash t_1 : \mathsf{T}_1 \quad \Gamma_2 \vdash t_2 : \mathsf{T}_2}{\Gamma_1,\ \Gamma_2 \vdash (t_1\,,\ t_2) : \mathsf{T}_1 \otimes \mathsf{T}_2}\ \otimes\mathrm{I} \qquad \frac{\Gamma_1 \vdash t : \mathsf{T} \quad \Gamma_2 \vdash t' : \mathsf{T} \multimap \mathsf{U}}{\Gamma_1,\ \Gamma_2 \vdash t'\ t : \mathsf{U}}\ \multimap\mathrm{E}$$

$$\frac{\Gamma_1 \vdash t : \mathbf{1} \qquad \Gamma_2 \vdash u : \mathsf{U}}{\Gamma_1,\ \Gamma_2 \vdash t \fatsemi u : \mathsf{U}}\ \mathbf{1}\mathrm{E} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \oplus \mathsf{T}_2 \\ \Gamma_2,\ x_1 : \mathsf{T}_1 \vdash u_1 : \mathsf{U} \\ \Gamma_2,\ x_2 : \mathsf{T}_2 \vdash u_2 : \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{case}\ t\ \mathsf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1\,,\ \mathsf{Inr}\, x_2 \mapsto u_2\} : \mathsf{U}}\ \oplus\mathrm{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \\ \Gamma_2,\ x_1 : \mathsf{T}_1,\ x_2 : \mathsf{T}_2 \vdash u : \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{case}\ t\ \mathsf{of}\ (x_1\,,\ x_2) \mapsto u : \mathsf{U}}\ \otimes\mathrm{E} \qquad \frac{!\Gamma \vdash t : \mathsf{T}}{!\Gamma \vdash \mathsf{Many}\, t : !\mathsf{T}}\ !\mathrm{P} \qquad \frac{\Gamma \vdash t : !\mathsf{T}}{\Gamma \vdash \mathsf{derelict}\ t : \mathsf{T}}\ !\mathrm{D}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : !\mathsf{T} \\ \Gamma_2,\ x_1 : !\mathsf{T},\ x_2 : !\mathsf{T} \vdash u : \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{dup}\ t\ \mathsf{as}\ x_1\,,\ x_2\ \mathsf{in}\ u : \mathsf{U}}\ !\mathrm{C} \qquad \frac{\Gamma_1 \vdash t : !\mathsf{T} \quad \Gamma_2 \vdash u : \mathsf{U}}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{drop}\ t\ \mathsf{in}\ u : \mathsf{U}}\ !\mathrm{W}$$

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

language includes a data constructor for unrestricted terms and values, denoted by $\mathsf{Many}\, t$ and $\mathsf{Many}\, v$. Promotion of a term to an unrestricted form is made by direct application of constructor Many. For easier exposition, we allow ourselves to use syntactic sugar $\mathbf{let}\ x := t\ \mathbf{in}\ u$ and encode it as $(\boldsymbol{\lambda} x \mapsto u)\ (t)$.

The derivation of the linear implication $!\mathsf{T} \multimap (\mathsf{T} \otimes \mathsf{T})$ from previous section now translates into a typing derivation for a well-typed term of type $!\mathsf{T} \multimap (\mathsf{T} \otimes \mathsf{T})$. Let's see how it goes:

$$\dfrac{\dfrac{\dfrac{}{x : !\mathsf{T} \vdash x : !\mathsf{T}}\ \mathrm{ID} \qquad \dfrac{\dfrac{\dfrac{}{x_1 : !\mathsf{T} \vdash x_1 : !\mathsf{T}}\ \mathrm{ID}}{x_1 : !\mathsf{T} \vdash \mathsf{derelict}\ x_1 : \mathsf{T}}\ !\mathrm{D} \qquad \dfrac{\dfrac{}{x_2 : !\mathsf{T} \vdash x_2 : !\mathsf{T}}\ \mathrm{ID}}{x_2 : !\mathsf{T} \vdash \mathsf{derelict}\ x_2 : \mathsf{T}}\ !\mathrm{D}}{x_1 : !\mathsf{T},\ x_2 : !\mathsf{T} \vdash (\mathsf{derelict}\ x_1\,,\ \mathsf{derelict}\ x_2) : \mathsf{T} \otimes \mathsf{T}}\ \otimes\mathrm{I}}{x : !\mathsf{T} \vdash \mathsf{dup}\ x\ \mathsf{as}\ x_1\,,\ x_2\ \mathsf{in}\ (\mathsf{derelict}\ x_1\,,\ \mathsf{derelict}\ x_2) : \mathsf{T} \otimes \mathsf{T}}\ !\mathrm{C}}{\bullet \vdash \boldsymbol{\lambda} x \mapsto \mathsf{dup}\ x\ \mathsf{as}\ x_1\,,\ x_2\ \mathsf{in}\ (\mathsf{derelict}\ x_1\,,\ \mathsf{derelict}\ x_2) : !\mathsf{T} \multimap (\mathsf{T} \otimes \mathsf{T})}\ \multimap\mathrm{I}$$

This translation is very direct. The only change is that we have to give distinct variable names to hypotheses of the same type inside the typing context, while there was no concept of named copies of a same formula in ILL. We also see that dealing with unrestricted hypotheses can be rather heavyweight because of the need of manual duplication and dereliction. Next section present a way to circumvent this issue.

$$
\begin{array}{rcl}
v & ::= & \boldsymbol{\lambda} x \mapsto u \mid () \mid \mathsf{Inl}\, v \mid \mathsf{Inr}\, v \mid (v_1\,,\, v_2) \mid \mathsf{Many}\, v \\
t, u & ::= & v \mid x \mid \mathsf{Inl}\, t \mid \mathsf{Inr}\, t \mid (t_1\,,\, t_2) \mid \mathsf{Many}\, t \mid t\ t' \mid t \fatsemi t' \\
& \mid & \mathsf{case}\ t\ \mathsf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1\,,\ \mathsf{Inr}\, x_2 \mapsto u_2\} \mid \mathsf{case}\ t\ \mathsf{of}\ (x_1\,,\, x_2) \mapsto u \mid \mathsf{case}\ t\ \mathsf{of}\ \mathsf{Many}\, x \mapsto u
\end{array}
$$

$$
\mathsf{T}, \mathsf{U} \quad ::= \quad \mathsf{T} \multimap \mathsf{U} \mid \mathsf{1} \mid \mathsf{T}_1 \oplus \mathsf{T}_2 \mid \mathsf{T}_1 \otimes \mathsf{T}_2 \mid\ !\mathsf{T}
$$

$$
\begin{array}{rcl}
\Gamma & ::= & \bullet \mid x : \mathsf{T} \mid \Gamma_1,\ \Gamma_2 \\
\mho & ::= & \bullet \mid x : \mathsf{T} \mid \mho_1,\ \mho_2
\end{array}
$$

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

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

$$\frac{}{x : \mathsf{T}\ ;\ \mho \vdash x : \mathsf{T}}\ \mathrm{ID_{LIN}} \qquad \frac{}{\bullet\ ;\ \mho,\ x : \mathsf{T} \vdash x : \mathsf{T}}\ \mathrm{ID_{UR}} \qquad \frac{\Gamma,\ x : \mathsf{T}\ ;\ \mho \vdash u : \mathsf{U}}{\Gamma\ ;\ \mho \vdash \boldsymbol{\lambda} x \mapsto u : \mathsf{T} \multimap \mathsf{U}}\ \multimap\mathrm{I}$$

$$\frac{}{\bullet\ ;\ \mho \vdash () : \mathsf{1}}\ \mathsf{1}\mathrm{I} \qquad \frac{\Gamma\ ;\ \mho \vdash t_1 : \mathsf{T}_1}{\Gamma\ ;\ \mho \vdash \mathsf{Inl}\, t_1 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \oplus\mathrm{I}_1 \qquad \frac{\Gamma\ ;\ \mho \vdash t_2 : \mathsf{T}_2}{\Gamma\ ;\ \mho \vdash \mathsf{Inr}\, t_2 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \oplus\mathrm{I}_2$$

$$\frac{\Gamma_1\ ;\ \mho \vdash t_1 : \mathsf{T}_1 \quad \Gamma_2\ ;\ \mho \vdash t_2 : \mathsf{T}_2}{\Gamma_1,\ \Gamma_2\ ;\ \mho \vdash (t_1\,,\, t_2) : \mathsf{T}_1 \otimes \mathsf{T}_2}\ \otimes\mathrm{I} \qquad \frac{\bullet\ ;\ \mho \vdash t : \mathsf{T}}{\bullet\ ;\ \mho \vdash \mathsf{Many}\, t : !\mathsf{T}}\ !\mathrm{I} \qquad \frac{\Gamma_1\ ;\ \mho \vdash t : \mathsf{T} \quad \Gamma_2\ ;\ \mho \vdash t' : \mathsf{T} \multimap \mathsf{U}}{\Gamma_1,\ \Gamma_2\ ;\ \mho \vdash t'\ t : \mathsf{U}}\ \multimap\mathrm{E}$$

$$\frac{\Gamma_1\ ;\ \mho \vdash t : \mathsf{1} \quad \Gamma_2\ ;\ \mho \vdash u : \mathsf{U}}{\Gamma_1,\ \Gamma_2\ ;\ \mho \vdash t \fatsemi u : \mathsf{U}}\ \mathsf{1}\mathrm{E} \qquad \frac{\begin{array}{c}\Gamma_1\ ;\ \mho \vdash t : \mathsf{T}_1 \oplus \mathsf{T}_2 \\ \Gamma_2,\ x_1 : \mathsf{T}_1\ ;\ \mho \vdash u_1 : \mathsf{U} \\ \Gamma_2,\ x_2 : \mathsf{T}_2\ ;\ \mho \vdash u_2 : \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2\ ;\ \mho \vdash \mathsf{case}\ t\ \mathsf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1\,,\ \mathsf{Inr}\, x_2 \mapsto u_2\} : \mathsf{U}}\ \oplus\mathrm{E}$$

$$\frac{\Gamma_1\ ;\ \mho \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \quad \Gamma_2,\ x_1 : \mathsf{T}_1,\ x_2 : \mathsf{T}_2\ ;\ \mho \vdash u : \mathsf{U}}{\Gamma_1,\ \Gamma_2\ ;\ \mho \vdash \mathsf{case}\ t\ \mathsf{of}\ (x_1\,,\, x_2) \mapsto u : \mathsf{U}}\ \otimes\mathrm{E} \qquad \frac{\Gamma_1\ ;\ \mho \vdash t : !\mathsf{T} \quad \Gamma_2\ ;\ \mho,\ x : \mathsf{T} \vdash u : \mathsf{U}}{\Gamma_1,\ \Gamma_2\ ;\ \mho \vdash \mathsf{case}\ t\ \mathsf{of}\ \mathsf{Many}\, x \mapsto u : \mathsf{U}}\ !\mathrm{E}$$

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

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 the linear λ-calculus, named *dyadic* presentation or λ_{L2}, tends to alleviate this burden 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 are 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 $\mathsf{case}\ t\ \mathsf{of}\ \mathsf{Many}\ x \mapsto u$ construct let us access a term of type $!\mathsf{T}$ as a variable x of type T that lives in the unrestricted context $\mho$. It's important to note that $\mathsf{case}\ t\ \mathsf{of}\ \mathsf{Many}\ x \mapsto u$ 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 with $\mathsf{let}\ x := \mathsf{derelict}\ t\ \mathsf{in}\ u$ 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}–$\mathrm{TY}/\mathrm{ID}_{\mathrm{LIN}}$, λ_{L2}–$\mathrm{TY}/\mathrm{ID}_{\mathrm{UR}}$, and λ_{L2}–$\mathrm{TY}/\mathbf{1}\mathrm{I}$. 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}–$\mathrm{TY}/\mathrm{ID}_{\mathrm{LIN}}$, is the usual linear identity: it consumes the variable x present in the linear typing context. Because the linear typing context must be split into disjoint parts between sibling subterms (when a rule has several premises, like the product rule), it means x can only be used in one of them. The second identity rule, λ_{L2}–$\mathrm{TY}/\mathrm{ID}_{\mathrm{UR}}$, 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. Because the unrestricted typing context is duplicated between sibling subterms (again, when a rule has several premises), an unrestricted variable can be used in several of them. For instance, the following derivation is *not valid* because we don't respect the disjointness condition of the comma operator for the resulting typing context of the pair:

$$\frac{\dfrac{}{x' : \mathsf{T}\ ;\ \bullet \vdash x' : \mathsf{T}}\ \mathrm{ID}_{\mathrm{LIN}} \qquad \dfrac{}{x' : \mathsf{T}\ ;\ \bullet \vdash x' : \mathsf{T}}\ \mathrm{ID}_{\mathrm{LIN}}}{x' : \mathsf{T},\ x' : \mathsf{T}\ ;\ \bullet \vdash (x', x') : \mathsf{T} \otimes \mathsf{T}}\ \otimes\mathrm{I}$$

but the following derivation is valid, as the unrestricted typing context is duplicated between sibling subterms:

$$\frac{\dfrac{}{\bullet\ ;\ x' : \mathsf{T} \vdash x' : \mathsf{T}}\ \mathrm{ID}_{\mathrm{UR}} \qquad \dfrac{}{\bullet\ ;\ x' : \mathsf{T} \vdash x' : \mathsf{T}}\ \mathrm{ID}_{\mathrm{UR}}}{\bullet\ ;\ x' : \mathsf{T} \vdash (x', x') : \mathsf{T} \otimes \mathsf{T}}\ \otimes\mathrm{I}$$

At this point, we are very close from recreating a term of type $!\mathsf{T} \multimap (\mathsf{T} \otimes \mathsf{T})$ as we did in previous section. Let's finish the example:

$$\frac{\dfrac{\dfrac{}{x : !\mathsf{T}\ ;\ \bullet \vdash x : !\mathsf{T}}\ \mathrm{ID}_{\mathrm{LIN}} \qquad \dfrac{\dfrac{}{\bullet\ ;\ x' : \mathsf{T} \vdash x' : \mathsf{T}}\ \mathrm{ID}_{\mathrm{UR}} \qquad \dfrac{}{\bullet\ ;\ x' : \mathsf{T} \vdash x' : \mathsf{T}}\ \mathrm{ID}_{\mathrm{UR}}}{\bullet\ ;\ x' : \mathsf{T} \vdash (x', x') : \mathsf{T} \otimes \mathsf{T}}\ \otimes\mathrm{I}}{x : !\mathsf{T}\ ;\ \bullet \vdash \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Many}\ x' \mapsto (x', x') : \mathsf{T} \otimes \mathsf{T}}\ !\mathrm{E}}{\bullet\ ;\ \bullet \vdash \lambda x \mapsto \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Many}\ x' \mapsto (x', x') : !\mathsf{T} \multimap \mathsf{T} \otimes \mathsf{T}}\ \multimap\mathrm{I}$$

If we had any useless variable binding polluting our unrestricted typing context (let's say $\mathcal{U} = y : !\mathsf{U}$), the derivation would still hold without any change at term level; the useless bindings would just be carried over throughout the typing tree, and eliminated at the leaves (via implicit weakening happening in leaf rules):

$$\dfrac{\dfrac{\dfrac{}{x : !\mathsf{T} \,;\, y : !\mathsf{U} \vdash x : !\mathsf{T}}\,\mathrm{ID}_{\mathrm{LIN}} \qquad \dfrac{\dfrac{}{\bullet \,;\, y : !\mathsf{U},\ x' : \mathsf{T} \vdash x' : \mathsf{T}}\,\mathrm{ID}_{\mathrm{UR}} \qquad \dfrac{}{\bullet \,;\, y : !\mathsf{U},\ x' : \mathsf{T} \vdash x' : \mathsf{T}}\,\mathrm{ID}_{\mathrm{UR}}}{\bullet \,;\, y : !\mathsf{U},\ x' : \mathsf{T} \vdash (x', x') : \mathsf{T} \otimes \mathsf{T}}\,\otimes\mathrm{I}}{x : !\mathsf{T} \,;\, y : !\mathsf{U} \vdash \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Many}\, x' \mapsto (x', x') : \mathsf{T} \otimes \mathsf{T}}\,!\mathrm{E}}{\bullet \,;\, y : !\mathsf{U} \vdash \lambda x \mapsto \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Many}\, x' \mapsto (x', x') : !\mathsf{T} \multimap \mathsf{T} \otimes \mathsf{T}}\,\multimap\mathrm{I}$$

Managing unrestricted hypotheses is, as we just demonstrated, much easier in the dyadic system λ_{L2}. Typing trees are shorter, and probably easier to read too.

Also, we don't loose anything by going from λ_{L1} presentation to λ_{L2}: Andreoli [And92] has a detailed proof that λ_{L1}–TY and λ_{L2}–TY are equivalent, in other terms, that a program t types in the pair of contexts $\Gamma; \mathcal{U}$ in λ_{L2}–TY if and only if it types in context $\Gamma,\ !\mathcal{U}$ in λ_{L1}–TY.

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 respect to usual simply-typed λ-calculus. Anticipating on our future needs, further in this document we'll need to carry more information and restrictions throughout the type system than just linearity alone.

Naively, we could just multiply the typing contexts, for each new modality that we need. The problem is, that approach is not really scalable: at the end we need one context per possible combination of modalities (if modalities represent orthogonal principles), so it grows really quickly.

In fact, without thinking too far away, 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 one or unlimited? In [GSS92], Girard considers the *bounded* extension of linear logic, where the number of uses of hypotheses can be restricted to any value m —named *multiplicity*— 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 when 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 $x :_1 \mathsf{S}, y :_{\mathsf{m}} \mathsf{T}, z :_{\mathsf{n}} \mathsf{U} \vdash$. 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 a linear type 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 the 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). By abstracting away the structural restrictions of the type system as a mode system with a semiring structure, we make the type system much more scalable: if we were to enrich the type system, and if we are able to express the extension as a new semiring, then we almost don't have to modify the typing rules; just the underlying mode semiring.

We then lift the mode operators to variable bindings and typing contexts in the following way:

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

We'll now show what the concrete *modal* presentation for intuitionistic λ-calculus or λ_{Lm} looks like. We use 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 a 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

Now that we have a completely defined the ringoid for modes, we can move to the grammar and typing rules of λ_{Lm}, given in Figures 1.6 and 1.7. The main change compared to λ_{L2} is that the constructor for $!_{\mathsf{m}}\mathsf{T}$ is now $\mathsf{Mod}_{\mathsf{m}}\ t$ (instead of $\mathsf{Many}\ t$ for $!\mathsf{T}$).

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 m that must be *compatible with a single linear use* (we note $1 \precsim \mathsf{m}$). In our ringoid, with only two elements, we have both $1 \precsim 1$ and $1 \precsim \omega$, 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

[2]This uniform approach where every binding receives a mode seems to originate from [GS14] and [POM14].
[3]We still get all the algebraic benefits of [GS14] even without a zero for our structure.

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

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

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

$$\frac{1 \leq m}{\omega \cdot \Gamma,\ x :_m \mathsf{T} \vdash x : \mathsf{T}}\ \mathrm{ID} \qquad \frac{\Gamma,\ x :_m \mathsf{T} \vdash u : \mathsf{U}}{\Gamma \vdash \boldsymbol{\lambda} x\,{}_{m}\!\mapsto u : \mathsf{T}\,{}_{m}\!\multimap \mathsf{U}}\ \multimap\mathrm{I} \qquad \frac{}{\omega \cdot \Gamma \vdash () : \mathsf{1}}\ \mathsf{1}\mathrm{I}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{T}_1}{\Gamma \vdash \mathsf{Inl}\, t_1 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \oplus\mathrm{I}_1 \qquad \frac{\Gamma \vdash t_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{Inr}\, t_2 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \oplus\mathrm{I}_2 \qquad \frac{\Gamma_1 \vdash t_1 : \mathsf{T}_1 \quad \Gamma_2 \vdash t_2 : \mathsf{T}_2}{\Gamma_1 + \Gamma_2 \vdash (t_1\,,\, t_2) : \mathsf{T}_1 \otimes \mathsf{T}_2}\ \otimes\mathrm{I}$$

$$\frac{\Gamma \vdash t : \mathsf{T}}{n \cdot \Gamma \vdash \mathsf{Mod}_n\, t : !_n \mathsf{T}}\ !\mathrm{I} \qquad \frac{\Gamma_1 \vdash t : \mathsf{T} \quad \Gamma_2 \vdash t' : \mathsf{T}\,{}_{m}\!\multimap \mathsf{U}}{m \cdot \Gamma_1 + \Gamma_2 \vdash t'\ t : \mathsf{U}}\ \multimap\mathrm{E} \qquad \frac{\Gamma_1 \vdash t : \mathsf{1} \quad \Gamma_2 \vdash u : \mathsf{U}}{\Gamma_1 + \Gamma_2 \vdash t \fatsemi u : \mathsf{U}}\ \mathsf{1}\mathrm{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \oplus \mathsf{T}_2 \\ \Gamma_2,\ x_1 :_1 \mathsf{T}_1 \vdash u_1 : \mathsf{U} \\ \Gamma_2,\ x_2 :_1 \mathsf{T}_2 \vdash u_2 : \mathsf{U}\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{case}\ t\ \mathbf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1\,,\ \mathsf{Inr}\, x_2 \mapsto u_2\} : \mathsf{U}}\ \oplus\mathrm{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \\ \Gamma_2,\ x_1 :_1 \mathsf{T}_1,\ x_2 :_1 \mathsf{T}_2 \vdash u : \mathsf{U}\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{case}\ t\ \mathbf{of}\ (x_1\,,\, x_2) \mapsto u : \mathsf{U}}\ \otimes\mathrm{E} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash t : !_n \mathsf{T} \\ \Gamma_2,\ x :_n \mathsf{T} \vdash u : \mathsf{U}\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{case}\ t\ \mathbf{of}\ \mathsf{Mod}_n\, x \mapsto u : \mathsf{U}}\ !\mathrm{E}$$

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

ringoid[4]. Rules λ_{Lm}–TY/ID and λ_{Lm}–TY/1I also allow 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), then $\Gamma_1 + \Gamma_2$ will contain binding $x :_\omega \mathsf{T}$. Said differently, the parent term will automatically deduce whether 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 linear context split for subterms in $\lambda_{L1,2}$ and unrestricted context duplication for subterms in λ_{L2}. I would argue that, thanks to the algebraic properties of the mode semiring, it makes the presentation of the system smoother (at the theoretical level at least; implementing type checking for such a linear type system can be trickier).

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

The last specificity of this system is that the function arrow $\multimap$ now has a mode m to which it binds its argument. Previously, in $\lambda_{L1,2}$, the argument of a function was always linear, in the sense that the corresponding variable binding in the body of the function was linear. Here, in λ_{Lm}, we can bind argument at any mode m (and the rule λ_{Lm}–TY/$\multimap$E for function application reflects that: the typing context Γ_1 for the argument passed to the function is multiplied by m in the conclusion of the rule). 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 anyway, then it makes sense to allow this mode m to be whatever the programmer wants, and not just default to 1.

Let's update our running example. The term syntax doesn't change, we only see an evolution in typing contexts (as we now have only one, with mode annotations, instead of two):

$$\dfrac{\dfrac{\dfrac{}{y :_\omega \mathsf{U},\ x :_1 !\mathsf{T} \vdash x : !\mathsf{T}}\ \mathrm{ID}\ (1 \leq 1) \qquad \dfrac{\dfrac{}{x' :_1 \mathsf{T} \vdash x' : \mathsf{T}}\ \mathrm{ID}\ (1 \leq 1) \qquad \dfrac{}{x' :_1 \mathsf{T} \vdash x' : \mathsf{T}}\ \mathrm{ID}\ (1 \leq 1)}{x' :_\omega \mathsf{T} \vdash (x', x') : \mathsf{T} \otimes \mathsf{T}}\ \otimes\mathrm{I}}{y :_\omega \mathsf{U},\ x :_1 !\mathsf{T} \vdash \mathtt{case}\ x\ \mathtt{of}\ \mathsf{Mod}_\omega\ x' \mapsto (x', x') : \mathsf{T} \otimes \mathsf{T}}\ !\mathrm{E}}{y :_\omega \mathsf{U} \vdash \lambda x \,{}_1\!\mapsto \mathtt{case}\ x\ \mathtt{of}\ \mathsf{Mod}_\omega\ x' \mapsto (x', x') : !\mathsf{T} \,{}_1\!\multimap \mathsf{T} \otimes \mathsf{T}}\ \multimap\mathrm{I}$$

Now the unrestricted, unused part of the typing context ($y :_\omega \mathsf{U}$ in this case) doesn't have to be propagated towards all leaves, but only towards at least one of them (though we can still choose to propagate it towards all of them), thanks to the way the $+$ works on contexts. We also could have chosen to type x' at mode ω in the two identical instances of the λ_{Lm}–TY/ID rule at the top of the tree; it has no impact on the rest of the derivation as both $1 + 1$ and $\omega + \omega$ gives ω as a result.

Finally, we can give an alternative and more concise version of this function, taking advantage of the *mode on function arrow* feature that we just discussed above. This new version has type $\mathsf{T} \,{}_\omega\!\multimap \mathsf{T} \otimes \mathsf{T}$ instead of $!\mathsf{T} \,{}_1\!\multimap \mathsf{T} \otimes \mathsf{T}$:

[4]See Section 2.5.2 for a case of a ringoid where not all modes are compatible with the unit.

$$\dfrac{\dfrac{}{y :_\omega \mathtt{U},\ x' :_1 \mathtt{T} \vdash x' : \mathtt{T}}\ \mathrm{ID} \quad \dfrac{}{y :_\omega \mathtt{U},\ x' :_1 \mathtt{T} \vdash x' : \mathtt{T}}\ \mathrm{ID}}{\dfrac{y :_\omega \mathtt{U},\ x' :_\omega \mathtt{T} \vdash (x', x') : \mathtt{T} \otimes \mathtt{T}}{y :_\omega \mathtt{U} \vdash \lambda x' \,{}_\omega\!\mapsto (x', x') : \mathtt{T} \,{}_\omega\!\multimap \mathtt{T} \otimes \mathtt{T}}\ \multimap\mathrm{I}}\ \otimes\mathrm{I}$$

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 $!\mathtt{T} \otimes !\mathtt{U}$ and $!(\mathtt{T} \otimes \mathtt{U})$; we cannot go from $!(\mathtt{T} \otimes \mathtt{U})$ to $!\mathtt{T} \otimes !\mathtt{U}$. In other terms, an unrestricted pair $!(\mathtt{T} \otimes \mathtt{U})$ 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 $\mathtt{T} \otimes \mathtt{U}$, that no longer allow to duplicate or discard any of $\mathtt{T}$ or $\mathtt{U}$. As a result, $\mathtt{T}$ and $\mathtt{U}$ 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 $!(\mathtt{T} \otimes \mathtt{U})$ can be bound in an unrestricted binding $x : \mathtt{T} \otimes \mathtt{U} \in \mathcal{U}$ (resp. $x :_\omega \mathtt{T} \otimes \mathtt{U}$ in λ_{Lm}) and be used as this without need for dereliction, when it will be pattern-matched on (through the rule $\lambda_{L2,m}$–TY/$\otimes$E), implicit dereliction will still happen, and linear-only bindings will be made for its components: $x_1 : \mathtt{T},\ x_2 : \mathtt{U}$ in the linear typing context (resp. $x_1 :_1 \mathtt{T},\ x_2 :_1 \mathtt{U}$ in λ_{Lm}).

Let's say we want to write the **fst** function, that extracts the first element of a pair. In a non-linear language, we would probably write it that way:

$$\begin{aligned}
&\mathtt{fst}_{\mathtt{nl}} \ :\ \mathtt{T} \otimes \mathtt{U} \to \mathtt{T} \\
&\mathtt{fst}_{\mathtt{nl}} \ \triangleq\ \mathtt{case}\ \lambda x \mapsto x\ \mathtt{of}\ (x_1\,,\, x_2) \mapsto x_1
\end{aligned}$$

But in λ_{Lm}, or even $\lambda_{L1,2}$, we are not allowed to do this. A naive idea would be to give the function the signature $\mathtt{T} \otimes \mathtt{U} \,{}_\omega\!\multimap \mathtt{T}$, but that still doesn't help: as we just said, even with an unrestricted pair, we have to consume both elements of the pair the same number of times. The only valid solution is to indicate, in the signature, that the second element of the pair –that we discard– will not be used linearly:

$$\begin{aligned}
&\mathtt{fst}_{\mathtt{m}} \ :\ \mathtt{T} \otimes (!_\omega \mathtt{U}) \,{}_1\!\multimap \mathtt{T} \\
&\mathtt{fst}_{\mathtt{m}} \ \triangleq\ \lambda x \,{}_1\!\mapsto \mathtt{case}\ x\ \mathtt{of}\ (x_1\,,\, ux_2) \mapsto \\
&\qquad\qquad \mathtt{case}\ ux_2\ \mathtt{of}\ \mathsf{Mod}_\omega\ x_2 \mapsto x_1
\end{aligned}$$

It's often desirable to lift this limitation in a real programming language, and let the $!_{\mathtt{m}}$ modality, and more generally, the modes of variables, distribute over the other connectives like $\otimes$ or $\oplus$. The main 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 (further denoted λ_{Ldm}), functions of type $\mathtt{T} \,{}_\omega\!\multimap \mathtt{U}$ gain exactly the same expressive power as the ones with type $\mathtt{T} \to \mathtt{U}$ in STLC. In other terms, any program that is valid in STLC but doesn't abide by linearity can still type in

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

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

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \\ \Gamma_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u : \mathsf{U}\end{array}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{case}_{\mathsf{m}}\ t\ \mathbf{of}\ (x_1\ ,\ x_2) \mapsto u : \mathsf{U}}\ \otimes\mathrm{E} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash t : !_{\mathsf{n}} \mathsf{T} \\ \Gamma_2,\ x :_{\mathsf{m}\cdot\mathsf{n}} \mathsf{T} \vdash u : \mathsf{U}\end{array}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{case}_{\mathsf{m}}\ t\ \mathbf{of}\ \mathsf{Mod}_{\mathsf{n}}\, x \mapsto u : \mathsf{U}}\ !\mathrm{E}$$

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

λ_{Ldm} without any restructuration needed, we just need to add the proper mode annotations! This was clearly not the case in a faithful interpretation of Girard's linear logic, like we just showed before with limitations of λ_{Lm} when implementing $\mathbf{fst}_{\mathsf{m}}$.

Revisiting this **fst** function, with deep modes in λ_{Ldm}, we are able to write:

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

This is precisely the same implementation as $\mathbf{fst}_{\mathsf{nl}}$, with just extra mode annotations!

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

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

The only change needed on the grammar between λ_{Lm} and λ_{Ldm} is that the **case** constructs in λ_{Ldm} take a mode m to which they consume the scrutinee , which is propagated to the variable bindings for 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 identical to those of Figure 1.7.

For these new **case** rules, we observe that the typing context Γ_1 in which the scrutinee types is scaled by m in the conclusion of these rules. This is very similar to the application rule λ_{Lm}–TY/⊸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 $x_2 :_1 \mathsf{T}_2 \vdash ((), x_2) : 1 \otimes \mathsf{T}_2$ (the pair uses x_2 exactly once), if we want to extract the pair components with mode ω to drop x_2 (as in $\mathbf{fst}_{\mathsf{dm}}$), 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 having dependencies on linear variables 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 we 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.

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

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 straightforward reduction system, focusing rules can only trigger when the subterm that would be focused 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, $\mathtt{Mod}_{\mathtt{m}}\ t$ 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 $\big(E \circ \mathtt{case}_{\mathtt{m}}\ [\,]\ \mathtt{of}\ (x_1\,,\ x_2) \mapsto u\big)\big[(v_1\,,\ v_2)\big]$, without the need to recreate the term $\mathtt{case}_{\mathtt{m}}\ (v_1\,,\ v_2)\ \mathtt{of}\ (x_1\,,\ x_2) \mapsto u$. However, I prefer the presentation in three distinct steps, that despite being more verbose, clearly 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 lambda-calculi with sum and product types (and strict evaluation strategy).

For instance, let's see the reduction for function application. We reuse the same example of our function with signature $!\mathtt{T} \multimap (\mathtt{T} \otimes \mathtt{T})$, here applied to the argument $\mathtt{Mod}_{\omega}\ (\mathtt{Inl}\ ())$:

$$
\begin{array}{lll}
 & [\,]\big[(\boldsymbol{\lambda} x_1 \mapsto \mathtt{case}\ x\ \mathtt{of}\ \mathtt{Mod}_{\omega}\ x' \mapsto (x'\,,\ x'))\ (\mathtt{Mod}_{\omega}\ (\mathtt{Inl}\ ()))\big] & \\
\longrightarrow & [\,]\big[(\mathtt{case}\ \mathtt{Mod}_{\omega}\ (\mathtt{Inl}\ ())\ \mathtt{of}\ \mathtt{Mod}_{\omega}\ x' \mapsto (x'\,,\ x'))\big] & \multimap\mathrm{EC} \quad \text{with substitution } x := \mathtt{Mod}_{\omega}\ (\mathtt{Inl}\ ()) \\
\longrightarrow & [\,]\big[(\mathtt{Inl}\ ()\,,\ \mathtt{Inl}\ ())\big] & !\mathrm{EC} \quad \text{with substitution } x' := \mathtt{Inl}\ ()
\end{array}
$$

Because we have a function in value form already, with simple body, applied to a value, we don't need to focus into a subterm and unfocus back later; we see only see a succession of contraction steps.

$\boxed{E\,[\,t\,] \longrightarrow E'\,[\,t'\,]}$		*(Small-step evaluation)*
$E\,[\,\mathsf{Inl}\,t\,] \longrightarrow (E \circ \mathsf{Inl}\,[\,])\,[\,t\,]$	$\star$	$\oplus\mathrm{I}_1\mathrm{F}$
$(E \circ \mathsf{Inl}\,[\,])\,[\,v\,] \longrightarrow E\,[\,\mathsf{Inl}\,v\,]$		$\oplus\mathrm{I}_1\mathrm{U}$
$E\,[\,\mathsf{Inr}\,t\,] \longrightarrow (E \circ \mathsf{Inr}\,[\,])\,[\,t\,]$	$\star$	$\oplus\mathrm{I}_2\mathrm{F}$
$(E \circ \mathsf{Inr}\,[\,])\,[\,v\,] \longrightarrow E\,[\,\mathsf{Inr}\,v\,]$		$\oplus\mathrm{I}_2\mathrm{U}$
$E\,[\,(t_1\,,\,t_2)\,] \longrightarrow (E \circ ([\,]\,,\,t_2))\,[\,t_1\,]$	$\star$	$\otimes\mathrm{IF}_1$
$(E \circ ([\,]\,,\,t_2))\,[\,v_1\,] \longrightarrow E\,[\,(v_1\,,\,t_2)\,]$		$\otimes\mathrm{IU}_1$
$E\,[\,(v_1\,,\,t_2)\,] \longrightarrow (E \circ (v_1\,,\,[\,]))\,[\,t_2\,]$	$\star$	$\otimes\mathrm{IF}_2$
$(E \circ (v_1\,,\,[\,]))\,[\,v_2\,] \longrightarrow E\,[\,(v_1\,,\,v_2)\,]$		$\otimes\mathrm{IU}_2$
$E\,[\,\mathsf{Mod}_n\,t\,] \longrightarrow (E \circ \mathsf{Mod}_n\,[\,])\,[\,t\,]$	$\star$	!IF
$(E \circ \mathsf{Mod}_n\,[\,])\,[\,v\,] \longrightarrow E\,[\,\mathsf{Mod}_n\,v\,]$		!IU
$E\,[\,t'\;t\,] \longrightarrow (E \circ t'\,[\,])\,[\,t\,]$	$\star$	$\multimap\mathrm{EF}_1$
$(E \circ t'\,[\,])\,[\,v\,] \longrightarrow E\,[\,t'\;v\,]$		$\multimap\mathrm{EU}_1$
$E\,[\,t'\;v\,] \longrightarrow (E \circ [\,]\;v)\,[\,t'\,]$	$\star$	$\multimap\mathrm{EF}_2$
$(E \circ [\,]\;v)\,[\,v'\,] \longrightarrow E\,[\,v'\;v\,]$		$\multimap\mathrm{EU}_2$
$E\,[\,(\boldsymbol{\lambda} x \,{}_m\!\mapsto u)\;v\,] \longrightarrow E\,[\,u[x := v]\,]$		$\multimap$EC
$E\,[\,t \fatsemi u\,] \longrightarrow (E \circ [\,] \fatsemi u)\,[\,t\,]$	$\star$	1EF
$(E \circ [\,] \fatsemi u)\,[\,v\,] \longrightarrow E\,[\,v \fatsemi u\,]$		1EU
$E\,[\,() \fatsemi u\,] \longrightarrow E\,[\,u\,]$		1EC
$E\,[\,\mathsf{case}_m\;t\;\mathbf{of}\;\{\mathsf{Inl}\,x_1 \mapsto u_1\,,\;\mathsf{Inr}\,x_2 \mapsto u_2\}\,] \longrightarrow (E \circ \mathsf{case}_m\;[\,]\;\mathbf{of}\;\{\mathsf{Inl}\,x_1 \mapsto u_1\,,\;\mathsf{Inr}\,x_2 \mapsto u_2\})\,[\,t\,]$	$\star$	$\oplus$EF
$(E \circ \mathsf{case}_m\;[\,]\;\mathbf{of}\;\{\mathsf{Inl}\,x_1 \mapsto u_1\,,\;\mathsf{Inr}\,x_2 \mapsto u_2\})\,[\,v\,] \longrightarrow E\,[\,\mathsf{case}_m\;v\;\mathbf{of}\;\{\mathsf{Inl}\,x_1 \mapsto u_1\,,\;\mathsf{Inr}\,x_2 \mapsto u_2\}\,]$		$\oplus$EU
$E\,[\,\mathsf{case}_m\;(\mathsf{Inl}\,v_1)\;\mathbf{of}\;\{\mathsf{Inl}\,x_1 \mapsto u_1\,,\;\mathsf{Inr}\,x_2 \mapsto u_2\}\,] \longrightarrow E\,[\,u_1[x_1 := v_1]\,]$		$\oplus\mathrm{EC}_1$
$E\,[\,\mathsf{case}_m\;(\mathsf{Inr}\,v_2)\;\mathbf{of}\;\{\mathsf{Inl}\,x_1 \mapsto u_1\,,\;\mathsf{Inr}\,x_2 \mapsto u_2\}\,] \longrightarrow E\,[\,u_2[x_2 := v_2]\,]$		$\oplus\mathrm{EC}_2$
$E\,[\,\mathsf{case}_m\;t\;\mathbf{of}\;(x_1\,,\,x_2) \mapsto u\,] \longrightarrow (E \circ \mathsf{case}_m\;[\,]\;\mathbf{of}\;(x_1\,,\,x_2) \mapsto u)\,[\,t\,]$	$\star$	$\otimes$EF
$(E \circ \mathsf{case}_m\;[\,]\;\mathbf{of}\;(x_1\,,\,x_2) \mapsto u)\,[\,v\,] \longrightarrow E\,[\,\mathsf{case}_m\;v\;\mathbf{of}\;(x_1\,,\,x_2) \mapsto u\,]$		$\otimes$EU
$E\,[\,\mathsf{case}_m\;(v_1\,,\,v_2)\;\mathbf{of}\;(x_1\,,\,x_2) \mapsto u\,] \longrightarrow E\,[\,u[x_1 := v_1][x_2 := v_2]\,]$		$\otimes$EC
$E\,[\,\mathsf{case}_m\;t\;\mathbf{of}\;\mathsf{Mod}_n\,x \mapsto u\,] \longrightarrow (E \circ \mathsf{case}_m\;[\,]\;\mathbf{of}\;\mathsf{Mod}_n\,x \mapsto u)\,[\,t\,]$	$\star$	!EF
$(E \circ \mathsf{case}_m\;[\,]\;\mathbf{of}\;\mathsf{Mod}_n\,x \mapsto u)\,[\,v\,] \longrightarrow E\,[\,\mathsf{case}_m\;v\;\mathbf{of}\;\mathsf{Mod}_n\,x \mapsto u\,]$		!EU
$E\,[\,\mathsf{case}_m\;\mathsf{Mod}_n\,v\;\mathbf{of}\;\mathsf{Mod}_n\,x \mapsto u\,] \longrightarrow E\,[\,u[x := v]\,]$		!EC

$\star$: only allowed if the term that would become the new focus is not already a value

Figure 1.9: Small-step semantics for λ_{Ldm} (λ_{Ldm}–sem)

A command of the form $[]\left[v\right]$ with v a value is the stopping point of the reduction for all well-typed programs. If we have a non-value term t in $[]\left[t\right]$ then a focusing or contraction step should trigger; and if we have a non-empty evaluation context E in $E\left[v\right]$ then an unfocusing step should trigger. Only ill-shaped term or value can cause the reduction to be stuck, which is a symptom of a wrongly-typed program.

1.7 Linear Types in Haskell

It's no surprise that we intend to implement prototypes in a industrial functional programming language along our journey in the destination-passing world. And Haskell is a suitable target for that, thanks to its support for linear types.

Linear Haskell [Ber+18] is a language extension for the Glasgow Haskell Compiler (GHC) that introduces the linear function arrow, `t ⊸ u` and modifies the type checker of GHC so that it can enforce linearity requirements. A linear function of type `t ⊸ u` guarantees that the argument of the function will be consumed exactly once when the result of the function is consumed exactly once. The regular function arrow `t → u` is still available when Linear Haskell is enabled; this one doesn't guarantee how many times its argument will be consumed when its result is consumed once. Actually Linear Haskell is based on a modal formalism, like λ_{Ldm}, so there is also a multiplicity-polymorphic arrow, `t %m → u`, much like $\mathtt{T}\ {}_{\mathtt{m}}\!\multimap \mathtt{U}$; so `t ⊸ u` and `t → u` are sugar for `t %1 → u` and `t %ω → u` respectively.

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. This is no more than a practical view of the typing rules given for λ_{Ldm}.

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 t` is the Haskell equivalent for `!T` in $\lambda_{L1,2}$ or $!_{\omega}\mathtt{T}$ in λ_{Ldm}, and there is an equivalence between `Ur t ⊸ u` and `t → u`. As in λ_{L2} and λ_{Ldm}, we can pattern-match on a term of type `Ur t` with `case term of Ur x ⊸ term'` to obtain an unrestricted variable binding `x` of type `t`.

As usual with linear type systems, 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. This echoes rules λ_{L1}–TY/!P or $\lambda_{L2,m}$–TY/!I where a term wrapped in `Many` or $\mathtt{Mod}_{\omega}$ must only depend on an unrestricted context. The only exceptions to this rule are terms of types that implement the `Movable` typeclass[5] such as `Int` or `()`. `Movable` provides the function `move :: t ⊸ Ur t`, that is, unconditional promotion to `Ur` for types implementing this typeclass.

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

`(⨟) :: () ⊸ u ⊸ u` is used to chain a linear operation returning `()` with one returning a value of type `u` without breaking linearity. It's the Haskell equivalent of λ_{Ldm}–TY/1E. It isn't part of the core Haskell syntax, but can be defined simply as `case term1 of () -> term2`.

[5] The `Movable` typeclass is not part of the Linear Haskell language extension, but instead defined in the `linear-base` library.

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

`@t` in `f @t ...` is a type application; it allows to specify an explicit type for polymorphic functions.

In code excerpts, types are in blue, while terms, variables, and data constructors (sometimes having the same name as the type they belong to, like `Ur` (type) / `Ur` (data constructor)) are in black, to stay as consistent as possible with previous chapters. Unfortunately, while type variables `T`, `U` are in uppercase in our formalism, we have no choice than to use lowercase letters in Haskell `t`, `u`.

1.8 Uniqueness and Linear Scopes

In the introduction we said that we will use linear type systems to control the number of times a resource –in particular a destination– is used. This is indeed a promising idea in order to forgo monads entirely when doing resource management, for a less contaminating programming style... or so it seems.

But linear type systems, at first, may seem insufficient to do that job fully. In a linear type system, we can only promise what we do to a value we receive. We cannot say much about what has been done to it previously. And thus, we cannot ensure it has been used exactly once *overall*.

For instance, given a function of type `t ⊸ u`, if we apply it to an argument of type `t`, we know we won't duplicate this value of type `t` (as long as we use the result of type `u` exactly once). But we can't say whether the `t` has been duplicated or not beforehand: that's a major issue.

Stating facts about *the past* of a value is the land of *uniqueness typing* or unique type systems. In a unique type system, the main property is not how many times a function uses a value to produce its output, but rather, whether a value has been aliased or not. Uniqueness (or non-uniqueness) is a property on values, while linearity is a property on functions[6]. Having uniqueness or non-uniqueness property on values allows for easier control of resources that should always stay unique during their lifetime. The most common example is efficient implementation of functional arrays. If an array is handled uniquely and isn't shared, then we can implement array updates as in-place memory mutations[MVO22]. By being able to state uniqueness of values, statically through the type system, we can allow predictable compiler optimizations.

The thing is, uniqueness typing doesn't have a theoretical ground as rich as linear types. Unlike linearity, uniqueness properties have first been studied on the computational side and only later studied on the logical side, with [Har06]. There are also two downsides of uniqueness typing for what we want to do later. First, there isn't an easy way with unique type system to force the use of something (and we definitely want a way to force a destination to be used and not just be dropped). We can state or prove that it isn't aliased, but that's all. Secondly, and more importantly perhaps, higher-order functions (capturing lambda abstractions) are way more finicky in unique type systems, as detailed in Section 5 of [Har06] or mentioned briefly in [MVO22] (that's one reason why they prefer a linear system as a basis with a uniqueness topping instead of a uniqueness system with a linearity topping).

[6]At least in the presentations above that follows from Girard's Linear Logic. There exists approaches of Linear Type systems that are more focused on the programming side, for which linearity is a property of values and their types, not functions.

Actually, Marshall, Vollmer, and Orchard [MVO22] argue that modern programming languages would benefit from integrating both linear and uniqueness typing, as linearity and uniqueness appear to be half-dual, half-complementary notions that together are enough for a very precise control of resource usage in a language.

In Chapter 3 however, our main target language will be Haskell, which doesn't have uniqueness typing, only linear types, as detailed in Section 1.7. Fortunately, we can emulate uniqueness control in a language equipped with linear types, using *linear scopes* or the *scope function trick*, as one might call it.

Linear scopes As we said before, in Linear Haskell, as in most linear type systems, functions can be made to use their arguments linearly or not. In other terms, linearity is a contract that links the input and output of a function. But we cannot say anything a priori about what happened to the value passed as a parameter before the function call, or what happens to the function result after. Linearity control can only be enforced on *variables*.

Let say we want to design a file API without monads (that's actually one interesting use of linear/unique types). Assuming the following simplified API, one could write this small program:

```
openFile :: String ⊸ File
closeFile :: File ⊸ ()

file :: File  -- no way to force this value to be used exactly once
file = openFile "test.txt"

nonLinearUseButValid :: () =
  closeFile file ; closeFile file  -- valid even if file is consumed twice
```

With call-by-value or call-by-need semantics, the side-effects of `openFile` will be produced only once, while the side effects of `closeFile` will be produced twice, resulting in an error.

The solution to that is to forbid the consumer from creating and directly accessing a value of the resource type for which we want to enforce linearity/uniqueness. Instead, we force the consumer of a resource to pass a continuation representing what they want to do on the resource, so that we can check through its signature that it is indeed a linear continuation:

```
withFile :: String ⊸ (File ⊸ t) ⊸ t
closeFile :: File ⊸ ()
nonLinearUseAndEffectivelyRejected :: () =
  withFile "test.txt" (\file → closeFile file ; closeFile file) -- not linear
```

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

Still, this is not enough; because `\file → file` is indeed a linear continuation, one could use `withFile "test.txt" (\file → file)` to leak a `File`, and then use it in a non-linear fashion in the outside world. Hence we must forbid `File` from appearing anywhere in the return type of the continuation.

To do that, we ask the return type to be wrapped in the unrestricted modality `Ur` (the equivalent in the Linear Haskell realm to the `!` modality): because the value of type `File` comes from the left of a linear arrow, it cannot be promoted to an `Ur` without breaking linearity, so it cannot leave the scope and has to be used exactly once in this scope. That's why we speak of *linear scopes*. Here's the updated example:

```
withFile' :: String ⊸ (File ⊸ Ur t) ⊸ Ur t
closeFile :: File ⊸ ()
exampleOk :: Ur () = withFile' "test.txt" (\file → closeFile file ; Ur ())
exampleFail :: Ur File =
  withFile' "test.txt" (\file → Ur file) -- leaking file, effectively rejected
```

Still, this solution is not perfect. We effectively prevent any linear resource from leaving the scope, even resources that have nothing to do with the file API and have been created in outer scopes! This issue is talked in length by Spiwack [Spi23b]. It manifests itself in the following example, which gets rejected:

```
readLine :: File ⊸ (Ur String, File)
writeLine File ⊸ String ⊸ File

exampleFail' :: Ur () =
  withFile' "test.txt" (\file1 →
    ...
    withFile' "test2.txt" (\file2 →
      let (Ur line, file1') = readLine file1
          file2' = writeLine file2 line
       in closeFile file2' ; Ur file1')
    ... -- other operations with file1' here
  )
```

The problem here is that we cannot return `file1'` from the inner scope, because it's a linear resource that cannot be promoted to `Ur`. However, it would be perfectly safe to do so; it's just that we don't have precise enough tools at type level to convince the compiler of this fact.

In that case, the most obvious workaround is to extend the scope of `withFile' "test2.txt" (file2 → ...)` as long as we need to operate on `file1'`. Very unfortunate and unpractical indeed to have to prolongate the inner scope because we still need to operate on resources from the outer one![7]

One easy improvement we can do is to have a single linear scope, providing a linear token type, and let this token type be used to instantiate any resource for which we want to enforce linearity/uniqueness. That way, we can just have one big linear scope, and write most of the code in direct style, still without monads!

[7] This is what is referred to as *sticky end of linear scopes* in [Spi23b]

```
type Token
dup :: Token ⊸ (Token, Token)
drop :: Token ⊸ ()
withToken :: (Token ⊸ Ur t) ⊸ Ur t

-- A first linear API
openFile' :: String ⊸ Token ⊸ File
closeFile :: File ⊸ ()

-- A second linear API
openDB :: String ⊸ String ⊸ Token ⊸ DB
closeDB :: DB ⊸ ()

exampleOk1 :: Ur () =
  withToken (\tok → let file = openFile' "test.txt" tok in closeFile file ; Ur ())
exampleOk2 :: Ur () =
  withToken (\tok → let (tok1, tok2) = dup tok
                        file = openFile' "test.txt" tok1
                        db = openDB "localhost" 3306 tok2
                    in closeFile file ; closeDB db ; Ur ())
exampleRejected :: Ur File =
  withToken (\tok →  Ur (openFile' "test.txt" tok)) -- leaking file which linearly depends on tok
```

What might be surprising here is that we are allowed to duplicate or drop a token (with `dup` or `drop`), which we announced as a linear resource!

In fact, the goal of the `Token` type is to tie every resource to a linearity requirement. The important part is that the `Token` type should not have a way to be promoted to `Ur`, otherwise tokens and other linear resources could leak and live outside of the linear scope of `withToken`.

The linear `dup` function creates two tokens out of one, but still tie the resulting tokens to the input one through a linear arrow. Allowing to duplicate tokens doesn't allow the consumer to duplicate the resources we want to control (files, database handles, etc.). So we're safe on that front.

With this approach, dealing with multiple linear resource types is much more uniform. We can use any remaining token to instantiate one or several new linear resources. As we only have one big linear scope, with every linear resource in it, there is no issue with nested scopes that need to be extended. When a program is sliced into many sub-functions, these functions don't even have to deal or know about the linear scope; they just have to pass around and receive tokens, an can otherwise be written in direct style.

Although it's out of scope for this document, Spiwack et al. [Spi+22] goes further on the idea of linear tokens and scopes, with *Linear constraints*, which are roughly a way to pass tokens around in an implicit fashion, to make programming with linear types even more convenient[8].

In next chapters, we will reuse the *linear scope with tokens* strategy to enforce linearity and uniqueness on linear resources, to allow safe effects in pure, direct-style APIs.

[8] At the time of writing this document, Linear Constraints are seriously considered to be added into GHC, the main Haskell compiler[Spi23a].

1.9 Conclusion

In this chapter we introduced the linear λ-calculus, from its roots in linear logic, to more practical and computation-centric presentations. In particular, we laid theoretical foundations for a modal linear λ-calculus, which will be reused in the rest of this document.

Furthermore, we explored Linear Haskell, as a concrete target for experimenting with linear types. In particular, we illustrated how linear types can help with managing unique resources, that must be used but also must not be duplicated, even though it requires a bit of care to avoid the inherent limitations of linear types (in that, they can't guarantee per se what happened to a value in the past).

We are now in comfortable ground and have all the tool needed to explore both theoretical and practical aspect of functional destination passing.

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 develop here, namely λ_d. Let's see why destination passing is valuable in a pure functional programming context, and how λ_d constitutes a good foundational layer for this.

The results presented in this chapter have also been published at OOPSLA1 2025 conference[BS25].

2.1 Destination-passing style programming, in a functional setting

In destination-passing style, a function doesn't return a value: it takes as an argument a location —named *destination*— 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 $\mathtt{T} \to \lfloor \mathtt{U} \rfloor \to \mathtt{1}$ when transformed into destination-passing style. Instead of returning an $\mathtt{U}$, it consumes a destination of type $\lfloor \mathtt{U} \rfloor$ and returns just *unit* (type $\mathtt{1}$). This style is common in systems 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 systems programs rely on destinations so much is that using destinations can save calls to the memory allocator, which are costly, in a context where every gain of performance is highly valuable. 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 (though it might be slightly more verbose); there are 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 modify a standard functional programming language just enough to be able to build immutable structures by destination passing without endangering purity and memory safety. Destinations in this 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 contribute to 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 not only that the order in which a structure is built is arbitrary, but also that this order can be determined dynamically during the runtime of the program. Here is the main novelty of our approach compared to earlier work.

In this chapter, we introduce λ_d, 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 subsumes 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. Indeed, we proved type safety theorems for λ_d with the Coq proof assistant, as described in Section 2.7.

2.2 Working with destinations in λ_d

Let's introduce and get familiar with λ_d, a modal, linear, simply typed λ-calculus with destinations. We borrow most of the syntax of the previous chapter, especially from λ_{Ldm}. We still use linear logic's $\mathtt{T} \oplus \mathtt{U}$ and $\mathtt{T} \otimes \mathtt{U}$ 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 $\mathtt{Inl}\, t$ 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. For instance, the following is a destination-passing version of the identity function:

```
dId : T ⊸ ⌊T⌋ ⊸ 1
dId x d ≜ d ◀ x
```

We can 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.

Performing $d \blacktriangleleft x$ is the simplest way to use a destination: with that we fill it with a value directly. But we can also fill a destination piecewise, by specifying just the outermost constructor that we want to fill it with:

```
fillWithInl : ⌊T⊕U⌋ ⊸ ⌊T⌋
fillWithInl d ≜ d ◁ Inl
```

In this example, we're filling a destination for type `T⊕U` by setting the outermost constructor to left variant `Inl`. We think of $d \triangleleft$ `Inl` (read "fill d with `Inl`") as allocating memory to store a block of the form `Inl` □, and write the address of that block to the location that d points to. Because we've just created an `Inl` constructor with no argument yet, we return a new destination of type `⌊T⌋` pointing to the uninitialized argument of `Inl`. Uninitialized memory, when part of a structure or value, like □ in `Inl` □, 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 `Inl` □ into a hole, but we have yet to describe what goes in the new hole □. Such data constructors with uninitialized arguments are called *hollow constructors*[9]. This is opposite to how functional programming usually works, where values are built from the innermost constructors outward: first we build a value v and only then can we use `Inl` to make `Inl` v. Being able to build structures from the outermost constructor inward is a key ingredient to the extra expressiveness that we promised 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 2.9). But where destinations really differ from continuations is in the case where 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:

```
fillWithPair : ⌊T⊗U⌋ ⊸ ⌊T⌋⊗⌊U⌋
fillWithPair d ≜ d ◁ (,)
```

After using **`fillWithPair`**, both the first field *and* the second field of the pair must be filled, using the destinations of type `⌊T⌋` and `⌊U⌋` respectively. If we were to replace destinations by continuations in **`fillWithPair`**, we wouldn't be able to use the two returned continuations easily.

[9]The full triangle ◀ is used to fill a destination with a fully-formed value, while the *hollow* triangle ◁ is used to fill a destination with a *hollow constructor*.

We can already note there that there is a sort of duality between destination-filling operators and associated constructors. Usual `Inl` constructor has signature $\mathtt{T} \multimap \mathtt{T} \oplus \mathtt{U}$, while destination-filling $\triangleleft\mathtt{Inl}$ has signature $\lfloor \mathtt{T} \oplus \mathtt{U} \rfloor \multimap \lfloor \mathtt{T} \rfloor$. Similarly, pair constructor (,) has signature $\mathtt{T} \multimap \mathtt{U} \multimap \mathtt{T} \otimes \mathtt{U}$, while destination-filling $\triangleleft(,)$ has signature $\lfloor \mathtt{T} \otimes \mathtt{U} \rfloor \multimap \lfloor \mathtt{T} \rfloor \otimes \lfloor \mathtt{U} \rfloor$. Assuming some flexibility with currying, we see that the types of arguments and results switch sides around the arrow, and get wrapped/unwrapped from $\lfloor \bullet \rfloor$ 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 $\lfloor \mathtt{T} \rfloor$ only indicates that some bigger structure has at least a hole of type $\mathtt{T}$ 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 $\mathtt{T} \otimes \mathtt{U}$ 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 $\mathtt{S} \ltimes \lfloor \mathtt{T} \rfloor$ for a structure of type $\mathtt{S}$ missing a value of type $\mathtt{T}$ to be complete. There can be several holes in $\mathtt{S}$ — resulting in several destinations on the right hand side — and as long as there remain holes in $\mathtt{S}$, it cannot be read. For instance, $\mathtt{S} \ltimes (\lfloor \mathtt{T} \rfloor \otimes \lfloor \mathtt{U} \rfloor)$ represents an $\mathtt{S}$ that misses both a $\mathtt{T}$ and a $\mathtt{U}$ 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 $\mathtt{S} \ltimes \mathtt{T}$ is read "$\mathtt{S}$ ampar $\mathtt{T}$". The name "ampar" stands for "asymmetric memory par". The "par" $\parr$ 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: $\mathtt{T} \multimap \mathtt{S}$ is equivalent to $\mathtt{T}^\perp \parr \mathtt{S}$ or $\mathtt{S} \parr \mathtt{T}^\perp$. Similarly, $\mathtt{T}_1 \multimap \mathtt{T}_2 \multimap \mathtt{S}$ is equivalent to $\mathtt{T1}^\perp \parr \mathtt{T2}^\perp \parr \mathtt{S}$. Function input's types —which are in negative position— are wrapped in the dualizing operator $\bullet^\perp$ when a function is put into "par" form. Here we take a similar approach: Minamide [Min98] first observed that structures with holes are akin to linear functions $\mathtt{T} \multimap \mathtt{S}$, where $\mathtt{T}$ represents the missing part; so we decide to represent them in a more flexible fashion under the form $\mathtt{S} \ltimes \lfloor \mathtt{T} \rfloor$. In this presentation, the *missing part of type* $\mathtt{T}$ appears as a first-class type $\lfloor \mathtt{T} \rfloor$, a.k.a the *destination* type. The asymmetric nature of our memory par is a bit disappointing, compared to usual linear logic, but it comes from limitations that we are working in an intuitionistic setting, not a classical one like $\parr$ 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 that lives in the right side of the corresponding ampar. To access the destinations of an ampar, λ_d provides a $\mathtt{upd}_\ltimes$ construction, which lets us apply a function to the right-hand side of the ampar. It is in the body of $\mathtt{upd}_\ltimes : \mathtt{S} \ltimes \mathtt{T} \multimap (\mathtt{T} \multimap \mathtt{U}) \multimap \mathtt{S} \ltimes \mathtt{U}$ that functions operating on destinations can be called to update the structure:

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

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 somewhat similar to the linear 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[10] $\mathtt{from}'_{\ltimes} : \mathtt{S} \ltimes \mathtt{1} \multimap \mathtt{S}$: 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 data structure.

Equipped with these new operators, we can finally show how to 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 Inl and (,), but the other constructors are derived similarly. Operator $\mathbin{⨟} : \mathtt{1} \multimap \mathtt{U} \multimap \mathtt{U}$, present in second example, is used to chain operations returning unit type $\mathtt{1}$. For easier reading, we also provide type annotation when $\mathtt{new}_{\ltimes}$ is used:

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

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:

  $$\mathtt{forget} : \mathtt{T}$$
  $$\mathtt{forget} \triangleq \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes}\ (\mathtt{new}_{\ltimes} : \mathtt{T} \ltimes \lfloor \mathtt{T} \rfloor)\ \mathtt{with}\ d \mapsto ())$$

  then reading the result of `forget` would lead to reading a location pointed to by a destination that we never used, in other words, reading uninitialized memory. This must be prevented at all cost;
- if destinations are written several times, as in:

  $$\mathtt{ambiguous1} : \mathtt{Bool}$$
  $$\mathtt{ambiguous1} \triangleq \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes}\ (\mathtt{new}_{\ltimes} : \mathtt{Bool} \ltimes \lfloor \mathtt{Bool} \rfloor)\ \mathtt{with}\ d \mapsto d \blacktriangleleft \mathtt{true} \mathbin{⨟} d \blacktriangleleft \mathtt{false})$$
  $$\mathtt{ambiguous2} : \mathtt{Bool}$$
  $$\mathtt{ambiguous2} \triangleq \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes}\ (\mathtt{new}_{\ltimes} : \mathtt{Bool} \ltimes \lfloor \mathtt{Bool} \rfloor)\ \mathtt{with}\ d \mapsto \mathtt{let}\ x := (d \blacktriangleleft \mathtt{false})\ \mathtt{in}\ d \blacktriangleleft \mathtt{true} \mathbin{⨟} x)$$

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

then, we have **ambiguous1** that returns `false` and **ambiguous2** that returns `true` due to evaluation order, so we break *let expansion* that is supposed to be valid in a functional programming language (including 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 this, let's get more familiar with λ_d through more complex examples.

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 (though this isn't included in the formal system of Section 2.5).

Linked lists We define lists as a fixpoint, as usual: $\mathtt{List\ T} \triangleq \mathtt{1} \oplus (\mathtt{T} \otimes (\mathtt{List\ T}))$. 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:

$$\triangleleft[] \ : \ \lfloor \mathtt{List\ T} \rfloor \multimap \mathtt{1}$$
$$d \triangleleft [] \triangleq d \triangleleft \mathsf{Inl} \triangleleft ()$$

$$\triangleleft(::) \ : \ \lfloor \mathtt{List\ T} \rfloor \multimap \lfloor \mathtt{T} \rfloor \otimes \lfloor \mathtt{List\ T} \rfloor$$
$$d \triangleleft (::) \triangleq d \triangleleft \mathsf{Inr} \triangleleft (,)$$

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

$$(::) \ : \ \mathtt{T} \otimes (\mathtt{List\ T}) \multimap \mathtt{List\ T}$$
$$x :: xs \triangleq \mathtt{from}'_{\ltimes}\,(\mathtt{upd}_{\ltimes}\,(\mathtt{new}_{\ltimes} : (\mathtt{List\ T}) \ltimes \lfloor \mathtt{List\ T} \rfloor)\ \mathtt{with}\ d \mapsto \mathtt{case}\,(d \triangleleft (::))\ \mathtt{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft x \fatsemi dxs \blacktriangleleft xs)$$

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 order, then reverse it. But thanks to destinations, we are 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 **map′**, it has type $(\mathtt{T} \multimap \mathtt{U}) \multimap_{\omega} \mathtt{List\ T} \multimap \lfloor \mathtt{List\ U} \rfloor \multimap \mathtt{1}$[11]. 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, **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 **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[12].

[11] In fact, we'll develop a more complex system of modes starting from Section 2.5 to ensure scope control of destinations, and thus the proper signature will be $(\mathtt{T} \multimap \mathtt{U}) \multimap_{\omega\infty} \mathtt{List\ T} \multimap_{1\uparrow} \lfloor \mathtt{List\ U} \rfloor \multimap \mathtt{1}$.

[12] We consider data structures in λ_d to be immutable thanks to 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.

```
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

To obtain the regular `map` function, all is left to do is to call $\texttt{new}_{\ltimes}$ to create an initial destination, and $\texttt{from}'_{\ltimes}$ 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 of the previous works, 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 structure with a (single) hole. We'll slowly works towards an example which uses first-class destinations at their full potential.

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 this complexity is to use *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 $\boldsymbol{\lambda} ys \mapsto x :: ys$, and concatenation of difference lists is function composition. A difference list 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 $x::\square$, 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.

```
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

Efficient queue using difference lists In an immutable functional language, a queue can be implemented as a pair of lists (*front* , *back*) [HM81]. The list *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.

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 made 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 $\lfloor\lfloor \mathsf{T} \rfloor\rfloor$: the type of a destination pointing to a hole where a destination is expected. Think of it as an equivalent of the pointer type $\mathsf{T} **$ 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 $d : \lfloor \mathsf{T} \rfloor$ and $dd : \lfloor\lfloor \mathsf{T} \rfloor\rfloor$, we can form $dd \blacktriangleleft d$: 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 $dd \blacktriangleleft d$ is rejected since destinations (like d here) can only be used linearly. This choice is fairly limiting, as it would prevent us from storing destinations in structures with holes, as we will do, crucially, in Section 2.4.
- If we do not count this use of d at all, we can write $dd \blacktriangleleft d \fatsemi d \blacktriangleleft v$ 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 at the end of Section 2.2.1.

So linear use it is. But this creates a problem: there's no way, within our linear type system, to distinguish between "a destination that has been used on the left of a triangle so its corresponding hole has been filled" and "a destination that 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. A reduction step goes from a pair $\mathcal{S}|t$ of a store and a term to a new pair $\mathcal{S}'|t'$, and store are composed of named bindings of the form $\{l := v\}$ with v a value that may contain holes. We'll also need the $\mathbf{alloc} : (\lfloor \mathsf{T} \rfloor \multimap \mathbf{1}) \multimap \mathsf{T}$ 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. $\mathbf{alloc}\ f$ corresponds, in type and behavior, to $\mathbf{from}'_{\ltimes}\,(\mathbf{upd}_{\ltimes}\ \mathbf{new}_{\ltimes}\ \mathbf{with}\ d \mapsto f\ d)$.

In the following snippets, structures with holes are given names l and ld in the store; holes are given names too and denoted by $\boxed{h}$ and $\boxed{hd}$, and concrete destinations are denoted by $\to h$ and $\to hd$.

When the building scope of $l : \mathsf{Bool}$ is parent to the one of $ld : \lfloor \mathsf{Bool} \rfloor$, everything works well because ld, that contains destination pointing to $\boxed{h}$, has to be consumed before l can be read. In other terms, storing an older destination into a fresher one is not a problem:

$$
\begin{array}{rr|l}
 & \{\ \} & \mathbf{alloc}\ (\lambda d \mapsto (\mathbf{alloc}\ (\lambda dd \mapsto dd \blacktriangleleft d) : \lfloor \mathsf{Bool} \rfloor) \blacktriangleleft \mathsf{true}) \\
\longrightarrow & \{l := \boxed{h}\} & (\mathbf{alloc}\ (\lambda dd \mapsto dd \blacktriangleleft \to h) : \lfloor \mathsf{Bool} \rfloor) \blacktriangleleft \mathsf{true} \fatsemi \mathbf{deref}\ l \\
\longrightarrow & \{l := \boxed{h}, ld := \boxed{hd}\} & (\to hd \blacktriangleleft \to h \fatsemi \mathbf{deref}\ ld) \blacktriangleleft \mathsf{true} \fatsemi \mathbf{deref}\ l \\
\longrightarrow & \{l := \boxed{h}, ld := \to h\} & \mathbf{deref}\ ld \blacktriangleleft \mathsf{true} \fatsemi \mathbf{deref}\ l \\
\longrightarrow & \{l := \boxed{h}\} & \to h \blacktriangleleft \mathsf{true} \fatsemi \mathbf{deref}\ l \\
\longrightarrow & \{l := \mathsf{true}\} & \mathbf{deref}\ l \\
\longrightarrow & \{\ \} & \mathsf{true}
\end{array}
$$

However, when ld's scope is parent to l's, i.e. when a newer destination is stored into an older one, then we can write a linearly typed yet unsound program:

$$
\begin{array}{rr|l}
 & \{\ \} & \mathbf{alloc}\ (\lambda dd \mapsto \mathbf{case}\ (\mathbf{alloc}\ (\lambda d \mapsto dd \blacktriangleleft d) : \mathsf{Bool})\ \mathbf{of}\ \{\mathsf{true} \mapsto ()\,,\ \mathsf{false} \mapsto ()\}) \\
\longrightarrow & \{ld := \boxed{hd}\} & \mathbf{case}\ (\mathbf{alloc}\ (\lambda d \mapsto \to hd \blacktriangleleft d) : \mathsf{Bool})\ \mathbf{of}\ \{\mathsf{true} \mapsto ()\,,\ \mathsf{false} \mapsto ()\} \fatsemi \mathbf{deref}\ ld \\
\longrightarrow & \{ld := \boxed{hd}, l := \boxed{h}\} & \mathbf{case}\ (\to hd \blacktriangleleft \to h \fatsemi \mathbf{deref}\ l)\ \mathbf{of}\ \{\mathsf{true} \mapsto ()\,,\ \mathsf{false} \mapsto ()\} \fatsemi \mathbf{deref}\ ld \\
\longrightarrow & \{ld := \to h, l := \boxed{h}\} & \mathbf{case}\ (\mathbf{deref}\ l)\ \mathbf{of}\ \{\mathsf{true} \mapsto ()\,,\ \mathsf{false} \mapsto ()\} \fatsemi \mathbf{deref}\ ld \\
\longrightarrow & \{ld := \to h\} & \mathbf{case}\ \boxed{h}\ \mathbf{of}\ \{\mathsf{true} \mapsto ()\,,\ \mathsf{false} \mapsto ()\} \fatsemi \mathbf{deref}\ ld \qquad \text{💣💣💣}
\end{array}
$$

Here the expression $dd \blacktriangleleft d$ results in d escaping its scope for the parent one, so l 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 that, 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.

General considerations on linearity, scopes, and references Scope escape, that we just described, is not an issue specific to destinations. In fact, it seems that any form of storage, be it destinations, or `refs` *à la* OCaml, or anything that can be written to, is susceptible to let scope escape happens if we give it a linear write interface.

Most of the time, linear interfaces rely on the fact that there is a specific set of functions available acting as destructors for a given resource type, and we want to be sure that one of them is called for each resource at some point. Linearity forces us to consume the resource in the scope it is made available, and if the resource type is opaque, then the only way to consume it is by eventually calling one of these destructors. Except if we are given linear storage, in which case we can consume the resource by storing it away; this is the essence of scope escape.

2.4 Breadth-first tree traversal

As a full-fledged example, which uses the full expressive power of λ_d, we focus on 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* $1 \dots |T|$ *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's system[Min98] 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` has been defined, unsurprisingly, as `Tree T` $\triangleq$ `1⊕(T⊗((Tree T)⊗(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 numbers. 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.

```
go : (S ω∞⊸ T1 ⊸ (!ω∞S)⊗T2) ω∞⊸ S ω∞⊸ Queue (Tree T1⊗⌊Tree T2⌋) ⊸ 1
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
                    (Mod_ω∞ 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 ⨟ (Mod_ω∞ (succ st) , st)) 1 tree
```

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

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, otherwise we run into the issues of scope escape described in the previous section. We'll combine both linearity and the aforementioned ages system in the same *mode* *ringoid*[13]. You already know the linearity annotations 1 and ω from Chapter 1; 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 ringoid; in particular they are linear, and can capture destinations.

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} described in Chapter 1. It also features a mode-polymorphic function arrow $_m\!\multimap$. Its most distinctive features however are the destination $\lfloor_m \mathsf{T}\rfloor$ and ampar $\mathsf{S} \bowtie \mathsf{T}$ types which we've introduced in Sections 2.2 to 2.4. Don't pay too much attention to the mode annotation on the destination type; so far it was hidden in the examples because we only considered linear destinations.

[13] We said earlier that the modal approach would make the type system easier to extend, here is the proof.

Core grammar of terms:

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

Syntactic sugar for terms:

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

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

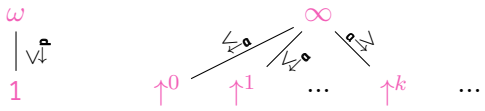
Grammar of types, modes and contexts:

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

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

Ordering on modes:

$$\mathsf{p}\mathsf{a} \leqslant \mathsf{p}'\mathsf{a}' \iff \mathsf{p} \leqslant_{\mathsf{p}} \mathsf{p}' \wedge\ \mathsf{a} \leqslant_{\mathsf{a}} \mathsf{a}'$$



Operations on modes:

+	1	ω
1	ω	ω
ω	ω	ω

·	1	ω
1	1	ω
ω	ω	ω

$$(\mathsf{p}\mathsf{a}) \cdot (\mathsf{p}'\mathsf{a}') \triangleq (\mathsf{p} \cdot \mathsf{p}')(\mathsf{a} \cdot \mathsf{a}')$$

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

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

$$(\mathsf{p}\mathsf{a}) + (\mathsf{p}'\mathsf{a}') \triangleq (\mathsf{p} + \mathsf{p}')(\mathsf{a} + \mathsf{a}')$$

Operations on typing contexts:

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

Figure 2.4: Terms, types and modes of λ_d

To ensure that destinations are used soundly, we need both to enforce the linearity of destinations 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 scopes 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 $\mathsf{upd}_{\ltimes}\ t\ \mathsf{with}\ x \mapsto t'$ construct. For instance, if we have a term $\mathsf{upd}_{\ltimes}\ t_1\ \mathsf{with}\ x_1 \mapsto \mathsf{upd}_{\ltimes}\ t_2\ \mathsf{with}\ x_2 \mapsto \mathsf{upd}_{\ltimes}\ t_3\ \mathsf{with}\ x_3 \mapsto x_1$, then we say that the innermost occurrence of x_1 has age $\uparrow^2$ because two nested $\mathsf{upd}_{\ltimes}$ 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 used in Sections 2.2 to 2.4.

2.5.1 Modes and the age ringoid

Our mode ringoid (more precisely, a commutative additive semigroup $+$ and a multiplicative monoid $(\cdot, 1)$, with the usual distributivity law $\mathsf{n}\cdot(\mathsf{m}_1 + \mathsf{m}_2) = \mathsf{n}\cdot\mathsf{m}_1 + \mathsf{n}\cdot\mathsf{m}_2$, and equipped with a partial order $\leq$, 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 $\mathsf{upd}_{\ltimes}\ t\ \mathsf{with}\ x \mapsto t'$ 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. Indeed, it's crucial that λ_d tracks the precise age of variables, and usually, ordering on modes usually implies a form of subtyping (where a variable having mode n can be used in place where a variable with mode m is expected if $\mathsf{m} \leq \mathsf{n}$). Here we don't want variables from two scopes ago to be used as if they were from one scope ago. The ordering reflects this with finite ages being arranged in a flat order (so $\uparrow^1$ and $\uparrow^2$ aren't comparable), 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 on modes are lifted to typing contexts, still following the insights from [GS14] (see Figure 2.4).

Anticipating on next section, let's see how *plus* works on context, modes, and especially ages. Let's type the expression $d \blacktriangleleft x \fatsemi x$. Most of the typing here behaves the same as λ_{Ldm}–TY, the only particularity is that the operator $\blacktriangleleft$ scales the typing context of its right operand by $\uparrow$ (which corresponds to asking it to be from the previous scope, to prevent scope escape of destinations). We have the following typing tree:

$$\dfrac{\dfrac{\dfrac{1\nu \leqslant 1\nu}{d :_{1\nu} \lfloor \mathsf{T} \rfloor \vdash d : \lfloor \mathsf{T} \rfloor}\,\mathrm{ID_V} \quad \dfrac{1\nu \leqslant 1\nu}{x :_{1\nu} \mathsf{T} \vdash x : \mathsf{T}}\,\mathrm{ID_V}}{d :_{1\nu} \lfloor \mathsf{T} \rfloor,\ x :_{1\uparrow} \mathsf{T} \vdash d \blacktriangleleft x : 1}\,\lfloor\,\rfloor\mathrm{E_L} \quad \dfrac{1\nu \leqslant 1\nu}{x :_{1\nu} \mathsf{T} \vdash x : \mathsf{T}}\,\mathrm{ID_V}}{d :_{1\nu} \lfloor \mathsf{T} \rfloor,\ x :_{\omega\infty} \mathsf{T} \vdash d \blacktriangleleft x \fatsemi x : \mathsf{T}}\,1\mathrm{E}$$

The "default choice" is to type variables with mode 1ν at the leaves of the tree. At the root of the tree, with operator $\fatsemi$, the typing context of the two initial branches are summed, so we have $(d :_{1\nu} \lfloor \mathsf{T} \rfloor,\ x :_{1\uparrow} \mathsf{T}) + (x :_{1\nu} \mathsf{T}) = d :_{1\nu} \lfloor \mathsf{T} \rfloor,\ x :_{(1+1)(\nu+\uparrow)} \mathsf{T} = d :_{1\nu} \lfloor \mathsf{T} \rfloor,\ x :_{\omega\infty} \mathsf{T}$. In other terms, because we need x to be of age $\uparrow$ in the first branch, but of age ν in the second branch —as, we recall, $\nu = \uparrow^0$ and $\uparrow = \uparrow^1$ are *not* comparable, so we cannot use the rule λ_d–TY/ID$_\mathrm{V}$ on a variable with mode $1\uparrow$— then x must be of age ∞[14].

Alternatively, we can have:

$$\dfrac{\dfrac{\dfrac{1\nu \leqslant 1\nu}{d :_{1\nu} \lfloor \mathsf{T} \rfloor \vdash d : \lfloor \mathsf{T} \rfloor}\,\mathrm{ID_V} \quad \dfrac{1\nu \leqslant \omega\infty}{x :_{\omega\infty} \mathsf{T} \vdash x : \mathsf{T}}\,\mathrm{ID_V}}{d :_{1\nu} \lfloor \mathsf{T} \rfloor,\ x :_{\omega\infty} \mathsf{T} \vdash d \blacktriangleleft x : 1}\,\lfloor\,\rfloor\mathrm{E_L} \quad \dfrac{1\nu \leqslant \omega\infty}{x :_{\omega\infty} \mathsf{T} \vdash x : \mathsf{T}}\,\mathrm{ID_V}}{d :_{1\nu} \lfloor \mathsf{T} \rfloor,\ x :_{\omega\infty} \mathsf{T} \vdash d \blacktriangleleft x \fatsemi x : \mathsf{T}}\,1\mathrm{E}$$

where x is given mode $\omega\infty$ from the leaves to the root. We can make use of a variable with mode $\omega\infty$ because we have $1\nu \leqslant \omega\infty$, which matches the premise of rule λ_d–TY/ID$_\mathrm{V}$.

2.5.2 Typing rules

Core typing rules of λ_d are given in Figure 2.5. Rules for elimination of $\multimap$, 1, $\oplus$, $\otimes$ and $!_\mathsf{m}$ 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 $!_\mathsf{m}$, are derived from elimination rules of destinations, and thus are not part of the core language. They are presented on the distinct Figure 2.6. The introduction form for functions λ_d–TY$_\mathrm{S}$/$\multimap$I is also derived from elimination form λ_d–TY/$\lfloor\multimap\rfloor$E for the corresponding destinations of function, although functions are not purely "data".

Rules λ_d–TY/ID$_\mathrm{V}$, λ_d–TY/$\ltimes$NEW and λ_d–TY$_\mathrm{S}$/1I, 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}–TY/1I, except that Γ is scaled by $\omega\nu$ instead of ω now.

[14] Age requirements are only there for scope-sensitive resources, that is, just destinations and the structures they are stored in. Age ∞ is meant to designate non scope-sensitive resources, and consequently, is also the result of incompatible age requirements (as only non scope-sensitive resources can fit incompatible age requirements).

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

$$\frac{1\nu \leq \mathsf{m}}{\omega\nu \cdot \Gamma,\ x :_{\mathsf{m}} \mathsf{T} \vdash x : \mathsf{T}}\ \mathrm{ID_V} \qquad \frac{\Gamma_1 \vdash t : \mathsf{T} \qquad \Gamma_2 \vdash t' : \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash t'\ t : \mathsf{U}}\ \multimap\mathrm{E} \qquad \frac{\Gamma_1 \vdash t : \mathsf{1} \qquad \Gamma_2 \vdash u : \mathsf{U}}{\Gamma_1 + \Gamma_2 \vdash t \fatsemi u : \mathsf{U}}\ \mathsf{1}\mathrm{E}$$

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

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \\ \Gamma_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u : \mathsf{U}\end{array}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathsf{case}_{\mathsf{m}}\ t\ \mathsf{of}\ (x_1\ ,\ x_2) \mapsto u : \mathsf{U}}\ \otimes\mathrm{E} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash t : !_{\mathsf{n}} \mathsf{T} \\ \Gamma_2,\ x :_{\mathsf{m} \cdot \mathsf{n}} \mathsf{T} \vdash u : \mathsf{U}\end{array}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathsf{case}_{\mathsf{m}}\ t\ \mathsf{of}\ \mathsf{Mod}_{\mathsf{n}}\, x \mapsto u : \mathsf{U}}\ !\mathrm{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{U} \ltimes \mathsf{T} \\ 1\uparrow \cdot \Gamma_2,\ x :_{1\nu} \mathsf{T} \vdash t' : \mathsf{T}'\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathsf{upd}_{\ltimes}\ t\ \mathsf{with}\ x \mapsto t' : \mathsf{U} \ltimes \mathsf{T}'}\ \ltimes\mathrm{UPD} \qquad \frac{\Gamma \vdash u : \mathsf{U}}{\Gamma \vdash \mathsf{to}_{\ltimes}\ u : \mathsf{U} \ltimes \mathsf{1}}\ \ltimes\mathrm{TO}$$

$$\frac{\Gamma \vdash t : \mathsf{U} \ltimes (!_{1\infty} \mathsf{T})}{\Gamma \vdash \mathsf{from}_{\ltimes}\ t : \mathsf{U} \otimes (!_{1\infty} \mathsf{T})}\ \ltimes\mathrm{FROM} \qquad \frac{}{\omega\nu \cdot \Gamma \vdash \mathsf{new}_{\ltimes} : \mathsf{T} \ltimes \lfloor \mathsf{T} \rfloor}\ \ltimes\mathrm{NEW} \qquad \frac{\Gamma \vdash t : \lfloor_{\mathsf{n}} \mathsf{1} \rfloor}{\Gamma \vdash t \triangleleft () : \mathsf{1}}\ \lfloor \mathsf{1} \rfloor\mathrm{E}$$

$$\frac{\Gamma \vdash t : \lfloor_{\mathsf{n}} \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor}{\Gamma \vdash t \triangleleft \mathsf{Inl} : \lfloor_{\mathsf{n}} \mathsf{T}_1 \rfloor}\ \lfloor\oplus\rfloor\mathrm{E}_1 \qquad \frac{\Gamma \vdash t : \lfloor_{\mathsf{n}} \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor}{\Gamma \vdash t \triangleleft \mathsf{Inr} : \lfloor_{\mathsf{n}} \mathsf{T}_2 \rfloor}\ \lfloor\oplus\rfloor\mathrm{E}_2 \qquad \frac{\Gamma \vdash t : \lfloor_{\mathsf{n}} \mathsf{T}_1 \otimes \mathsf{T}_2 \rfloor}{\Gamma \vdash t \triangleleft (,) : \lfloor_{\mathsf{n}} \mathsf{T}_1 \rfloor \otimes \lfloor_{\mathsf{n}} \mathsf{T}_2 \rfloor}\ \lfloor\otimes\rfloor\mathrm{E}$$

$$\frac{\Gamma \vdash t : \lfloor_{\mathsf{n}} !_{\mathsf{n}'} \mathsf{T} \rfloor}{\Gamma \vdash t \triangleleft \mathsf{Mod}_{\mathsf{n}'} : \lfloor_{\mathsf{n}' \cdot \mathsf{n}} \mathsf{T} \rfloor}\ \lfloor!\rfloor\mathrm{E} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash t : \lfloor_{\mathsf{n}} \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U} \rfloor \\ \Gamma_2,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}\end{array}}{\Gamma_1 + (1\uparrow \cdot \mathsf{n}) \cdot \Gamma_2 \vdash t \triangleleft (\boldsymbol{\lambda} x \mapsto_{\mathsf{m}} u) : \mathsf{1}}\ \lfloor\multimap\rfloor\mathrm{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \lfloor \mathsf{U} \rfloor \\ \Gamma_2 \vdash t' : \mathsf{U} \ltimes \mathsf{T}\end{array}}{\Gamma_1 + 1\uparrow \cdot \Gamma_2 \vdash t \lhd\!\!\rhd t' : \mathsf{T}}\ \lfloor\ \rfloor\mathrm{E_C} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash t : \lfloor_{\mathsf{n}} \mathsf{T} \rfloor \\ \Gamma_2 \vdash t' : \mathsf{T}\end{array}}{\Gamma_1 + (1\uparrow \cdot \mathsf{n}) \cdot \Gamma_2 \vdash t \blacktriangleleft t' : \mathsf{1}}\ \lfloor\ \rfloor\mathrm{E_L}$$

Figure 2.5: Typing rules of λ_d (λ_d–TY)

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

$$\frac{\Gamma \vdash t : \mathsf{T} \ltimes \mathsf{1}}{\Gamma \vdash \mathbf{from}'_{\ltimes}\, t : \mathsf{T}} \ltimes\text{FROM'} \qquad \frac{}{\omega\nu \cdot \Gamma \vdash () : \mathsf{1}}\ 1\text{I} \qquad \frac{\Gamma_2,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}}{\Gamma_2 \vdash \boldsymbol{\lambda} x \,{}_{\mathsf{m}}\!\mapsto u : \mathsf{T} \,{}_{\mathsf{m}}\!\multimap \mathsf{U}} \multimap\text{I}$$

$$\frac{\Gamma_2 \vdash t : \mathsf{T}_1}{\Gamma_2 \vdash \mathsf{Inl}\, t : \mathsf{T}_1 \oplus \mathsf{T}_2} \oplus\text{I}_1 \qquad \frac{\Gamma_2 \vdash t : \mathsf{T}_2}{\Gamma_2 \vdash \mathsf{Inr}\, t : \mathsf{T}_1 \oplus \mathsf{T}_2} \oplus\text{I}_2 \qquad \frac{\Gamma_2 \vdash t : \mathsf{T}}{\mathsf{m} \cdot \Gamma_2 \vdash \mathsf{Mod}_{\mathsf{m}}\, t : !_{\mathsf{m}} \mathsf{T}} !\text{I}$$

$$\frac{\begin{array}{c}\Gamma_{21} \vdash t_1 : \mathsf{T}_1 \\ \Gamma_{22} \vdash t_2 : \mathsf{T}_2\end{array}}{\Gamma_{21} + \Gamma_{22} \vdash (t_1\,,\ t_2) : \mathsf{T}_1 \otimes \mathsf{T}_2} \otimes\text{I}$$

Figure 2.6: Derived typing rules for syntactic sugar (λ_d–TYS)

Rule λ_d–TY/IDV, in addition to weakening, allows for coercion of the mode m of the variable used, with ordering constraint $1\nu \leqslant \mathsf{m}$ as defined in Figure 2.4. Unlike in λ_{Ldm}, here not all modes 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. $\leqslant_{\mathfrak{a}}$ 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 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, $\mathbf{upd}_{\ltimes}\ t\ \mathbf{with}\ x \mapsto t'$ 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 $\mathbf{upd}_{\ltimes}\ t\ \mathbf{with}\ x \mapsto t'$ 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 2.7 illustrates scopes introduced by $\mathbf{upd}_{\ltimes}$, and how the typing rules for $\mathbf{upd}_{\ltimes}$ and $\blacktriangleleft$ interact. For the first time here we see ampar values $\langle v_2 \frown v_1 \rangle$, represented as pair-like objects with a structure with holes v_2 on the left, and an arbitrary structure v_1 containing destinations on the right (ampar values will be formally introduced in Section 2.6.1). 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/$\lfloor\ \rfloor$EL, for instance $d_{11} \blacktriangleleft x_0$ in Figure 2.7, 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.

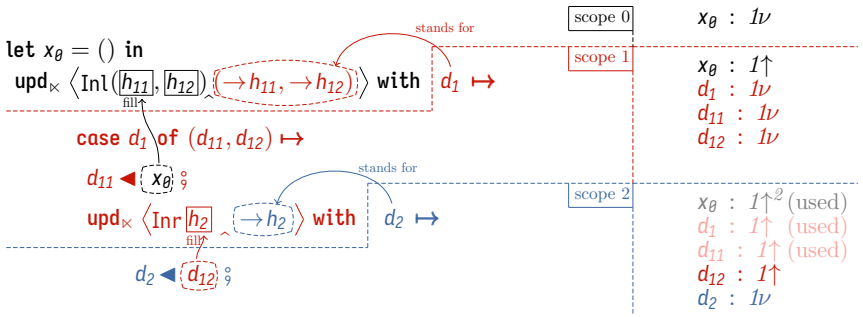


Figure 2.7: Evolution of ages through nested scopes

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 $\triangleleft(\boldsymbol{\lambda} x\ {}_{m}\!\mapsto u)$, 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 $\mathbf{from}'_{\ltimes}$ (rule λ_d–TY$_S$/$\ltimes$FROM'), the left of an ampar is extracted to the current scope. 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. $\mathbf{from}'_{\ltimes}$ is implemented in terms of $\mathbf{from}_{\ltimes}$ 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 $\mathbf{from}_{\ltimes}$ 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 $!_{1\infty}\mathsf{T}$, which means it is scope-insensitive: it can't contain any scope-controlled resource. Furthermore, it ensures that the right-hand side cannot contain destinations, so the structure has to be complete and thus is ready to be read.

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

The remaining operators $\triangleleft(), \triangleleft\mathsf{Inl}, \triangleleft\mathsf{Inr}, \triangleleft\mathsf{Mod}_{m}, \triangleleft(,)$ 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).

Putting into practice As an exercise, now that we have all the typing rules in place, we can work out the typing tree for the synthetic destination-filling operator $\triangleleft(::)$ and the corresponding synthetic constructor $(::)$ that we used since Section 2.2.2. Starting with $\triangleleft(::)$, we have the following definition.

$$\triangleleft(::) \ : \ \lfloor{}_{m}\mathsf{List\ T}\rfloor \multimap (\lfloor{}_{m}\mathsf{T}\rfloor \otimes \lfloor{}_{m}\mathsf{List\ T}\rfloor)$$
$$d \triangleleft (::) \triangleq d \triangleleft \mathsf{Inr} \triangleleft (,)$$

We recall that $\mathsf{List\ T} \triangleq \mathsf{1} \oplus (\mathsf{T} \otimes (\mathsf{List\ T}))$. Let's see the typing tree:

A-TY:

$$\dfrac{\dfrac{\dfrac{\dfrac{1\nu \lesssim 1\nu}{d :_{1\nu} \lfloor_{\mathsf{m}} \mathtt{List\ T} \rfloor \vdash d : \lfloor_{\mathsf{m}} \mathtt{List\ T} \rfloor}\,\mathrm{ID_V}}{d :_{1\nu} \lfloor_{\mathsf{m}} \mathtt{List\ T} \rfloor \vdash d : \lfloor_{\mathsf{m}} \mathtt{1} \oplus (\mathtt{T} \otimes (\mathtt{List\ T})) \rfloor}\ \text{unfold List T definition on the right}}{d :_{1\nu} \lfloor_{\mathsf{m}} \mathtt{List\ T} \rfloor \vdash d \triangleleft \mathsf{Inr} : \lfloor_{\mathsf{m}} \mathtt{T} \otimes (\mathtt{List\ T}) \rfloor}\,\lfloor\oplus\rfloor\mathrm{E}_2}{d :_{1\nu} \lfloor_{\mathsf{m}} \mathtt{List\ T} \rfloor \vdash d \triangleleft \mathsf{Inr} \triangleleft (,) : \lfloor_{\mathsf{m}} \mathtt{T} \rfloor \otimes \lfloor_{\mathsf{m}} \mathtt{List\ T} \rfloor}\,\lfloor\otimes\rfloor\mathrm{E}}{\bullet \vdash \lambda d \mapsto d \triangleleft \mathsf{Inr} \triangleleft (,) : \lfloor_{\mathsf{m}} \mathtt{List\ T} \rfloor \multimap \lfloor_{\mathsf{m}} \mathtt{T} \rfloor \otimes \lfloor_{\mathsf{m}} \mathtt{List\ T} \rfloor}\,\lambda_d\text{–}\mathrm{TY_S}/\multimap\mathrm{I}$$

Now, let's move on to (::) synthetic constructor:

$$(::)\ :\ \mathtt{T} \multimap \mathtt{List\ T} \multimap \mathtt{List\ T}$$
$$x :: xs \triangleq \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes}\ \mathtt{new}_{\ltimes}\ \mathtt{with}\ d \mapsto \mathtt{case}\ (d \triangleleft (::))\ \mathtt{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft x \fatsemi dxs \blacktriangleleft xs)$$

We will first type the subexpression $B \triangleq \mathtt{case}\ (d \triangleleft (::))\ \mathtt{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft x \fatsemi dxs \blacktriangleleft xs$, then type the whole expression corresponding to operator (::).

B-TY:

$$\dfrac{\dfrac{\dfrac{1\nu \lesssim 1\nu}{d :_{1\nu} \lfloor \mathtt{List\ T} \rfloor \vdash d : \lfloor \mathtt{List\ T} \rfloor}\,\mathrm{ID_V}}{d :_{1\nu} \lfloor \mathtt{List\ T} \rfloor \vdash d \triangleleft (::) : \lfloor \mathtt{T} \rfloor \otimes \lfloor \mathtt{List\ T} \rfloor}\,\text{A-TY} \qquad \dfrac{\dfrac{\dfrac{1\nu \lesssim 1\nu}{dx :_{1\nu} \lfloor \mathtt{T} \rfloor \vdash dx : \lfloor \mathtt{T} \rfloor}\,\mathrm{ID_V} \quad \dfrac{1\nu \lesssim 1\nu}{x :_{1\nu} \mathtt{T} \vdash x : \mathtt{T}}\,\mathrm{ID_V}}{dx :_{1\nu} \lfloor \mathtt{T} \rfloor,\ 1\uparrow \cdot x :_{1\nu} \mathtt{T} \vdash dx \blacktriangleleft x : 1}\,\lfloor\ \rfloor\mathrm{E_L} \qquad \dfrac{\dfrac{1\nu \lesssim 1\nu}{dxs :_{1\nu} \lfloor \mathtt{List\ T} \rfloor \vdash dxs : \lfloor \mathtt{List\ T} \rfloor}\,\mathrm{ID_V} \quad \dfrac{1\nu \lesssim 1\nu}{xs :_{1\nu} \mathtt{List\ T} \vdash xs : \mathtt{List\ T}}\,\mathrm{ID_V}}{dxs :_{1\nu} \lfloor \mathtt{List\ T} \rfloor,\ 1\uparrow \cdot xs :_{1\nu} \mathtt{List\ T} \vdash dxs \blacktriangleleft xs : 1}\,\lfloor\ \rfloor\mathrm{E_L}}{dx :_{1\nu} \lfloor \mathtt{T} \rfloor,\ dxs :_{1\nu} \lfloor \mathtt{List\ T} \rfloor,\ 1\uparrow \cdot (x :_{1\nu} \mathtt{T},\ xs :_{1\nu} \mathtt{List\ T}) \vdash dx \blacktriangleleft x \fatsemi dxs \blacktriangleleft xs : 1}\,1\mathrm{E}}{d :_{1\nu} \lfloor \mathtt{List\ T} \rfloor,\ 1\uparrow \cdot (x :_{1\nu} \mathtt{T},\ xs :_{1\nu} \mathtt{List\ T}) \vdash \mathtt{case}\ (d \triangleleft (::))\ \mathtt{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft x \fatsemi dxs \blacktriangleleft xs : 1}\,\otimes\mathrm{E}$$

C-TY:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bullet \vdash \mathtt{new}_{\ltimes} : \mathtt{List\ T} \ltimes \lfloor \mathtt{List\ T} \rfloor}\,\ltimes\mathrm{NEW} \qquad \dfrac{}{d :_{1\nu} \lfloor \mathtt{List\ T} \rfloor,\ 1\uparrow \cdot (x :_{1\nu} \mathtt{T},\ xs :_{1\nu} \mathtt{List\ T}) \vdash B : 1}\,\text{B-TY}}{x :_{1\nu} \mathtt{T},\ xs :_{1\nu} \mathtt{List\ T} \vdash \mathtt{upd}_{\ltimes}\ \mathtt{new}_{\ltimes}\ \mathtt{with}\ d \mapsto \mathtt{case}\ (d \triangleleft (::))\ \mathtt{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft x \fatsemi dxs \blacktriangleleft xs : \mathtt{List\ T} \ltimes 1}\,\ltimes\mathrm{UPD}}{x :_{1\nu} \mathtt{T},\ xs :_{1\nu} \mathtt{List\ T} \vdash \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes}\ \mathtt{new}_{\ltimes}\ \mathtt{with}\ d \mapsto \mathtt{case}\ (d \triangleleft (::))\ \mathtt{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft x \fatsemi dxs \blacktriangleleft xs) : \mathtt{List\ T}}\,\ltimes\mathrm{FROM'}}{x :_{1\nu} \mathtt{T} \vdash \lambda xs \mapsto \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes}\ \mathtt{new}_{\ltimes}\ \mathtt{with}\ d \mapsto \mathtt{case}\ (d \triangleleft (::))\ \mathtt{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft x \fatsemi dxs \blacktriangleleft xs) : \mathtt{List\ T} \multimap \mathtt{List\ T}}\,\lambda_d\text{–}\mathrm{TY_S}/\multimap\mathrm{I}}{\bullet \vdash \lambda x \mapsto \lambda xs \mapsto \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes}\ \mathtt{new}_{\ltimes}\ \mathtt{with}\ d \mapsto \mathtt{case}\ (d \triangleleft (::))\ \mathtt{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft x \fatsemi dxs \blacktriangleleft xs) : \mathtt{T} \multimap \mathtt{List\ T} \multimap \mathtt{List\ T}}\,\lambda_d\text{–}\mathrm{TY_S}/\multimap\mathrm{I}$$

If look at the typing tree C-TY from the root to the leaves, we see that when going under $\mathtt{upd}_{\ltimes}$, variable bindings for x and xs are multiplied by $1\uparrow$ (their age is increased by one). But this is not an issue; in fact, it's required by the typing rule λ_d–TY/$\lfloor\ \rfloor\mathrm{E_L}$ that the right operand is of age $\uparrow$ (as always, to prevent scope escape), as we see in typing tree B-TY.

Also, as in a regular functional programming language, modes and more specifically ages do *not* show up in the signature of synthetic constructor (::); only in its internal typing derivations, because the use of destinations to define (::) is, from this point of view, just implementation details hidden to the user.

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 this time we'll be more thorough. In particular, we'll introduce the grammar of evaluation contexts formally, and we'll also define typing rules for evaluation contexts and commands, so as to prove that typing is preserved throughout the reduction. As before, commands $E\lceil t \rceil$ are used to represent running programs; they're described in detail in Section 2.6.2.

Grammar extended with values:

$t, u ::= \ldots \mid v$

$v ::= \boxed{h}$ *(hole)*
$\mid \to h$ *(destination)*
$\mid {}_H\langle v_2 \frown v_1 \rangle$ *(ampar value)*
$\mid () \mid \boldsymbol{\lambda}_{\mathbf{v}} x :_m \mapsto u \mid \mathsf{Inl}\, v$
$\mid \mathsf{Inr}\, v \mid \mathsf{Mod}_m\, v \mid (v_1, v_2)$

Extended grammar of typing contexts:

$\Delta ::= \bullet \mid \to h :_m \lfloor {}_n\mathsf{T} \rfloor \mid \Delta_1, \Delta_2$
$\Gamma ::= \bullet \mid \to h :_m \lfloor {}_n\mathsf{T} \rfloor \mid x :_m \mathsf{T} \mid \Gamma_1, \Gamma_2$
$\Theta ::= \bullet \mid \to h :_m \lfloor {}_n\mathsf{T} \rfloor \mid \boxed{h} :_n \mathsf{T} \mid \Theta_1, \Theta_2$

Operations on new typing context forms:

$n' \cdot (\boxed{h} :_n \mathsf{T}, \Theta) \triangleq (\boxed{h} :_{n' \cdot n} \mathsf{T}), n' \cdot \Theta$
$n' \cdot (\to h :_m \lfloor {}_n\mathsf{T} \rfloor, \Gamma) \triangleq (\to h :_{n' \cdot m} \lfloor {}_n\mathsf{T} \rfloor), n' \cdot \Gamma$ †

$(\boxed{h} :_n \mathsf{T}, \Theta_1) + \Theta_2 \triangleq \boxed{h} :_n \mathsf{T}, (\Theta_1 + \Theta_2)$ if $h \notin \Theta_2$
$(\boxed{h} :_n \mathsf{T}, \Theta_1) + (\boxed{h} :_{n'} \mathsf{T}, \Theta_2) \triangleq \boxed{h} :_{n+n'} \mathsf{T}, (\Theta_1 + \Theta_2)$
$(\to h :_m \lfloor {}_n\mathsf{T} \rfloor, \Gamma_1) + \Gamma_2 \triangleq \to h :_m \lfloor {}_n\mathsf{T} \rfloor, (\Gamma_1 + \Gamma_2)$ if $h \notin \Gamma_2$ †
$(\to h :_m \lfloor {}_n\mathsf{T} \rfloor, \Gamma_1) + (\to h :_{m'} \lfloor {}_n\mathsf{T} \rfloor, \Gamma_2) \triangleq \to h :_{m+m'} \lfloor {}_n\mathsf{T} \rfloor, (\Gamma_1 + \Gamma_2)$ †

$\to^{-1}(\bullet) \triangleq \bullet$
$\to^{-1}(\to h :_{1\nu} \lfloor {}_n\mathsf{T} \rfloor, \Delta) \triangleq (\boxed{h} :_n \mathsf{T}), \to^{-1}(\Delta)$

† : *same rule is also true for* Θ *or* Δ *replacing* Γ

Figure 2.8: Runtime values, new typing context forms, and operators

But first, we need a class of runtime values (we'll often just say *values*), and their corresponding typing rules, as λ_d currently lacks any way to represent destinations or holes, or really any kind of value (for instance $\mathsf{Inl}\,()$ has been, so far, just syntactic sugar for a term $\mathbf{from}'_{\ltimes}(\mathbf{upd}_{\ltimes}\, \mathbf{new}_{\ltimes}\ \mathbf{with}\ d \mapsto \ldots)$). It's a peculiarity of λ_d that values (in particular, data constructors) are just used during reduction; usually they are also allowed in the term syntax of functional languages as in λ_{Ldm}.

2.6.1 Runtime values and new typing context forms

The syntax of runtime values is given in Figure 2.8. It features constructors for all of our basic types, as well as functions (note that in $\boldsymbol{\lambda}_{\mathbf{v}} x :_m \mapsto u$, u is a term, not a value). The more interesting values are holes $\boxed{h}$, destinations $\to h$, and ampars ${}_H\langle v_2 \frown v_1 \rangle$, which were not present in λ_{Ldm}, and that 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, and we add the corresponding typing rule λ_d–TY/FROMVAL (at the top of Figure 2.9).

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* $\boxed{h} :_n \mathsf{T}$ and *destination bindings* $\to h :_m \lfloor {}_n\mathsf{T} \rfloor$ in typing contexts in addition to the usual variable bindings $x :_m \mathsf{T}$. In both cases, we call h a *hole name*. We complement this by new typing context forms: we now have Γ, Θ and Δ. Γ is the form of typing context used when typing terms; we allow it to contain variable bindings (as before) but also destination bindings. On the other

Typing values as terms:

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

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

$$\frac{}{\boxed{h} :_{1\nu} \mathsf{T} \vdash_{\mathsf{v}} \boxed{h} : \mathsf{T}}\ \mathrm{ID_H} \qquad \frac{1\nu \leqslant \mathsf{m}}{\to h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor \vdash_{\mathsf{v}} \to h : \lfloor_{\mathsf{n}} \mathsf{T} \rfloor}\ \mathrm{ID_D} \qquad \frac{}{\cdot \vdash_{\mathsf{v}} () : \mathsf{1}}\ \mathrm{1I}$$

$$\frac{\Delta,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}}{\Delta \vdash_{\mathsf{v}} \boldsymbol{\lambda}_{\mathsf{v}} x \,{}_{\mathsf{m}}\!\mapsto u : \mathsf{T} \,{}_{\mathsf{m}}\!\multimap \mathsf{U}}\ \multimap\mathrm{I} \qquad \frac{\Theta \vdash_{\mathsf{v}} v_1 : \mathsf{T}_1}{\Theta \vdash_{\mathsf{v}} \mathsf{Inl}\, v_1 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \oplus\mathrm{I}_1 \qquad \frac{\Theta \vdash_{\mathsf{v}} v_2 : \mathsf{T}_2}{\Theta \vdash_{\mathsf{v}} \mathsf{Inr}\, v_2 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \oplus\mathrm{I}_2$$

$$\frac{\Theta_1 \vdash_{\mathsf{v}} v_1 : \mathsf{T}_1 \quad \Theta_2 \vdash_{\mathsf{v}} v_2 : \mathsf{T}_2}{\Theta_1 + \Theta_2 \vdash_{\mathsf{v}} (v_1\,,\ v_2) : \mathsf{T}_1 \otimes \mathsf{T}_2}\ \otimes\mathrm{I} \qquad \frac{\Theta \vdash_{\mathsf{v}} v' : \mathsf{T}}{\mathsf{n} \cdot \Theta \vdash_{\mathsf{v}} \mathsf{Mod}_{\mathsf{n}}\, v' : !_{\mathsf{n}} \mathsf{T}}\ !\mathrm{I}$$

$$\frac{1\uparrow \cdot \Delta_1,\ \Delta_3 \vdash_{\mathsf{v}} v_1 : \mathsf{T} \quad \Delta_2,\ \to^{-1} \Delta_3 \vdash_{\mathsf{v}} v_2 : \mathsf{U}}{\Delta_1,\ \Delta_2 \vdash_{\mathsf{v}} \mathsf{hnames}(\Delta_3)\langle v_2 \frown v_1 \rangle : \mathsf{U} \ltimes \mathsf{T}}\ \ltimes\mathrm{I}$$

Figure 2.9: Typing rules for runtime values (λ_d–TY$_{\mathrm{V}}$)

hand, the new context form Θ can contain both destination bindings and hole bindings, but *not a destination binding and a hole binding for the same hole name.* Θ cannot contain variable bindings either. Finally, a typing context Δ can only contain destination bindings.

We extend our previous context operations $+$ and $\cdot$ to act on the new binding forms, as described in Figure 2.8. Context addition is still very partial; for instance, $(\boxed{h} :_{\mathsf{n}} \mathsf{T}) + (\to h :_{\mathsf{m}} \lfloor_{\mathsf{n}'} \mathsf{T} \rfloor)$ 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. We don't distinguish values with holes from fully-defined values at the syntactic level: instead types prevent holes from being read. The type system maintains this invariant by simply not allowing any hole bindings in the context when typing terms (indeed, typing rules λ_d–TY from Figure 2.5 still hold, and for instance, typing contexts must be of shape Γ). In fact, the only place where holes are introduced, is in typing rules for values, in Figure 2.9, and more precisely, in the left-hand side v_2 of an ampar $_H\langle v_2 \frown v_1 \rangle$ in rule λ_d–TY$_{\mathrm{V}}$/$\ltimes$I.

Specifically, holes come from the operator $\to^{-1}$, which represents the hole bindings matching the specified set of destination bindings. It's a partial, pointwise operation on typing contexts Δ, as defined in Figure 2.8. Note that $\to^{-1}\Delta$ is undefined if any destination binding in Δ has a mode other than 1ν.

Furthermore, in λ_d–$\mathrm{TY}_\mathrm{V}/{\ltimes}\mathrm{I}$, the holes $\to^{-1}\Delta_3$ and the corresponding destinations Δ_3 of the ampar 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. We annotate, with subscript H, the set of hole names that an ampar ${}_H\langle v_2 \frown v_1\rangle$ bounds. That's why, in rule λ_d–$\mathrm{TY}_\mathrm{V}/{\ltimes}\mathrm{I}$, the resulting ampar carries the set of hole names $\mathsf{hnames}(\Delta_3)$ (we use operator $\mathsf{hnames}(\bullet)$ to denote the holes names present in a given typing context).

That being said, an ampar can also store destinations that aren't its own, from other scopes; they are represented by $1{\uparrow}\cdot\Delta_1$ and Δ_2 in the respective typing contexts of v_1 and v_2 inλ_d–$\mathrm{TY}_\mathrm{V}/{\ltimes}\mathrm{I}$.

Rule λ_d–$\mathrm{TY}_\mathrm{V}/\mathrm{ID}_\mathrm{H}$ indicates that a hole must have mode 1ν in the 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–$\mathrm{TY}_\mathrm{V}/\mathrm{ID}_\mathrm{D}$, we see that a destination can be typed with any mode m coercible to 1ν. We did this to mimic the rule λ_d–$\mathrm{TY}/\mathrm{ID}_\mathrm{V}$ and make the general modal substitution lemma expressible for λ_d[15]. We formally proved however that throughout the reduction of 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 at our advantage to make the formal proof of the substitution lemma much easier. The other mode n, appearing in λ_d–$\mathrm{TY}_\mathrm{V}/\mathrm{ID}_\mathrm{D}$, 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 While values are typed in contexts Θ allowing both destination and hole bindings, when using a value as a term in λ_d–$\mathrm{TY}/\mathrm{FROMVAL}$, it's only allowed to have free destinations, but no free holes (as we require a typing context of form Δ for the value).

Notice, also, that values (used as terms, or not) can't have free variables, since contexts Θ only contain hole and destination bindings, but 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–$\mathrm{TY}_\mathrm{V}/{\multimap}\mathrm{I}$), 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 $\boldsymbol{\lambda} x \mapsto \boldsymbol{\lambda} y \mapsto x\ \mathsf{concat}\ y$ 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 $\boldsymbol{\lambda}_\mathsf{v} x \mapsto \mathsf{from}'_\ltimes(\mathsf{upd}_\ltimes\ \mathsf{new}_\ltimes\ \mathsf{with}\ d \mapsto d \triangleleft (\boldsymbol{\lambda} y \mapsto x\ \mathsf{concat}\ y))$, where the inner closure is not yet in value form. As the form $d \triangleleft (\boldsymbol{\lambda} y \mapsto x\ \mathsf{concat}\ y)$ 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 *command* $E[t]$, that is, a pair of an evaluation context E, and an (extended) term t under focus, as in Section 1.6.

[15] 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 $x :_{\omega\infty} \lfloor_\mathsf{n}\mathsf{T}\rfloor \vdash () : \mathsf{1}$ and ${\to}h :_{1\nu} \lfloor_\mathsf{n}\mathsf{T}\rfloor \vdash {\to}h : \lfloor_\mathsf{n}\mathsf{T}\rfloor$ so $\omega\infty\cdot({\to}h :_{1\nu} \lfloor_\mathsf{n}\mathsf{T}\rfloor) \vdash ()[x := {\to}h] : \mathsf{1}$ should be valid.

Grammar of evaluation contexts:

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

Figure 2.10: Evaluation contexts

The grammar of evaluation contexts is given in Figure 2.10. As for 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 requires 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.

It's important to note, however, that unlike in λ_{Ldm}, command $E[t]$ of λ_d won't always have a corresponding term. The reason is that the focusing components are all directly derived from the term syntax, except for the *open ampar* focusing component ${}^{\mathrm{op}}_{H}\langle v_2 \frown [\,]\rangle$ 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.

Typing of evaluation contexts Evaluation contexts are typed in a context Δ that can only contain destination bindings. The typing context Δ of an evaluation context E is exactly the typing context that the term t has to use to form a valid $E[t]$. 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, and the sole typing rule for commands, are given in Figure 2.11.

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 $t : \mathtt{T}$, $E[t]$ 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–$\mathrm{TY_E}/{\ltimes}\mathrm{OP}$, as there is no direct term equivalent to the *open ampar* forcusing component). Let's take the rule λ_d–$\mathrm{TY_E}/{\otimes}\mathrm{E}$ as an example:

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

$$
\lambda_d\text{–}\mathrm{TY}/{\otimes}\mathrm{E}\ \frac{\Gamma_1 \vdash t : \mathtt{T}_1 \otimes \mathtt{T}_2 \qquad \Gamma_2,\ x_1 :_{\mathtt{m}} \mathtt{T}_1,\ x_2 :_{\mathtt{m}} \mathtt{T}_2 \vdash u : \mathtt{U}}{\mathtt{m}\cdot\Gamma_1 + \Gamma_2 \vdash \mathtt{case}_{\mathtt{m}}\,t\,\mathtt{of}\,(x_1\,,\,x_2) \mapsto u : \mathtt{U}}
$$

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

$$\frac{}{\bullet \dashv [\,] : \mathsf{U}_0 \rightarrowtail \mathsf{U}_0}\ \mathrm{ID} \qquad \frac{m \cdot \Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_2 \vdash t' : \mathsf{T}\ {}_m\!\multimap \mathsf{U}}{\Delta_1 \dashv E \circ t'\,[\,] : \mathsf{T} \rightarrowtail \mathsf{U}_0}\ \multimap\mathrm{E}_1 \qquad \frac{m \cdot \Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_1 \vdash v : \mathsf{T}}{\Delta_2 \dashv E \circ [\,]\ v : (\mathsf{T}\ {}_m\!\multimap \mathsf{U}) \rightarrowtail \mathsf{U}_0}\ \multimap\mathrm{E}_2$$

$$\frac{\Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_2 \vdash u : \mathsf{U}}{\Delta_1 \dashv E \circ [\,] \fatsemi u : \mathsf{1} \rightarrowtail \mathsf{U}_0}\ 1\mathrm{E} \qquad \frac{m \cdot \Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_2,\ x_1 :_m \mathsf{T}_1 \vdash u_1 : \mathsf{U} \quad \Delta_2,\ x_2 :_m \mathsf{T}_2 \vdash u_2 : \mathsf{U}}{\Delta_1 \dashv E \circ \mathsf{case}_m\ [\,]\ \mathsf{of}\ \{\mathsf{Inl}\,x_1 \mapsto u_1\ ,\ \mathsf{Inr}\,x_2 \mapsto u_2\} : (\mathsf{T}_1 \oplus \mathsf{T}_2) \rightarrowtail \mathsf{U}_0}\ \oplus\mathrm{E}$$

$$\frac{m \cdot \Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_2,\ x_1 :_m \mathsf{T}_1,\ x_2 :_m \mathsf{T}_2 \vdash u : \mathsf{U}}{\Delta_1 \dashv E \circ \mathsf{case}_m\ [\,]\ \mathsf{of}\ (x_1\ ,\ x_2) \mapsto u : (\mathsf{T}_1 \otimes \mathsf{T}_2) \rightarrowtail \mathsf{U}_0}\ \otimes\mathrm{E} \qquad \frac{m \cdot \Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_2,\ x :_{m \cdot m'} \mathsf{T} \vdash u : \mathsf{U}}{\Delta_1 \dashv E \circ \mathsf{case}_m\ [\,]\ \mathsf{of}\ \mathsf{Mod}_{m'}\,x \mapsto u : !_{m'}\mathsf{T} \rightarrowtail \mathsf{U}_0}\ !\mathrm{E}$$

$$\frac{\Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \ltimes \mathsf{T}' \rightarrowtail \mathsf{U}_0 \quad 1\!\uparrow \cdot \Delta_2,\ x :_{1\nu} \mathsf{T} \vdash t' : \mathsf{T}'}{\Delta_1 \dashv E \circ \mathsf{upd}_\ltimes\ [\,]\ \mathsf{with}\ x \mapsto t' : (\mathsf{U} \ltimes \mathsf{T}) \rightarrowtail \mathsf{U}_0}\ \ltimes\mathrm{UPD} \qquad \frac{\Delta \dashv E : (\mathsf{U} \ltimes \mathsf{1}) \rightarrowtail \mathsf{U}_0}{\Delta \dashv E \circ \mathsf{to}_\ltimes\ [\,] : \mathsf{U} \rightarrowtail \mathsf{U}_0}\ \ltimes\mathrm{TO}$$

$$\frac{\Delta \dashv E : (\mathsf{U} \otimes (!_{1\infty}\mathsf{T})) \rightarrowtail \mathsf{U}_0}{\Delta \dashv E \circ \mathsf{from}_\ltimes\ [\,] : (\mathsf{U} \ltimes (!_{1\infty}\mathsf{T})) \rightarrowtail \mathsf{U}_0}\ \ltimes\mathrm{FROM} \qquad \frac{\Delta \dashv E : \mathsf{1} \rightarrowtail \mathsf{U}_0}{\Delta \dashv E \circ [\,] \lhd () : \lfloor_n \mathsf{1} \rfloor \rightarrowtail \mathsf{U}_0}\ \lfloor\mathsf{1}\rfloor\mathrm{E}$$

$$\frac{\Delta \dashv E : \lfloor_n \mathsf{T}_1 \rfloor \rightarrowtail \mathsf{U}_0}{\Delta \dashv E \circ [\,] \lhd \mathsf{Inl} : \lfloor_n \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_0}\ \lfloor\oplus\rfloor\mathrm{E}_1 \qquad \frac{\Delta \dashv E : \lfloor_n \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_0}{\Delta \dashv E \circ [\,] \lhd \mathsf{Inr} : \lfloor_n \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_0}\ \lfloor\oplus\rfloor\mathrm{E}_2$$

$$\frac{\Delta \dashv E : (\lfloor_n \mathsf{T}_1 \rfloor \otimes \lfloor_n \mathsf{T}_2 \rfloor) \rightarrowtail \mathsf{U}_0}{\Delta \dashv E \circ [\,] \lhd (,) : \lfloor_n \mathsf{T}_1 \otimes \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_0}\ \lfloor\otimes\rfloor\mathrm{E} \qquad \frac{\Delta \dashv E : \lfloor_{m \cdot n} \mathsf{T} \rfloor \rightarrowtail \mathsf{U}_0}{\Delta \dashv E \circ [\,] \lhd \mathsf{Mod}_m : \lfloor_n !_m \mathsf{T} \rfloor \rightarrowtail \mathsf{U}_0}\ \lfloor!\rfloor\mathrm{E}$$

$$\frac{\Delta_1,\ (1\!\uparrow \cdot n) \cdot \Delta_2 \dashv E : \mathsf{1} \rightarrowtail \mathsf{U}_0 \quad \Delta_2,\ x :_m \mathsf{T} \vdash u : \mathsf{U}}{\Delta_1 \dashv E \circ [\,] \lhd (\lambda x\ {}_m\!\mapsto u) : \lfloor_n \mathsf{T}\ {}_m\!\multimap \mathsf{U} \rfloor \rightarrowtail \mathsf{U}_0}\ \lfloor\multimap\rfloor\mathrm{E} \qquad \frac{\Delta_1,\ 1\!\uparrow \cdot \Delta_2 \dashv E : \mathsf{T} \rightarrowtail \mathsf{U}_0 \quad \Delta_2 \vdash t' : \mathsf{U} \ltimes \mathsf{T}}{\Delta_1 \dashv E \circ [\,] \lhd\!\circ\, t' : \lfloor \mathsf{U} \rfloor \rightarrowtail \mathsf{U}_0}\ \lfloor\ \rfloor\mathrm{E}_{\mathrm{C1}}$$

$$\frac{\Delta_1,\ 1\!\uparrow \cdot \Delta_2 \dashv E : \mathsf{T} \rightarrowtail \mathsf{U}_0 \quad \Delta_1 \vdash v : \lfloor \mathsf{U} \rfloor}{\Delta_2 \dashv E \circ v \lhd\!\circ [\,] : \mathsf{U} \ltimes \mathsf{T} \rightarrowtail \mathsf{U}_0}\ \lfloor\ \rfloor\mathrm{E}_{\mathrm{C2}} \qquad \frac{\Delta_1,\ (1\!\uparrow \cdot n) \cdot \Delta_2 \dashv E : \mathsf{1} \rightarrowtail \mathsf{U}_0 \quad \Delta_2 \vdash t' : \mathsf{T}}{\Delta_1 \dashv E \circ [\,] \blacktriangleleft t' : \lfloor_n \mathsf{T} \rfloor \rightarrowtail \mathsf{U}_0}\ \lfloor\ \rfloor\mathrm{E}_{\mathrm{L1}}$$

$$\frac{\Delta_1,\ (1\!\uparrow \cdot n) \cdot \Delta_2 \dashv E : \mathsf{1} \rightarrowtail \mathsf{U}_0 \quad \Delta_1 \vdash v : \lfloor_n \mathsf{T} \rfloor}{\Delta_2 \dashv E \circ v \blacktriangleleft [\,] : \mathsf{T} \rightarrowtail \mathsf{U}_0}\ \lfloor\ \rfloor\mathrm{E}_{\mathrm{L2}} \qquad \frac{\mathsf{hnames}(E)\ \#\#\ \mathsf{hnames}(\Delta_3) \quad \Delta_1,\ \Delta_2 \dashv E : (\mathsf{U} \ltimes \mathsf{T}') \rightarrowtail \mathsf{U}_0 \quad \Delta_2,\ \rightarrow^{-1}\!\Delta_3 \vdash_{\blacktriangledown} v_2 : \mathsf{U}}{1\!\uparrow \cdot \Delta_1,\ \Delta_3 \dashv E \circ {}^{\mathrm{op}}_{\mathsf{hnames}(\Delta_3)}\langle v_2 \frown [\,]\rangle : \mathsf{T}' \rightarrowtail \mathsf{U}_0}\ \ltimes\mathrm{OP}$$

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

$$\frac{\Delta \dashv E : \mathsf{T} \rightarrowtail \mathsf{U}_0 \quad \Delta \vdash t : \mathsf{T}}{\vdash E[\,t\,] : \mathsf{U}_0}\ \lambda_d\text{–}\mathrm{TY}_{\mathrm{CMD}}$$

Figure 2.11: Typing rules of evaluation contexts (λ_d–TY_{E}) and commands (λ_d–$\mathrm{TY}_{\mathrm{CMD}}$)

- the typing context $\mathtt{m}\cdot\Delta_1,\ \Delta_2$ in the premise for E in λ_d–TY$_\text{E}$/$\otimes$E corresponds to $\mathtt{m}\cdot\Gamma_1+\Gamma_2$ in the conclusion of λ_d–TY/$\otimes$E;
- the typing context $\Delta_2,\ x_1 :_\mathtt{m} \mathtt{T}_1,\ x_2 :_\mathtt{m} \mathtt{T}_2$ in the premise for term u in λ_d–TY$_\text{E}$/$\otimes$E corresponds to the typing context $\Gamma_2,\ x_1 :_\mathtt{m} \mathtt{T}_1,\ x_2 :_\mathtt{m} \mathtt{T}_2$ for the same term in λ_d–TY/$\otimes$E;
- the typing context Δ_1 in the conclusion for $E\ \circ\ \mathtt{case}_\mathtt{m}\ [\,]\ \mathtt{of}\ (x_1\,,\,x_2) \mapsto u$ 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 the reduction rules of λ_d, displayed in Figures 2.12 and 2.13. As in λ_{Ldm}, focus (F suffix), unfocus (U suffix) and contraction (C suffix) rules of λ_d are triggered in a purely deterministic fashion. Once a subterm is a value, it cannot be focused on again.

Figure 2.12 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 $\mathtt{to}_\ltimes$ 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 $\mathtt{from}_\ltimes$ are fairly symmetric: once we have an ampar with a value of the shape $\mathtt{Mod}_{1\infty}\ v_1$ on the right, we can extract both the left and right side out of the ampar shell, into a normal pair.

Finally, $\mathtt{new}_\ltimes$ 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 2.13 are all related to $\mathtt{upd}_\ltimes$ 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 $\to h$ 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 $\mathtt{Inl}\ \boxed{h'+1}$ 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 bottom of Figure 2.12.

λ_d–SEM/$\lfloor\mathtt{1}\rfloor$EC and λ_d–SEM/$\lfloor\multimap\rfloor$EC of Figure 2.13 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.

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 in value form (which is why $\mathtt{new}_\ltimes$ is allowed to reduce to the same $_{\{1\}}\langle\boxed{1}\ ,\ \to 1\rangle$ every time), but as soon as an ampar is open (when it becomes an open ampar focusing

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

$E\,[\,t'\ t\,] \longrightarrow (E \circ t'\ [\,])\,[\,t\,]$	⋆	$\multimap$EF$_1$
$(E \circ t'\ [\,])\,[\,v\,] \longrightarrow E\,[\,t'\ v\,]$		$\multimap$EU$_1$
$E\,[\,t'\ v\,] \longrightarrow (E \circ [\,]\ v)\,[\,t'\,]$	⋆	$\multimap$EF$_2$
$(E \circ [\,]\ v)\,[\,v'\,] \longrightarrow E\,[\,v'\ v\,]$		$\multimap$EU$_2$
$E\,[\,(\boldsymbol{\lambda}_{\bullet} x\,{}_{m}\!\mapsto u)\ v\,] \longrightarrow E\,[\,u[x := v]\,]$		$\multimap$EC
$E\,[\,t \fatsemi u\,] \longrightarrow (E \circ [\,] \fatsemi u)\,[\,t\,]$	⋆	1EF
$(E \circ [\,] \fatsemi u)\,[\,v\,] \longrightarrow E\,[\,v \fatsemi u\,]$		1EU
$E\,[\,() \fatsemi u\,] \longrightarrow E\,[\,u\,]$		1EC
$E\,[\,\mathsf{case}_m\ t\ \mathsf{of}\ \{\mathsf{Inl}\,x_1 \mapsto u_1\,,\ \mathsf{Inr}\,x_2 \mapsto u_2\}\,] \longrightarrow (E \circ \mathsf{case}_m\ [\,]\ \mathsf{of}\ \{\mathsf{Inl}\,x_1 \mapsto u_1\,,\ \mathsf{Inr}\,x_2 \mapsto u_2\})\,[\,t\,]$	⋆	$\oplus$EF
$(E \circ \mathsf{case}_m\ [\,]\ \mathsf{of}\ \{\mathsf{Inl}\,x_1 \mapsto u_1\,,\ \mathsf{Inr}\,x_2 \mapsto u_2\})\,[\,v\,] \longrightarrow E\,[\,\mathsf{case}_m\ v\ \mathsf{of}\ \{\mathsf{Inl}\,x_1 \mapsto u_1\,,\ \mathsf{Inr}\,x_2 \mapsto u_2\}\,]$		$\oplus$EU
$E\,[\,\mathsf{case}_m\,(\mathsf{Inl}\,v_1)\ \mathsf{of}\ \{\mathsf{Inl}\,x_1 \mapsto u_1\,,\ \mathsf{Inr}\,x_2 \mapsto u_2\}\,] \longrightarrow E\,[\,u_1[x_1 := v_1]\,]$		$\oplus$EC$_1$
$E\,[\,\mathsf{case}_m\,(\mathsf{Inr}\,v_2)\ \mathsf{of}\ \{\mathsf{Inl}\,x_1 \mapsto u_1\,,\ \mathsf{Inr}\,x_2 \mapsto u_2\}\,] \longrightarrow E\,[\,u_2[x_2 := v_2]\,]$		$\oplus$EC$_2$
$E\,[\,\mathsf{case}_m\ t\ \mathsf{of}\ (x_1\,,\,x_2) \mapsto u\,] \longrightarrow (E \circ \mathsf{case}_m\ [\,]\ \mathsf{of}\ (x_1\,,\,x_2) \mapsto u)\,[\,t\,]$	⋆	$\otimes$EF
$(E \circ \mathsf{case}_m\ [\,]\ \mathsf{of}\ (x_1\,,\,x_2) \mapsto u)\,[\,v\,] \longrightarrow E\,[\,\mathsf{case}_m\ v\ \mathsf{of}\ (x_1\,,\,x_2) \mapsto u\,]$		$\otimes$EU
$E\,[\,\mathsf{case}_m\,(v_1\,,\,v_2)\ \mathsf{of}\ (x_1\,,\,x_2) \mapsto u\,] \longrightarrow E\,[\,u[x_1 := v_1][x_2 := v_2]\,]$		$\otimes$EC
$E\,[\,\mathsf{case}_m\ t\ \mathsf{of}\ \mathsf{Mod}_n\,x \mapsto u\,] \longrightarrow (E \circ \mathsf{case}_m\ [\,]\ \mathsf{of}\ \mathsf{Mod}_n\,x \mapsto u)\,[\,t\,]$	⋆	!EF
$(E \circ \mathsf{case}_m\ [\,]\ \mathsf{of}\ \mathsf{Mod}_n\,x \mapsto u)\,[\,v\,] \longrightarrow E\,[\,\mathsf{case}_m\ v\ \mathsf{of}\ \mathsf{Mod}_n\,x \mapsto u\,]$		!EU
$E\,[\,\mathsf{case}_m\ \mathsf{Mod}_n\,v'\ \mathsf{of}\ \mathsf{Mod}_n\,x \mapsto u\,] \longrightarrow E\,[\,u[x := v']\,]$		!EC
$E\,[\,\mathsf{to}_{\ltimes}\,u\,] \longrightarrow (E \circ \mathsf{to}_{\ltimes}\,[\,])\,[\,u\,]$	⋆	$\ltimes$TOF
$(E \circ \mathsf{to}_{\ltimes}\,[\,])\,[\,v_2\,] \longrightarrow E\,[\,\mathsf{to}_{\ltimes}\,v_2\,]$		$\ltimes$TOU
$E\,[\,\mathsf{to}_{\ltimes}\,v_2\,] \longrightarrow E\,[\,\{\,\}\langle v_2 \frown ()\rangle\,]$		$\ltimes$TOC
$E\,[\,\mathsf{from}_{\ltimes}\,t\,] \longrightarrow (E \circ \mathsf{from}_{\ltimes}\,[\,])\,[\,t\,]$	⋆	$\ltimes$FROMF
$(E \circ \mathsf{from}_{\ltimes}\,[\,])\,[\,v\,] \longrightarrow E\,[\,\mathsf{from}_{\ltimes}\,v\,]$		$\ltimes$FROMU
$E\,[\,\mathsf{from}_{\ltimes}\,\{\,\}\langle v_2 \frown \mathsf{Mod}_{1\infty}\,v_1\rangle\,] \longrightarrow E\,[\,(v_2\,,\,\mathsf{Mod}_{1\infty}\,v_1)\,]$		$\ltimes$FROMC
$E\,[\,\mathsf{new}_{\ltimes}\,] \longrightarrow E\,[\,\{1\}\langle\boxed{1} \frown \rightarrow 1\rangle\,]$		$\ltimes$NEWC

⋆ : only allowed if the term that would become the new focus is not already a value

---

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:

$$\begin{aligned} (E \circ {}^{\mathrm{op}}_{\{h\} \sqcup H}\langle v_2 \frown [\,]\rangle)\,(\!|h :=_{H'} v'|\!) &= E \circ {}^{\mathrm{op}}_{H \sqcup H'}\langle v_2\,(\!|h :=_{H'} v'|\!) \frown [\,]\rangle \\ (E \circ e)\,(\!|h :=_{H'} v'|\!) &= E\,(\!|h :=_{H'} v'|\!) \circ e \qquad \text{if } h \notin e \end{aligned}$$

Figure 2.12: Small-step reduction of commands for λ_d (λ_d–SEM, 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\ \ltimes\mathrm{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] & \ltimes\mathrm{UPDU} \\
E\left[\,\mathsf{upd}_{\ltimes}\ {}_{H}\langle v_2 \frown v_1\rangle\ \mathsf{with}\ x \mapsto t'\,\right] & \longrightarrow & \left(E \circ {}^{\mathrm{op}}_{H \doteq h'''}\langle v_2[H \doteq h'''] \frown [\,]\rangle\right)\left[\,t'[x := v_1[H \doteq h''']]\,\right] & \star\ \ltimes\mathrm{OP} \\
\left(E \circ {}^{\mathrm{op}}_{H}\langle v_2 \frown [\,]\rangle\right)\left[\,v_1\,\right] & \longrightarrow & E\left[\,{}_{H}\langle v_2 \frown v_1\rangle\,\right] & \ltimes\mathrm{CL} \\
E\left[\,t \lhd ()\,\right] & \longrightarrow & \left(E \circ [\,] \lhd ()\right)\left[\,t\,\right] & \star\ \lfloor 1 \rfloor\mathrm{EF} \\
\left(E \circ [\,] \lhd ()\right)\left[\,v\,\right] & \longrightarrow & E\left[\,v \lhd ()\,\right] & \lfloor 1 \rfloor\mathrm{EU} \\
E\left[\,{\to}h \lhd ()\,\right] & \longrightarrow & E\left(\!\left|h :=_{\{\}}\ ()\right|\!\right)\left[\,()\,\right] & \lfloor 1 \rfloor\mathrm{EC} \\
E\left[\,t \lhd \mathsf{Inl}\,\right] & \longrightarrow & \left(E \circ [\,] \lhd \mathsf{Inl}\right)\left[\,t\,\right] & \star\ \lfloor \oplus \rfloor\mathrm{E_1F} \\
\left(E \circ [\,] \lhd \mathsf{Inl}\right)\left[\,v\,\right] & \longrightarrow & E\left[\,v \lhd \mathsf{Inl}\,\right] & \lfloor \oplus \rfloor\mathrm{E_1U} \\
E\left[\,{\to}h \lhd \mathsf{Inl}\,\right] & \longrightarrow & E\left(\!\left|h :=_{\{h'+1\}}\ \mathsf{Inl}\ \boxed{h'+1}\right|\!\right)\left[\,{\to}h'+1\,\right] & \star\ \lfloor \oplus \rfloor\mathrm{E_1C} \\
E\left[\,t \lhd \mathsf{Inr}\,\right] & \longrightarrow & \left(E \circ [\,] \lhd \mathsf{Inr}\right)\left[\,t\,\right] & \star\ \lfloor \oplus \rfloor\mathrm{E_2F} \\
\left(E \circ [\,] \lhd \mathsf{Inr}\right)\left[\,v\,\right] & \longrightarrow & E\left[\,v \lhd \mathsf{Inr}\,\right] & \lfloor \oplus \rfloor\mathrm{E_2U} \\
E\left[\,{\to}h \lhd \mathsf{Inr}\,\right] & \longrightarrow & E\left(\!\left|h :=_{\{h'+1\}}\ \mathsf{Inr}\ \boxed{h'+1}\right|\!\right)\left[\,{\to}h'+1\,\right] & \star\ \lfloor \oplus \rfloor\mathrm{E_2C} \\
E\left[\,t \lhd \mathsf{Mod}_{m}\,\right] & \longrightarrow & \left(E \circ [\,] \lhd \mathsf{Mod}_{m}\right)\left[\,t\,\right] & \star\ \lfloor ! \rfloor\mathrm{EF} \\
\left(E \circ [\,] \lhd \mathsf{Mod}_{m}\right)\left[\,v\,\right] & \longrightarrow & E\left[\,v \lhd \mathsf{Mod}_{m}\,\right] & \lfloor ! \rfloor\mathrm{EU} \\
E\left[\,{\to}h \lhd \mathsf{Mod}_{m}\,\right] & \longrightarrow & E\left(\!\left|h :=_{\{h'+1\}}\ \mathsf{Mod}_{m}\ \boxed{h'+1}\right|\!\right)\left[\,{\to}h'+1\,\right] & \star\ \lfloor ! \rfloor\mathrm{EC} \\
E\left[\,t \lhd (,)\,\right] & \longrightarrow & \left(E \circ [\,] \lhd (,)\right)\left[\,t\,\right] & \star\ \lfloor \otimes \rfloor\mathrm{EF} \\
\left(E \circ [\,] \lhd (,)\right)\left[\,v\,\right] & \longrightarrow & E\left[\,v \lhd (,)\,\right] & \lfloor \otimes \rfloor\mathrm{EU} \\
E\left[\,{\to}h \lhd (,)\,\right] & \longrightarrow & E\left(\!\left|h :=_{\{h'+1,h'+2\}}\ \left(\boxed{h'+1}\,,\ \boxed{h'+2}\right)\right|\!\right)\left[\,({\to}h'+1\,,\ {\to}h'+2)\,\right] & \star\ \lfloor \otimes \rfloor\mathrm{EC} \\
E\left[\,t \lhd (\boldsymbol{\lambda} x\ {}_{m}{\mapsto}\ u)\,\right] & \longrightarrow & \left(E \circ [\,] \lhd (\boldsymbol{\lambda} x\ {}_{m}{\mapsto}\ u)\right)\left[\,t\,\right] & \star\ \lfloor {\multimap} \rfloor\mathrm{EF} \\
\left(E \circ [\,] \lhd (\boldsymbol{\lambda} x\ {}_{m}{\mapsto}\ u)\right)\left[\,v\,\right] & \longrightarrow & E\left[\,v \lhd (\boldsymbol{\lambda} x\ {}_{m}{\mapsto}\ u)\,\right] & \lfloor {\multimap} \rfloor\mathrm{EU} \\
E\left[\,{\to}h \lhd (\boldsymbol{\lambda} x\ {}_{m}{\mapsto}\ u)\,\right] & \longrightarrow & E\left(\!\left|h :=_{\{\}}\ \boldsymbol{\lambda}_{\mathbf{v}} x\ {}_{m}{\mapsto}\ u\right|\!\right)\left[\,()\,\right] & \lfloor {\multimap} \rfloor\mathrm{EC} \\
E\left[\,t \lhd\!\circ\, t'\,\right] & \longrightarrow & \left(E \circ [\,] \lhd\!\circ\, t'\right)\left[\,t\,\right] & \star\ \lfloor\ \rfloor\mathrm{E_CF_1} \\
\left(E \circ [\,] \lhd\!\circ\, t'\right)\left[\,v\,\right] & \longrightarrow & E\left[\,v \lhd\!\circ\, t'\,\right] & \lfloor\ \rfloor\mathrm{E_CU_1} \\
E\left[\,v \lhd\!\circ\, t'\,\right] & \longrightarrow & \left(E \circ v \lhd\!\circ [\,]\right)\left[\,t'\,\right] & \star\ \lfloor\ \rfloor\mathrm{E_CF_2} \\
\left(E \circ v \lhd\!\circ [\,]\right)\left[\,v'\,\right] & \longrightarrow & E\left[\,v \lhd\!\circ\, v'\,\right] & \lfloor\ \rfloor\mathrm{E_CU_2} \\
E\left[\,{\to}h \lhd\!\circ_{H}\langle v_2 \frown v_1\rangle\,\right] & \longrightarrow & E\left(\!\left|h :=_{(H \doteq h'')}\ v_2[H \doteq h'']\right|\!\right)\left[\,v_1[H \doteq h'']\,\right] & \star\ \lfloor\ \rfloor\mathrm{E_CC} \\
E\left[\,t \blacktriangleleft t'\,\right] & \longrightarrow & \left(E \circ [\,] \blacktriangleleft t'\right)\left[\,t\,\right] & \star\ \lfloor\ \rfloor\mathrm{E_LF_1} \\
\left(E \circ [\,] \blacktriangleleft t'\right)\left[\,v\,\right] & \longrightarrow & E\left[\,v \blacktriangleleft t'\,\right] & \lfloor\ \rfloor\mathrm{E_LU_1} \\
E\left[\,v \blacktriangleleft t'\,\right] & \longrightarrow & \left(E \circ v \blacktriangleleft [\,]\right)\left[\,t'\,\right] & \star\ \lfloor\ \rfloor\mathrm{E_LF_2} \\
\left(E \circ v \blacktriangleleft [\,]\right)\left[\,v'\,\right] & \longrightarrow & E\left[\,v \blacktriangleleft v'\,\right] & \lfloor\ \rfloor\mathrm{E_LU_2} \\
E\left[\,{\to}h \blacktriangleleft v\,\right] & \longrightarrow & E\left(\!\left|h :=_{\{\}}\ v\right|\!\right)\left[\,()\,\right] & \lfloor\ \rfloor\mathrm{E_LC}
\end{array}
$$

$$
\text{where} \quad \begin{cases} h' &= \mathsf{max}(\mathsf{hnames}(E) \cup \{h\}) + 1 \\ h'' &= \mathsf{max}(H \cup (\mathsf{hnames}(E) \cup \{h\})) + 1 \\ h''' &= \mathsf{max}(H \cup \mathsf{hnames}(E)) + 1 \end{cases}
$$

$\star$: only allowed if the term that would become the new focus is not already a value

Figure 2.13: Small-step reduction of commands for λ_d (λ_d–SEM, part 2)

component), it should have globally unique hole names. This restriction is enforced in rule λ_d–TYE/$\ltimes$OP by premise $\mathsf{hnames}(E) \mathbin{\#\#} \mathsf{hnames}(\Delta_3)$, requiring hole name sets from E and Δ_3 to be disjoint when an open ampar focusing component is created during reduction of $\mathbf{upd}_\ltimes$. 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 $h' = \mathsf{max}(\mathsf{hnames}(E) \cup \{h\}) + 1$ or similar definitions for h'' and h''' (see at the bottom of Figure 2.13) to find a new unused name[16]. Then we use either the *shifted set* $H \pm h'$ or the *conditional shift operator* $h[H \pm h']$ as defined at the bottom of Figure 2.12 to replace all names or just a specific one with fresh unused name(s). We extend *conditional shift* $\bullet[H \pm h']$ to arbitrary values, terms, and typing contexts in the obvious way (keeping in mind that ${}_{H'}\langle v_2 \frown v_1 \rangle$ binds the names in H').

Rules λ_d–SEM/$\ltimes$OP and λ_d–SEM/$\ltimes$CL dictate how and when an 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 ${}^{\mathrm{op}}_{H \pm h''}\langle v_2[H \pm h''] \frown [\,] \rangle$ while the right-hand side (containing destinations) is substituted in the body of the $\mathbf{upd}_\ltimes$ statement (which becomes the new term under focus). The rule λ_d–SEM/$\ltimes$CL triggers when the body of a $\mathbf{upd}_\ltimes$ 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 an ampar value again.

In rule λ_d–SEM/$\lfloor\,\rfloor$EcC, we write the left-hand side v_2 of an ampar value ${}_H\langle v_2 \frown v_1 \rangle$ 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 $\mathbf{upd}_\ltimes$. This renaming is carried out by the conditional shift $v_2[H \pm h'']$ and $v_1[H \pm h'']$.

Putting into practice Let's reuse the example from Section 2.5.2 with the synthetic (::) *cons* constructor, and see the reduction of expression $() :: \mathtt{Inl}\,()$. Here, $\mathtt{Inl}\,()$ is the value form corresponding to *nil* (given our encoding of lists), and we also assume that () in the expression is the value form of Figure 2.8, not the syntactic constructor defined in Figure 2.4. Finally, we consider $\mathbf{from}'_\ltimes$ to be a primitive (for brevity) with its contraction rule λ_d–SEM/$\ltimes$FROM'C : $E\left[\,\mathbf{from}'_\ltimes(\{\,\}\langle v_2 \frown ()\rangle)\,\right] \longrightarrow E\left[\,v_2\,\right]$ and trivial focus and unfocus rules.

[16] Sometimes we don't take the smallest unused name possible; it's mostly the result of using whatever made the formal proofs easier. For instance, in the formal proofs with Coq, renaming is implemented in terms of more general permutations, which make it more easily interoperable with typing contexts, represented as functions, but adds a few extra requirements to make lemma hold

$$
\begin{array}{lll}
 & [\,]\big[(\,) :: \mathsf{Inl}\,()\big] & \\
\longrightarrow & [\,]\big[\mathbf{from}'_{\ltimes}\,(\mathbf{upd}_{\ltimes}\,\mathbf{new}_{\ltimes}\ \mathbf{with}\ d \mapsto \mathbf{case}\,(d \triangleleft \mathsf{Inr} \triangleleft (,))\ \mathbf{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft () \fatsemi dxs \blacktriangleleft \mathsf{Inl}\,())\big] & \text{expanding } (::) \text{ and } \triangleleft(::) \text{ definitions} \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,])\big[\mathbf{upd}_{\ltimes}\,\mathbf{new}_{\ltimes}\ \mathbf{with}\ d \mapsto \mathbf{case}\,(d \triangleleft \mathsf{Inr} \triangleleft (,))\ \mathbf{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft () \fatsemi dxs \blacktriangleleft \mathsf{Inl}\,()\big] & \ltimes\textsc{from'F} \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,] \circ \mathbf{upd}_{\ltimes}\,[\,]\ \mathbf{with}\ d \mapsto \mathbf{case}\,(d \triangleleft \mathsf{Inr} \triangleleft (,))\ \mathbf{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft () \fatsemi dxs \blacktriangleleft \mathsf{Inl}\,())\big[\mathbf{new}_{\ltimes}\big] & \ltimes\textsc{updF} \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,] \circ \mathbf{upd}_{\ltimes}\,[\,]\ \mathbf{with}\ d \mapsto \mathbf{case}\,(d \triangleleft \mathsf{Inr} \triangleleft (,))\ \mathbf{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft () \fatsemi dxs \blacktriangleleft \mathsf{Inl}\,())\big[{}_{\{1\}}\langle \boxed{1} \frown \to 1\rangle\big] & \ltimes\textsc{newC} \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,])\big[\mathbf{upd}_{\ltimes}\,{}_{\{1\}}\langle \boxed{1} \frown \to 1\rangle\ \mathbf{with}\ d \mapsto \mathbf{case}\,(d \triangleleft \mathsf{Inr} \triangleleft (,))\ \mathbf{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft () \fatsemi dxs \blacktriangleleft \mathsf{Inl}\,()\big] & \ltimes\textsc{updU} \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,] \circ {}^{\mathrm{op}}_{\{2\}}\langle \boxed{2} \frown [\,]\rangle)\big[\mathbf{case}\,(\to 2 \triangleleft \mathsf{Inr} \triangleleft (,))\ \mathbf{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft () \fatsemi dxs \blacktriangleleft \mathsf{Inl}\,()\big] & \ltimes\textsc{op} \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,] \circ {}^{\mathrm{op}}_{\{2\}}\langle \boxed{2} \frown [\,]\rangle \circ \mathbf{case}_{\mathit{lr}}\,[\,]\ \mathbf{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft () \fatsemi dxs \blacktriangleleft \mathsf{Inl}\,())\big[\to 2 \triangleleft \mathsf{Inr} \triangleleft (,)\big] & \otimes\textsc{EF} \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,] \circ {}^{\mathrm{op}}_{\{2\}}\langle \boxed{2} \frown [\,]\rangle \circ \mathbf{case}_{\mathit{lr}}\,[\,]\ \mathbf{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft () \fatsemi dxs \blacktriangleleft \mathsf{Inl}\,() \circ [\,] \triangleleft (,))\big[\to 2 \triangleleft \mathsf{Inr}\big] & \lfloor\otimes\rfloor\textsc{EF} \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,] \circ {}^{\mathrm{op}}_{\{2\}}\langle \mathsf{Inr}\ \boxed{4} \frown [\,]\rangle \circ \mathbf{case}_{\mathit{lr}}\,[\,]\ \mathbf{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft () \fatsemi dxs \blacktriangleleft \mathsf{Inl}\,() \circ [\,] \triangleleft (,))\big[\to 4\big] & \lfloor\oplus\rfloor\textsc{E}_2\textsc{C with sub. } E\,(\!|2 :=_{\{4\}} \mathsf{Inr}\ \boxed{4}|\!) \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,] \circ {}^{\mathrm{op}}_{\{4\}}\langle \mathsf{Inr}\ \boxed{4} \frown [\,]\rangle \circ \mathbf{case}_{\mathit{lr}}\,[\,]\ \mathbf{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft () \fatsemi dxs \blacktriangleleft \mathsf{Inl}\,())\big[\to 4 \triangleleft (,)\big] & \lfloor\otimes\rfloor\textsc{EU} \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,] \circ {}^{\mathrm{op}}_{\{6,7\}}\langle \mathsf{Inr}\ (\boxed{6}, \boxed{7}) \frown [\,]\rangle \circ \mathbf{case}_{\mathit{lr}}\,[\,]\ \mathbf{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft () \fatsemi dxs \blacktriangleleft \mathsf{Inl}\,())\big[(\to 6\,,\,\to 7)\big] & \lfloor\otimes\rfloor\textsc{EC with sub. } E\,(\!|4 :=_{\{6,7\}} (\boxed{6}, \boxed{7})|\!) \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,] \circ {}^{\mathrm{op}}_{\{6,7\}}\langle \mathsf{Inr}\ (\boxed{6}, \boxed{7}) \frown [\,]\rangle)\big[\mathbf{case}\,(\to 6\,,\,\to 7)\ \mathbf{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft () \fatsemi dxs \blacktriangleleft \mathsf{Inl}\,()\big] & \otimes\textsc{EU} \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,] \circ {}^{\mathrm{op}}_{\{6,7\}}\langle \mathsf{Inr}\ (\boxed{6}, \boxed{7}) \frown [\,]\rangle)\big[\to 6 \blacktriangleleft () \fatsemi \to 7 \blacktriangleleft \mathsf{Inl}\,()\big] & \otimes\textsc{EC} \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,] \circ {}^{\mathrm{op}}_{\{6,7\}}\langle \mathsf{Inr}\ (\boxed{6}, \boxed{7}) \frown [\,]\rangle \circ [\,] \fatsemi \to 7 \blacktriangleleft \mathsf{Inl}\,())\big[\to 6 \blacktriangleleft ()\big] & 1\textsc{EF} \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,] \circ {}^{\mathrm{op}}_{\{7\}}\langle \mathsf{Inr}\ ((), \boxed{7}) \frown [\,]\rangle \circ [\,] \fatsemi \to 7 \blacktriangleleft \mathsf{Inl}\,())\big[()\big] & \lfloor\ \rfloor\textsc{E}_{\textsc{L}}\textsc{C with sub. } E\,(\!|6 :=_{\{\}} ()|\!) \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,] \circ {}^{\mathrm{op}}_{\{7\}}\langle \mathsf{Inr}\ ((), \boxed{7}) \frown [\,]\rangle)\big[() \fatsemi \to 7 \blacktriangleleft \mathsf{Inl}\,()\big] & 1\textsc{EU} \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,] \circ {}^{\mathrm{op}}_{\{7\}}\langle \mathsf{Inr}\ ((), \boxed{7}) \frown [\,]\rangle)\big[\to 7 \blacktriangleleft \mathsf{Inl}\,()\big] & 1\textsc{EC} \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,] \circ {}^{\mathrm{op}}_{\{\}}\langle \mathsf{Inr}\ ((), \mathsf{Inl}\,()) \frown [\,]\rangle)\big[()\big] & \lfloor\ \rfloor\textsc{E}_{\textsc{L}}\textsc{C with sub. } E\,(\!|7 :=_{\{\}} \mathsf{Inl}\,()|\!) \\
\longrightarrow & ([\,] \circ \mathbf{from}'_{\ltimes}\,[\,])\big[{}_{\{\}}\langle \mathsf{Inr}\ ((), \mathsf{Inl}\,()) \frown ()\rangle\big] & \ltimes\textsc{CL} \\
\longrightarrow & [\,]\big[\mathbf{from}'_{\ltimes}\,({}_{\{\}}\langle \mathsf{Inr}\ ((), \mathsf{Inl}\,()) \frown ()\rangle)\big] & \ltimes\textsc{from'U} \\
\longrightarrow & [\,]\big[\mathsf{Inr}\ ((), \mathsf{Inl}\,())\big] & \ltimes\textsc{from'C}
\end{array}
$$

At the end of the reduction, we get the empty evaluation context with value $\mathsf{Inr}\,((), \mathsf{Inl}\,())$ on focus, which is the value form for the singleton list containing just unit, as expected.

We also see how, throughout the reduction, the contractions of destination-filling operations under focus will trigger global substitutions $E\,(\!|h :=_{\mathcal{H}} v|\!)$ on the evaluation context E. As said before, these substitutions are responsible for mutating the structure under construction, which is (always) attached on *open ampar* focusing component, somewhere in the evaluation context E.

2.6.4 Type safety

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

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

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

As seen in the example above, a command of the form $[\,]\big[v\big]$ 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. 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, at the detriment of elegance of the proofs. 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 contexts 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.

2.8 Translation of prior work into λ_d terms and types

We announced in Section 2.1 that λ_d subsumes existing systems. Let's see how.

A Functional Representation of Data Structures with a Hole Minamide's system[Min98] introduces two primitives, `happ` and `hcomp`, as well as the hole abstraction form, $\hat{\boldsymbol{\lambda}} x \mapsto t$, representing structures with (a) hole, where x appears exactly once in t.

The paper indicates that hole abstractions cannot have lambda abstraction, application, or case expression operating on the variable of the abstraction, but otherwise allow these constructs to appear in the body of the hole abstraction if they are operating on other variables or values.

In a well-typed, closed program, we can evaluate the body of the hole abstraction (under the $\hat{\boldsymbol{\lambda}}$) as far as possible, until it is almost a value except for the single occurence of x (we know x cannot cause this reduction to be stuck, as it cannot appear in the evaluation path of an application or case expression). At that point hole abstractions directly translate to our ampar value form: we transform $\hat{\boldsymbol{\lambda}} x \mapsto t$ to ${}_{\{h\}}\langle t[x := \boxed{h}] \frown \rightarrow h \rangle$. In other terms, we take the body of the hole abstraction as the left-hand side of the ampar, with the single occurrence of the variable replaced by a hole $\boxed{h}$, and just use the corresponding destination $\rightarrow h$ as the right-hand side. The type of hole abstractions, $(\mathtt{T}, \mathtt{S})\mathtt{hfun}$, is translated as $\mathtt{S} \ltimes \lfloor \mathtt{T} \rfloor$.

If we don't allow for this eager evaluation of the hole abstraction's body to happen before the translation, we need to encode an hole abstraction by:

- spawning a new ampar with $\mathtt{new}_{\ltimes}$;
- opening it with $\mathtt{upd}_{\ltimes}$;
- building the data constructor in which the hole variable appear by a chain of fill expressions $d \triangleleft \ldots$ or $d \blacktriangleleft \ldots$ (following the same recipe that we used to recover normal data constructors or compound constructors from destination-filling operators; see (::) at the beginning of Section 2.2.2);
- returning, as the result of the body of $\mathtt{upd}_{\ltimes}$, the destination corresponding to the field of the data constructor in which we want the hole to be;
- other constructs, like case expressions or applications related to other variables or values are transposed untouched into the body of $\mathtt{upd}_{\ltimes}$.

It's rather hard to give a formal description of the translation, but it's quite clear on an example, with Minamide's term on the right, and λ_d one on the left:

```
ha : (T, List T)hfun
ha ≜ λ̂x ↦ 0 :: x :: (f 2)
```

```
ha_λd : (List T) ⋉ ⌊T⌋
ha_λd ≜ upd⋉ new⋉ with d ↦ case
          (d ◁ (::)) of (d0 , dt) ↦
            case (dt ◁ (::)) of (d1 , dt') ↦
              d0 ◀ 0 ⨟ dt' ◀ (f 2) ⨟ d1
```

Once $(f\ 2)$ has reduced to a value v, the λ_d's version will be evaluated to the promised ampar form ${}_{H}\langle 0 :: \boxed{h} :: v \frown \rightarrow h \rangle$.

Because destinations aren't part of [Min98], any variable in a program from Minamide's system can be given age ∞ (so mode 1∞ or $\omega\infty$ depending on whether it is linear or not), except of course the variables bound by hole abstractions.

The translation of **happ** and **hcomp** is even more direct:

$$\begin{aligned}
&\mathbf{happ} \;:\; (\mathtt{T},\mathtt{S})\mathtt{hfun} \multimap \mathtt{T} \multimap \mathtt{S}\\
&\mathbf{happ}_{\lambda_d} \;:\; \mathtt{S} \ltimes \lfloor \mathtt{T} \rfloor \multimap \mathtt{T} \multimap \mathtt{S}\\
&\mathbf{happ}_{\lambda_d}\ \mathit{ampar}\ x \triangleq \mathbf{from}'_{\ltimes}(\mathbf{upd}_{\ltimes}\ \mathit{ampar}\ \mathbf{with}\ d \mapsto d \blacktriangleleft x)
\end{aligned}$$

$$\begin{aligned}
&\mathbf{hcomp} \;:\; (\mathtt{T},\mathtt{S})\mathtt{hfun} \multimap (\mathtt{U},\mathtt{T})\mathtt{hfun} \multimap (\mathtt{U},\mathtt{S})\mathtt{hfun}\\
&\mathbf{hcomp}_{\lambda_d} \;:\; \mathtt{S} \ltimes \lfloor \mathtt{T} \rfloor \multimap \mathtt{T} \ltimes \lfloor \mathtt{U} \rfloor \multimap \mathtt{S} \ltimes \lfloor \mathtt{U} \rfloor\\
&\mathbf{hcomp}_{\lambda_d}\ \mathit{ampar}_1\ \mathit{ampar}_2 \triangleq \mathbf{upd}_{\ltimes}\ \mathit{ampar}_1\ \mathbf{with}\ d \mapsto d \triangleleft\!\circ\ \mathit{ampar}_2
\end{aligned}$$

With this translation we can completly express Minamide's system in λ_d with no loss of flexiblity.

The Functional Essence of Imperative Binary Search Trees The portion of the work from Lorenzen et al. [Lor+24b] and Leijen and Lorenzen [LL] about *first-class constructor contexts* is fairly similar, in both expressiveness and presentation, to Minamide's *hole abstractions*. A hole abstraction $\hat{\boldsymbol{\lambda}}x \mapsto t$ is written **ctx** $t[x := \_]$, whose body is the same as t but with an underscore _ (or a square □ in some versions of their work) in place of the named hole variable x. Similarly, **happ** is replaced by the ++. operator, and **hcomp** by the ++ operator. Consequently, the translation for terms from [Lor+24b; LL] system to ours is the same as for Minamide's system.

For types, [Lor+24b; LL] only define **ctx**<T>, where the hole and the resulting structure (once the hole will be filled) are of the same type T. This translates to $\mathtt{T} \ltimes \lfloor \mathtt{T} \rfloor$ in λ_d.

2.9 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. $\mathtt{Mod}_{\omega\nu}\ \{h\}\langle 0 :: \boxed{h}_{\frown} \to h\rangle$ (representing a difference list $0::\square$ that can be used non-linearly), and then used twice, each time in a different way:

$$\begin{aligned}
&\mathbf{case}\ \mathtt{Mod}_{\omega\nu}\ \{h\}\langle 0 :: \boxed{h}_{\frown} \to h\rangle\ \mathbf{of}\ \mathtt{Mod}_{\omega\nu}\ x \mapsto\\
&\quad \mathbf{let}\ x_1 := x\ \mathbf{append}\ 1\ \mathbf{in}\\
&\quad \mathbf{let}\ x_2 := x\ \mathbf{append}\ 2\ \mathbf{in} \qquad\qquad \longrightarrow^* \quad 0 :: 1 :: 0 :: 2 :: []\\
&\quad\quad \mathbf{toList}\ (x_1\ \mathbf{concat}\ x_2)
\end{aligned}$$

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 $\mathbf{upd}_{\ltimes}$, its hole names are renamed to fresh ones. One way we can implement this in practice is

to allocate a fresh copy of `x` every time we call $\mathtt{upd}_{\ltimes}$ on it (recall that `append` is implemented in terms of $\mathtt{upd}_{\ltimes}$), 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. This idea is further explored, for structure with holes specifically, in Chapter 7 of their more recent work[LL]. 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 the recipe we developed in Section 1.8, highly inspired from [Spi+22; Spi23b], and introduce a new built-in type `Token`, together with the primitive `dup`, `drop`, and `withToken`. We also switch $\mathtt{new}_{\ltimes}$ for $\mathtt{new}_{\ltimes\mathtt{IP}}$:

$$
\begin{aligned}
&\mathtt{dup} : \mathtt{Token} \multimap \mathtt{Token} \otimes \mathtt{Token} \\
&\mathtt{drop} : \mathtt{Token} \multimap 1 \\
&\mathtt{withToken} : (\mathtt{Token} \,{}_{1\infty}\!\!\multimap\, !_{\omega\infty}\mathtt{T}) \multimap\, !_{\omega\infty}\mathtt{T} \\
&\mathtt{new}_{\ltimes\mathtt{IP}} : \mathtt{Token} \multimap \mathtt{T} \ltimes \lfloor \mathtt{T} \rfloor
\end{aligned}
$$

Ampars produced by $\mathtt{new}_{\ltimes\mathtt{IP}}$ have a linear dependency on a `Token`. If the same ampar, originally created with $\mathtt{new}_{\ltimes\mathtt{IP}}$ *tok*, were to be used twice in a block t, then t would require a typing context $\{\mathit{tok} :_{\omega\infty} \mathtt{Token}\}$, thus the block would be rejected by `withToken`. In the other hand, duplicating *tok* into tok_1 and tok_2 first, and then using each new token to create a different ampar would be linearly valid.

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 requires 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 $\lhd\!\circ$ no longer requires any renaming: we have the guarantee that all the names involved are globally fresh, and can only be used once, so it actually corresponds to an in-place memory update.

In Chapters 3 and 4, dedicated to the implementation of DPS in a functional setting, we will treat ampars as linear resources, in the same style as λ_d extended with `Token`s and $\mathtt{new}_{\ltimes\mathtt{IP}}$.

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 $1 \oplus (\mathtt{S} \otimes (\mathtt{T} \otimes \mathtt{U}))$. We will do that as soon as next chapter.

However, we require 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' (through operator $\lhd\circ$). 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.

2.10 Conclusion

λ_d is a purely functional calculus which supports destinations in an extremely flexible way. It supports data structures with multiple holes, and allows both composition of data structures with holes and storing destinations in data structures that, themselves, have holes.

With this, λ_d let us implement easily imperative-like algorithms in a functional setting, without sacrificing safety. Indeed, thanks to a linear type system augmented with a system of ages, the mutations introduced by destination use are opaque and controlled.

However, 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 a destination passing system can be expressed in terms of λ_d primitives, then it is 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

A first implementation of first-class destination passing in Haskell

In the previous chapter, we saw how we can build a λ-calculus from the ground up with first-class support for destination passing, and that still achieve memory safety. It's now time to put that into practice in an industrial-grade programming language.

The main ideas of this chapter have resulted in a publication at JFLA 2024[Bag24a]. Several sections have been reworked though, since the theoretical work described in Chapter 2 was still in very early stage when the article [Bag24a] got published.

3.1 Linear Haskell is our implementation target

Designing an industrial-grade programming language *just* to add support for destinations would be a gigantic task, with much risk to fail. Instead, we'll base our work on an existing functional programming language. The Haskell programming language is equipped with support for linear types through its main compiler, *GHC*, since version 9.0.1 [Ber+18]. We kindly redirect the reader to Section 1.7 for a primer on Linear Haskell and how to use it to control resource use.

There aren't many other industrial-grade functional programming languages, and most of them don't have support for linear types, which are the first piece required to make our system safe. OCaml, thanks to recent work from Lorenzen et al. [Lor+24a], has support for *affinity*, but not for *linearity* (despite the aforementioned work using the term linearity): a resource can be restricted to at most one use, but there is no way to ensure it will be used exactly once; it could be dropped without being used at all.

As a result, Haskell appeared to be a promising target for experimenting with destination passing in a practical setting, and try to implement as many ideas from λ_d (detailed in length-Chapter 2) as possible. In addition, Haskell is also a *pure* functional language, which is rather in line with the idea of hiding impure memory effects related to destinations behind a *pure* API (as presented in the introduction) by preventing any read before a structure has been completed.

However, the main challenge is that Haskell, albeit incorporating linear types, doesn't have any concept of scope or age control. So we won't be able to avoid the fundamental issues presented in Section 2.3 without imposing limitations on the flexibility of the system.

3.2 Restricting λ_d so that it can be made safe with just linear types

As we've seen in Section 2.3, linear types are not enough, alone, to make λ_d —or any other very flexible destination-passing system— safe. We definitely need ages and scope control. In this chapter, we will take a radical decision: we will make it so that destinations can only be used to build non-linear (a.k.a. unrestricted) data structures, so that the issue of scope escape disappear completely. In exchange, we won't be able to store destinations inside data structures built using ampars.

In practice, that means that a destination can only be filled with an unrestricted value (which forbids filling a destination with another destination, as destinations are linear resources). We will use the Haskell type `data UDest t` to represent those destinations for unrestricted values, and type `data UAmpar s t` to represent these ampars where the `s` side is unrestricted.

Because there is no difference when creating the spine of a data structures that is linear or unrestricted, our fill functions that adds a new hollow constructor to an existing structure will be exactly the same as in λ_d. Only the signature of fill operators acting on the leaves of data structures will change in the implementation compared to the formal definition.

The mapping of operators and types between λ_d and DPS Haskell is given in Figure 3.1

In λ_d, destinations always have finite ages. As a result, we use age ∞ as a marker that something doesn't contain destinations. That way, can still enforce scope control independently of linearity. This is particularly visible in rule λ_d–TY/$\ltimes$FROM, where a linear resource can be extracted from the right of an ampar as long as it has age ∞.

In DPS Haskell however, we don't have ages. So we use the fact that destinations are always linear too, and we employ the ω multiplicity as a marker that something doesn't contain destinations. Of course, this isn't cheap; by doing so we conflate the absence of destination with non-linearity. Controlling scopes that way means we are overly conservative (to preserve safety), which restrain the possibilities of a few operators.

Now, let's see how programs look like in DPS Haskell. We'll start by reimplementing examples from the previous chapter, to see how they take form in a practical setting.

3.3 A glimpse of DPS Haskell programming

The following subsections present three typical cases in which DPS programming brings expressiveness or performance benefits over a more traditional functional implementation. We'll start by revisiting examples from Section 2.2.

3.3.1 Efficient difference lists, and proper memory model

Linked lists are a staple of functional programming, but they aren't efficient for concatenation, as we've seen in Section 2.2.3, 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.

Destination-filling operators:

λ_d	DPS Haskell		
$d \triangleleft ()$	`fill @'() d`	or	`d &fill @'()`
$d \triangleleft$ Inl	`fill @'Inl d`	or	`d &fill @'Inl`
$d \triangleleft$ Inr	`fill @'Inr d`	or	`d &fill @'Inr`
$d \triangleleft (,)$	`fill @'(,) d`	or	`d &fill @'(,)`
$d \triangleleft \mathsf{Mod}_{\omega\infty}$	`fill @'Ur d`	or	`d &fill @'Ur`
$d \triangleleft (\lambda x \,{}_{m}\!\mapsto term)$	`fillLeaf (\x -> term) d`	or	`d &fillLeaf (\x -> term)`
$d \triangleleft\!\circ\, term$	`fillComp term d`	or	`d &fillComp term`
$d \blacktriangleleft term$	`fillLeaf term d`	or	`d &fillLeaf term`

(one can see `&fill @'` as the equivalent of $\triangleleft$ in λ_d, `&fillLeaf` as $\blacktriangleleft$, and `&fillComp` as $\triangleleft\!\circ$)

Types:

λ_d	DPS Haskell
$\lfloor_{\omega\infty} T \rfloor$	`UDest t`
$!_{\omega\infty} S \ltimes T$	`UAmpar s t`

Figure 3.1: Mappings between λ_d and DPS Haskell

In a functional setting, *difference lists* are the closest equivalent, offering constant-time concatenation, and constant-time conversion to a normal linked list. Usually, as we've seen previously, functional difference lists are encoded as functions, but thanks to destination-passing style, we can represent them as an actual list with a hole at the end.

This example will be the opportunity to show how DPS Haskell operates on memory. So far, with λ_d, we worked on a theoretical memory model with global substitutions (on the whole evaluation context) and no representation of allocations. Here, with DPS Haskell, we assume that we have a proper heap. We pay no attention right now to garbage collection / deallocation; instead we will deal with this topic in Section 3.5.1.

Following the mapping tables of Section 3.2, we know the DPS Haskell representation of difference lists is `type DList t = UAmpar [t] (UDest [t])`. As announced previously, because our difference lists are built using UAmpars, they cannot host linear resources. That's the major limitation of this first approach of DPS programming for Haskell.

As with ampars in λ_d, the left side of the UAmpar carries the structures being built, while the right side carries the destinations of the structure: the `UDest [t]` must be filled (with an unrestricted `[t]`) to get a readable (unrestricted) `[t]`.

The implementation of destination-backed difference lists is presented in Listing 1. Note that Haskell uses `::` for typing, and `(:)` for the list *cons* constructor, which is the opposite of what these symbols mean in λ_d.

```
-- we recall here the definition of list type in Haskell
data [t] = []        -- nil constructor
         | (:) t [t] -- cons constructor

-- we also reuse the same linear token API as in Section 1.8
withToken :: (Token ⊸ Ur t) ⊸ Ur t

type DList t = UAmpar [t] (UDest [t])

newDList :: Token ⊸ DList t
newDList = newUAmpar @[t]

append :: DList t ⊸ t → DList t
dlist `append` x =
  dlist `updWith` \d → case (d &fill @'(:)) of
    (dh, dt) → (dh &fillLeaf x) ; dt

concat :: DList t ⊸ DList t ⊸ DList t
dlist1 `concat` dlist2 = dlist1 `updWith` \dt1 → (dt1 &fillComp dlist2)

toList :: DList t ⊸ Ur [t]
toList dlist = fromUAmpar' (dlist `updWith` \dt → dt &fill @'[])
```

Listing 1: Implementation of difference lists in DPS Haskell

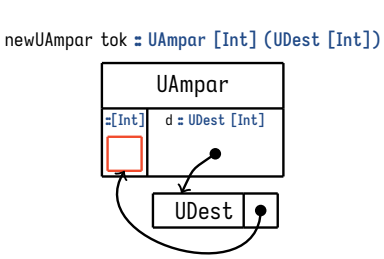


Figure 3.2: `newUAmpar tok`

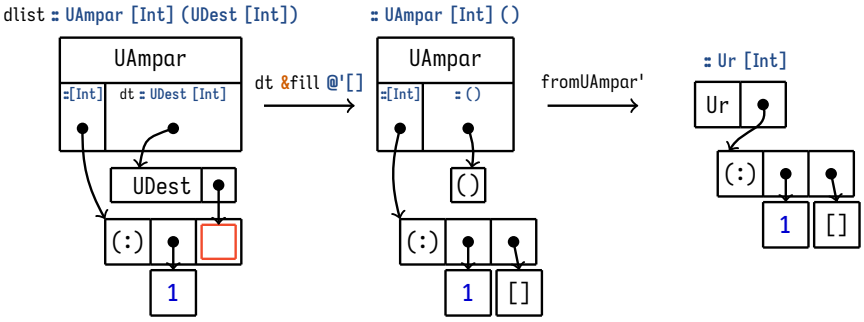


Figure 3.3: Memory behavior of `toList dlist`

- `newDList` is just an alias for `newUAmpar`, to linearly exchange a token for a new `UAmpar [t] (UDest [t])` —that is, exactly `DList t`. Recall that UAmpars are treated as linear resources now, and are created in exchange of a linear token, following the principles described in Sections 1.8 and 2.9. There is no data in this new UAmpar yet, it's just a hole and a destination pointing to it, as we see on Figure 3.2;

- `append` (Figure 3.4) 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 :: UDest [t]` with a hollow `(:)` constructor that has two new holes, pointed to by `dh :: UDest t` and `dt :: UDest [t]`.

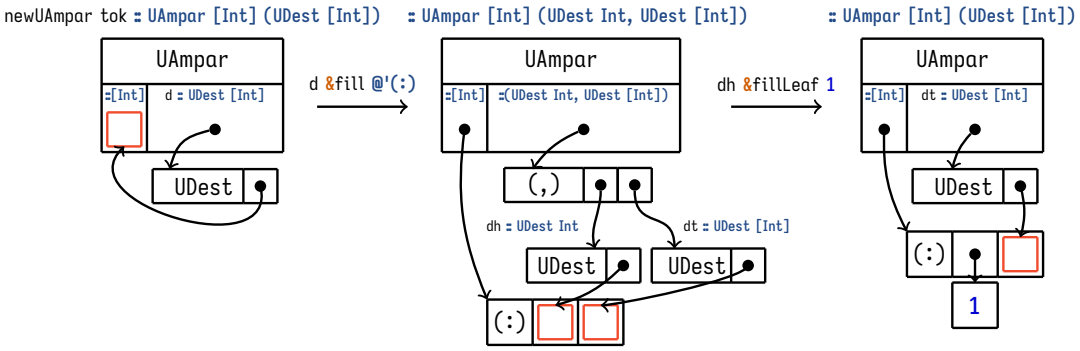


Figure 3.4: Memory behavior of `append newUAmpar 1`

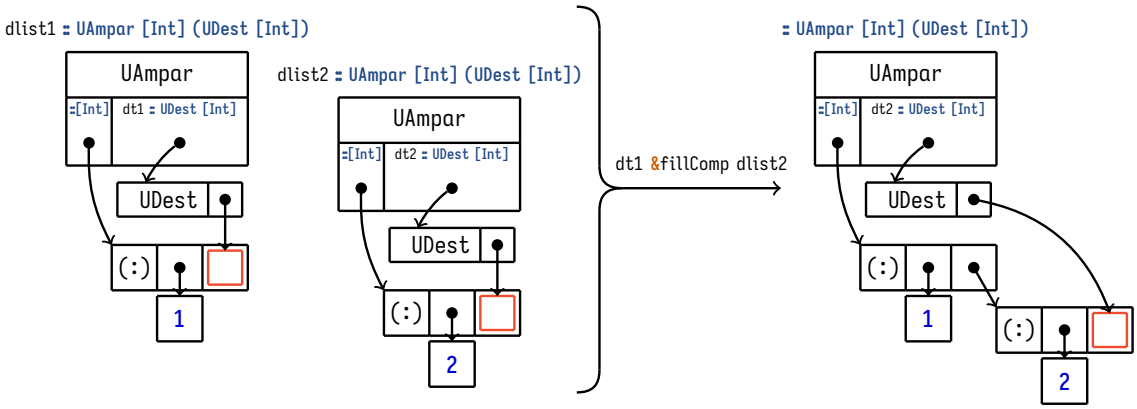


Figure 3.5: Memory behavior of `concat dlist1 dlist2` (based on `fillComp`)

Then, `fillLeaf` fills the hole represented by `dh` with the value of type `t` to append. The hole at the end of the resulting difference list is the one pointed by `dt : UDest [t]` which hasn't been filled yet, and stays on the right side of the resulting UAmpar.

- `concat` (Figure 3.5) concatenates two difference lists, `dlist1` and `dlist2`. It uses `fillComp` to fill the destination `dt1` of the first difference list with the root of the second difference list `dlist2`. The resulting `UAmpar` 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 the address of the second list into the hole of the first one.
- `toList` (Figure 3.3) completes the UAmpar structure by plugging a new *nil* `[]` constructor into its hole with `fill @'[]`, and then removes the `UAmpar` wrapper as the structure is now complete, using `fromUAmpar'`. The completed list is wrapped in `Ur`, as it is allowed to be used non-linearly —it's always the case with UAmpars— and so that it can also escape the linear scope created by `withToken` if needed.

Linearity of UAmpars, enforced by the linear token technique, is essential[17] to allow safe, in-place, memory updates when destination-filling operations occur, as shown in Figures 3.2 to 3.5. Otherwise, we could first complete a difference list with `let Ur l = toList dlist`, then add

[17] In λ_d we could choose to use a smart renaming technique for hole names (morally equivalent to copy-on-write) so that ampars don't have to be managed linearly. Here, as we aim for an efficient implementation, the idea of copying ampars and updating the target of destinations on-the-go doesn't seem promising.

a new cons cell to `dlist` with `append dlist x` (actually reusing the destination inside `dlist` for the second time). This would create a hole inside `l`, although it is of type `[t]` so we would be able to pattern-match on it, and might get a segfault!

We could expect the implementation of difference list described above to be more efficient than the functional encoding (where a difference list `x :` □ is represented by the function `\ys -> x : ys`). We'll see in Section 3.6 that the prototype implementation of DPS Haskell primitives (covered in Section 3.5) cannot yet demonstrate these performance improvements.

3.3.2 Breadth-first tree traversal

Let's move on now to breadth-first tree traversal. We want to traverse a binary tree and update its nodes values so that they are numbered in a breadth-first order.

In Section 2.4, we took full advantage of λ_d's flexibility, and implemented the breadth-first traversal using a queue to drive the processing order, as we would do in an imperative programming language. This queue would contain both input subtrees and destination to output subtrees, in breadth-first order. Furthermore, the queue was implemented efficiently using a pair of a queue and a ampar-based difference list.

Unfortunately, in DPS Haskell, destinations cannot be filled with linear elements. So an ampar-based difference list wouldn't be able to store the destinations we need to store (the one representing the output subtrees), and consequently, neither would an ampar-based efficient queue.

However, we can still use the regular, non-ampar-based data structures of Haskell to build a suitable Hood-Melville[HM81] queue. That's at least one thing we get from using a language that isn't fully based on destinations to build data structures[18]. The implementation of Hood-Melville queues is presented in Listing 2.

As before, for the breadth-first traversal, we'll keep a queue of pairs of a tree to be relabeled and of the destination where the relabeled result is expected, and process each of them when their turn comes. Nothing other than choice of implementation for the queue will change. The DPS Haskell implementation of breadth-first tree traversal is provided in Listing 3.

Except from the choice of queue implementation, and the fact that ampars must be created from `Token`s, this implementation is very much the same as in Section 2.4. In the signature of `go` function, the linear arrow enforces the fact that every destination ever put in the queue is eventually consumed at some point, which guarantees that the output tree is complete after the function has run.

We will see in Section 3.6 that this imperative-like implementation of breadth-first traversal in DPS Haskell, albeit not using difference-list-based efficient queues, still presents great performance gains compared to the fancy functional implementation from Gibbons et al. [Gib+23].

[18] Destination-based data structure building is more general than constructor-based structure building as long as we can fill destinations with linear resources, as in λ_d. When we add the restriction that destination can only be filled with unrestricted elements, as we do in DPS Haskell, then it is no longer more general than constructor-based structure building, and we need to conserve both building ways in the language to stay at the same level of expressiveness.

```haskell
data HMQueue t = HMQueue [t] [t]

newHMQueue :: HMQueue t
newHMQueue = HMQueue [] []

singleton :: t ⊸ HMQueue t
singleton x = HMQueue [x] []

toList :: HMQueue t ⊸ [t]
toList (HMQueue front backRev) = front ++ reverse backRev

enqueue :: HMQueue t ⊸ t ⊸ HMQueue t
enqueue (HMQueue front backRev) x = HMQueue front (x : backRev)

dequeue :: HMQueue t ⊸ Maybe (t, HMQueue t)
dequeue (HMQueue front backRev) = case front of
  [] -> case reverse backRev of
    [] -> Nothing
    (x : front') -> Just (x, HMQueue front' [])
  (x : front') -> Just (x, HMQueue front' backRev)
```

Listing 2: Implementation of Hood-Melville queue in Haskell

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

relabelDPS :: Token ⊸ Tree t → Tree Int
relabelDPS tree = fst (mapAccumBFS (\st _ → (st + 1, st)) 1 tree)

mapAccumBFS :: ∀ s t u. Token ⊸ (s → t → (s, u)) → s → Tree t → Ur (Tree u, s)
mapAccumBFS tok f s0 tree =
  fromUAmpar (newUAmpar @(Tree u) tok `updWith` \dtree → go s0 (singleton (Ur tree, dtree)))
  where
    go :: s → Queue (Ur (Tree t), UDest (Tree u)) ⊸ Ur s
    go st q = case dequeue q of
      Nothing → Ur st
      Just ((utree, dtree), q') → case utree of
        Ur Nil → dtree &fill @'Nil ; go st q'
        Ur (Node x tl tr) → case (dtree &fill @'Node) of
          (dy, dtl, dtr) →
            let q'' = q' `enqueue` (Ur tl, dtl) `enqueue` (Ur tr, dtr)
                (st', y) = f st x
             in dy &fillLeaf y ; go st' q''
```

Listing 3: Implementation of breadth-first tree traversal in DPS Haskell

```
data Token
dup :: Token ⊸ (Token, Token)
drop :: Token ⊸ ()
withToken :: ∀ t. (Token ⊸ Ur t) ⊸ Ur t

data UAmpar s t
newUAmpar :: ∀ s. Token ⊸ UAmpar s (UDest s)
tokenBesides :: ∀ s t. UAmpar s t ⊸ (UAmpar s t, Token)
toUAmpar :: ∀ s. Token ⊸ s → UAmpar s ()
fromUAmpar :: ∀ s t. UAmpar s (Ur t) ⊸ Ur (s, t)
fromUAmpar' :: ∀ s. UAmpar s () ⊸ Ur s
updWith :: ∀ s t u. UAmpar s t ⊸ (t ⊸ u) ⊸ UAmpar s u

data UDest t
type family UDestsOf lCtor t  -- returns dests associated to fields of constructor
fill :: ∀ lCtor t. UDest t ⊸ UDestsOf lCtor t
fillComp :: ∀ s t. UAmpar s t ⊸ UDest s ⊸ t
fillLeaf :: ∀ t. t → UDest t ⊸ ()
```

Listing 4: Destination API for Haskell

3.4 DPS Haskell API and Design concerns

Listing 4 presents the pure API of DPS Haskell. This API is sufficient to implement all the examples of Section 3.3. This section explains its various parts in detail and how it compares to Chapter 2.

3.4.1 The `UAmpar` type

As with Ampars from the previous chapter, UAmpar structures, that serve as a wrapper for structures with holes, can be freely passed around and stored, but need to be completed before any reading i.e. pattern-matching can be made on them. As a result, `UAmpar s t` is defined as an opaque data type, with no public constructor.

In `UAmpar s t`, `s` stands for the type of the structure being built, and `t` is the type of what needs to be linearly consumed before the structure can be read. Eventually, when complete, the structure will be wrapped in `Ur`; this illustrates the fact that it has been made only from unrestricted elements.

The `newUAmpar` operator is the main way for a user of the API to create a new UAmpar. Its signature is equivalent to the one of $\texttt{new}_{\ltimes \texttt{IP}}$ from Section 2.9.

Also, sometimes, when programming in DPS style, we won't have a `Token` at hand, but only an existing `UAmpar`. In that case, we can piggyback on the linearity of that existing UAmpar (as it linearly depends, directly or indirectly, on a token) to get a new linear `Token` so that we can spawn a new UAmpar, or any other linear resource really. This is the goal of the `tokenBesides` function.

The operator `tokenBesides` is particularly useful when implementing efficient queues in DPS Haskell (as in Section 2.2.3):

```
type DList t = UAmpar [t] (UDest [t])
data EffQueue t = EffQueue [t] (DList t)

newEffQueue :: Token ⊸ EffQueue t
newEffQueue tok = EffQueue [] (newUAmpar @[t] tok)

dequeue :: EffQueue t ⊸ Maybe (t, EffQueue t)
dequeue (EffQueue front back) = case front of
  [] -> case (toList back) of
    Ur [] -> Nothing
    Ur (x : xs) -> Just (x, (EffQueue xs (newUAmpar @[t] tok)))
  (x : xs) -> Just (x, (EffQueue xs back))
```

In the second branch of the inner-most `case` in `dequeue`, when we have transformed the back difference list (from which we used to write) into the new front list (from which we will now read), we have to create a new back difference list, that is, a new ampar. However, we don't have a token at hand. It would be quite unpractical for a function like `dequeue` to ask for a linear token to be passed; it's better if tokens are only requested when spawning a new data structure, not when operating on one. That's why `tokenBesides` comes handy: we can instead reuse the linearity requirement of the existing UAmpar:

```
tokenBesides :: ∀ s t. UAmpar s t ⊸ (UAmpar s t, Token)

dequeue :: EffQueue t ⊸ Maybe (t, EffQueue t)
dequeue (EffQueue front back) = case front of
  [] -> case tokenBesides back of (back, tok) -> case (toList back) of
    Ur [] -> drop tok ⨟ Nothing
    Ur (x : xs) -> Just (x, (EffQueue xs (newDList tok)))
  (x : xs) -> Just (x, (EffQueue xs back))
```

This new version is not perfect either though. We have to create a new token before even knowing if we will need it. Indeed, we can only know if a new token is really needed when we have read the content of the old UAmpar, and we can only read it when it is no longer a UAmpar... so at that point we cannot spawn a new token any longer. So the solution is to always spawn a new token before consuming the old UAmpar, taking the risk that if the queue is really empty, we will have to discard the fresh token immediately (with `drop`) before returning `Nothing`. Implicit token passing, as proposed by Linear Constraints[Spi+22; Spi23a], seems to really be the way forward to get rid of this kind of issues altogether.

Getting out of an UAmpar Same as in λ_d, the structure encapsulated within a UAmpar can be released in two ways: with `fromUAmpar'`, the value on the `t` side must be unit (`()`), and just the complete `s` is returned, wrapped in `Ur`. With `fromUAmpar`, the type on the `t` side must be of the form `Ur t'`, and then a pair `Ur (s, t')` is returned.

It is actually safe to wrap the structure that has been built in `Ur` because, as we've said previously, its leaves either come from non-linear sources (as `fillLeaf :: t → UDest t ⊸ ()` 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; and its spine, made of hollow constructors, can be duplicated at will.

3.4.2 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 `UDest t`, three functions are available:

`fillLeaf :: ∀ t. t → UDest t ⊸ ()` uses a value of type `t` to fill the hole represented by the destination, as ◀ from λ_d. However, if the destination is consumed linearly, as in λ_d, the value to fill the hole isn't (as indicated by the first non-linear arrow). This is key to the fact that UAmpar only host unrestricted data. Memory-wise, when `fillLeaf` is used, the address of the object of type `t` is written into the memory cell pointed to by the destination of type `UDest t` (see Figure 3.8).

`fillComp :: ∀ s t. UAmpar s t ⊸ UDest s ⊸ t` is used to plug two `UAmpar` objects together. The parent `UAmpar` isn't represented in the signature of the function, as with the similar operator ◁⚬ from λ_d. Instead, only the hole of the parent UAmpar that will receive the address of the child UAmpar is represented in the signature of the function by `UDest s`; while `UAmpar s t` in the signature refers to the child UAmpar. A call to `fillComp` always takes place in the scope of `` `updWith` `` over the parent one:

```
1 parent :: UAmpar BigStruct (UDest SmallStruct, UDest OtherStruct)
2 child :: UAmpar SmallStruct (UDest Int)
3 comp = parent `updWith` \(ds, do) → (ds &fillComp child, do)
4       :: UAmpar BigStruct (UDest Int, UDest OtherStruct)
```

The resulting structure `comp` is morally a `BigStruct` like `parent`, that inherited the hole from the child structure (`UDest Int`) and still has its other hole (`UDest OtherStruct`) waiting to be filled. An example of memory behavior of `fillComp` in action can be seen in Figure 3.5.

Finally, `fill :: ∀ lCtor t. UDest t ⊸ UDestsOf lCtor t` is the generic function to fill a destination with hollow constructors. It takes a data constructor as a type parameter (`lCtor`) and allocates a corresponding hollow constructor, that is, a heap object that has the same header as the specified constructor but unspecified fields. The address of the allocated hollow constructor is written in the destination that is passed to `fill`. An example of the memory behavior of `fill @'(:) :: UDest [t] ⊸ (UDest t, UDest [t])` is given in Figure 3.6 and the one for `fill @'[] :: UDest [t] ⊸ ()` is given in Figure 3.7.

The `fill` function also returns one destination of matching type for each of the fields of the constructor; in Haskell this is represented by `UDestsOf lCtor t`. `UDestsOf` is a type family (i.e. a function from types to types) whose role is to map a constructor (lifted as a type) to the type of destinations for its fields. For example, `UDestsOf '[] [t] = ()` and `UDestsOf '(:) [t] = (UDest t, UDest [t])`. More generally, the `UDestsOf` typeclass reflects the duality between the types of fields of a constructor and the ones of destinations for a hollow constructor, as first evoked in Section 2.2.1.

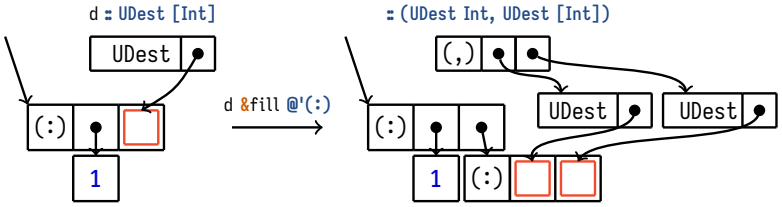


Figure 3.6: Memory behavior of `fill @'(:) :: UDest [t] ⊸ (UDest t, UDest [t])`

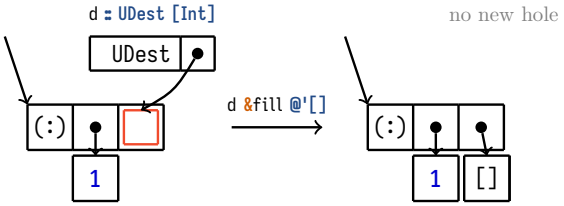


Figure 3.7: Memory behavior of `fill @'[] :: UDest [t] ⊸ ()`

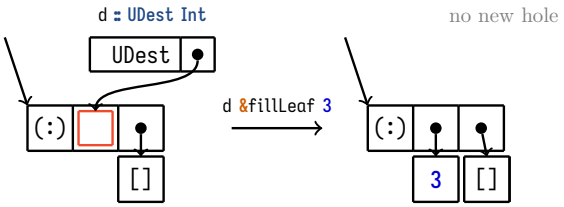


Figure 3.8: Memory behavior of `fillLeaf :: t → UDest [t] ⊸ ()`

3.5 Compact Regions: a playground to implement the DPS Haskell API

Having incomplete structures in the memory inherently introduces a lot of tension with both the garbage collector and compiler. Indeed, the garbage collector of GHC assumes that every heap object it traverses is well-formed, whereas UAmpar structures are absolutely ill-formed: they contain uninitialized pointers, which the GC should absolutely not follow. Also, the compiler and GC can make some optimizations because they assume 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). Consequently, we looked for an alternative memory management scheme, mostly independent from the garbage collector, that would let us implement the DPS Haskell API more easily.

3.5.1 Compact Regions

Compact regions from Yang et al. [Yan+15] are special memory *arenas* for 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, this doesn't change the principle). Also, compact regions are immobile in memory; the GC won't move them, so a destination to a hole in the compact region can just be implemented as a raw pointer (type `Addr#` in Haskell): `data UDest r t = UDest Addr#`, as we have the guarantee that the UAmpar containing the pointed hole won't move.

By using compact regions to implement DPS programming, we completely elude the concerns of tension between the garbage collector and UAmpar structures that we just mentioned above. In exchange, we get two extra restrictions imposed by compact regions. First, every structure in a region must be in a fully-evaluated form. This is contrasting with the usual Haskell evaluation scheme, where everything is lazy by default. Consequently, as regions are strict, any heap object that is copied to a region is first forced into normal form (i.e. it is being fully evaluated). This might not always be a win, sometimes laziness is preferable for better performance.

Secondly, data in a region cannot contain pointers to the garbage-collected heap, or pointers to other regions: it must be self-contained. This forces us to slightly modify the API, to add a phantom type parameter `r` that tags each object with the identifier of the region it belongs to so that our API remains memory-safe, without runtime checks. There are two related consequences: first, when a value from the garbage-collected heap is used as an argument to `fillLeaf`, it has to be fully evaluated and copied into the region instead of making just a pointer update; and secondly, `fillComp` can only plug together two `UAmpar`s 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 `inRegion` function is the new addition to the modified API presented in Listing 5. It receives an expression of arbitrary type in which `r` must be a free type variable. Internally, when used, it spawns a new compact region and a fresh type `r` (not a variable), and uses the `reflection` library to provide an instance of `Region r` on-the-fly that links `r` and the `RegionInfo` for the new region, and then make type application for the supplied expression at the concrete type `r`. Actually, `inRegion` is a form of scope function (although this time, it isn't linear).

3.5.2 DPS programming in Compact regions: Deserializing, lifetime, and garbage collection

We will now study an example, related to implementation concerns, for which DPS programming really shines (and for which compact regions are a really nice fit).

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 can be freed as a block. Compact regions are, in fact, the perfect tool for this job, as the GC never follows pointers into a compact region and consider each region as a having the same lifetime for all its contents.

```
data Token
dup :: Token ⊸ (Token, Token)
drop :: Token ⊸ ()
withToken :: ∀ t. (Token ⊸ Ur t) ⊸ Ur t

type Region r :: Constraint
inRegion :: ∀ t. (∀ r. Region r ⇒ t) ⊸ t

data UAmpar r s t
newUAmpar :: ∀ r s. Region r ⇒ Token ⊸ UAmpar s (UDest r s)
tokenBesides :: ∀ r s t. Region r ⇒ UAmpar r s t ⊸ (UAmpar r s t, Token)
toUAmpar :: ∀ r s. Region r ⇒ Token ⊸ s → UAmpar r s ()
fromUAmpar  :: ∀ r s t. Region r ⇒ UAmpar r s (Ur t) ⊸ Ur (s, t)
fromUAmpar' :: ∀ r s. Region r ⇒ UAmpar r s () ⊸ Ur s
updWith :: ∀ r s t u. Region r ⇒ UAmpar r s t ⊸ (t ⊸ u) ⊸ UAmpar r s u

data UDest r t
type family UDestsOf lCtor r t  -- returns dests associated to fields of constructor
fill :: ∀ lCtor r t. Region r ⇒ UDest r t ⊸ UDestsOf lCtor r t
fillComp :: ∀ r s t. Region r ⇒ UAmpar r s t ⊸ UDest r s ⊸ t
fillLeaf :: ∀ r t. Region r ⇒ t → UDest r t ⊸ ()
```

Listing 5: Destination API using compact regions

If we use compact regions as-is, with the API provided with GHC, we would first deserialize the payload normally, in the GC heap, then copy it into a compact region and then 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 one 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. Fortunately, with our implementation of DPS Haskell with compact regions, we now have a much better way to allocate and build structures directly into these compact regions!

Let's see how using destinations and compact regions for a parser of S-expressions (representing our request payload) 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:

- `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.

```
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 6: Implementation of the S-expression parser without destinations

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 6 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 also store, in the parsed result, the index `i` corresponding to the end position of the structure in the input string.

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 and in the right order!

Code for `parseSListDPS` is presented in Listing 7. 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);

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

Listing 7: Implementation of the S-expression parser with destinations

- 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 writing a new cons cell with `fill @'(:)` into the hole represented by the `UDest [SExpr]`, writing an element to the new *head* destination, and then doing a recursive call with the new *tail* destination passed as an argument (which has type `UDest [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;
- ultimately, DPS functions just return the offset of the next character to read instead of returning a parsed value (as the parsed value is written directly into the destination received as a parameter).

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.

We see here that compact regions and destination-passing style are quite symbiotic; compact regions makes the DPS Haskell API easy to implement, and destination-passing style makes compact regions more efficient and flexible to use.

3.5.3 Memory representation of `UAmpar` objects in Compact Regions

As we detailed in Sections 2.6.1 and 3.3.1, we want `UAmpar r s t` to contains a value of type `s` and one of type `t`, and let the value of type `s` free when the one of type `t` has been fully consumed (or linearly transformed into `Ur t'`).

We also know that `newUAmpar` returns an `UAmpar r s (UDest s)`: there is nothing more here than an empty memory cell that will host the root of the future structure of type `s`, which the associated destination of type `UDest s` points to, as presented in Figure 3.2. As we said earlier, whatever goes in the destination is exactly what will be retrieved in the `s` side.

The naive and most direct idea is to represent `UAmpar r s t` in memory as a pair-like constructor with one field of type `s` and one of type `t`, with the first field being there to host the root of the structure being built (as in Figure 3.2). However, the root hole, that will later host the structure of type `s`, might contain garbage data, so the GC must be prevented to read it and follow uninitialized pointers (otherwise we risk a segmentation fault). In our compact region implementation, we can prevent this issue by having the root hole live inside the compact region (so that the GC doesn't look into it). But we also want the `UAmpar` constructor to live in the garbage-collected heap so that it can sometimes be optimized away by the compiler, and always deallocated as soon as possible. These two requirements are, apparently, incompatible, as the root hole corresponds to the first field of the `UAmpar` constructor in this representation.

Given that UAmpars hold unrestricted data, one potential workaround is to represent `UAmpar r s t` in memory as a constructor with one field of type `Ur s` and one of type `t` (instead of one of type `s` and one of type `t` previously). It means we need an intermediary `Ur` constructor between the UAmpar wrapper and the structure of type `s` being built, and we can decide to allocate this `Ur` constructor inside the region, with its field of type `s`, also inside the region, serving as the root hole. With this approach, illustrated in Figure 3.9, we have a root hole that won't be read by the GC, while the `UAmpar` wrapper can live in the GC heap without issues

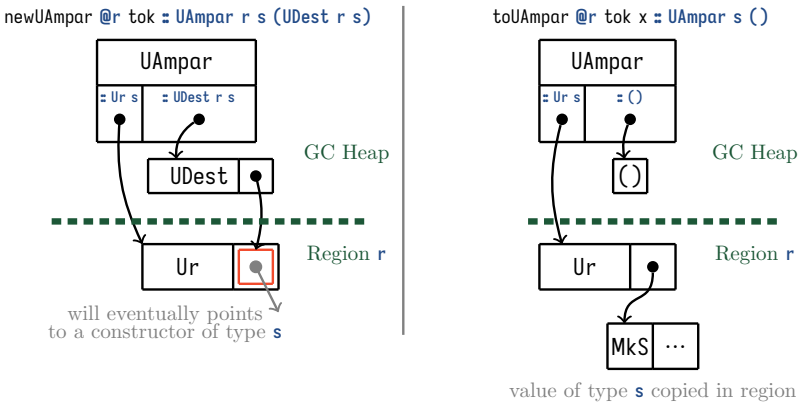


Figure 3.9: Representation of UAmpars in the compact region implementation, using `Ur` (not chosen)

and be discarded as soon as possible, as we wanted! We also get a very efficient implementation of `fromUAmpar'`, as the first field of a completed UAmpar is directly what `fromUAmpar'` should return (it doesn't help for `fromUAmpar` though). The downside is that every UAmpar will now allocate a few words in the region (to host the `Ur` constructor[19]) that won't be collected by the GC for a long time even if the parent `UAmpar` is collected. In particular, this makes `toUAmpar` quite inefficient memory-wise, as it will have to allocate a `Ur` wrapper in the region that is useless for already complete structures (because already complete structures don't need to have a root hole living in the compact region, as they don't have holes at all).

Another option, that we decided to go for in our real implementation, is to allocate an object inside the compact region to host the root hole of the structure *only* for truly incomplete structures created with `newUAmpar`. For already complete structures that are turned into an `UAmpar` with `toUAmpar`, we skip this allocation; but we preserve the same underlying types for the fields of the UAmpar in both cases (`newUAmpar` and `toUAmpar`). This is made possible by the use of the special indirection object (`stg_IND` in GHC codebase) in the UAmpar returned by `newUAmpar`. A `stg_IND` object is a kind of one-field constructor that is considered to have type `s` although its field also has type `s`. We illustrate this in Figure 3.10:

- in the pair-like structure returned by `newUAmpar`, the `s` side points to an indirection object, which is allocated in the region, and serves as the root hole for the incomplete structure (because it is in the compact region, the GC won't follow garbage pointers that it might contain at the moment);
- in the pair-like structure returned by `toUAmpar`, the `s` side directly points to the object of type `s` that has been copied to the region, without indirection.

Compared to the previous solution, this one is more efficient when using `toUAmpar` (as no long-lived garbage is produced), but does no longer give efficiency benefits for `fromUAmpar'` (as we no longer produce an `Ur` object). This is the solution we arbitrarily chose to implement in the original artifact [Bag23b], but it could be worth comparing and benchmarking it with the one based on `Ur` indirection in a variety of real-world use-cases.

[19]One might ask why `Ur` can't be a zero-cost `newtype` wrapper. First, if `Ur` were a `newtype`, it wouldn't work for what we are trying to achieve here, as we are relying on the fact that it creates an indirection in memory. Secondly, and more importantly, there are semantics motivations, detailed in[Spi24].

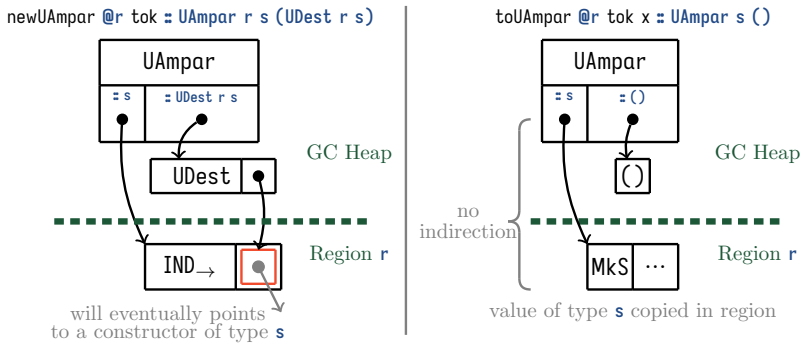


Figure 3.10: Chosen representation for UAmpars in the compact region implementation, using indirections

In the end, in [Bag23b], we have the following private definitions[20] for our API types:

```
1 data Token = Token  -- same representation as ()
2 data UAmpar r s t = UAmpar s t  -- same representation as (s, t),
3                                 -- with sometimes an indirection on the s field
4 data UDest r t = UDest Addr#  -- boxed Addr#, same representation as Ptr t
```

3.5.4 Deriving fill for all constructors with Generics

In λ_d, it was quite easy to have a dedicated destination-filling function for each corresponding data constructor. Indeed, we only had to account for a finite number of them (one for unit, two for sums, one for pair, and one for exponential; with the one for function being quite optional).

In Haskell however, we want to be able to do destination passing with arbitrary data types, even user-defined ones, so it wouldn't be feasible to implement, manually, one filling function per possible data constructor. So instead, we have to generalize all filling functions into a polymorphic `fill` function that is able to work for any data constructor.

To this extend, we create a typeclass `Fill lCtor t`, and we define `fill` as the only method of this class. Having such a typeclass let us tie the behaviour of `fill` to the type parameter `lCtor`. The idea is that `lCtor` represents the type-level name of the data constructor we want to use `fill` with. The syntax at call site is of the form `fill @'Ctor`: we lift the constructor `Ctor` into a type-level name with the `'` operator, then make a type application with `@`. We recall that the `fill` function should plug a new hollow constructor of the specified variant into the hole of a structure whose corresponding destination is received as an argument.

Let's now see how we can use the type-level name of the constructor `lCtor` and its base type `t` to obtain all the information we need to write the concrete implementation of the corresponding `fill` function. The first thing we need to know is the shape of the constructor and more precisely the number and types of its fields, as every data constructor for a linked data type[21] can be seen as a n-ary product constructor. So we will leverage `GHC.Generics` to find the required information.

[20] In the sense that they are opaque for the consumer of the API.

[21] Dealing with primitive or unboxed types is, on the other hand, quite challenging, see the mention at the end of Section 2.9.

`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 (almost) any type! Here's, for example, the `Generic` representation of `Maybe t`:

```
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 t)))
```

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 `t` (indicated by `K1 R t`).

As illustrated with the example above, the generic representation of a type contains most of what we want to know about its constructors. So, with type-level programming techniques[22], in particular with type families, we can extract the parts of this generic representation that are relevant to us (as type-level expressions), and refer to them in the head of our typeclass' instances, to have several specialized implementation of `fill`. For instance, in [Bag23b], we have one specialized implementation of `fill` for every possible number of fields in `lCtor`, from 0 to 7[23]. The main reason is that `fill` should return a tuple containing as many new destinations as there are fields in the the chosen constructor, and at the moment, there is no way to abstract over the length of a tuple in Haskell, so we need to harcode each possible case.

The resolution of the `UDestsOf lCtor t` type family into a concrete tuple type of destinations is made similarly, by type-level expressions based on the generic representation of the base type `t` which `lCtor` belongs to.

At this point we still miss a major ingredient though: the internal Haskell machinery to allocate the proper hollow constructor object in the compact region.

3.5.5 Changes to GHC internals and its RTS

We will see here how to allocate a hollow constructor, that is, 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 that can be used in Haskell code. 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.

In Haskell, a heap object is a piece of data in memory, that corresponds to a given instance of a data constructor, or a (partially-applied) function. The info table is what defines both the layout and behavior of a heap object. All heap objects representing a same data constructor (let's say `Just`) have the same info table, which acts as the identity card for this constructor, even when the associated types are different (e.g. `Just x :: Maybe Int` and `Just y :: Maybe Bool` shares the

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

[23] We could go higher, 7 is just an arbitrary limit for the number of fields that is rather common in standard Haskell libraries.

same info table). In the compilation process, during the code generation, all instances of the `Just` constructor will hold the label `Just_con_info`, that is later resolved by the linker into an actual pointer to the shared info table, that all corresponding heap objects for this constructor will carry.

On the other hand, the RTS is a static piece of code that is compiled once when GHC is built, and is then copied into every executable built by GHC. Only when a program is launched does the RTS run (to overwatch the program execution), so the RTS has no direct way to access the information that had been emitted during the compilation of the program. In particular, it has no way to inspect the info table labels have long been replaced by actual pointers (which are indistinguishable in memory from any other data). But on the other hand, the RTS is the one which knows how to allocate space inside a compact region for our new hollow constructor.

As a result, we need a way to pass information, or more precisely, the info table pointer, from compile time to the runtime. Consequently, we manage the creation of a new hollow constructor in a compact region through two new primitives:

- 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*[24] that resolves, at compile time, a constructor name into a static value representing the info table pointer of this 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 8. 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 at compile time, is to first lift the constructor into a type-level literal. Then, when wanting to pass a type as an input to a function, we then use a `Proxy t` (the unit type with a phantom type parameter `t`) as a vehicle for the . Unfortunately, due to a quirk of the compiler, primops don't have access to the type of their arguments, so using the `Proxy` trick wouldn't let us read the supplied type. Primops can, however, access their return type. So we use a phantom type `InfoPtrPlaceholder# t` as the return type for our, to somehow pass the contructor as an input!

The gist of this implementation is presented in Listing 9. 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 allocate a `stg_IND` object as mentionned in Section 3.5.3). In the case it reads a lifted data constructor, it resolves the

[24] Internal primops are macros which generates C– code at compile time.

primop call into the label which corresponds to the info table pointer of that constructor. The returned `InfoPtrPlaceholder# t` 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 `UDestsOf` 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 newUAmpar 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 10.

Putting it all together, and further evolutions Combining all these elements, we are able to implement the low-level operations needed for destination passing and hollow constructor allocation.

The implementation presented above has been thought to require the minimal number of changes on GHC, as we considered destination passing to be a niche concern that couldn't justify large exotic changes to this industrial-grade compiler.

However, since this first experimentation[Bag23a], a pull request has been opened[Bag24b] to try to merge support for hollow constructor building in compact regions into the main, publicly-distributed branch of GHC. This PR tends to take the path of a slightly more complete set of primitives than the three aforementioned ones, in particular with primitive forms for `fill` being part of it (instead of being only in library code, as it is now). Still, much work remains to be done before reaching a mergeable state. I hope to dedicate a few weeks to work on the matter at the very end or just after the end of my PhD.

3.6 Evaluating the performance of DPS programming

Benchmarking methodology It's now time to compare programs in both naive style and DPS style, and see if our claims of performance are verified with our prototype destination passing implementation[Bag23b]. With DPS programs,the output produced by the program is stored in a compact region; we cannot chance that. But as we said in Section 3.5.1, compact regions also force strictness i.e. the structure will be automatically in fully evaluated form. It might be counterproductive in some use-cases, but at least it makes things simpler for a benchmark (in which we want to force the result to a fully evaluated form).

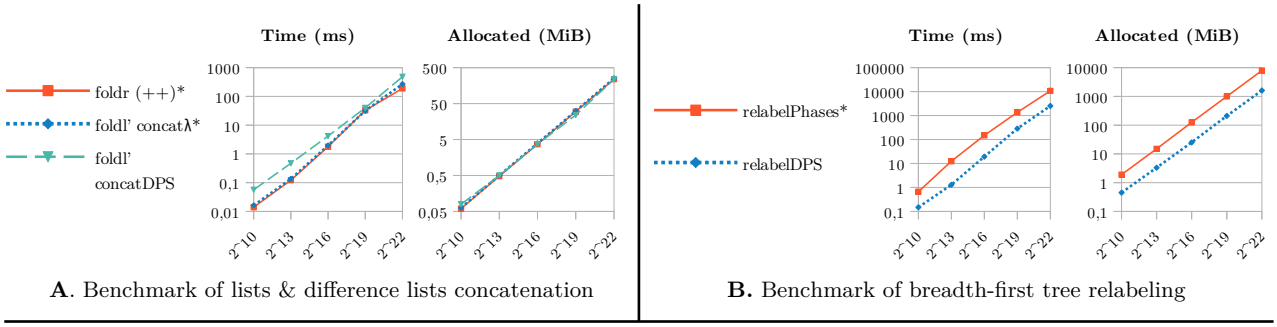
For naive versions of the programs, we have a choice to make on how to fully evaluate the result: either force each chunk of the result inside the GC heap (using `Control.DeepSeq.force`), or copy the result in a compact region that is strict by default and thus will force evaluation (using `Data.Compact.compact`).

```
// 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 8: `compactAddHollow#` implementation in `rts/Compact.cmm`

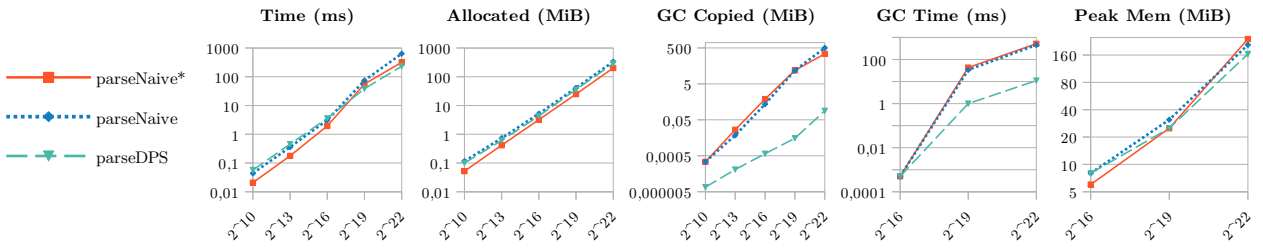
```
case primop of
  [...]
  ReifyStgInfoPtrOp → \_ →  -- we don't care about the function argument (# #)
    opIntoRegsTy $ \[res] resTy → emitAssign (CmmLocal res) $ case resTy of
      -- when 'a' is a Symbol, and extracts the symbol value in 'sym'
      TyConApp _addrLikeTyCon [_typeParamKind, LitTy (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 9: `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 3.11: Benchmarks performed on AMD EPYC 7401P @ 2.0 GHz (single core, `-N1 -O2`)

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 for list concatenation.
`foldr (++)*` has calls to `(++)` nested to the right, giving the most optimal context for list concatenation (it should run in $\mathcal{O}(n)$ time). We only use it as a reference, as it is quite unrealistic: we often want to concatenate lists with nesting to the left, e.g. when appending lines to a log. On the other hand, `foldl' concatλ*` uses function-backed difference lists, and `foldl' concatDPS` uses destination-backed ones (from Section 3.3.1), both with calls to concat nested to the left. They should still run in $\mathcal{O}(n)$, thanks to difference list magic.

We see in part **A** of Figure 3.11 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.

It's still unclear where the performance loss comes from for the destination-based version using compact regions. We tried to profile the program but we couldn't identify clear culprit that would cause a particular slowdown. We must admit that programming with destinations and

linear types in general still require a bit of embarassing boilerplate compared to naive versions, where some performance might be lost. These tests would need to be redone though in a more recent version of GHC, as performance of linear types has been improved in GHC 9.8.

Breadth-first relabeling For breadth-first tree traversal, we benchmark the implementation of Section 3.3.2 against a fancy functional implementation based on *Phases* applicatives presented in [Gib+23]:

We see in part **B** of Figure 3.11 that the destination-based tree traversal is almost one order of magnitude more efficient, both time-wise and memory-wise, compared to the implementation from Gibbons et al. [Gib+23].

Parsing S-expressions In part **C** of Figure 3.11, we compare the naive implementation of the S-expression parser and the DPS one (as presented in Section 3.5.2). 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 to produce a result in the GC heap that we evaluate fully with `force` (its name is suffixed with a star), and once to produce a result that we copy in a compact region using `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 is wrappers such as `UAmpar`s and `UDest`s that just last one generation and thus can be discarded very early by the GC, without needing to be copied into the next generation, unlike most data nodes allocated by the naive versions.

Finally, we see that 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.

3.6.1 Mapping a function over a list

It seemed important to also measure the performance of a `map` function implementation using destinations. In a strict functional language such as OCaml, the choice of implementation for `map` is crucial as the naive one makes the stack grow linearly with the size of the processed list. A strict tail-recursive version (`mapTR'`) takes $\mathcal{O}(1)$ space, but it requires an extra $\mathcal{O}(n)$ operation at the end of the processing (reversing the accumulator).

With destinations, `map` can be implemented in a tail-recursive fashion, as explained in Section 2.2.2, without the need for the reverse operation (as the list is built in a top-down approach). It can alternatively be implemented as a fold, as shown below:

```
1 append :: UDest [a] ⊸ a → UDest [a]
2 append d x = let !(dh, dt) = fill @'(:) d in fillLeaf x dh ⨟ dt
3
4 mapDPS' _ [] d = fill @'[] d
```

```
5 | mapDPS' f (x : xs) d = let !y = f x ; !d' = append d y in mapDPS' f xs d'
6 |
7 | mapDPSFold' f l dl = fill @'[] (foldl' (\d x → let !y = f x in append d y) d l)
```

We see in part 4 of Figure 3.11 that the destination-based implementations takes 1.5-4× more time than `map` and `mapTR'` (depending on the dataset size), but memory-wise both `mapDPS'` and `mapDPSFold'` are more efficient than `map`; and `mapDPS'` even manage to make 13% fewer allocs than `mapTR'` on the largest dataset.

In OCaml, `mapDPS'` is actually more performant time-wise than `mapTR'` $(0.5 - 0.85\times)$ even for small lists, as detailed in benchmarks of [BCS21].

3.7 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 UAmpar 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.

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

Chapter 4

Extending DPS support for linear data structures

One issue with the approach of the previous chapter is that we are not able to build data structures hosting linear data with a destination-based approach. This is the consequence of having a lot of flexibility in the handling of destinations, without a system of ages to prevent scope escape in a fine-grained manner. So we had to use *unrestrictedness* as a barrier for destinations.

Previous work however, such as [Min98], or [Lor+24b], would allow for efficient queues (see Section 2.2.3) or any structure with holes to store linear data, without issues with scope escape. Mainly because they don't have a concept of destinations; we can only fill holes in a structure by interacting with the incomplete structure (ampar) itself. So we ought to find a way for our Haskell implementation to be at least as expressive as these works.

4.1 Challenges of destinations for linear data

If we now allow destinations to be filled with linear data[25], we have to be extremely careful about potential scope escape. One observation is that scope escape can only happen when we let the user fill an older linear destination in a newer scope.

A potential remedy would be to not allow older linear destinations to be filled using `fillLeaf` or `fillComp` functions. Instead, linear destinations could only be filled with a value by operating on the ampar they belong to, as in [Min98] and [Lor+24b]. We will use the base name `extend` for this new family of operators. Unrestricted destinations, on the other hand, could still be used with `fill`-like functions. Filling a destination with a hollow constructor cannot cause any scope escape issues, so it's safe to do with `fill @lCtor` for both linear and unrestricted destinations. Let's see what the API could look like:

[25]Starting from this point, we will call *linear destination* an object of type $\lfloor 1 \mathtt{T} \rfloor$ or corresponding `Dest 1 t`, that is, a destination that can host linear data. Similarly, an *unrestricted destination* will be an object of type $\lfloor \omega \mathtt{T} \rfloor$ or `Dest ω t`. Regardless, a destination is always, itself, a resource that must be managed linearly.

```
data Ampar s t
data Dest m t  -- m is the multiplicity, either 1 or ω
newAmpar :: Token ⊸ Ampar s (Dest 1 s)
updWith :: Ampar s t ⊸ (t ⊸ u) ⊸ Ampar s u

-- for unrestricted destinations
fillLeaf :: t → Dest ω t → ()

-- for linear destinations
extendLeaf :: t ⊸ Ampar s (Dest 1 t) ⊸ Ampar s ()
```

The idea is that `extendLeaf` is a way of opening a new scope, and filling the linear destination, belonging to that fresh scope, immediately, without letting the user interfere. So we are confident that the linear destination filled by `extendLeaf` is strictly younger than whatever ends up stored into it, which prevents scope escape (recall the sane example of Section 2.3, where an older destination is stored into a younger one with no issues).

Except...it doesn't work:

```
newAmpar tok0 `updWith` \(d0 :: Dest 1 (Dest 1 t)) ->
  newAmpar tok1 `updWith` \(d1 :: Dest 1 t) ->
    (newAmpar tok2 `updWith` \(d2 :: Dest ω ()) -> d2 &fillLeaf () ⨟ d0) &extendLeaf d1
```

In this particularly cursed but valid excerpt[26], we hide an old destination `d0` inside the nest of a fresher one `d2` that we've just consumed. So we have an ampar `(newAmpar `updWith` \d2 -> d2 &fillLeaf () ⨟ d0)` whose right-hand side contains a destination `d0` which isn't its own; and we can use that together with `extendLeaf` to cause scope escape (as `d1` get stored in the outermost ampar)!

There seems to be an inevitable friction between having a flexible `updWith` function, and having safe linear destinations, when we don't have a proper age system to ensure there is no cheating with scopes.

Restricting `updWith` If we want to prevent older destinations from hiding in newer ampars, we can ask for `updWith` to *not* capture any linear resource. That way, the only destinations accessible and usable in the scope of `updWith` are the one coming immediately from the ampar being opened.

This is done by changing the signature from `updWith :: Ampar s t ⊸ (t ⊸ u) ⊸ Ampar s u` to `updWith :: Ampar s t ⊸ (t ⊸ u) → Ampar s u`. Notice the unrestricted function arrow →: it means the linear function supplied by the user must not capture anything linear (but still must use its argument linearly). Adding this drastic restrictions has many consequences.

First, obviously, we cannot use any linear resource coming from outer scopes inside the scope of `updWith`. In particular, we cannot use a `Token` coming from the top-level linear scope. Instead, we have to start a new nested linear scope (by calling `withToken`) if we want to create new ampars in that scope, or tweak `updWith` to also provide a linear token to the inner scope: `updWith :: Ampar s t ⊸ ((t, Token) ⊸ u) → Ampar s u`.

[26] It's valid in Linear Haskell, but the example would be rejected by λ_d, as `d0` would have age $\uparrow^2$ when it is used as a variable in the innermost scope, which isn't allowed by rule λ_d–TY/ID.

Also, we still cannot fill linear destinations using a `fillLeaf` or `fillComp` function, for two main reasons. One being that we cannot bring linear resources in the scope anyway, so we wouldn't have something useful to fill into the linear destination. The second being that in addition to scope escape, we must be careful about sibling destinations from the same scope interacting with one another:

```
fromAmpar' (
  newAmpar @(t, Dest 1 t) tok `updWith` \(d :: Dest 1 (t, Dest 1 t)) ->
    case (d &fill @'(,)) of (dt :: Dest 1 t, ddt :: Dest 1 (Dest 1 t)) ->
      ddt &fillLeaf dt
)
```

Here, we create a new ampar for a pair of a `t` and a `Dest 1 t`. After filling the root destination with a hollow pair constructor, we get two destinations, `dt` and `ddt`. If we have access to a `fillLeaf` function operating with a linear destination on the left, we can actually put `dt` into `ddt`. Because we've consumed the two destinations and unit is returned, it seems that the ampar is complete, while in fact we've never provided a value of type `t` for the first field of the pair! So the structure is still incomplete while we are allowed to call `fromAmpar'` on it and read it!

So far we hadn't spoke about that danger, because mechanisms used to stop scope escape in Chapters 2 and 3, that is, the age system, or absence of linear destinations, are enough to also prevent sibling destinations from interacting with one another. But here, the limitation put on `updWith` (not allowing capture of linear resources) only stops scope escape, but not this other issue, because both sibling destinations are coming from the same, freshest scope. Hence, we still cannot have a function `fillLeaf` or `fillComp` operating on linear destinations.

4.2 extend functions for ampars with a single hole

We've just seen that to use linear destinations safely, we need operators that work on closed ampars, instead of destinations. In fact, that is exactly what happens in [Min98], or [Lor+24b]: they only allow incomplete structures with one hole, and extending the structure happens by manipulating the whole incomplete structure itself, not destinations (as they don't exist in these works).

For starters, let's start on this way, and design operators for ampars with only one hole, so exactly one destination on their right-hand side. If we want always have only one hole in the ampar, then, when plugging a new hollow constructor into the existing hole, we must fill all of the new fields of the hollow constructor except one, so that only one new hole remains[27]. In other terms, it means that we must ask the user to specify a value for all except one field of the newly added constructor. The signature of such a function would be as follows:

```
extend :: ∀ lCtor k t s m fiTys kthFiTy othFiTys.
          (OnlyLinear lCtor t,
          FieldTypes lCtor t ~ fiTys,
          At k fiTys ~ kthFiTy, Remove k fiTys ~ othFiTys) ⇒
          othFiTys %m→ Ampar s (Dest m t) ⊸ Ampar s (Dest m kthFiTy)
```

[27] It's still possible to use `fill @'(:)` on a linear destination, creating two new linear destinations. But then we can't use the `extendLeaf` function teased above to fill these destinations with complete values, because `extendLeaf` only works on an ampar with just a single destination on the right-hand side, and `fillLeaf` or `fillComp` only work on unrestricted destinations.

There's a lot to decipher here. `FieldsTypes` is a type family that returns the types of the fields for a given constructor, specified in its type-lifted representation `lCtor` and the type `t` to which the constructor belongs. For example, `FieldsTypes '(:) [Int]` resolves to `(Int, [Int])`. `At k` is a type family that returns the k^{th} element (i.e. type) of a tuple type, and similarly, `Remove k` returns the same tuple type it is given, with the k^{th} removed.

The symbol `~` is for type equality constraint, and can be used as above as a way to bind type variables to complex type-level expressions in a function or type class signature. Above, we say that `fiTys` represents all field types of `lCtor`, `kthFiTy` is the k^{th} of these, and `othfiTys` is all the other field types in a tuple except the k^{th} one.

For instance, in `extend @'(:) @0 @[Int]`, `kthFiTy` is `Int`, and `othFiTys` is the one-element tuple type `Solo [Int]`[28]. Conversely, in `extend @'(:) @1 @[Int]`, `kthFiTy` is `[Int]`, and `othFiTys` is `Solo Int`.

In practice, `extend` takes an ampar whose right side is a destination of type `t`, with multiplicity `m`. It then allocates a new hollow constructor (specified by `lCtor`), consumes `othFiTys` at multiplicity `m` to fill all the fields of the new hollow constructor except the k^{th}, and then returns a destination with same multiplicity `m` for that remaining field of type `kthFiTy`.

Note that `extend` doesn't support constructors with 0 fields (it would make the signature of the function way to complex), as these can be plugged into the structure, like any other complete value, with `extendLeaf` anyway:

```
1 | extendLeaf :: ∀ s t m. t %m→ Ampar s (Dest m t) ⊸ Ampar s ()
```

Here we've made `extendLeaf` multiplicity-polymorphic, compared to the initial version presented above. But its behavior is the same.

Because we only have one destination on the right-hand side of any ampar, that is consumed and not replaced with a new one when calling `extendLeaf`, we could alternatively return the completed structure, instead of returning an `Ampar s ()`:

```
1 | fromAmparWithLeaf :: ∀ s t m. t %m→ Ampar s (Dest m t) ⊸ s
2 | fromAmparWithLeaf = fromAmpar' . extendLeaf
```

Note also that above, for `extend`, we assumed that the constructor in question only has linear fields[29], as the multiplicity `m` at which we consume the values for the fields, and at which we return the new destination, are the same as the multiplicity of the parent destination. We reflect that by the pseudo-constraint `OnlyLinear lCtor t`. Ideally, we could go further and be polymorphic over the multiplicity of each field, however, to this date, we cannot get that information through GHC.Generics (GHC.Generics is only implemented for constructors having just linear fields[30]). That also means we have to build a custom version of `extend` for the `Ur` constructor:

```
1 | extendUr :: ∀ s t m. Ampar s (Dest m (Ur t)) ⊸ Ampar s (Dest ω t)
```

The last function that we need, in this world is `extendComp`, to plug two ampars together:

[28] We could also decide to use the type `t` directly instead of `Solo t` for when there is only one element remaining, as we did for `UDestsOf` in Chapter 3.

[29] In λ_d, every data constructor has linear fields, except the exponential constructor $\mathtt{Mod}_{\mathtt{m}}$ which has a single field of mode `m`. In Haskell however, constructors have only linear fields by default, but can have unrestricted fields when the corresponding datatype is defined with GADT syntax and that an unrestricted arrow → is following the field's type. An unrestricted field is a way to hold unrestricted data without explicit need for `Ur`.

[30] So the constraint `OnlyLinear lCtor t` would only be needed in practice if GHC.Generics were to provide, in the future, an instance of `Generic t` for types `t` defined with GADT syntax.

```
extendComp :: ∀ s t u. Ampar s (Dest 1 t) ⊸ Ampar t u ⊸ Ampar s u
```

In `extendComp`, we need the destination in the parent ampar to be linear, as we fill it with an ampar, which is a linear resource[31].

4.3 Arbitrary type on the right-hand side of ampars for linear data

Being limited to only one destination on the right-hand side of ampars if we want to be able to use `extendLeaf` is not ideal. Indeed, we stay at the same expressivity level as existing work.

Going back to arbitrary types on the right hand-side of ampars, for instance to allow multiple destinations, as we did in Chapters 2 and 3, creates new challenges. Indeed, when we had only one destination for each ampar, we could operate on the destination through the ampar itself, with no choice to be made: there was only one destination that could be filled. If we have now an arbitrary type on the right-hand side of ampars, and we still want to operate on the destinations without giving direct access to them to the user (the motivations are still the same; work around the fact that `updWith` is now non-capturing, and also prevent scope escape/malicious interaction between sibling destinations), then we need a way for the user to select which destination inside the right-hand side of the ampar they want to operate on.

One idea for that is to ask the user for a linear lens. First, let's define what it is.

Lenses are a pattern for defining functional getters and setters that compose easily in an immutable functional programming language. A lens of type `Lens t1 t2 u1 u2` has two components (at least conceptually):

- a getter for a value of type `u1` somewhere inside the structure of type `t1`, which establishes the *focus* of the lens;
- a way to transform a structure of type `t1` into a structure of type `t2` by replacing the (focused) value of type `u1` by a value of type `u2`.

Although it isn't the most efficient and practical representation, non-linear lenses can be implemented by the following datatype:

```
data LensNL t1 t2 u1 u2 = LensNL
  { view :: t1 → u1  -- just get u1 out of t1, discard the rest
  , update :: t1 → u2 → t2  -- discard the u1 inside t1 and replace it with a u2 to make a t2
  }
```

For linear lenses, we are not allowed to use the `view` and `update` components separately; otherwise we could drop parts of the original structure of type `t1`. So instead, we group the two functions into one:

```
newtype Lens t1 t2 u1 u2 = Lens (t1 ⊸ (u1, u2 ⊸ t2))
```

[31] Even in a λ_d-like world, where ampars don't have to be linear, the destination used for composition using `◁⊸` / `fillComp` / `extendComp` must be linear. The reason is that if we write the child ampar into an unrestricted destination of the parent ampar, then we would need to update the remaining destinations of this child ampar to be unrestricted too, and this is hard to express when we can have arbitrarily complex types on the right-hand side of an ampar.

With a linear lens of type `Lens t1 t2 u1 u2`, we can split a structure of type `t1` into a pair of a value of type `u1` (the focus), and a closure of type `u2 ⊸ t2` that carries all the rest of the original `t1`, and is ready to produce a structure of type `t2` if we give it a `u2`. Both of these have to be consumed if we want to respect linearity.

The simplest lens is the standard getter and setter, where `t1 = t2` and `u1 = u2`, and thus, `Lens t t u u` becomes an alias for `t ⊸ (u, u ⊸ t)`. There, the second element of the pair, of type `u ⊸ t`, is really just a functional representation of a `t` missing a `u` to be complete! For example:

```
data Triple t1 t2 t3 = Triple { first :: t1, second :: t2, third :: t3 }

fstMono :: Lens (Triple t1 t2 t3) (Triple t1 t2 t3) t1 t1
fstMono (Triple { first, second, third }) = Lens (first, \newFirst -> Pair { newFirst, second, third })
```

As we see, when we are operating on a polymorphic type such as `Triple`, it is a bit silly to restrict our lens to the same type `t1` for both the original value at the focus, and the updated one. We gain much more flexibility by doing:

```
fst :: Lens (Triple t1 t2 t3) (Triple t4 t2 t3) t1 t4
fst (Triple { first, second, third }) = Lens (first, \newFirst -> Pair { newFirst, second, third })
```

Lenses to select destinations to operate on With linear lenses, the user can communicate which destination they want to focus, and we gain a way to inject back the potential newly produced destinations at the right place on the right-hand side of the ampar:

```
extendFocused :: ∀ lCtor u t1 t2 s m.
                 (OnlyLinear lCtor u) ⇒
                 Lens t1 t2 (Dest m u) (DestsOf lCtor m u) → Ampar s t1 ⊸ Ampar s t2

extendWithPairOnFirst :: Ampar s (Triple (Dest m (t11, t12)) t2 t3) ⊸ Ampar s (Triple (Dest m t11, Dest m t12) t2
extendWithPairOnFirst ampar = ampar &(fst &extendFocused @'(,))
```

As stated above, this technique gets even more useful for `extendFocusedLeaf`, as we had no way previously to fill a linear destination contained in an arbitrary structure inside the right-hand side of an ampar. With `extendFocusedLeaf`, it is now possible:

```
extendFocusedLeaf :: ∀ u t1 t2 s m. u %m→ Lens t1 t2 (Dest m u) () ⊸ Ampar s t1 ⊸ Ampar s t2

extendWithValOnFirst :: t1 ⊸ Ampar s (Triple (Dest 1 t1) t2 t3) ⊸ Ampar s (Triple () t2 t3)
extendWithPairOnFirst x ampar = ampar &(fst &extendFocusedLeaf x)
```

4.4 Limitations of this system

The situation is still not perfectly satisfying, but we don't think that a single perfect balance between expressiveness and simplicity exists for DPS programming in Haskell, given the constraint of using an existing type system.

Here, we can have at most two usable layers of nesting for destinations. But that's enough for most use-cases, and in particular, it let us implement the breadth-first traversal of binary trees using efficient queues!

4.5 Full API and breadth-first tree traversal, updated

The implementation of difference lists here is very different from the one from Section 3.3.1, because we are no longer using the full flexibility of destinations and `updWith`, and instead, use primitives very similar in behavior to the ones from Minamide [Min98] and Lorenzen et al. [Lor+24b].

On the other hand, the implementation of efficient queues displayed here is very similar to the one described in Section 3.4, except that we are using `Ampar` instead of `UAmpar` so we can store linear elements inside the queue (but this almost doesn't affect the implementation).

```
data Token
dup :: Token ⊸ (Token, Token)
drop :: Token ⊸ ()
withToken :: ∀ t. (Token ⊸ Ur t) ⊸ Ur t

type Region r :: Constraint
inRegion :: ∀ t. (∀ r. Region r ⇒ t) ⊸ t

data Ampar r s t
newAmpar :: ∀ r s. Region r ⇒ Token ⊸ Ampar s (Dest r 1 s)
tokenBesides :: ∀ r s t. Region r ⇒ Ampar r s t ⊸ (Ampar r s t, Token)
toAmpar :: ∀ r s. Region r ⇒ Token ⊸ s ⊸ Ampar r s () -- now linear in s
fromAmpar  :: ∀ r s t. Region r ⇒ Ampar r s (Ur t) ⊸ (s, Ur t)
fromAmpar' :: ∀ r s. Region r ⇒ Ampar r s () ⊸ s
updWith :: ∀ r s t u. Region r ⇒ Ampar r s t ⊸ (t ⊸ u) → Ampar r s u -- non-capturing now
updWithToken :: ∀ r s t u. Region r ⇒ Ampar r s t ⊸ ((t, Token) ⊸ u) → Ampar r s u

data Dest r m t
type family DestsOf lCtor r m t -- now multiplicity-polymorphic

-- === API for unrestricted destinations only ===
fillLeaf :: ∀ r t. Region r ⇒ t → Dest r ω t ⊸ ()
-- we can no longer have fillComp for unrestricted destinations

-- === API for ampars with linear or unrestricted destinations ===
fill :: ∀ lCtor r m t. (Region r, OnlyLinear lCtor t) ⇒ Dest r m t ⊸ DestsOf lCtor r m t
fillUr :: ∀ r m t. Region r ⇒ Dest r m (Ur t) ⊸ Dest r ω t

-- only one destination on the right (simplified but less expressive)
extend :: ∀ lCtor r k t s m fiTy kthFiTy othFiTys.
          (Region r, OnlyLinear lCtor t,
          FieldTypes lCtor t ~ fiTys,
          At k fiTys ~ kthFiTy, Remove k fiTys ~ othFiTys) ⇒
          othFiTys %m→ Ampar r s (Dest r m t) ⊸ Ampar r s (Dest r m kthFiTy)
extendUr :: ∀ r s t m. Region r ⇒ Ampar r s (Dest r m (Ur t)) ⊸ Ampar r s (Dest r ω t)
extendLeaf :: ∀ r s t m. Region r ⇒ t %m→ Ampar r s (Dest r m t) ⊸ Ampar r s ()
extendComp :: ∀ r s t u. Region r ⇒ Ampar r s (Dest r 1 t) ⊸ Ampar r t u ⊸ Ampar r s u

-- with arbitrary type on the right; destination focused with a linear lens
extendFocused :: ∀ lCtor r u t1 t2 s m.
                 (Region r, OnlyLinear lCtor u) ⇒
                 Lens t1 t2 (Dest r m u) (DestsOf lCtor r m u) ⊸ Ampar r s t1 ⊸ Ampar r s t2
extendFocusedUr :: ∀ r u t1 t2 s m. (Region r) ⇒
                   Lens t1 t2 (Dest r m (Ur u)) (Dest r ω u) ⊸ Ampar r s t1 ⊸ Ampar r s t2
extendFocusedLeaf :: ∀ r u t1 t2 s m. (Region r) ⇒
                     u %m→ Lens t1 t2 (Dest r m u) () ⊸ Ampar r s t1 ⊸ Ampar r s t2
extendFocusedComp :: ∀ r u t1 t2 s u1 u2. (Region r) ⇒
                     Ampar r u1 u2 ⊸ Lens t1 t2 (Dest r 1 u1) u2 ⊸ Ampar r s t1 ⊸ Ampar r s t2
```

Listing 11: Destination-passing API allowing linear destinations

```
type DList r t = Ampar r [t] (Dest r 1 [t])

newDList :: Region r ⇒ Token ⊸ DList r t
newDList = newAmpar @[t]

dListToList :: Region r ⇒ DList r t ⊸ [t]
dListToList dlist = fromAmpar' (dlist &extendLeaf [])

append :: Region r ⇒ DList r t ⊸ t ⊸ DList r t
append dlist x = dlist &extend @'(:) @1 (MkSolo x)

-------------------------------------------------------------------------

data EffQueue r t = EffQueue [t] (DList r t)

newEffQueue :: Region r ⇒ Token ⊸ EffQueue r t
newEffQueue tok = EffQueue [] (newDList tok)

singleton :: Region r ⇒ Token ⊸ t ⊸ EffQueue r t
singleton tok x = EffQueue [x] (newDList tok)

queueToList :: Region r ⇒ EffQueue r t ⊸ [t]
queueToList (EffQueue front back) = front ++ dListToList back

enqueue :: Region r ⇒ EffQueue r t ⊸ t ⊸ EffQueue r t
enqueue (EffQueue front back) x = EffQueue front (back `append` x)

dequeue :: Region r ⇒ EffQueue r t ⊸ Maybe (t, EffQueue r t)
dequeue (EffQueue front back) = case front of
  [] -> case tokenBesides back of (back, tok) -> case (toList back) of
    [] -> drop tok ⨟ Nothing
    (x : xs) -> Just (x, (EffQueue xs (newDList tok)))
  (x : xs) -> Just (x, (EffQueue xs back))

-------------------------------------------------------------------------

data Tree t = Nil | Node t (Tree t) (Tree t)

relabelDPS :: Region r ⇒ Token ⊸ Tree t → Tree Int
relabelDPS tree = fst (mapAccumBFS (\st _ → (st + 1, st)) 1 tree)

mapAccumBFS :: ∀ r s t u. Region r ⇒ Token ⊸ (s → t → (s, u)) → s → Tree t → (Tree u, s)
mapAccumBFS tok f s0 tree =
  case fromAmpar (newAmpar @(Ur (Tree u)) tok `updWith` \du → go s0 (singleton (Ur tree, du &fillUr)))
  of (Ur outTree, Ur st) → (outTree, st)
  where
    go :: s → EffQueue r (Ur (Tree t), Dest r ω (Tree u)) ⊸ Ur s
    go st q = case dequeue q of
      Nothing → Ur st
      Just ((utree, dtree), q') → case utree of
        Ur Nil → dtree &fill @'Nil ⨟ go st q'
        Ur (Node x tl tr) → case (dtree &fill @'Node) of
          (dy, dtl, dtr) →
            let q'' = q' `enqueue` (Ur tl, dtl) `enqueue` (Ur tr, dtr)
                (st', y) = f st x
```

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 Programming with Permissions in Mezzo

Protzenko and Pottier [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].

5.3 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.

5.4 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, $(\mathtt{T},\mathtt{S})\mathtt{hfun}$ is thus a weak form of linear function type $\mathtt{T} \multimap \mathtt{S}$.

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 $\mathtt{T} \multimap \mathtt{S}$ is reinterpreted as $\mathtt{S} \parr \mathtt{T}^{\bot}$. We, likewise, reinterpret $(\mathtt{T},\mathtt{S})\mathtt{hfun}$ as $\mathtt{S} \ltimes \lfloor \mathtt{T} \rfloor$ (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) `(T, S)hfun`; so in λ_d, we can't act directly on the left-hand side `S` of $\mathtt{S} \ltimes \mathtt{T}$, 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.

5.5 Destination-passing style programming: a Haskell implementation

Bagrel [Bag24a] 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 [Bag24a], 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 [Bag24a] itself), so goals are quite different.

5.6 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.

5.7 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.

5.8 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

[How69] W H Howard. *The formulae-as-types notion of construction*. en. Transcribed in 2017 to LATEX by Armando B. Matos. 1969. URL: https://www.dcc.fc.up.pt/~acm/howard2.pdf (visited on 05/05/2025).

[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.

[Gir87] Jean-Yves Girard. "Linear logic." In: *Theoretical Computer Science* 50.1 (1987), pp. 1–101. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(87)90045-4. URL: https://www.sciencedirect.com/science/article/pii/0304397587900454.

[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.

[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).

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091.

[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).

[Har06] Dana Harrington. "Uniqueness logic." en. In: *Theoretical Computer Science* 354.1 (Mar. 2006), pp. 24–41. ISSN: 03043975. DOI: 10.1016/j.tcs.2005.11.006. URL: https://linkinghub.elsevier.com/retrieve/pii/S0304397505008522 (visited on 03/24/2025).

[SU06] Morten Heine B. Sørensen and Paweł Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. en-US. 1st. Vol. 149. Studies in Logic and the Foundations of Mathematics. Elsevier Science, July 2006. ISBN: 978-0-444-52077-7. URL: https://www.cs.cmu.edu/~rwh/courses/clogic/www/handouts/curry-howard.pdf (visited on 05/05/2025).

[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).

[MVO22] Danielle Marshall, Michael Vollmer, and Dominic Orchard. "Linearity and Uniqueness: An Entente Cordiale." en. In: *Programming Languages and Systems*. Ed. by Ilya Sergey. Vol. 13240. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 346–375. ISBN: 978-3-030-99335-1 978-3-030-99336-8. DOI: 10.1007/978-3-030-99336-8_13. URL: https://link.springer.com/10.1007/978-3-030-99336-8_13 (visited on 03/24/2025).

[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).

[Spi23a] Arnaud Spiwack. *Linear constraints proposal by aspiwack · Pull Request #621 · ghc-proposals/ghc-proposals*. en. Nov. 2023. URL: https://github.com/ghc-proposals/ghc-proposals/pull/621 (visited on 03/25/2025).

[Spi23b] Arnaud Spiwack. *Linear Constraints: the problem with scopes*. en. Blog. Mar. 2023. URL: https://tweag.io/blog/2023-03-23-linear-constraints-linearly/ (visited on 03/25/2025).

[Bag24a] 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.

[Bag24b] Thomas Bagrel. *Primitives for zero-copy compact regions by tbagrel1 · Pull Request #683 · ghc-proposals/ghc-proposals*. en. Nov. 2024. URL: https://github.com/ghc-proposals/ghc-proposals/pull/683 (visited on 04/01/2025).

[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.

[Spi24] Arnaud Spiwack. *Ur can't be a newtype*. en. Blog. Jan. 2024. URL: https://www.tweag.io/blog/2024-01-18-linear-desugaring/#ur-cant-be-a-newtype (visited on 04/01/2025).

[BS25] Thomas Bagrel and Arnaud Spiwack. "Destination Calculus: A Linear ⊠-Calculus for Purely Functional Memory Writes." In: *Proc. ACM Program. Lang.* 9.OOPSLA1 (Apr. 2025). DOI: 10.1145/3720423. URL: https://doi.org/10.1145/3720423.

[LL] Daan Leijen and Anton Lorenzen. "Tail Recursion Modulo Context: An Equational Approach (extended version)." In: *under submission to JFP'25* (). URL: https://antonlorenzen.de/trmc-jfp.pdf (visited on 05/13/2025).