

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

<i>Président :</i>	Le président
<i>Rapporteurs :</i>	Le rapporteur 1 de Paris
	Le rapporteur 2
	Le rapporteur 3
<i>Examineurs :</i>	L'examineur 1 d'ici
	L'examineur 2

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.

Table of Contents

Frontmatter	i
Abstract	i
Acknowledgments	iii
Table of Contents	ix
List of Figures	xi
List of Listings	xiii
Introduction	1
1 Memory management: a core design axis for programming languages	1
2 Destination passing style: taking roots in the old imperative world	2
3 Functional programming languages	3
4 Functional structures with holes: pioneered by Minamide and quite active since .	4
5 Unifying and formalizing Functional DPS frameworks	6
6 Reconciling write pointers and immutability	6
Chapter 1 – Linear λ-calculus and Linear Haskell	7
1.1 From λ -calculus to linear λ -calculus	7
1.2 Linear λ -calculus: a computational interpretation of linear logic	9
1.3 Implicit structural rules for unrestricted resources	11
1.4 Back to a single context with the graded modal approach	13
1.5 Deep modes: projecting modes through fields of data structures	17
1.6 Semantics of linear λ -calculus (with deep modes)	19
1.7 Linear Types in Haskell	21
1.8 Uniqueness and Linear Scopes	22
1.9 Conclusion	26
Chapter 2 – A formal functional language based on first-class destinations: λ_d	27
2.1 Destination-passing style programming, in a functional setting	27
2.2 Working with destinations in λ_d	28
2.2.1 Building up a vocabulary	29

2.2.2	Tail-recursive map	32
2.2.3	Functional queues, with destinations	33
2.3	Scope escape of destinations	34
2.4	Breadth-first tree traversal	36
2.5	Type system	37
2.5.1	Modes and the age ringoid	39
2.5.2	Typing rules	40
2.6	Operational semantics	44
2.6.1	Runtime values and new typing context forms	45
2.6.2	Evaluation contexts and commands	48
2.6.3	Reduction semantics	50
2.6.4	Type safety	54
2.7	Formal proof of type safety	55
2.8	Translation of prior work into λ_d terms and types	56
2.9	Implementation of λ_d using in-place memory mutations	57
2.10	Conclusion	59

Chapter 3 – A first implementation of first-class destination passing in Haskell 61

3.1	Linear Haskell is our implementation target	61
3.2	Restricting λ_d so that it can be made safe with just linear types	62
3.3	A glimpse of DPS Haskell programming	62
3.3.1	Efficient difference lists, and proper memory model	62
3.3.2	Breadth-first tree traversal	66
3.4	DPS Haskell API and Design concerns	68
3.4.1	The <code>UAmper</code> type	68
3.4.2	Filling functions for destinations	70
3.5	Compact Regions: a playground to implement the DPS Haskell API	71
3.5.1	Compact Regions	71
3.5.2	DPS programming in Compact regions: Deserializing, lifetime, and garbage collection	72
3.5.3	Memory representation of <code>UAmper</code> objects in Compact Regions	75
3.5.4	Deriving <code>fill</code> for all constructors with <code>Generics</code>	77
3.5.5	Changes to GHC internals and its RTS	78
3.6	Evaluating the performance of DPS programming	82
3.6.1	Concatenating lists and difference lists	83
3.6.2	Breadth-first relabeling	84
3.6.3	Parsing S-expressions	84
3.6.4	Mapping a function over a list	85
3.7	Conclusion	86

Chapter 4 – Extending DPS support in Haskell for linear data structures	87
4.1 Challenges of destinations for linear data – scope escape, updated	87
4.2 <code>/extend/</code> functions for ampars with a single hole	89
4.3 Arbitrary type on the right-hand side of ampars storing linear data	92
4.4 Breadth-first tree traversal, updated, and final API	95
4.5 Conclusion	95
Related Work	101
1 Destination-passing style for efficient memory management	101
2 Programming with Permissions in Mezzo	101
3 Tail modulo constructor	101
4 A functional representation of data structures with a hole	102
5 Semi-axiomatic sequent calculus	103
6 Rust lifetimes	103
7 Oxidizing OCaml	103
Conclusion and future work	105
Bibliography	109

List of Figures

1.1	Intuitionistic linear logic in natural deduction presentation, sequent-style (ILL)	9
1.2	Grammar of linear λ -calculus in monadic presentation (λ_{L1})	9
1.3	Typing rules for linear λ -calculus in monadic presentation (λ_{L1} -TY)	10
1.4	Grammar of linear λ -calculus in dyadic presentation (λ_{L2})	11
1.5	Typing rules for linear λ -calculus in dyadic presentation (λ_{L2} -TY)	12
1.6	Grammar of linear λ -calculus in modal presentation (λ_{Lm})	15
1.7	Typing rules for linear λ -calculus in modal presentation (λ_{Lm} -TY)	15
1.8	Altered typing rules for deep modes in linear λ -calculus in modal presentation (λ_{Ldm} -TY)	18
1.9	Small-step semantics for λ_{Ldm} (λ_{Ldm} -sem)	20
2.1	Terms, types and modes of λ_d	38
2.2	Typing rules of λ_d (λ_d -TY)	41
2.3	Derived typing rules for syntactic sugar (λ_d -TY _S)	42
2.4	Evolution of ages through nested scopes	43
2.5	Runtime values, new typing context forms, and operators	45
2.6	Typing rules for runtime values (λ_d -TY _V)	46
2.7	Grammar of evaluation contexts	48
2.8	Typing rules of evaluation contexts (λ_d -TY _E) and commands (λ_d -TY _{CMD})	49
2.9	Small-step reduction of commands for λ_d (λ_d -SEM, part 1)	51
2.10	Small-step reduction of commands for λ_d (λ_d -SEM, part 2)	52
3.1	Mappings between λ_d and DPS Haskell	63
3.2	/newUAmper tok/	64
3.3	Memory behavior of /toList dlist/	64
3.4	Memory behavior of /append newUAmper 1/	65
3.5	Memory behavior of /concat dlist1 dlist2/ (based on /fillComp/)	65
3.6	Memory behavior of /fill @'(:) :: UDest [t] \multimap (UDest t, UDest [t])/	71
3.7	Memory behavior of /fill @'[] :: UDest [t] \multimap ()/	71
3.8	Memory behavior of /fillLeaf :: t \rightarrow UDest [t] \multimap ()/	71

3.9	Representation of UAmpars in the compact region implementation, using <i>/Ur/</i> (not chosen)	76
3.10	Chosen representation for UAmpars in the compact region implementation, using indirections	77
3.11	Benchmarks performed on AMD EPYC 7401P @ 2.0 GHz (single core, -N1 -02) .	83

List of Listings

2.1	Tail-recursive <code>map</code> function on lists	33
2.2	Difference list and queue implementation in equirecursive λ_d	34
2.3	Breadth-first tree traversal in destination-passing style	37
2.4	Syntactic sugar definitions for terms	38
3.1	Implementation of difference lists in DPS Haskell	64
3.2	Implementation of Hood-Melville queue in Haskell	67
3.3	Implementation of breadth-first tree traversal in DPS Haskell	67
3.4	Destination API for Haskell	68
3.5	Destination API using compact regions	73
3.6	Implementation of the S-expression parser without destinations	74
3.7	Implementation of the S-expression parser with destinations	74
3.8	<code>reifyInfoPtr#</code> implementation in <code>compiler/GHC/StgToCmm/Prim.hs</code>	80
3.9	<code>compactAddHollow#</code> implementation in <code>rts/Compact.cmm</code>	81
3.10	<code>/LCtorToSymbol/</code> implementation in <code>compiler/GHC/Builtin/Types/Literal.hs</code>	82
4.1	Efficient queue for linear data	96
4.2	Breadth-first tree traversal revisited, with efficient queue for storing destinations	98
4.3	Destination-passing API for Haskell with support for linear destinations	99

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 that 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 capture parts of their parent environment (closures), can be stored, and can be 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 [Lorenzen, White, et al., 2024]. 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.

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

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 [1998] 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: $(\lambda x \mapsto 1 :: x) \circ (\lambda y \mapsto 2 :: y) \rightsquigarrow \lambda y \mapsto 1 :: 2 :: y$. 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 [Leijen and Lorenzen, 2023] and [Lorenzen, Leijen, Swierstra, and Lindley, 2024], 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. [Bour, Clément, and Scherer, 2021] 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 [Minamide, 1998], 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 [Bour, Clément, and Scherer, 2021] or [Leijen and Lorenzen, 2023], 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 [Bour, Clément, and Scherer, 2021; Leijen and Lorenzen, 2023]) 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 [Pierce, 2002] and [Sørensen and Urzyczyn, 2006] 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 [Howard, 1969] 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 [Jean-Yves Girard, 1987], 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 calculus¹, hypotheses are duplicated or discarded explicitly using *structural* rules of the system, named respectively *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 $\mathbf{T} \rightarrow \mathbf{U}$ is not part of linear logic, but is replaced by linear implication $\mathbf{T} \multimap \mathbf{U}$, where \mathbf{T} must be used exactly once to prove \mathbf{U} . Linear logic also introduces a modality $!$, pronounced *of course* or *bang*, to allow weakening and contraction on specific formulae: $!\mathbf{T}$ denotes that \mathbf{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. $\mathbf{1}$ 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 \mathbf{T} can become an unrestricted formula $!\mathbf{T}$ 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 $!\mathbf{T}$ can be used in a place expecting a linear formula \mathbf{T} . The contraction rule ILL/!C and weakening rule ILL/!W state respectively that an unrestricted formula $!\mathbf{T}$ 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 [J.-Y. Girard, 1995], whose main idea is to prefix most formulae with $!$ and encode the non-linear implication $\mathbf{T} \rightarrow \mathbf{U}$ as $!\mathbf{T} \multimap \mathbf{U}$.

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

$$\begin{array}{c}
 \frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID} \quad \frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID} \\
 \frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID} \quad \frac{\frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID} \quad \frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID}}{!\mathbf{T} \vdash \mathbf{T}} \text{!D} \quad \frac{\frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID} \quad \frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID}}{!\mathbf{T} \vdash \mathbf{T}} \text{!D} \\
 \frac{\frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID} \quad \frac{\frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID} \quad \frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID}}{!\mathbf{T} \vdash \mathbf{T}} \text{!D} \quad \frac{\frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID} \quad \frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID}}{!\mathbf{T} \vdash \mathbf{T}} \text{!D}}{!\mathbf{T}, !\mathbf{T} \vdash \mathbf{T} \otimes \mathbf{T}} \otimes \text{I} \\
 \frac{\frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID} \quad \frac{\frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID} \quad \frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID}}{!\mathbf{T} \vdash \mathbf{T}} \text{!D} \quad \frac{\frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID} \quad \frac{}{!\mathbf{T} \vdash !\mathbf{T}} \text{ID}}{!\mathbf{T} \vdash \mathbf{T}} \text{!D}}{!\mathbf{T} \vdash \mathbf{T} \otimes \mathbf{T}} \text{!C} \\
 \frac{}{\bullet \vdash !\mathbf{T} \multimap (\mathbf{T} \otimes \mathbf{T})} \multimap \text{I}
 \end{array}$$

¹Sequent 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

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

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

$$\begin{array}{c}
\frac{}{T \vdash T} \text{ID} \quad \frac{\Gamma, T \vdash U}{\Gamma \vdash T \multimap U} \multimap\text{I} \quad \frac{}{\bullet \vdash 1} 1\text{I} \quad \frac{\Gamma \vdash T_1}{\Gamma \vdash T_1 \oplus T_2} \oplus\text{I}_1 \quad \frac{\Gamma \vdash T_2}{\Gamma \vdash T_1 \oplus T_2} \oplus\text{I}_2 \\
\\
\frac{\Gamma_1 \vdash T_1 \quad \Gamma_2 \vdash T_2}{\Gamma_1, \Gamma_2 \vdash T_1 \otimes T_2} \otimes\text{I} \quad \frac{\Gamma_1 \vdash T \quad \Gamma_2 \vdash T \multimap U}{\Gamma_1, \Gamma_2 \vdash U} \multimap\text{E} \quad \frac{\Gamma_1 \vdash 1 \quad \Gamma_2 \vdash U}{\Gamma_1, \Gamma_2 \vdash U} 1\text{E} \\
\\
\frac{\Gamma_1 \vdash T_1 \oplus T_2 \quad \Gamma_2, T_1 \vdash U \quad \Gamma_2, T_2 \vdash U}{\Gamma_1, \Gamma_2 \vdash U} \oplus\text{E} \quad \frac{\Gamma_1 \vdash T_1 \otimes T_2 \quad \Gamma_2, T_1, T_2 \vdash U}{\Gamma_1, \Gamma_2 \vdash U} \otimes\text{E} \quad \frac{! \Gamma \vdash T}{! \Gamma \vdash !T} !\text{P} \quad \frac{\Gamma \vdash !T}{\Gamma \vdash T} !\text{D} \quad \frac{\Gamma_1 \vdash !T \quad \Gamma_2, !T, !T \vdash U}{\Gamma_1, \Gamma_2 \vdash U} !\text{C} \\
\\
\frac{\Gamma_1 \vdash !T \quad \Gamma_2 \vdash U}{\Gamma_1, \Gamma_2 \vdash U} !\text{W}
\end{array}$$

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

$$\begin{array}{l}
v ::= \lambda x \mapsto u \mid () \mid \text{Inl } v \mid \text{Inr } v \mid (v_1, v_2) \mid \text{Many } v \\
t, u ::= v \mid x \mid \text{Inl } t \mid \text{Inr } t \mid (t_1, t_2) \mid \text{Many } t \mid t \, t' \mid t \, ; \, t' \\
\quad \mid \text{case } t \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\} \mid \text{case } t \text{ of } (x_1, x_2) \mapsto u \\
\quad \mid \text{dup } t \text{ as } x_1, x_2 \text{ in } u \mid \text{drop } t \text{ in } u \mid \text{derelict } t \\
\\
T, U ::= T \multimap U \mid 1 \mid T_1 \oplus T_2 \mid T_1 \otimes T_2 \mid !T \\
\\
\Gamma ::= \bullet \mid x : T \mid \Gamma_1, \Gamma_2
\end{array}$$

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

We use dereliction and identity rules in both branches of the pair, to transform the unrestricted T into a regular one. That way, the product $T \otimes T$ requires the context $!T, !T$ 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.

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 [Andreoli, 1992], 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 [Bierman, 1994], are given in Figures 1.2 and 1.3.

$\boxed{\Gamma \vdash t : \mathbb{T}}$

(Typing judgment for terms)

$$\begin{array}{c}
 \frac{}{x : \mathbb{T} \vdash x : \mathbb{T}} \text{ID} \quad \frac{\Gamma, x : \mathbb{T} \vdash u : \mathbb{U}}{\Gamma \vdash \lambda x \mapsto u : \mathbb{T} \multimap \mathbb{U}} \multimap\text{I} \quad \frac{}{\bullet \vdash () : \mathbf{1}} \mathbf{1}\text{I} \quad \frac{\Gamma \vdash t_1 : \mathbb{T}_1}{\Gamma \vdash \text{Inl } t_1 : \mathbb{T}_1 \oplus \mathbb{T}_2} \oplus\text{I}_1 \\
 \\
 \frac{\Gamma \vdash t_2 : \mathbb{T}_2}{\Gamma \vdash \text{Inr } t_2 : \mathbb{T}_1 \oplus \mathbb{T}_2} \oplus\text{I}_2 \quad \frac{\Gamma_1 \vdash t_1 : \mathbb{T}_1 \quad \Gamma_2 \vdash t_2 : \mathbb{T}_2}{\Gamma_1, \Gamma_2 \vdash (t_1, t_2) : \mathbb{T}_1 \otimes \mathbb{T}_2} \otimes\text{I} \quad \frac{\Gamma_1 \vdash t : \mathbb{T} \quad \Gamma_2 \vdash t' : \mathbb{T} \multimap \mathbb{U}}{\Gamma_1, \Gamma_2 \vdash t' t : \mathbb{U}} \multimap\text{E} \\
 \\
 \frac{\Gamma_1 \vdash t : \mathbf{1} \quad \Gamma_2 \vdash u : \mathbb{U}}{\Gamma_1, \Gamma_2 \vdash t \mathbin{\circ} u : \mathbb{U}} \mathbf{1}\text{E} \quad \frac{\Gamma_1 \vdash t : \mathbb{T}_1 \oplus \mathbb{T}_2 \quad \Gamma_2, x_1 : \mathbb{T}_1 \vdash u_1 : \mathbb{U} \quad \Gamma_2, x_2 : \mathbb{T}_2 \vdash u_2 : \mathbb{U}}{\Gamma_1, \Gamma_2 \vdash \text{case } t \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\} : \mathbb{U}} \oplus\text{E} \\
 \\
 \frac{\Gamma_1 \vdash t : \mathbb{T}_1 \otimes \mathbb{T}_2 \quad \Gamma_2, x_1 : \mathbb{T}_1, x_2 : \mathbb{T}_2 \vdash u : \mathbb{U}}{\Gamma_1, \Gamma_2 \vdash \text{case } t \text{ of } (x_1, x_2) \mapsto u : \mathbb{U}} \otimes\text{E} \quad \frac{! \Gamma \vdash t : \mathbb{T}}{! \Gamma \vdash \text{Many } t : ! \mathbb{T}} !\text{P} \quad \frac{\Gamma \vdash t : ! \mathbb{T}}{\Gamma \vdash \text{derelict } t : \mathbb{T}} !\text{D} \\
 \\
 \frac{\Gamma_1 \vdash t : ! \mathbb{T} \quad \Gamma_2, x_1 : ! \mathbb{T}, x_2 : ! \mathbb{T} \vdash u : \mathbb{U}}{\Gamma_1, \Gamma_2 \vdash \text{dup } t \text{ as } x_1, x_2 \text{ in } u : \mathbb{U}} !\text{C} \quad \frac{\Gamma_1 \vdash t : ! \mathbb{T} \quad \Gamma_2 \vdash u : \mathbb{U}}{\Gamma_1, \Gamma_2 \vdash \text{drop } t \text{ in } u : \mathbb{U}} !\text{W}
 \end{array}$$

 Figure 1.3: Typing rules for linear λ -calculus in monadic presentation ($\lambda_{L1}\text{-TY}$)

In λ_{L1} , Γ is now a finite map from variables to types that can be represented as a set of variable bindings $x : \mathbb{T}$. 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 $\mathbf{1}$ is made with the $\mathbin{\circ}$ 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 $! \mathbb{T}$. In addition, the language includes a data constructor for unrestricted terms and values, denoted by **Many** t and **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 **let** $x := t$ **in** u and encode it as $(\lambda x \mapsto u) (t)$.

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

$$\begin{array}{c}
 \frac{}{x_1 : ! \mathbb{T} \vdash x_1 : ! \mathbb{T}} \text{ID} \quad \frac{}{x_2 : ! \mathbb{T} \vdash x_2 : ! \mathbb{T}} \text{ID} \\
 \frac{}{x : ! \mathbb{T} \vdash x : ! \mathbb{T}} \text{ID} \quad \frac{x_1 : ! \mathbb{T} \vdash \text{derelict } x_1 : \mathbb{T} \quad x_2 : ! \mathbb{T} \vdash \text{derelict } x_2 : \mathbb{T}}{x_1 : ! \mathbb{T}, x_2 : ! \mathbb{T} \vdash (\text{derelict } x_1, \text{derelict } x_2) : \mathbb{T} \otimes \mathbb{T}} \otimes\text{I} \\
 \frac{}{x : ! \mathbb{T} \vdash x : ! \mathbb{T}} \text{ID} \quad \frac{x_1 : ! \mathbb{T}, x_2 : ! \mathbb{T} \vdash (\text{derelict } x_1, \text{derelict } x_2) : \mathbb{T} \otimes \mathbb{T}}{x : ! \mathbb{T} \vdash \text{dup } x \text{ as } x_1, x_2 \text{ in } (\text{derelict } x_1, \text{derelict } x_2) : \mathbb{T} \otimes \mathbb{T}} !\text{C} \\
 \frac{}{\bullet \vdash \lambda x \mapsto \text{dup } x \text{ as } x_1, x_2 \text{ in } (\text{derelict } x_1, \text{derelict } x_2) : ! \mathbb{T} \multimap (\mathbb{T} \otimes \mathbb{T})} \multimap\text{I}
 \end{array}$$

1.3. Implicit structural rules for unrestricted resources

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

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

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.

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 U that holds variable bindings that can be used in an unrestricted fashion. The **case** t of **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 U . It's important to note that **case** t of **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 **Many** x ; while that wouldn't be possible with **let** $x := \text{derelict } t \text{ in } u$ of λ_{L1} . Morally, we can view the pair of contexts $\Gamma; \mathsf{U}$ of λ_{L2} as a single context $\Gamma, !\mathsf{U}$ of λ_{L1} , where $!\mathsf{U}$ is the context with the same variable bindings as U , except that all types are prefixed by $!$.

In λ_{L2} , contraction for unrestricted resources happens implicitly every time a rule has two sub-terms as premises. Indeed, the unrestricted context U 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 $\lambda_{L2}\text{-TY}/\text{ID}_{\text{LIN}}$, $\lambda_{L2}\text{-TY}/\text{ID}_{\text{UR}}$, and $\lambda_{L2}\text{-TY}/\mathbf{1I}$. These three rules discard all bindings of the unrestricted context U that aren't used, performing several implicit weakening steps.

Finally, this system has two identity rules. The first one, $\lambda_{L2}\text{-TY}/\text{ID}_{\text{LIN}}$, 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

$\boxed{\Gamma; \mathcal{U} \vdash t : \mathbb{T}}$

(Typing judgment for terms)

$$\begin{array}{c}
 \frac{}{x : \mathbb{T}; \mathcal{U} \vdash x : \mathbb{T}} \text{ID}_{\text{LIN}} \quad \frac{}{\bullet; \mathcal{U}, x : \mathbb{T} \vdash x : \mathbb{T}} \text{ID}_{\text{UR}} \quad \frac{\Gamma, x : \mathbb{T}; \mathcal{U} \vdash u : \mathbb{U}}{\Gamma; \mathcal{U} \vdash \lambda x \mapsto u : \mathbb{T} \multimap \mathbb{U}} \multimap \text{I} \\
 \\
 \frac{}{\bullet; \mathcal{U} \vdash () : \mathbb{1}} \mathbb{1}\text{I} \quad \frac{\Gamma; \mathcal{U} \vdash t_1 : \mathbb{T}_1}{\Gamma; \mathcal{U} \vdash \text{Inl } t_1 : \mathbb{T}_1 \oplus \mathbb{T}_2} \oplus \text{I}_1 \quad \frac{\Gamma; \mathcal{U} \vdash t_2 : \mathbb{T}_2}{\Gamma; \mathcal{U} \vdash \text{Inr } t_2 : \mathbb{T}_1 \oplus \mathbb{T}_2} \oplus \text{I}_2 \\
 \\
 \frac{\Gamma_1; \mathcal{U} \vdash t_1 : \mathbb{T}_1 \quad \Gamma_2; \mathcal{U} \vdash t_2 : \mathbb{T}_2}{\Gamma_1, \Gamma_2; \mathcal{U} \vdash (t_1, t_2) : \mathbb{T}_1 \otimes \mathbb{T}_2} \otimes \text{I} \quad \frac{\bullet; \mathcal{U} \vdash t : \mathbb{T}}{\bullet; \mathcal{U} \vdash \text{Many } t : \mathbb{!T}} \mathbb{!}\text{I} \quad \frac{\Gamma_1; \mathcal{U} \vdash t : \mathbb{T} \quad \Gamma_2; \mathcal{U} \vdash t' : \mathbb{T} \multimap \mathbb{U}}{\Gamma_1, \Gamma_2; \mathcal{U} \vdash t' t : \mathbb{U}} \multimap \text{E} \\
 \\
 \frac{\Gamma_1; \mathcal{U} \vdash t : \mathbb{1} \quad \Gamma_2; \mathcal{U} \vdash u : \mathbb{U}}{\Gamma_1, \Gamma_2; \mathcal{U} \vdash t \mathbin{\text{\tiny ;}} u : \mathbb{U}} \mathbb{1}\text{E} \quad \frac{\Gamma_1; \mathcal{U} \vdash t : \mathbb{T}_1 \oplus \mathbb{T}_2 \quad \Gamma_2, x_1 : \mathbb{T}_1; \mathcal{U} \vdash u_1 : \mathbb{U} \quad \Gamma_2, x_2 : \mathbb{T}_2; \mathcal{U} \vdash u_2 : \mathbb{U}}{\Gamma_1, \Gamma_2; \mathcal{U} \vdash \text{case } t \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\} : \mathbb{U}} \oplus \text{E} \\
 \\
 \frac{\Gamma_1; \mathcal{U} \vdash t : \mathbb{T}_1 \otimes \mathbb{T}_2 \quad \Gamma_2, x_1 : \mathbb{T}_1, x_2 : \mathbb{T}_2; \mathcal{U} \vdash u : \mathbb{U}}{\Gamma_1, \Gamma_2; \mathcal{U} \vdash \text{case } t \text{ of } (x_1, x_2) \mapsto u : \mathbb{U}} \otimes \text{E} \quad \frac{\Gamma_1; \mathcal{U} \vdash t : \mathbb{!T} \quad \Gamma_2; \mathcal{U}, x : \mathbb{T} \vdash u : \mathbb{U}}{\Gamma_1, \Gamma_2; \mathcal{U} \vdash \text{case } t \text{ of } \text{Many } x \mapsto u : \mathbb{U}} \mathbb{!}\text{E}
 \end{array}$$

 Figure 1.5: Typing rules for linear λ -calculus in dyadic presentation ($\lambda_{L2}\text{-TY}$)

identity rule, $\lambda_{L2}\text{-TY}/\text{ID}_{\text{UR}}$, is the unrestricted identity: it lets us use the variable x from the unrestricted typing context in a place where a linear variable is expected, performing a sort of implicit dereliction. 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:

$$\frac{\frac{}{x' : \mathbb{T}; \bullet \vdash x' : \mathbb{T}} \text{ID}_{\text{LIN}} \quad \frac{}{x' : \mathbb{T}; \bullet \vdash x' : \mathbb{T}} \text{ID}_{\text{LIN}}}{x' : \mathbb{T}, x' : \mathbb{T}; \bullet \vdash (x', x') : \mathbb{T} \otimes \mathbb{T}} \otimes \text{I}$$

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

$$\frac{\frac{}{\bullet; x' : \mathbb{T} \vdash x' : \mathbb{T}} \text{ID}_{\text{UR}} \quad \frac{}{\bullet; x' : \mathbb{T} \vdash x' : \mathbb{T}} \text{ID}_{\text{UR}}}{\bullet; x' : \mathbb{T} \vdash (x', x') : \mathbb{T} \otimes \mathbb{T}} \otimes \text{I}$$

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

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

$$\begin{array}{c}
\frac{}{x : !T ; \bullet \vdash x : !T} \text{ID}_{\text{LIN}} \quad \frac{\frac{}{\bullet ; x' : T \vdash x' : T} \text{ID}_{\text{UR}} \quad \frac{}{\bullet ; x' : T \vdash x' : T} \text{ID}_{\text{UR}}}{\bullet ; x' : T \vdash (x', x') : T \otimes T} \otimes \text{I} \\
\hline
\frac{}{x : !T ; \bullet \vdash \text{case } x \text{ of } \text{Many } x' \mapsto (x', x') : T \otimes T} \text{!E} \\
\hline
\bullet ; \bullet \vdash \lambda x \mapsto \text{case } x \text{ of } \text{Many } x' \mapsto (x', x') : !T \multimap T \otimes T \quad \multimap \text{I}
\end{array}$$

If we had any useless variable binding polluting our unrestricted typing context (let's say $\bar{U} = y : !U$), 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):

$$\begin{array}{c}
\frac{}{x : !T ; y : !U \vdash x : !T} \text{ID}_{\text{LIN}} \quad \frac{\frac{}{\bullet ; y : !U, x' : T \vdash x' : T} \text{ID}_{\text{UR}} \quad \frac{}{\bullet ; y : !U, x' : T \vdash x' : T} \text{ID}_{\text{UR}}}{\bullet ; y : !U, x' : T \vdash (x', x') : T \otimes T} \otimes \text{I} \\
\hline
\frac{}{x : !T ; y : !U \vdash \text{case } x \text{ of } \text{Many } x' \mapsto (x', x') : T \otimes T} \text{!E} \\
\hline
\bullet ; y : !U \vdash \lambda x \mapsto \text{case } x \text{ of } \text{Many } x' \mapsto (x', x') : !T \multimap T \otimes T \quad \multimap \text{I}
\end{array}$$

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 lose anything by going from λ_{L1} presentation to λ_{L2} : Andreoli [1992] has a detailed proof that $\lambda_{L1}\text{-TY}$ and $\lambda_{L2}\text{-TY}$ are equivalent, in other terms, that a program t types in the pair of contexts $\Gamma; \bar{U}$ in $\lambda_{L2}\text{-TY}$ if and only if it types in context $\Gamma, !\bar{U}$ in $\lambda_{L1}\text{-TY}$.

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 [Jean-Yves Girard, Scedrov, and Scott, 1992], 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 $!_m$ is a *graded modality*.

Having a family of modalities $(!_m)_{m \in \mathbb{M}}$ 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 $!_m T$ to a pair $!_{m-1} T, T$ 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 $x : S, y : !_m T, z : !_n U \vdash \dots$, we can have $x :_1 S, y :_m T, z :_n U \vdash$. 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 types². 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 losing 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 [Ghica and Smith, 2014], 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 $t \ ; \ u$ 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:

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

We'll now show what the concrete *modal* presentation for intuitionistic λ -calculus or λ_{Lm} looks like. We use a very simple ringoid³ for modes, that just models linearity equivalently as the previous presentations λ_{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

²This uniform approach where every binding receives a mode seems to originate from [Ghica and Smith, 2014] and [Petricek, Orchard, and Mycroft, 2014].

³We still get all the algebraic benefits of [Ghica and Smith, 2014] even without a zero for our structure.

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

$$\begin{aligned}
v &::= \lambda x_{\mathfrak{m}} \mapsto u \mid () \mid \text{Inl } v \mid \text{Inr } v \mid (v_1, v_2) \mid \text{Mod}_{\mathfrak{n}} v \\
t, u &::= v \mid x \mid \text{Inl } t \mid \text{Inr } t \mid (t_1, t_2) \mid \text{Mod}_{\mathfrak{n}} t \mid t \mathbin{\text{\textcircled{>}}} t' \mid t \mathbin{\text{\textcircled{>}}} t' \\
&\quad \mid \text{case } t \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\} \mid \text{case } t \text{ of } (x_1, x_2) \mapsto u \mid \text{case } t \text{ of } \text{Mod}_{\mathfrak{n}} x \mapsto u \\
\mathbb{T}, \mathbb{U} &::= \mathbb{T}_{\mathfrak{m}} \multimap \mathbb{U} \mid \mathbf{1} \mid \mathbb{T}_1 \oplus \mathbb{T}_2 \mid \mathbb{T}_1 \otimes \mathbb{T}_2 \mid !_{\mathfrak{n}} \mathbb{T} \\
\mathfrak{m}, \mathfrak{n} &::= \mathbf{1} \mid \omega \\
\Gamma &::= \bullet \mid x :_{\mathfrak{m}} \mathbb{T} \mid \Gamma_1, \Gamma_2
\end{aligned}$$

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

$$\boxed{\Gamma \vdash t : \mathbb{T}} \quad (\text{Typing judgment for terms})$$

$$\begin{array}{c}
\frac{\mathbf{1} \leq \mathfrak{m}}{\omega \cdot \Gamma, x :_{\mathfrak{m}} \mathbb{T} \vdash x : \mathbb{T}} \text{ID} \qquad \frac{\Gamma, x :_{\mathfrak{m}} \mathbb{T} \vdash u : \mathbb{U}}{\Gamma \vdash \lambda x_{\mathfrak{m}} \mapsto u : \mathbb{T}_{\mathfrak{m}} \multimap \mathbb{U}} \multimap \text{I} \qquad \frac{}{\omega \cdot \Gamma \vdash () : \mathbf{1}} \mathbf{1I} \\
\\
\frac{\Gamma \vdash t_1 : \mathbb{T}_1}{\Gamma \vdash \text{Inl } t_1 : \mathbb{T}_1 \oplus \mathbb{T}_2} \oplus \text{I}_1 \qquad \frac{\Gamma \vdash t_2 : \mathbb{T}_2}{\Gamma \vdash \text{Inr } t_2 : \mathbb{T}_1 \oplus \mathbb{T}_2} \oplus \text{I}_2 \qquad \frac{\Gamma_1 \vdash t_1 : \mathbb{T}_1 \quad \Gamma_2 \vdash t_2 : \mathbb{T}_2}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : \mathbb{T}_1 \otimes \mathbb{T}_2} \otimes \text{I} \\
\\
\frac{\Gamma \vdash t : \mathbb{T}}{\mathfrak{n} \cdot \Gamma \vdash \text{Mod}_{\mathfrak{n}} t : !_{\mathfrak{n}} \mathbb{T}} !\text{I} \qquad \frac{\Gamma_1 \vdash t : \mathbb{T} \quad \Gamma_2 \vdash t' : \mathbb{T}_{\mathfrak{m}} \multimap \mathbb{U}}{\mathfrak{m} \cdot \Gamma_1 + \Gamma_2 \vdash t' t : \mathbb{U}} \multimap \text{E} \qquad \frac{\Gamma_1 \vdash t : \mathbf{1} \quad \Gamma_2 \vdash u : \mathbb{U}}{\Gamma_1 + \Gamma_2 \vdash t \mathbin{\text{\textcircled{>}}} u : \mathbb{U}} \mathbf{1E} \\
\\
\frac{\Gamma_1 \vdash t : \mathbb{T}_1 \oplus \mathbb{T}_2 \quad \Gamma_2, x_1 :_{\mathbf{1}} \mathbb{T}_1 \vdash u_1 : \mathbb{U} \quad \Gamma_2, x_2 :_{\mathbf{1}} \mathbb{T}_2 \vdash u_2 : \mathbb{U}}{\Gamma_1 + \Gamma_2 \vdash \text{case } t \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\} : \mathbb{U}} \oplus \text{E} \\
\\
\frac{\Gamma_1 \vdash t : \mathbb{T}_1 \otimes \mathbb{T}_2 \quad \Gamma_2, x_1 :_{\mathbf{1}} \mathbb{T}_1, x_2 :_{\mathbf{1}} \mathbb{T}_2 \vdash u : \mathbb{U}}{\Gamma_1 + \Gamma_2 \vdash \text{case } t \text{ of } (x_1, x_2) \mapsto u : \mathbb{U}} \otimes \text{E} \qquad \frac{\Gamma_1 \vdash t : !_{\mathfrak{n}} \mathbb{T} \quad \Gamma_2, x :_{\mathfrak{n}} \mathbb{T} \vdash u : \mathbb{U}}{\Gamma_1 + \Gamma_2 \vdash \text{case } t \text{ of } \text{Mod}_{\mathfrak{n}} x \mapsto u : \mathbb{U}} !\text{E}
\end{array}$$

Figure 1.7: Typing rules for linear λ -calculus in modal presentation ($\lambda_{Lm}\text{-TY}$)

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 $!_{\mathfrak{m}} \mathbb{T}$ is now $\text{Mod}_{\mathfrak{m}} t$ (instead of $\text{Many } t$ for $!\mathbb{T}$).

In λ_{Lm} we're back to a single identity rule $\lambda_{Lm}\text{-TY/ID}$, that asks for x to be in the typing context with a mode \mathfrak{m} that must be *compatible with a single linear use* (we note $\mathbf{1} \leq \mathfrak{m}$). In our ringoid, with only two elements, we have both $\mathbf{1} \leq \mathbf{1}$ and $\mathbf{1} \leq \omega$, so x can actually have any mode (either linear or unrestricted, so encompassing both $\lambda_{L2}\text{-TY/ID}_{\text{LIN}}$ and $\lambda_{L2}\text{-TY/ID}_{\text{UR}}$), but in more complex modal systems, not all modes have to be compatible with the unit of the

ringoid⁴. Rules $\lambda_{Lm\text{-TY/ID}}$ and $\lambda_{Lm\text{-TY/1I}}$ 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 $x : \omega\mathbf{T}$. 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 $!_{\mathfrak{m}}$ is eliminated by a **case** t **of** $\text{Mod}_{\mathfrak{m}} x \mapsto u$ expression, that binds the scrutinee to a new variable x with mode \mathfrak{m} . The original boxed value can be recreated if needed with $\text{Mod}_{\mathfrak{m}} x$; indeed, as in λ_{L2} , elimination rule for $!_{\mathfrak{m}}$ is *not* dereliction.

The last specificity of this system is that the function arrow \multimap now has a mode \mathfrak{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 \mathfrak{m} (and the rule $\lambda_{Lm}\text{-TY}/\multimap\text{E}$ for function application reflects that: the typing context Γ_1 for the argument passed to the function is multiplied by \mathfrak{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 \mathfrak{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):

$$\frac{\frac{\frac{1 \leq 1}{y : \omega \mathbf{U}, x : 1^! \mathbf{T} \vdash x : !\mathbf{T}} \text{ID} \quad \frac{\frac{x' : 1^! \mathbf{T} \vdash x' : \mathbf{T}} \text{ID} \quad \frac{x' : 1^! \mathbf{T} \vdash x' : \mathbf{T}} \text{ID}}{x' : \omega \mathbf{T} \vdash (x', x') : \mathbf{T} \otimes \mathbf{T}} \otimes \text{I}}{y : \omega \mathbf{U}, x : 1^! \mathbf{T} \vdash \text{case } x \text{ of } \text{Mod}_{\omega} x' \mapsto (x', x') : \mathbf{T} \otimes \mathbf{T}} !\text{E}}{y : \omega \mathbf{U} \vdash \lambda x. 1 \mapsto \text{case } x \text{ of } \text{Mod}_{\omega} x' \mapsto (x', x') : !\mathbf{T} \multimap \mathbf{T} \otimes \mathbf{T}} \multimap \text{I}$$

Now the unrestricted, unused part of the typing context ($y : \omega\mathbf{U}$ 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 $\lambda_{Lm}\text{-TY/ID}$ 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 $\mathbb{T}_{\omega} \multimap \mathbb{T} \otimes \mathbb{T}$ instead of $! \mathbb{T}_{\omega} \multimap \mathbb{T} \otimes \mathbb{T}$:

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

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

$$\frac{\frac{\frac{1 \leq 1}{y : \omega \mathbf{U}, x' : \mathbf{T} \vdash x' : \mathbf{T}} \text{ID} \quad \frac{\frac{1 \leq 1}{y : \omega \mathbf{U}, x' : \mathbf{T} \vdash x' : \mathbf{T}} \text{ID}}{y : \omega \mathbf{U}, x' : \omega \mathbf{T} \vdash (x', x') : \mathbf{T} \otimes \mathbf{T}} \otimes \text{I}}{y : \omega \mathbf{U} \vdash \lambda x'_{\omega} \mapsto (x', x') : \mathbf{T}_{\omega \multimap} \mathbf{T} \otimes \mathbf{T}} \multimap \text{I}$$

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

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

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 $\mathbf{T} \otimes \mathbf{U}_{\omega \multimap} \mathbf{T}$, 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:

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

It's often desirable to lift this limitation in a real programming language, and let the $!_m$ 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 $\mathbf{T}_{\omega \multimap} \mathbf{U}$ gain exactly the same expressive power as the ones with type $\mathbf{T} \rightarrow \mathbf{U}$ in STLC. In other terms, any program that is valid in STLC but doesn't abide by linearity can still type in

$\boxed{\Gamma \vdash t : \mathbb{T}}$

(Typing judgment for terms)

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

$$\begin{array}{c}
 \Gamma_1 \vdash t : \mathbb{T}_1 \otimes \mathbb{T}_2 \\
 \Gamma_2, x_1 : \mathbb{m}\mathbb{T}_1, x_2 : \mathbb{m}\mathbb{T}_2 \vdash u : \mathbb{U} \\
 \hline
 \mathbb{m} \cdot \Gamma_1 + \Gamma_2 \vdash \text{case}_{\mathbb{m}} t \text{ of } (x_1, x_2) \mapsto u : \mathbb{U} \quad \otimes\text{E}
 \end{array}$$

$$\begin{array}{c}
 \Gamma_1 \vdash t : \mathbb{!}_n \mathbb{T} \\
 \Gamma_2, x : \mathbb{m} \cdot n \mathbb{T} \vdash u : \mathbb{U} \\
 \hline
 \mathbb{m} \cdot \Gamma_1 + \Gamma_2 \vdash \text{case}_{\mathbb{m}} t \text{ of } \text{Mod}_n x \mapsto u : \mathbb{U} \quad \mathbb{!}\text{E}
 \end{array}$$

 Figure 1.8: Altered typing rules for deep modes in linear λ -calculus in modal presentation ($\lambda_{Ldm}\text{-TY}$)

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 **fst**_m.

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

$$\begin{aligned}
 \text{fst}_{dm} &: \mathbb{T} \otimes \mathbb{U} \multimap \mathbb{T} \\
 \text{fst}_{dm} &\triangleq \lambda x_{\omega} \mapsto \text{case}_{\omega} x \text{ of } (x_1, x_2) \mapsto x_1
 \end{aligned}$$

This is precisely the same implementation as **fst**_{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 [Bernardy et al., 2018] and in recent work from Lorenzen, White, et al. [2024]. With deep modes, we get the following equivalences:

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

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 $\lambda_{Lm}\text{-TY}/\multimap\text{E}$: 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 $x_2 : \mathbb{!}_1 \mathbb{T}_2 \vdash ((), x_2) : \mathbb{1} \otimes \mathbb{T}_2$ (the pair uses x_2 exactly once), if we want to extract the pair components with mode ω to drop x_2 (as in **fst**_{dm}), then **case** _{ω} $((), x_2) \text{ of } (x_1, x_2) \mapsto x_2$ will require context $\omega \cdot (x_2 : \mathbb{!}_1 \mathbb{T}_2)$ i.e. $x_2 : \omega \mathbb{T}_2$. 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 [Felleisen, 1987] and subsequent [Danvy and Nielsen, 2004; Biernacka and Danvy, 2007], 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 $E[t]$ of an evaluation context E , and a term t under focus. We call such a pair $E[t]$ a *command*, borrowing the terminology from Curien and Herbelin [2000].

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, $\text{Mod}_m t$ 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 $(E \circ \text{case}_m [] \text{ of } (x_1, x_2) \mapsto u)[(v_1, v_2)]$, without the need to recreate the term $\text{case}_m(v_1, v_2) \text{ of } (x_1, x_2) \mapsto u$. 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 $!T \multimap (T \otimes T)$, here applied to the argument $\text{Mod}_\omega(\text{Inl}())$:

$$\begin{aligned}
& [][(\lambda x. 1 \mapsto \text{case } x \text{ of } \text{Mod}_\omega x' \mapsto (x', x')) (\text{Mod}_\omega(\text{Inl}()))] \\
\longrightarrow & [][(\text{case } \text{Mod}_\omega(\text{Inl}()) \text{ of } \text{Mod}_\omega x' \mapsto (x', x'))] && \multimap\text{EC} \quad \text{with substitution } x := \text{Mod}_\omega(\text{Inl}()) \\
\longrightarrow & [][(\text{Inl}(), \text{Inl}())] && !\text{EC} \quad \text{with substitution } x' := \text{Inl}()
\end{aligned}$$

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 a succession of contraction steps.

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

\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 $[] [v]$ 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 $[] [t]$ then a focusing or contraction step should trigger; and if we have a non-empty evaluation context E in $E[v]$ 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 [Bernardy et al., 2018] is a language extension for the Glasgow Haskell Compiler (GHC) that introduces the linear function arrow, $\mathbf{t} \multimap \mathbf{u}$ and modifies the type checker of GHC so that it can enforce linearity requirements. A linear function of type $\mathbf{t} \multimap \mathbf{u}$ 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 $\mathbf{t} \rightarrow \mathbf{u}$ 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, $\mathbf{t} \% \mathbf{m} \rightarrow \mathbf{u}$, much like $\mathbf{T}_m \multimap \mathbf{U}$; so $\mathbf{t} \multimap \mathbf{u}$ and $\mathbf{t} \rightarrow \mathbf{u}$ are sugar for $\mathbf{t} \% 1 \rightarrow \mathbf{u}$ and $\mathbf{t} \% \omega \rightarrow \mathbf{u}$ 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 \mathbf{u} when consuming \mathbf{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 $!_{\omega}T$ in λ_{Ldm} , and there is an equivalence between `Ur t \multimap u` and `t \rightarrow 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 \multimap 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 $\lambda_{L1-TY}/!P$ or $\lambda_{L2,m-TY}/!I$ where a term wrapped in `Many` or `Mod $_{\omega}$` must only depend on an unrestricted context. The only exceptions to this rule are terms of types that implement the `Movable` typeclass⁵ such as `Int` or `()`. `Movable` provides the function `move :: t \multimap 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:

`(;) :: () \multimap u \multimap 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 $\lambda_{Ldm-TY}/!E$. It isn't part of the core Haskell syntax, but can be defined simply as `case term1 of () -> term2`.

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

`Class \Rightarrow ...` 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 \rightarrow 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 functions⁶. 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 [Marshall, Vollmer, and Orchard, 2022]. 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 [Harrington, 2006]. 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 [Harrington, 2006] or mentioned briefly in [Marshall, Vollmer, and Orchard, 2022] (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).

⁶At 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 [2022] 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:

```

1 | openFile :: String → File
2 | closeFile :: File → ()
3 |
4 | file :: File -- no way to force this value to be used exactly once
5 | file = openFile "test.txt"
6 |
7 | nonLinearUseButValid :: () =
8 |   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:

```

1 | withFile :: String → (File → t) → t
2 | closeFile :: File → ()
3 | nonLinearUseAndEffectivelyRejected :: () =
4 |   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:

```

1 | withFile' :: String -> (File -> Ur t) -> Ur t
2 | closeFile :: File -> ()
3 | exampleOk :: Ur () = withFile' "test.txt" (\file -> closeFile file ; Ur ())
4 | exampleFail :: Ur File =
5 |   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 [2023b]. It manifests itself in the following example, which gets rejected:

```

1 | readLine :: File -> (Ur String, File)
2 | writeLine :: File -> String -> File
3 |
4 | exampleFail' :: Ur () =
5 |   withFile' "test.txt" (\file1 ->
6 |     ...
7 |     withFile' "test2.txt" (\file2 ->
8 |       let (Ur line, file1') = readLine file1
9 |         file2' = writeLine file2 line
10 |        in closeFile file2' ; Ur file1')
11 |     ... -- other operations with file1' here
12 |   )

```

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!

⁷This is what is referred to as *sticky end of linear scopes* in [Spiwack, 2023b]


```

1  type Token
2  dup :: Token → (Token, Token)
3  drop :: Token → ()
4  withToken :: (Token → Ur t) → Ur t
5
6  -- A first linear API
7  openFile' :: String → Token → File
8  closeFile :: File → ()
9
10 -- A second linear API
11 openDB :: String → String → Token → DB
12 closeDB :: DB → ()
13
14 exampleOk1 :: Ur () =
15   withToken (\tok → let file = openFile' "test.txt" tok in closeFile file) ; Ur ()
16 exampleOk2 :: Ur () =
17   withToken (\tok → let (tok1, tok2) = dup tok
18                       file = openFile' "test.txt" tok1
19                       db = openDB "localhost" 3306 tok2
20                       in closeFile file ; closeDB db) ; Ur ()
21 exampleRejected :: Ur File =
22   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, and can otherwise be written in direct style.

Although it's out of scope for this document, Spiwack et al. [2022] 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 convenient⁸.

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.

⁸At the time of writing this document, Linear Constraints are seriously considered to be added into GHC, the main Haskell compiler [Spiwack, 2023a].

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 [Bagrel and Spiwack, 2025].

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 $\llbracket \mathbf{T} \rrbracket$ where \mathbf{T} is the type of what can be stored in the destination. A function of type $\mathbf{T} \rightarrow \mathbf{U}$ would have signature $\mathbf{T} \rightarrow \llbracket \mathbf{U} \rrbracket \rightarrow \mathbf{1}$ when transformed into destination-passing style. Instead of returning an \mathbf{U} , it consumes a destination of type $\llbracket \mathbf{U} \rrbracket$ and returns just *unit* (type $\mathbf{1}$). This style is common in systems programming, where destinations $\llbracket \mathbf{U} \rrbracket$ are more commonly known as *out parameters* (in C, $\llbracket \mathbf{U} \rrbracket$ would typically be a pointer of type \mathbf{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 \mathbf{U} , it has to allocate the space for a \mathbf{U} . But with destinations, the caller is responsible for finding space for a \mathbf{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 \mathbf{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 [1998];
- 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 [Minamide, 1998] to [Lorenzen, Leijen, Swierstra, and Lindley, 2024]. 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 $\mathbf{T} \oplus \mathbf{U}$ and $\mathbf{T} \otimes \mathbf{U}$ 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 $\text{Inl } t$ 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 the output value instead of returning it proper. For instance, the following is a destination-passing version of the identity function:

dId : $\mathsf{T} \multimap [\mathsf{T}] \multimap 1$
dId $x\ d \triangleq d \blacktriangleleft x$

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 : $[\mathsf{T} \oplus \mathsf{U}] \multimap [\mathsf{T}]$
fillWithInl $d \triangleq d \triangleleft \text{Inl}$

In this example, we’re filling a destination for type $\mathsf{T} \oplus \mathsf{U}$ by setting the outermost constructor to left variant **Inl**. We think of $d \triangleleft \text{Inl}$ (read “fill d with **Inl**”) as allocating memory to store a block of the form **Inl** \square , 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 $[\mathsf{T}]$ pointing to the uninitialized argument of **Inl**. Uninitialized memory, when part of a structure or value, like \square in **Inl** \square , 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** \square into a hole, but we have yet to describe what goes in the new hole \square . 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 : $[\mathsf{T} \otimes \mathsf{U}] \multimap [\mathsf{T}] \otimes [\mathsf{U}]$
fillWithPair $d \triangleq d \triangleleft (,)$

After using **fillWithPair**, both the first field *and* the second field of the pair must be filled, using the destinations of type $[\mathsf{T}]$ and $[\mathsf{U}]$ respectively. If we were to replace destinations by continuations in **fillWithPair**, we wouldn’t be able to use the two returned continuations easily.

⁹The full triangle \blacktriangleleft is used to fill a destination with a fully-formed value, while the *hollow* triangle \triangleleft is used to fill a destination with a *hollow constructor*.

We can already note there that there is a sort of duality between destination-filling operators and associated constructors. Usual `Inl` constructor has signature $\mathbf{T} \multimap \mathbf{T} \oplus \mathbf{U}$, while destination-filling $\triangleleft \text{Inl}$ has signature $[\mathbf{T} \oplus \mathbf{U}] \multimap [\mathbf{T}]$. Similarly, pair constructor $(,)$ has signature $\mathbf{T} \multimap \mathbf{U} \multimap \mathbf{T} \otimes \mathbf{U}$, while destination-filling $\triangleleft (,)$ has signature $[\mathbf{T} \otimes \mathbf{U}] \multimap [\mathbf{T}] \otimes [\mathbf{U}]$. Assuming some flexibility with currying, we see that the types of arguments and results switch sides around the arrow, and get wrapped/unwrapped from $[\cdot]$ 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 $[\mathbf{T}]$ only indicates that some bigger structure has at least a hole of type \mathbf{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 $\mathbf{T} \otimes \mathbf{U}$ 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 $\mathbf{S} \ltimes [\mathbf{T}]$ for a structure of type \mathbf{S} missing a value of type \mathbf{T} to be complete. There can be several holes in \mathbf{S} — resulting in several destinations on the right hand side — and as long as there remain holes in \mathbf{S} , it cannot be read. For instance, $\mathbf{S} \ltimes ([\mathbf{T}] \otimes [\mathbf{U}])$ represents an \mathbf{S} that misses both a \mathbf{T} and a \mathbf{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 $\mathbf{S} \ltimes \mathbf{T}$ is read “ \mathbf{S} ampar \mathbf{T} ”. The name “ampar” stands for “asymmetric memory par”. The “par” \wp 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: $\mathbf{T} \multimap \mathbf{S}$ is equivalent to $\mathbf{T}^\perp \wp \mathbf{S}$ or $\mathbf{S} \wp \mathbf{T}^\perp$. Similarly, $\mathbf{T}_1 \multimap \mathbf{T}_2 \multimap \mathbf{S}$ is equivalent to $\mathbf{T}_1^\perp \wp \mathbf{T}_2^\perp \wp \mathbf{S}$. Function input's types — which are in negative position — are wrapped in the dualizing operator \cdot^\perp when a function is put into “par” form. Here we take a similar approach: Minamide [1998] first observed that structures with holes are akin to linear functions $\mathbf{T} \multimap \mathbf{S}$, where \mathbf{T} represents the missing part; so we decide to represent them in a more flexible fashion under the form $\mathbf{S} \ltimes [\mathbf{T}]$. In this presentation, the *missing part of type \mathbf{T}* appears as a first-class type $[\mathbf{T}]$, 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 \wp 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 `upd×` construction, which lets us apply a function to the right-hand side of the ampar. It is in the body of `upd× : $\mathbf{S} \ltimes \mathbf{T} \multimap (\mathbf{T} \multimap \mathbf{U}) \multimap \mathbf{S} \ltimes \mathbf{U}$` that functions operating on destinations can be called to update the structure:

$\text{fillWithInl}' : \mathbf{S} \ltimes [\mathbf{T} \oplus \mathbf{U}] \multimap \mathbf{S} \ltimes [\mathbf{T}]$
 $\text{fillWithInl}' x \triangleq \text{upd}_\ltimes x \text{ with } d \mapsto d \triangleleft \text{Inl}$
 $\text{fillWithPair}' : \mathbf{S} \ltimes [\mathbf{T} \otimes \mathbf{U}] \multimap \mathbf{S} \ltimes ([\mathbf{T}] \otimes [\mathbf{U}])$
 $\text{fillWithPair}' x \triangleq \text{upd}_\ltimes x \text{ with } d \mapsto \text{fillWithPair } d$

To tie this up, we need a way to introduce and to eliminate structures with holes. Structures with holes are introduced with new_\ltimes which creates a value of type $\mathbf{T} \ltimes [\mathbf{T}]$. new_\ltimes is somewhat similar to the linear identity function: it is a hole (of type \mathbf{T}) that needs a value of type \mathbf{T} to be a complete value of type \mathbf{T} . Memory-wise, it is an uninitialized block large enough to host a value of type \mathbf{T} , and a destination pointing to it. Conversely, structures with holes are eliminated with¹⁰ $\text{from}'_\ltimes : \mathbf{S} \ltimes \mathbf{1} \multimap \mathbf{S}$: if all the destinations have been consumed and only unit remains on the right side, then \mathbf{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 $\mathbin{\text{\textcircled{;}}} : \mathbf{1} \multimap \mathbf{U} \multimap \mathbf{U}$, present in second example, is used to chain operations returning unit type $\mathbf{1}$. For easier reading, we also provide type annotation when new_\ltimes is used:

$\text{Inl} : \mathbf{T} \multimap \mathbf{T} \oplus \mathbf{U}$
 $\text{Inl } x \triangleq \text{from}'_\ltimes (\text{upd}_\ltimes (\text{new}_\ltimes : (\mathbf{T} \oplus \mathbf{U}) \ltimes [\mathbf{T} \oplus \mathbf{U}]) \text{ with } d \mapsto d \triangleleft \text{Inl} \triangleleft x)$
 $(,) : \mathbf{T} \multimap \mathbf{U} \multimap \mathbf{T} \otimes \mathbf{U}$
 $(x, y) \triangleq \text{from}'_\ltimes (\text{upd}_\ltimes (\text{new}_\ltimes : (\mathbf{T} \otimes \mathbf{U}) \ltimes [\mathbf{T} \otimes \mathbf{U}]) \text{ with } d \mapsto \text{case } (d \triangleleft (,)) \text{ of } (d_1, d_2) \mapsto d_1 \triangleleft x \mathbin{\text{\textcircled{;}}} d_2 \triangleleft y)$

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:

$\text{forget} : \mathbf{T}$
 $\text{forget} \triangleq \text{from}'_\ltimes (\text{upd}_\ltimes (\text{new}_\ltimes : \mathbf{T} \ltimes [\mathbf{T}]) \text{ with } d \mapsto ())$

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:

$\text{ambiguous1} : \text{Bool}$
 $\text{ambiguous1} \triangleq \text{from}'_\ltimes (\text{upd}_\ltimes (\text{new}_\ltimes : \text{Bool} \ltimes [\text{Bool}]) \text{ with } d \mapsto d \triangleleft \text{true} \mathbin{\text{\textcircled{;}}} d \triangleleft \text{false})$
 $\text{ambiguous2} : \text{Bool}$
 $\text{ambiguous2} \triangleq \text{from}'_\ltimes (\text{upd}_\ltimes (\text{new}_\ltimes : \text{Bool} \ltimes [\text{Bool}]) \text{ with } d \mapsto \text{let } x := (d \triangleleft \text{false}) \text{ in } d \triangleleft \text{true} \mathbin{\text{\textcircled{;}}} x)$

¹⁰As the name suggest, there is a more general elimination 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: $\text{List } T \triangleq 1 \oplus (T \otimes (\text{List } T))$. 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:

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

Just like we did in Section 2.2.1 we can recover traditional constructors systematically from destination-filling operators, using new_\times , upd_\times and from'_\times :

$$\begin{array}{l} (::) : T \otimes (\text{List } T) \multimap \text{List } T \\ x :: xs \triangleq \text{from}'_\times (\text{upd}_\times (\text{new}_\times : (\text{List } T) \times [\text{List } T]) \text{ with } d \mapsto \\ \quad \text{case } (d \triangleleft (::)) \text{ of } (dx, dxs) \mapsto dx \triangleleft x \ ; \ dxs \triangleleft xs) \end{array}$$

A tail-recursive map function List being ubiquitous in functional programming, the fact that the most natural way to write a **map** function on lists isn't tail recursive (hence consumes unbounded stack space), is unpleasant. Map can be made tail-recursive in two passes: first build the result list in reverse 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 Listing 2.1.

The tail-recursive function is **map'**, it has type $(T \multimap U) \multimap \text{List } T \multimap [\text{List } U] \multimap 1$ 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 types¹².

¹¹In 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 $(T \multimap U) \multimap \text{List } T \multimap [\text{List } U] \multimap 1$.

¹²We 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  $\triangleq$  1  $\oplus$  (T  $\otimes$  (List T))
map' : (T  $\multimap$  U)  $\multimap$  List T  $\multimap$  [List U]  $\multimap$  1
map' f l dl  $\triangleq$ 
  case l of {
    []  $\mapsto$  dl  $\triangleleft$  [],
    x :: xs  $\mapsto$  case (dl  $\triangleleft$  (::)) of
      (dx, dxs)  $\mapsto$  dx  $\blacktriangleleft$  f x ; map' f xs dxs }
map : (T  $\multimap$  U)  $\multimap$  List T  $\multimap$  List U
map f l  $\triangleq$  from'_{\times}(\text{upd}_{\times}(\text{new}_{\times} : (\text{List U}) \times [\text{List U}]) \text{ with } dl \mapsto \text{map}' f l dl)
```

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

To obtain the regular **map** function, all is left to do is to call **new**_× to create an initial destination, and **from**'_× 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 [Minamide, 1998], to recent work [Bour, Clément, and Scherer, 2021; Leijen and Lorenzen, 2023]. 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 work towards an example which uses first-class destinations at their full potential.

Difference lists Just like in any language, iterated concatenation of linked lists $((xs_1 ++ xs_2) ++ \dots) ++ xs_n$ 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 [Hughes, 1986], as we usually don't have write pointers. A singleton difference list is then $\lambda ys \mapsto x :: ys$, and concatenation of difference lists is function composition. A difference 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 . The detailed implementation is given on the left of Listing 2.2. This encoding for difference lists makes no superfluous traversal: concatenation is just an $O(1)$ in-place update.

<pre> DList T \triangleq (List T) \times [List T] append : DList T \multimap T \multimap DList T ys append y \triangleq upd$_{\times}$ ys with dys \mapsto case (dys \triangleleft (::)) of (dy, dys') \mapsto dy \blacktriangleleft y \circ dys' concat : DList T \multimap DList T \multimap DList T ys concat ys' \triangleq upd$_{\times}$ ys with d \mapsto d \triangleleft ys' toList : DList T \multimap List T toList ys \triangleq from'$_{\times}$ (upd$_{\times}$ ys with d \mapsto d \triangleleft []) </pre>	<pre> Queue T \triangleq (List T) \otimes (DList T) singleton : T \multimap Queue T singleton x \triangleq (Inr (x :: []), (new$_{\times}$: DList T)) enqueue : Queue T \multimap T \multimap Queue T q enqueue y \triangleq case q of (xs, ys) \mapsto (xs, ys append y) dequeue : Queue T \multimap 1 \oplus (T \otimes (Queue T)) dequeue q \triangleq case q of { ((x :: xs), ys) \mapsto Inr (x, (xs, ys)), ([], ys) \mapsto case (toList ys) of { [] \mapsto Inl (), x :: xs \mapsto Inr (x, (xs, (new$_{\times}$: DList T))) } } </pre>
--	--

Listing 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*) [Hood and Melville, 1981]. 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 Listing 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 \blacktriangleleft or \triangleleft , so that the associated holes get written to. This turns out to be slightly incomplete.

To see why, let's consider the type $[[T]]$: the type of a destination pointing to a hole where a destination is expected. Think of it as an equivalent of the pointer type `T**` in the C language. Destinations are indeed ordinary values, so they can be stored in data structures, and before they get stored, holes stand in their place in the structure. For instance, if we have $d : [T]$ and $dd : [[T]]$, we can form $dd \blacktriangleleft d$: 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 $dd \blacktriangleleft d \ ; \ d \blacktriangleleft v$ 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 **alloc** : $([T] \multimap 1) \multimap T$ 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 . **alloc** f corresponds, in type and behavior, to **from**'_×(**upd**'_× **new**'_× 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 $\rightarrow h$ and $\rightarrow hd$.

When the building scope of l : **Bool** is parent to the one of ld : $\boxed{\text{Bool}}$, 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:

```

      { } | alloc (λd ↦ (alloc (λdd ↦ dd ◀ d) : [Bool]) ◀ true)
→ {l :=  $\boxed{h}$ } | (alloc (λdd ↦ dd ◀  $\rightarrow h$ ) : [Bool]) ◀ true ; deref l
→ {l :=  $\boxed{h}$ , ld :=  $\boxed{hd}$ } | ( $\rightarrow hd \blacktriangleleft \rightarrow h$  ; deref ld) ◀ true ; deref l
→ {l :=  $\boxed{h}$ , ld :=  $\rightarrow h$ } | deref ld ◀ true ; deref l
→ {l :=  $\boxed{h}$ } |  $\rightarrow h \blacktriangleleft$  true ; deref l
→ {l := true} | deref l
→ { } | true

```

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:

```

      { } | alloc (λdd ↦ case (alloc (λd ↦ dd ◀ d) : Bool) of {true ↦ (), false ↦ ()})
→ {ld :=  $\boxed{hd}$ } | case (alloc (λd ↦  $\rightarrow hd \blacktriangleleft d$ ) : Bool) of {true ↦ (), false ↦ ()} ; deref ld
→ {ld :=  $\boxed{hd}$ , l :=  $\boxed{h}$ } | case ( $\rightarrow hd \blacktriangleleft \rightarrow h$  ; deref l) of {true ↦ (), false ↦ ()} ; deref ld
→ {ld :=  $\rightarrow h$ , l :=  $\boxed{h}$ } | case (deref l) of {true ↦ (), false ↦ ()} ; deref ld
→ {ld :=  $\rightarrow h$ } | case  $\boxed{h}$  of {true ↦ (), false ↦ ()} ; deref ld

```

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 $\boxed{[T]}$ 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 example 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 [Gibbons, 1993; Okasaki, 2000; Gibbons et al., 2023].

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 [Minamide, 1998] 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.

Listing 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** $\triangleq 1 \oplus (T \otimes ((\text{Tree T}) \otimes (\text{Tree T})))$; 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  $\omega_{rec} \multimap T_1 \multimap (!_{\omega\infty} S) \otimes T_2$ )  $\omega_{\infty} \multimap S \omega_{\infty} \multimap \text{Queue } (\text{Tree } T_1 \otimes \lfloor \text{Tree } T_2 \rfloor) \multimap 1$ 
go f st q  $\triangleq$  case (dequeue q) of {
  Inl ()  $\mapsto$  (),
  Inr ((tree, dtree), q')  $\mapsto$  case tree of {
    Nil  $\mapsto$  dtree  $\triangleleft$  Nil ; go f st q',
    Node x tl tr  $\mapsto$  case (dtree  $\triangleleft$  Node) of
      (dy, (dtl, dtr))  $\mapsto$  case (f st x) of
        (Mod  $\omega_{\infty}$  st', y)  $\mapsto$ 
          dy  $\blacktriangleleft$  y ;
          go f st' (q' enqueue (tl, dtl) enqueue (tr, dtr)))
  }
}

mapAccumBFS : (S  $\omega_{\infty} \multimap T_1 \multimap (!_{\omega\infty} S) \otimes T_2$ )  $\omega_{\infty} \multimap S \omega_{\infty} \multimap \text{Tree } T_1 \mathbf{1}_{\infty} \multimap \text{Tree } T_2$ 
mapAccumBFS f st tree  $\triangleq$  from'  $\times$  (upd  $\times$  (new  $\times$  : (Tree T2)  $\times$   $\lfloor$  Tree T2  $\rfloor$ ) with dtree  $\mapsto$ 
  go f st (singleton (tree, dtree)))

relabelDPS : Tree 1  $\mathbf{1}_{\infty} \multimap \text{Tree Nat}$ 
relabelDPS tree  $\triangleq$  mapAccumBFS ( $\lambda st \omega_{\infty} \mapsto \lambda un \mapsto un ; (\text{Mod } \omega_{\infty} (\text{succ st}), st)$ ) 1 tree

```

Listing 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 (**1**), product (\otimes) and sum (\oplus) types, inspired from λ_{Ldm} described in Chapter 1. It also features a mode-polymorphic function arrow \multimap_m . Its most distinctive features however are the destination $\lfloor_m T \rfloor$ and ampar $S \ltimes T$ 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.

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

Core grammar of terms:

$$\begin{aligned}
 t, u ::= & x \mid t' t \mid t \circ t' \\
 & \mid \text{case}_m t \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\} \mid \text{case}_m t \text{ of } (x_1, x_2) \mapsto u \mid \text{case}_m t \text{ of } \text{Mod}_m x \mapsto u \\
 & \mid \text{upd}_\times t \text{ with } x \mapsto t' \mid \text{to}_\times t \mid \text{from}_\times t \mid \text{new}_\times \\
 & \mid t \triangleleft () \mid t \triangleleft \text{Inl} \mid t \triangleleft \text{Inr} \mid t \triangleleft (,) \mid t \triangleleft \text{Mod}_m \mid t \triangleleft (\lambda x_m \mapsto u) \mid t \triangleleft \circ t' \mid t \triangleleft t'
 \end{aligned}$$

Grammar of types, modes and contexts:

$$\begin{aligned}
 T, U, S ::= & \lfloor_n T \rfloor \quad (\text{destination}) \\
 & \mid S \times T \quad (\text{ampar}) \\
 & \mid 1 \mid T_1 \oplus T_2 \mid T_1 \otimes T_2 \mid !_m T \mid T_m \multimap U \\
 \Gamma ::= & \bullet \mid x :_m T \mid \Gamma_1, \Gamma_2
 \end{aligned}
 \quad \left| \quad \begin{aligned}
 m, n ::= & pa \quad (\text{pair of multiplicity and age}) \\
 p ::= & 1 \mid \omega \\
 a ::= & \uparrow^k \mid \infty \\
 \nu \triangleq & \uparrow^0 \quad \uparrow \triangleq \uparrow^1
 \end{aligned} \right.$$

Ordering on modes:

$$pa \preceq p'a' \iff p \preceq_p p' \wedge a \preceq_a a'$$



Operations on modes:

+	1	ω
1	ω	ω
ω	ω	ω

\cdot	1	ω
1	1	ω
ω	ω	ω

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

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

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

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

Operations on typing contexts:

$$\begin{aligned}
 n \cdot \bullet & \triangleq \bullet \\
 n \cdot (x :_m T, \Gamma) & \triangleq (x :_{n \cdot m} T), n \cdot \Gamma
 \end{aligned}
 \quad \left| \quad \begin{aligned}
 \bullet + \Gamma & \triangleq \Gamma \\
 (x :_m T, \Gamma_1) + \Gamma_2 & \triangleq x :_m T, (\Gamma_1 + \Gamma_2) \quad \text{if } x \notin \Gamma_2 \\
 (x :_m T, \Gamma_1) + (x :_{m'} T, \Gamma_2) & \triangleq x :_{m+m'} T, (\Gamma_1 + \Gamma_2)
 \end{aligned} \right.$$

Figure 2.1: Terms, types and modes of λ_d

$$\begin{aligned}
 \text{Inl } t & \triangleq \text{from}'_\times (\text{upd}_\times \text{new}_\times \text{ with } d \mapsto d \triangleleft \text{Inl} \triangleleft t) \\
 \text{Inr } t & \triangleq \text{from}'_\times (\text{upd}_\times \text{new}_\times \text{ with } d \mapsto d \triangleleft \text{Inr} \triangleleft t) \\
 \text{Mod}_m t & \triangleq \text{from}'_\times (\text{upd}_\times \text{new}_\times \text{ with } d \mapsto d \triangleleft \text{Mod}_m \triangleleft t) \\
 \lambda x_m \mapsto u & \triangleq \text{from}'_\times (\text{upd}_\times \text{new}_\times \text{ with } d \mapsto d \triangleleft (\lambda x_m \mapsto u))
 \end{aligned}
 \quad \left| \quad \begin{aligned}
 \text{from}'_\times t & \triangleq \\
 & \text{case } (\text{from}_\times (\text{upd}_\times t \text{ with } un \mapsto un \circ \text{Mod}_{1\infty} ())) \text{ of} \\
 & (st, ex) \mapsto \text{case } ex \text{ of} \\
 & \text{Mod}_{1\infty} un \mapsto un \circ st \\
 () & \triangleq \text{from}'_\times (\text{upd}_\times \text{new}_\times \text{ with } d \mapsto d \triangleleft ()) \\
 (t_1, t_2) & \triangleq \text{from}'_\times (\text{upd}_\times \text{new}_\times \text{ with } d \mapsto \text{case } (d \triangleleft (,)) \text{ of} \\
 & (d_1, d_2) \mapsto d_1 \triangleleft t_1 \circ d_2 \triangleleft t_2)
 \end{aligned} \right.$$

Listing 2.4: Syntactic sugar definitions for terms

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 $\text{upd}_\times t \text{ with } x \mapsto t'$ construct. For instance, if we have a term $\text{upd}_\times t_1 \text{ with } x_1 \mapsto \text{upd}_\times t_2 \text{ with } x_2 \mapsto \text{upd}_\times t_3 \text{ with } x_3 \mapsto x_1$, then we say that the innermost occurrence of x_1 has age \uparrow^2 because two nested upd_\times 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 2.1, and the syntactic sugar that we used in Sections 2.2 to 2.4 is presented in Listing 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 $n \cdot (m_1 + m_2) = n \cdot m_1 + n \cdot m_2$, 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 [Atkey, 2018] or [Bernardy et al., 2018], and the same as described in detail in Chapter 1. The full description of the multiplicity ringoid is given again in Figure 2.1.

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 $\text{upd}_\times t \text{ with } x \mapsto t'$ 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 $m \leq n$). 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 [Ghica and Smith, 2014] (see Figure 2.1).

Anticipating on next section, let's see how *plus* works on context, modes, and especially ages. Let's type the expression $d \blacktriangleleft x \ ; \ x$. 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:

$$\frac{\frac{\frac{1\nu \leq 1\nu}{d : 1\nu [T] \vdash d : [T]} \text{ID}_V \quad \frac{1\nu \leq 1\nu}{x : 1\nu T \vdash x : T} \text{ID}_V}{d : 1\nu [T], x : 1\uparrow T \vdash d \blacktriangleleft x : 1} \text{E}_L \quad \frac{1\nu \leq 1\nu}{x : 1\nu T \vdash x : T} \text{ID}_V}{d : 1\nu [T], x : \omega\infty T \vdash d \blacktriangleleft x \ ; \ x : T} 1E$$

The “default choice” is to type variables with mode 1ν at the leaves of the tree. At the root of the tree, with operator $\ ;$, the typing context of the two initial branches are summed, so we have $(d : 1\nu [T], x : 1\uparrow T) + (x : 1\nu T) = d : 1\nu [T], x : (1+1)(\nu+\uparrow) T = d : 1\nu [T], x : \omega\infty T$. 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_V on a variable with mode $1\uparrow$ — then x must be of age ∞ ¹⁴.

Alternatively, we can have:

$$\frac{\frac{1\nu \leq 1\nu}{d : 1\nu [T] \vdash d : [T]} \text{ID}_V \quad \frac{1\nu \leq \omega\infty}{x : \omega\infty T \vdash x : T} \text{ID}_V}{d : 1\nu [T], x : \omega\infty T \vdash d \blacktriangleleft x : 1} \text{E}_L \quad \frac{1\nu \leq \omega\infty}{x : \omega\infty T \vdash x : T} \text{ID}_V}{d : 1\nu [T], x : \omega\infty T \vdash d \blacktriangleleft x \ ; \ x : T} 1E$$

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 $1\nu \leq \omega\infty$, which matches the premise of rule λ_d-TY/ID_V .

2.5.2 Typing rules

Core typing rules of λ_d are given in Figure 2.2. Rules for elimination of \multimap , 1 , \oplus , \otimes and $!_m$ 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 $!_m$, are derived from elimination rules of destinations, and thus are not part of the core language. They are presented on the distinct Figure 2.3. The introduction form for functions $\lambda_d-TY_s/\multimap I$ is also derived from elimination form $\lambda_d-TY/[\multimap]E$ for the corresponding destinations of function, although functions are not purely “data”.

Rules λ_d-TY/ID_V , $\lambda_d-TY/\bowtie \text{NEW}$ and $\lambda_d-TY_s/1I$, 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 $\lambda_{Ldm-TY}/\text{ID}$ or $\lambda_{Ldm-TY}/1I$, except that Γ is scaled by $\omega\nu$ instead of ω now.

¹⁴Age 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).

(Typing judgment for terms)

$$\begin{array}{c}
\frac{1\nu \preceq m}{\omega\nu \cdot \Gamma, x : \textcolor{blue}{m}\textcolor{red}{T} \vdash x : \textcolor{blue}{T}} \text{IDv} \quad \frac{\Gamma_1 \vdash t : \textcolor{red}{T}}{\textcolor{blue}{m} \cdot \Gamma_1 + \Gamma_2 \vdash t' : \textcolor{red}{T} \textcolor{blue}{m} \multimap \textcolor{blue}{U}} \multimap\text{E} \quad \frac{\Gamma_1 \vdash t : \textcolor{blue}{1} \quad \Gamma_2 \vdash u : \textcolor{blue}{U}}{\Gamma_1 + \Gamma_2 \vdash t \textcolor{blue}{;} u : \textcolor{blue}{U}} \textcolor{blue}{1}\text{E} \\
\\
\frac{\Gamma_1 \vdash t : \textcolor{blue}{T}_1 \oplus \textcolor{blue}{T}_2 \quad \Gamma_2, x_1 : \textcolor{blue}{m}\textcolor{red}{T}_1 \vdash u_1 : \textcolor{blue}{U} \quad \Gamma_2, x_2 : \textcolor{blue}{m}\textcolor{red}{T}_2 \vdash u_2 : \textcolor{blue}{U}}{\textcolor{blue}{m} \cdot \Gamma_1 + \Gamma_2 \vdash \text{case}_{\textcolor{blue}{m}} t \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\} : \textcolor{blue}{U}} \oplus\text{E} \\
\\
\frac{\Gamma_1 \vdash t : \textcolor{blue}{T}_1 \otimes \textcolor{blue}{T}_2 \quad \Gamma_2, x_1 : \textcolor{blue}{m}\textcolor{red}{T}_1, x_2 : \textcolor{blue}{m}\textcolor{red}{T}_2 \vdash u : \textcolor{blue}{U}}{\textcolor{blue}{m} \cdot \Gamma_1 + \Gamma_2 \vdash \text{case}_{\textcolor{blue}{m}} t \text{ of } (x_1, x_2) \mapsto u : \textcolor{blue}{U}} \otimes\text{E} \quad \frac{\Gamma_1 \vdash t : \textcolor{blue}{!}_n \textcolor{red}{T} \quad \Gamma_2, x : \textcolor{blue}{m} \textcolor{red}{n}\textcolor{blue}{T} \vdash u : \textcolor{blue}{U}}{\textcolor{blue}{m} \cdot \Gamma_1 + \Gamma_2 \vdash \text{case}_{\textcolor{blue}{m}} t \text{ of } \text{Mod}_{\textcolor{blue}{n}} x \mapsto u : \textcolor{blue}{U}} \textcolor{blue}{!}\text{E} \\
\\
\frac{\Gamma_1 \vdash t : \textcolor{blue}{U} \ltimes \textcolor{red}{T} \quad \textcolor{red}{1}\uparrow \cdot \Gamma_2, x : \textcolor{blue}{1\nu}\textcolor{red}{T} \vdash t' : \textcolor{red}{T}'}{\Gamma_1 + \Gamma_2 \vdash \text{upd}_{\ltimes} t \text{ with } x \mapsto t' : \textcolor{blue}{U} \ltimes \textcolor{red}{T}'} \ltimes\text{UPD} \quad \frac{\Gamma \vdash u : \textcolor{blue}{U}}{\Gamma \vdash \text{to}_{\ltimes} u : \textcolor{blue}{U} \ltimes \textcolor{blue}{1}} \ltimes\text{TO} \\
\\
\frac{\Gamma \vdash t : \textcolor{blue}{U} \ltimes (\textcolor{blue}{!}_{\infty} \textcolor{red}{T})}{\Gamma \vdash \text{from}_{\ltimes} t : \textcolor{blue}{U} \otimes (\textcolor{blue}{!}_{\infty} \textcolor{red}{T})} \ltimes\text{FROM} \quad \frac{}{\omega\nu \cdot \Gamma \vdash \text{new}_{\ltimes} : \textcolor{red}{T} \ltimes \lfloor \textcolor{red}{T} \rfloor} \ltimes\text{NEW} \quad \frac{\Gamma \vdash t : \lfloor \textcolor{blue}{n}\textcolor{blue}{1} \rfloor}{\Gamma \vdash t \triangleleft () : \textcolor{blue}{1}} \textcolor{blue}{1}\text{E} \\
\\
\frac{\Gamma \vdash t : \lfloor \textcolor{blue}{n}\textcolor{blue}{T}_1 \oplus \textcolor{blue}{T}_2 \rfloor}{\Gamma \vdash t \triangleleft \text{Inl} : \lfloor \textcolor{blue}{n}\textcolor{blue}{T}_1 \rfloor} \lfloor \oplus \rfloor\text{E}_1 \quad \frac{\Gamma \vdash t : \lfloor \textcolor{blue}{n}\textcolor{blue}{T}_1 \oplus \textcolor{blue}{T}_2 \rfloor}{\Gamma \vdash t \triangleleft \text{Inr} : \lfloor \textcolor{blue}{n}\textcolor{blue}{T}_2 \rfloor} \lfloor \oplus \rfloor\text{E}_2 \quad \frac{\Gamma \vdash t : \lfloor \textcolor{blue}{n}\textcolor{blue}{T}_1 \otimes \textcolor{blue}{T}_2 \rfloor}{\Gamma \vdash t \triangleleft (,) : \lfloor \textcolor{blue}{n}\textcolor{blue}{T}_1 \rfloor \otimes \lfloor \textcolor{blue}{n}\textcolor{blue}{T}_2 \rfloor} \lfloor \otimes \rfloor\text{E} \\
\\
\frac{\Gamma \vdash t : \lfloor \textcolor{blue}{n}\textcolor{blue}{!}_{n'} \textcolor{red}{T} \rfloor}{\Gamma \vdash t \triangleleft \text{Mod}_{\textcolor{blue}{n}'} : \lfloor \textcolor{blue}{n}' \textcolor{blue}{n}\textcolor{blue}{T} \rfloor} \lfloor \textcolor{blue}{!} \rfloor\text{E} \quad \frac{\Gamma_1 \vdash t : \lfloor \textcolor{blue}{n}\textcolor{red}{T} \textcolor{blue}{m} \multimap \textcolor{blue}{U} \rfloor \quad \Gamma_2, x : \textcolor{blue}{m}\textcolor{red}{T} \vdash u : \textcolor{blue}{U}}{\Gamma_1 + (\textcolor{red}{1}\uparrow \cdot \textcolor{blue}{n}) \cdot \Gamma_2 \vdash t \triangleleft (\lambda x \textcolor{red}{m} \mapsto u) : \textcolor{blue}{1}} \lfloor \multimap \rfloor\text{E} \\
\\
\frac{\Gamma_1 \vdash t : \lfloor \textcolor{blue}{U} \rfloor \quad \Gamma_2 \vdash t' : \textcolor{blue}{U} \ltimes \textcolor{red}{T}}{\Gamma_1 + \textcolor{red}{1}\uparrow \cdot \Gamma_2 \vdash t \triangleleft_{\circ} t' : \textcolor{red}{T}} \lfloor \textcolor{blue}{\circ} \rfloor\text{E}_c \quad \frac{\Gamma_1 \vdash t : \lfloor \textcolor{blue}{n}\textcolor{red}{T} \rfloor \quad \Gamma_2 \vdash t' : \textcolor{red}{T}}{\Gamma_1 + (\textcolor{red}{1}\uparrow \cdot \textcolor{blue}{n}) \cdot \Gamma_2 \vdash t \blacktriangleleft t' : \textcolor{blue}{1}} \lfloor \textcolor{blue}{\circ} \rfloor\text{E}_L
\end{array}$$

Figure 2.2: Typing rules of λ_d (λ_d -TY)

$$\boxed{\Gamma \vdash t : \mathbb{T}} \quad (\text{Derived typing judgment for syntactic sugar forms})$$

$$\begin{array}{c}
 \frac{\Gamma \vdash t : \mathbb{T} \times \mathbf{1}}{\Gamma \vdash \text{from}'_{\times} t : \mathbb{T}} \times\text{FROM}' \quad \frac{}{\omega\nu \cdot \Gamma \vdash () : \mathbf{1}} \mathbf{1I} \quad \frac{\Gamma_2, x : \mathbf{m} \mathbb{T} \vdash u : \mathbb{U}}{\Gamma_2 \vdash \lambda x_{\mathbf{m}} \mapsto u : \mathbb{T}_{\mathbf{m} \circ} \mathbb{U}} \circ\text{I} \\
 \\
 \frac{\Gamma_2 \vdash t : \mathbb{T}_1}{\Gamma_2 \vdash \text{Inl } t : \mathbb{T}_1 \oplus \mathbb{T}_2} \oplus\text{I}_1 \quad \frac{\Gamma_2 \vdash t : \mathbb{T}_2}{\Gamma_2 \vdash \text{Inr } t : \mathbb{T}_1 \oplus \mathbb{T}_2} \oplus\text{I}_2 \quad \frac{\Gamma_2 \vdash t : \mathbb{T}}{\mathbf{m} \cdot \Gamma_2 \vdash \text{Mod}_{\mathbf{m}} t : !_{\mathbf{m}} \mathbb{T}} !\text{I} \\
 \\
 \frac{\Gamma_{21} \vdash t_1 : \mathbb{T}_1 \quad \Gamma_{22} \vdash t_2 : \mathbb{T}_2}{\Gamma_{21} + \Gamma_{22} \vdash (t_1, t_2) : \mathbb{T}_1 \otimes \mathbb{T}_2} \otimes\text{I}
 \end{array}$$

 Figure 2.3: Derived typing rules for syntactic sugar ($\lambda_d\text{-TY}_S$)

Rule $\lambda_d\text{-TY}/\text{ID}_V$, in addition to weakening, allows for coercion of the mode \mathbf{m} of the variable used, with ordering constraint $\mathbf{1}\nu \preceq \mathbf{m}$ as defined in Figure 2.1. Unlike in λ_{Ldm} , here not all modes \mathbf{m} are compatible with $\mathbf{1}\nu$. 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. $\preceq_{\mathbf{a}}$ when $j \neq k$. So for example we cannot use a variable with mode $\mathbf{1}\uparrow$ in most contexts.

For pattern-matching, with rules $\lambda_d\text{-TY}/\oplus\text{E}$, $\lambda_d\text{-TY}/\otimes\text{E}$ and $\lambda_d\text{-TY}/!\text{E}$, we keep the same *deep mode* approach as in λ_{Ldm} : **case** expressions are parametrized by a mode \mathbf{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 $\lambda_d\text{-TY}/\times\text{UPD}$, as destinations are only ever accessed through upd_{\times} . More precisely, $\text{upd}_{\times} t \text{ with } x \mapsto t'$ 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 numbers). That is represented by t' typing in $\mathbf{1}\uparrow \cdot \Gamma_2, x : \mathbf{1}\nu \mathbb{T}$ while the parent term $\text{upd}_{\times} t \text{ with } x \mapsto t'$ types in unscaled contexts $\Gamma_1 + \Gamma_2$. This difference of age between x — introduced by upd_{\times} , 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 2.4 illustrates scopes introduced by upd_{\times} , and how the typing rules for upd_{\times} and \blacktriangleleft interact. For the first time here we see ampar values $\langle v_2 \smallfrown v_1 \rangle$, 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 upd_{\times} 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 $\lambda_d\text{-TY}/\lfloor \rfloor\text{E}_L$, for instance $d_{11} \blacktriangleleft x_0$ in Figure 2.4, 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.

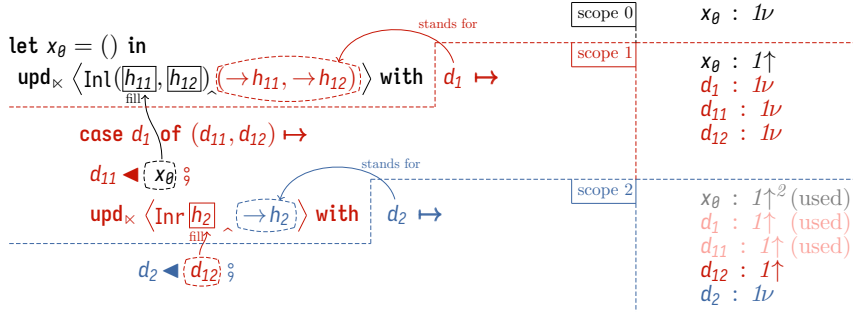


Figure 2.4: Evolution of ages through nested scopes

This is the opposite of the scaling that \mathbf{upd}_\times does: while \mathbf{upd}_\times creates a new scope for its body, operator \blacktriangleleft , and similarly, \blacktriangleleft_\circ and $\blacktriangleleft(\lambda x_m \mapsto u)$, transfer their right operand to the outer scope. In other words, the right-hand side of \blacktriangleleft or \blacktriangleleft is an enclave for the parent scope.

When using \mathbf{from}'_\times (rule $\lambda_d\text{-TY}_S/\times\text{FROM}'$), 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}'_\times is implemented in terms of \mathbf{from}_\times in Listing 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}_\times in rule $\lambda_d\text{-TY}/\times\text{FROM}$ 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 $!_{\infty}T$, 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 $\lambda_d\text{-TY}/\times\text{TO}$, on the other hand, there is no need to bother with scopes: the operator \mathbf{to}_\times 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 $\blacktriangleleft()$, $\blacktriangleleft\text{Inl}$, $\blacktriangleleft\text{Inr}$, $\blacktriangleleft\text{Mod}_m$, $\blacktriangleleft()$ from rules of the form $\lambda_d\text{-TY}/\lfloor \rfloor E$ 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 $\blacktriangleleft(::)$ and the corresponding synthetic constructor $(::)$ that we used since Section 2.2.2. Starting with $\blacktriangleleft(::)$, we have the following definition.

$$\begin{aligned} \blacktriangleleft(::) &: \lfloor_m \text{List } T \rfloor \multimap (\lfloor_m T \rfloor \otimes \lfloor_m \text{List } T \rfloor) \\ d \blacktriangleleft(::) &\triangleq d \blacktriangleleft \text{Inr} \blacktriangleleft () \end{aligned}$$

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

A-TY:

$$\begin{array}{c}
 \frac{1\nu \leq 1\nu}{d :_{1\nu} [\text{List } T] \vdash d : [\text{List } T]} \text{ID}_V \\
 \frac{d :_{1\nu} [\text{List } T] \vdash d : [\text{List } T]}{d :_{1\nu} [\text{List } T] \vdash d : [\text{List } T]} \text{unfold List } T \text{ definition on the right} \\
 \frac{d :_{1\nu} [\text{List } T] \vdash d : [\text{List } T]}{d :_{1\nu} [\text{List } T] \vdash d \triangleleft \text{Inr} : [\text{List } T \otimes (\text{List } T)]} [\oplus]E_2 \\
 \frac{d :_{1\nu} [\text{List } T] \vdash d \triangleleft \text{Inr} \triangleleft (,) : [\text{List } T] \otimes [\text{List } T]}{\bullet \vdash \lambda d \mapsto d \triangleleft \text{Inr} \triangleleft (,) : [\text{List } T] \multimap [\text{List } T]} [\otimes]E \\
 \lambda_d\text{-TY}_s / \multimap I
 \end{array}$$

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

$(::) : T \multimap \text{List } T \multimap \text{List } T$

$x :: xs \triangleq \text{from}'_{\times}(\text{upd}_{\times} \text{new}_{\times} \text{ with } d \mapsto \text{case } (d \triangleleft (::)) \text{ of } (dx, dxs) \mapsto dx \triangleleft x \ ; \ dxs \triangleleft xs)$

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

B-TY:

$$\begin{array}{c}
 \frac{1\nu \leq 1\nu}{d :_{1\nu} [\text{List } T] \vdash d : [\text{List } T]} \text{ID}_V \quad \frac{1\nu \leq 1\nu}{dx :_{1\nu} [T] \vdash dx : [T]} \text{ID}_V \quad \frac{1\nu \leq 1\nu}{x :_{1\nu} T \vdash x : T} \text{ID}_V \quad \frac{1\nu \leq 1\nu}{dxs :_{1\nu} [\text{List } T] \vdash dxs : [\text{List } T]} \text{ID}_V \quad \frac{1\nu \leq 1\nu}{xs :_{1\nu} \text{List } T \vdash xs : \text{List } T} \text{ID}_V \\
 \frac{d :_{1\nu} [\text{List } T] \vdash d : [\text{List } T]}{d :_{1\nu} [\text{List } T] \vdash d \triangleleft (:) : [\text{List } T] \otimes [\text{List } T]} \text{A-TY} \quad \frac{dx :_{1\nu} [T], \uparrow \cdot x :_{1\nu} T \vdash dx \triangleleft x : 1}{dx :_{1\nu} [T], dxs :_{1\nu} [\text{List } T], \uparrow \cdot (x :_{1\nu} T, xs :_{1\nu} \text{List } T) \vdash dx \triangleleft x \ ; \ dxs \triangleleft xs : 1} [\]E_L \quad \frac{dxs :_{1\nu} [\text{List } T], \uparrow \cdot xs :_{1\nu} \text{List } T \vdash dxs \triangleleft xs : 1}{dx :_{1\nu} [T], dxs :_{1\nu} [\text{List } T], \uparrow \cdot (x :_{1\nu} T, xs :_{1\nu} \text{List } T) \vdash dx \triangleleft x \ ; \ dxs \triangleleft xs : 1} [\]E_L \\
 d :_{1\nu} [\text{List } T], \uparrow \cdot (x :_{1\nu} T, xs :_{1\nu} \text{List } T) \vdash \text{case } (d \triangleleft (:)) \text{ of } (dx, dxs) \mapsto dx \triangleleft x \ ; \ dxs \triangleleft xs : 1 \quad \otimes E
 \end{array}$$

C-TY:

$$\begin{array}{c}
 \frac{\bullet \vdash \text{new}_{\times} : \text{List } T \times [\text{List } T]}{\bullet \vdash \text{new}_{\times} : \text{List } T \times [\text{List } T]} \times \text{NEW} \quad \frac{d :_{1\nu} [\text{List } T], \uparrow \cdot (x :_{1\nu} T, xs :_{1\nu} \text{List } T) \vdash B : 1}{d :_{1\nu} [\text{List } T], \uparrow \cdot (x :_{1\nu} T, xs :_{1\nu} \text{List } T) \vdash B : 1} \text{B-TY} \\
 \frac{x :_{1\nu} T, xs :_{1\nu} \text{List } T \vdash \text{upd}_{\times} \text{new}_{\times} \text{ with } d \mapsto \text{case } (d \triangleleft (:)) \text{ of } (dx, dxs) \mapsto dx \triangleleft x \ ; \ dxs \triangleleft xs : \text{List } T \times 1}{x :_{1\nu} T, xs :_{1\nu} \text{List } T \vdash \text{from}'_{\times}(\text{upd}_{\times} \text{new}_{\times} \text{ with } d \mapsto \text{case } (d \triangleleft (:)) \text{ of } (dx, dxs) \mapsto dx \triangleleft x \ ; \ dxs \triangleleft xs) : \text{List } T} \times \text{UPD} \\
 \frac{x :_{1\nu} T \vdash \lambda xs \mapsto \text{from}'_{\times}(\text{upd}_{\times} \text{new}_{\times} \text{ with } d \mapsto \text{case } (d \triangleleft (:)) \text{ of } (dx, dxs) \mapsto dx \triangleleft x \ ; \ dxs \triangleleft xs) : \text{List } T \multimap \text{List } T}{x :_{1\nu} T \vdash \lambda xs \mapsto \text{from}'_{\times}(\text{upd}_{\times} \text{new}_{\times} \text{ with } d \mapsto \text{case } (d \triangleleft (:)) \text{ of } (dx, dxs) \mapsto dx \triangleleft x \ ; \ dxs \triangleleft xs) : T \multimap \text{List } T \multimap \text{List } T} \lambda_d\text{-TY}_s / \multimap I \\
 \bullet \vdash \lambda x \mapsto \lambda xs \mapsto \text{from}'_{\times}(\text{upd}_{\times} \text{new}_{\times} \text{ with } d \mapsto \text{case } (d \triangleleft (:)) \text{ of } (dx, dxs) \mapsto dx \triangleleft x \ ; \ dxs \triangleleft xs) : T \multimap \text{List } T \multimap \text{List } T \quad \lambda_d\text{-TY}_s / \multimap I
 \end{array}$$

If look at the typing tree C-TY from the root to the leaves, we see that when going under upd_{\times} , variable bindings for x and xs are multiplied by \uparrow (their age is increased by one). But this is not an issue; in fact, it's required by the typing rule $\lambda_d\text{-TY} / [\]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 $E[t]$ are used to represent running programs; they're described in detail in Section 2.6.2.

Grammar extended with values:

$$\begin{aligned}
 t, u &::= \dots \mid v \\
 v &::= \boxed{h} \quad (\text{hole}) \\
 &\mid \rightarrow h \quad (\text{destination}) \\
 &\mid \mu \langle v_2 \wedge v_1 \rangle \quad (\text{ampar value}) \\
 &\mid () \mid \text{Inl } v \mid \text{Inr } v \mid \text{Mod}_m v \mid (v_1, v_2) \mid \lambda_v x_m \mapsto u
 \end{aligned}$$

Extended grammar of typing contexts:

$$\begin{aligned}
 \Delta &::= \bullet \mid \rightarrow h :_m \lfloor_n \mathbf{T} \rfloor \mid \Delta_1, \Delta_2 \\
 \Gamma &::= \bullet \mid \rightarrow h :_m \lfloor_n \mathbf{T} \rfloor \mid x :_m \mathbf{T} \mid \Gamma_1, \Gamma_2 \\
 \Theta &::= \bullet \mid \rightarrow h :_m \lfloor_n \mathbf{T} \rfloor \mid \boxed{h} :_n \mathbf{T} \mid \Theta_1, \Theta_2
 \end{aligned}$$

Operations on new typing context forms:

$$\begin{aligned}
 n' \cdot (\boxed{h} :_n \mathbf{T}, \Theta) &\triangleq (\boxed{h} :_{n' \cdot n} \mathbf{T}), n' \cdot \Theta \\
 n' \cdot (\rightarrow h :_m \lfloor_n \mathbf{T} \rfloor, \Gamma) &\triangleq (\rightarrow h :_{n' \cdot m} \lfloor_n \mathbf{T} \rfloor), n' \cdot \Gamma \quad \dagger \\
 (\boxed{h} :_n \mathbf{T}, \Theta_1) + \Theta_2 &\triangleq \boxed{h} :_n \mathbf{T}, (\Theta_1 + \Theta_2) \quad \text{if } h \notin \Theta_2 \\
 (\boxed{h} :_n \mathbf{T}, \Theta_1) + (\boxed{h} :_{n'} \mathbf{T}, \Theta_2) &\triangleq \boxed{h} :_{n+n'} \mathbf{T}, (\Theta_1 + \Theta_2) \\
 (\rightarrow h :_m \lfloor_n \mathbf{T} \rfloor, \Gamma_1) + \Gamma_2 &\triangleq \rightarrow h :_m \lfloor_n \mathbf{T} \rfloor, (\Gamma_1 + \Gamma_2) \quad \text{if } h \notin \Gamma_2 \quad \dagger \\
 (\rightarrow h :_m \lfloor_n \mathbf{T} \rfloor, \Gamma_1) + (\rightarrow h :_{m'} \lfloor_n \mathbf{T} \rfloor, \Gamma_2) &\triangleq \rightarrow h :_{m+m'} \lfloor_n \mathbf{T} \rfloor, (\Gamma_1 + \Gamma_2) \quad \dagger \\
 \rightarrow^{-1}(\bullet) &\triangleq \bullet \\
 \rightarrow^{-1}(\rightarrow h :_{1\nu} \lfloor_n \mathbf{T} \rfloor, \Delta) &\triangleq (\boxed{h} :_n \mathbf{T}), \rightarrow^{-1}(\Delta)
 \end{aligned}$$

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

Figure 2.5: 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 $\text{Inl}()$ has been, so far, just syntactic sugar for a term $\text{from}'_{\times}(\text{upd}_{\times} \text{new}_{\times} \text{with } d \mapsto \dots)$). 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 2.5. It features constructors for all of our basic types, as well as functions (note that in $\lambda_v x_m \mapsto u$, u is a term, not a value). The more interesting values are holes \boxed{h} , destinations $\rightarrow h$, and ampars $\mu \langle v_2 \wedge v_1 \rangle$, which were not present in

Typing values as terms:

$$\frac{\Delta \vdash_v v : \mathbb{T}}{\omega\nu \cdot \Gamma, \Delta \vdash v : \mathbb{T}} \lambda_d\text{-TY/FROMVAL}$$

$$\boxed{\Theta \vdash_v v : \mathbb{T}}$$

(Typing judgment for values)

$$\begin{array}{c} \frac{}{\boxed{h} :_{\mathbb{1}\nu} \mathbb{T} \vdash_v \boxed{h} : \mathbb{T}} \text{ID}_H \qquad \frac{\mathbb{1}\nu \preceq m}{\rightarrow h :_m \lfloor_n \mathbb{T} \rfloor \vdash_v \rightarrow h : \lfloor_n \mathbb{T} \rfloor} \text{ID}_D \qquad \frac{}{\cdot \vdash_v () : \mathbb{1}} \mathbb{1}I \\[10pt] \frac{\Delta, x :_m \mathbb{T} \vdash u : \mathbb{U}}{\Delta \vdash_v \lambda_{\nu} x \mapsto u : \mathbb{T}_m \multimap \mathbb{U}} \multimap I \qquad \frac{\Theta \vdash_v v_1 : \mathbb{T}_1}{\Theta \vdash_v \text{Inl } v_1 : \mathbb{T}_1 \oplus \mathbb{T}_2} \oplus I_1 \qquad \frac{\Theta \vdash_v v_2 : \mathbb{T}_2}{\Theta \vdash_v \text{Inr } v_2 : \mathbb{T}_1 \oplus \mathbb{T}_2} \oplus I_2 \\[10pt] \frac{\Theta_1 \vdash_v v_1 : \mathbb{T}_1 \quad \Theta_2 \vdash_v v_2 : \mathbb{T}_2}{\Theta_1 + \Theta_2 \vdash_v (v_1, v_2) : \mathbb{T}_1 \otimes \mathbb{T}_2} \otimes I \qquad \frac{\Theta \vdash_v v' : \mathbb{T}}{n \cdot \Theta \vdash_v \text{Mod}_n v' : \mathbb{!}_n \mathbb{T}} \mathbb{!}I \\[10pt] \frac{\mathbb{1}\uparrow \cdot \Delta_1, \Delta_3 \vdash_v v_1 : \mathbb{T} \quad \Delta_2, \rightarrow^{\mathbb{1}} \Delta_3 \vdash_v v_2 : \mathbb{U}}{\Delta_1, \Delta_2 \vdash_v \text{hnames}(\Delta_3) \langle v_2 \sim v_1 \rangle : \mathbb{U} \ltimes \mathbb{T}} \ltimes I \end{array}$$

Figure 2.6: Typing rules for runtime values ($\lambda_d\text{-TY}_V$)

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 $\lambda_d\text{-TY/FROMVAL}$ (at the top of Figure 2.6).

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* $\boxed{h} :_n \mathbb{T}$ and *destination bindings* $\rightarrow h :_m \lfloor_n \mathbb{T} \rfloor$ in typing contexts in addition to the usual variable bindings $x :_m \mathbb{T}$. 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 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 2.5. Context addition is still very partial; for instance, $(\boxed{h} :_n \mathbb{T}) + (\rightarrow h :_m \lfloor_{n'} \mathbb{T} \rfloor)$ 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 $\lambda_d\text{-TY}$ from Figure 2.2 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 2.6, and more precisely, in the left-hand side v_2 of an ampar $\textcolor{violet}{h}\langle v_2 \wedge v_1 \rangle$ in rule $\lambda_d\text{-TY}_V/\times I$.

Specifically, holes come from the operator \rightarrow^{-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 2.5. Note that $\rightarrow^{-1}\Delta$ is undefined if any destination binding in Δ has a mode other than $\textcolor{violet}{1}\nu$.

Furthermore, in $\lambda_d\text{-TY}_V/\times I$, the holes $\rightarrow^{-1}\Delta_3$ 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 $\textcolor{red}{H}$, the set of hole names that an ampar $\textcolor{violet}{h}\langle v_2 \wedge v_1 \rangle$ bounds. That's why, in rule $\lambda_d\text{-TY}_V/\times I$, the resulting ampar carries the set of hole names $\textcolor{red}{hnames}(\Delta_3)$ (we use operator $\textcolor{red}{hnames}(\cdot)$ 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 $\textcolor{violet}{1}\uparrow\cdot\Delta_1$ and Δ_2 in the respective typing contexts of v_1 and v_2 in $\lambda_d\text{-TY}_V/\times I$.

Rule $\lambda_d\text{-TY}_V/\text{ID}_H$ indicates that a hole must have mode $\textcolor{violet}{1}\nu$ 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 $\textcolor{violet}{n}$. The mode of a hole constrains which values can be written to it, e.g. in $\boxed{\textcolor{red}{h}} : \textcolor{violet}{n} \vdash \textcolor{violet}{V} \text{ Mod}_{\textcolor{violet}{n}} \boxed{\textcolor{red}{h}} : !\textcolor{violet}{n} \vdash$, only a value with mode $\textcolor{violet}{n}$ (more precisely, a value typed in a context of the form $\textcolor{violet}{n} \cdot \Theta$) can be written to $\boxed{\textcolor{red}{h}}$.

Surprisingly, in $\lambda_d\text{-TY}_V/\text{ID}_D$, we see that a destination can be typed with any mode $\textcolor{violet}{m}$ coercible to $\textcolor{violet}{1}\nu$. We did this to mimic the rule $\lambda_d\text{-TY}/\text{ID}_V$ 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, $\textcolor{violet}{m}$ 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 $\textcolor{violet}{n}$, appearing in $\lambda_d\text{-TY}_V/\text{ID}_D$, is not the mode of the destination binding; instead it is part of the type $\lfloor \textcolor{violet}{n} \vdash \rfloor$ and corresponds to the mode of values that we can write to the corresponding $\boxed{\textcolor{red}{h}}$; 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 $\lambda_d\text{-TY}/\text{FROMVAL}$, 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 ($\lambda_d\text{-TY}_V/\rightarrow I$), which, also is prevented from containing free holes. Since a

¹⁵Generally, in modal systems, if $x : \textcolor{violet}{n} \vdash$, $\Gamma \vdash u : \textcolor{violet}{U}$ and $\Delta \vdash v : \textcolor{violet}{T}$ then $\textcolor{violet}{m} \cdot \Delta, \Gamma \vdash u[x := v] : \textcolor{violet}{U}$ [Abel and Bernardy, 2020].

We have $x : \omega\infty \lfloor \textcolor{violet}{n} \vdash \rfloor \vdash () : \textcolor{violet}{1}$ and $\rightarrow\textcolor{red}{h} : \textcolor{violet}{1}\nu \lfloor \textcolor{violet}{n} \vdash \rfloor \vdash \rightarrow\textcolor{red}{h} : \lfloor \textcolor{violet}{n} \vdash \rfloor$ so $\omega\infty \cdot (\rightarrow\textcolor{red}{h} : \textcolor{violet}{1}\nu \lfloor \textcolor{violet}{n} \vdash \rfloor) \vdash ()[x := \rightarrow\textcolor{red}{h}] : \textcolor{violet}{1}$ should be valid.

$$\begin{aligned}
 e &::= t' [] \mid [] v \mid [] \mathbin{\text{\texttt{;}}} u \\
 &\mid \text{case}_{\mathfrak{m}} [] \text{ of } \{ \text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2 \} \mid \text{case}_{\mathfrak{m}} [] \text{ of } (x_1, x_2) \mapsto u \mid \text{case}_{\mathfrak{m}} [] \text{ of } \text{Mod}_{\mathfrak{n}} x \mapsto u \\
 &\mid \text{upd}_{\times} [] \text{ with } x \mapsto t' \mid \text{to}_{\times} [] \mid \text{from}_{\times} [] \mid [] \triangleleft_{\circ} t' \mid v \triangleleft_{\circ} [] \mid [] \triangleleft t' \mid v \triangleleft [] \\
 &\mid [] \triangleleft () \mid [] \triangleleft \text{Inl} \mid [] \triangleleft \text{Inr} \mid [] \triangleleft (,) \mid [] \triangleleft \text{Mod}_{\mathfrak{m}} \mid [] \triangleleft (\lambda x_{\mathfrak{m}} \mapsto u) \\
 &\mid \overset{\text{op}}{\mathfrak{H}} \langle v_2 \wedge [] \rangle \quad (\text{open ampar focusing component}) \\
 E &::= [] \mid E \circ e
 \end{aligned}$$

Figure 2.7: Grammar of evaluation contexts

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 $\lambda x \mapsto \lambda y \mapsto x \text{ concat } y$ 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 $\lambda_v x \mapsto \text{from}'_{\times} (\text{upd}_{\times} \text{ new}_{\times} \text{ with } d \mapsto d \triangleleft (\lambda y \mapsto x \text{ concat } y))$, where the inner closure is not yet in value form. As the form $d \triangleleft (\lambda y \mapsto x \text{ concat } y)$ 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.

The grammar of evaluation contexts is given in Figure 2.7. 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 $\overset{\text{op}}{\mathfrak{H}} \langle v_2 \wedge [] \rangle$ that doesn’t have a corresponding term construct. This focusing component indicates that an ampar is currently being processed by upd_{\times} , 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 upd_{\times} ’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 $\Gamma \vdash t : \mathbf{T}$ requires the bindings of Γ , judgment $\Delta \dashv E : \mathbf{T} \rightarrow \mathbf{U}_{\mathfrak{g}}$ provides the bindings of Δ . Typing rules for evaluation contexts, and the sole typing rule for commands, are given in Figure 2.8.

$\Delta \dashv E : \mathbb{T} \multimap \mathbb{U}_\theta$

(Typing judgment for evaluation contexts)

$$\begin{array}{c}
\frac{}{\bullet \dashv [] : \mathbb{U}_\theta \multimap \mathbb{U}_\theta} \text{ID} \quad \frac{\mathfrak{m} \cdot \Delta_1, \Delta_2 \dashv E : \mathbb{U} \multimap \mathbb{U}_\theta \quad \Delta_2 \vdash t' : \mathbb{T}_{\mathfrak{m}} \multimap \mathbb{U}}{\Delta_1 \dashv E \circ t' [] : \mathbb{T} \multimap \mathbb{U}_\theta} \multimap E_1 \quad \frac{\mathfrak{m} \cdot \Delta_1, \Delta_2 \dashv E : \mathbb{U} \multimap \mathbb{U}_\theta \quad \Delta_1 \vdash v : \mathbb{T}}{\Delta_2 \dashv E \circ [] v : (\mathbb{T}_{\mathfrak{m}} \multimap \mathbb{U}) \multimap \mathbb{U}_\theta} \multimap E_2 \\
\\
\frac{\Delta_1, \Delta_2 \dashv E : \mathbb{U} \multimap \mathbb{U}_\theta \quad \Delta_2 \vdash u : \mathbb{U}}{\Delta_1 \dashv E \circ [] \mathfrak{s} u : \mathbb{1} \multimap \mathbb{U}_\theta} \mathbb{1}E \quad \frac{\mathfrak{m} \cdot \Delta_1, \Delta_2 \dashv E : \mathbb{U} \multimap \mathbb{U}_\theta \quad \Delta_2, x_1 : \mathfrak{m} \mathbb{T}_1 \vdash u_1 : \mathbb{U} \quad \Delta_2, x_2 : \mathfrak{m} \mathbb{T}_2 \vdash u_2 : \mathbb{U}}{\Delta_1 \dashv E \circ \text{case}_{\mathfrak{m}} [] \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\} : (\mathbb{T}_1 \oplus \mathbb{T}_2) \multimap \mathbb{U}_\theta} \oplus E \\
\\
\frac{\mathfrak{m} \cdot \Delta_1, \Delta_2 \dashv E : \mathbb{U} \multimap \mathbb{U}_\theta \quad \Delta_2, x_1 : \mathfrak{m} \mathbb{T}_1, x_2 : \mathfrak{m} \mathbb{T}_2 \vdash u : \mathbb{U}}{\Delta_1 \dashv E \circ \text{case}_{\mathfrak{m}} [] \text{ of } (x_1, x_2) \mapsto u : (\mathbb{T}_1 \otimes \mathbb{T}_2) \multimap \mathbb{U}_\theta} \otimes E \quad \frac{\mathfrak{m} \cdot \Delta_1, \Delta_2 \dashv E : \mathbb{U} \multimap \mathbb{U}_\theta \quad \Delta_2, x : \mathfrak{m} \mathfrak{m}' \mathbb{T} \vdash u : \mathbb{U}}{\Delta_1 \dashv E \circ \text{case}_{\mathfrak{m}} [] \text{ of } \text{Mod}_{\mathfrak{m}'} x \mapsto u : \mathfrak{!}_{\mathfrak{m}'} \mathbb{T} \multimap \mathbb{U}_\theta} \mathfrak{!}E \\
\\
\frac{\Delta_1, \Delta_2 \dashv E : \mathbb{U} \times \mathbb{T}' \multimap \mathbb{U}_\theta \quad \mathfrak{1}\uparrow \cdot \Delta_2, x : \mathfrak{!}_{\mathfrak{U}} \mathbb{T} \vdash t' : \mathbb{T}'}{\Delta_1 \dashv E \circ \text{upd}_{\times} [] \text{ with } x \mapsto t' : (\mathbb{U} \times \mathbb{T}) \multimap \mathbb{U}_\theta} \times \text{UPD} \quad \frac{\Delta \dashv E : (\mathbb{U} \times \mathbb{1}) \multimap \mathbb{U}_\theta}{\Delta \dashv E \circ \text{to}_{\times} [] : \mathbb{U} \multimap \mathbb{U}_\theta} \times \text{TO} \\
\\
\frac{\Delta \dashv E : (\mathbb{U} \otimes (\mathfrak{!}_{\mathfrak{1}\infty} \mathbb{T})) \multimap \mathbb{U}_\theta}{\Delta \dashv E \circ \text{from}_{\times} [] : (\mathbb{U} \times (\mathfrak{!}_{\mathfrak{1}\infty} \mathbb{T})) \multimap \mathbb{U}_\theta} \times \text{FROM} \quad \frac{\Delta \dashv E : \mathbb{1} \multimap \mathbb{U}_\theta}{\Delta \dashv E \circ [] \triangleleft () : [\mathfrak{n} \mathbb{1}] \multimap \mathbb{U}_\theta} [\mathbb{1}]E \\
\\
\frac{\Delta \dashv E : [\mathfrak{n} \mathbb{T}_1] \multimap \mathbb{U}_\theta}{\Delta \dashv E \circ [] \triangleleft \text{Inl} : [\mathfrak{n} \mathbb{T}_1 \oplus \mathbb{T}_2] \multimap \mathbb{U}_\theta} [\oplus]E_1 \quad \frac{\Delta \dashv E : [\mathfrak{n} \mathbb{T}_2] \multimap \mathbb{U}_\theta}{\Delta \dashv E \circ [] \triangleleft \text{Inr} : [\mathfrak{n} \mathbb{T}_1 \oplus \mathbb{T}_2] \multimap \mathbb{U}_\theta} [\oplus]E_2 \\
\\
\frac{\Delta \dashv E : ([\mathfrak{n} \mathbb{T}_1] \otimes [\mathfrak{n} \mathbb{T}_2]) \multimap \mathbb{U}_\theta}{\Delta \dashv E \circ [] \triangleleft (,) : [\mathfrak{n} \mathbb{T}_1 \otimes \mathbb{T}_2] \multimap \mathbb{U}_\theta} [\otimes]E \quad \frac{\Delta \dashv E : [\mathfrak{m} \mathbb{T}] \multimap \mathbb{U}_\theta}{\Delta \dashv E \circ [] \triangleleft \text{Mod}_{\mathfrak{m}} : [\mathfrak{n} \mathfrak{!}_{\mathfrak{m}} \mathbb{T}] \multimap \mathbb{U}_\theta} [\mathfrak{!}]E \\
\\
\frac{\Delta_1, (\mathfrak{1}\uparrow \cdot \mathfrak{n}) \cdot \Delta_2 \dashv E : \mathbb{1} \multimap \mathbb{U}_\theta \quad \Delta_2, x : \mathfrak{m} \mathbb{T} \vdash u : \mathbb{U}}{\Delta_1 \dashv E \circ [] \triangleleft (\lambda x_{\mathfrak{m}} \mapsto u) : [\mathfrak{n} \mathbb{T}_{\mathfrak{m}} \multimap \mathbb{U}] \multimap \mathbb{U}_\theta} [\multimap]E \quad \frac{\Delta_1, \mathfrak{1}\uparrow \cdot \Delta_2 \dashv E : \mathbb{T} \multimap \mathbb{U}_\theta \quad \Delta_2 \vdash t' : \mathbb{U} \times \mathbb{T}}{\Delta_1 \dashv E \circ [] \triangleleft \circ t' : [\mathbb{U}] \multimap \mathbb{U}_\theta} [\circ]E_{c1} \\
\\
\frac{\Delta_1, \mathfrak{1}\uparrow \cdot \Delta_2 \dashv E : \mathbb{T} \multimap \mathbb{U}_\theta \quad \Delta_1 \vdash v : [\mathbb{U}]}{\Delta_2 \dashv E \circ v \triangleleft \circ [] : \mathbb{U} \times \mathbb{T} \multimap \mathbb{U}_\theta} [\circ]E_{c2} \quad \frac{\Delta_1, (\mathfrak{1}\uparrow \cdot \mathfrak{n}) \cdot \Delta_2 \dashv E : \mathbb{1} \multimap \mathbb{U}_\theta \quad \Delta_2 \vdash t' : \mathbb{T}}{\Delta_1 \dashv E \circ [] \blacktriangleleft t' : [\mathfrak{n} \mathbb{T}] \multimap \mathbb{U}_\theta} [\blacktriangleleft]E_{L1} \\
\\
\frac{\Delta_1, (\mathfrak{1}\uparrow \cdot \mathfrak{n}) \cdot \Delta_2 \dashv E : \mathbb{1} \multimap \mathbb{U}_\theta \quad \Delta_1 \vdash v : [\mathfrak{n} \mathbb{T}]}{\Delta_2 \dashv E \circ v \blacktriangleleft [] : \mathbb{T} \multimap \mathbb{U}_\theta} [\blacktriangleleft]E_{L2} \quad \frac{\text{hnames}(E) \# \# \text{hnames}(\Delta_3) \quad \Delta_1, \Delta_2 \dashv E : (\mathbb{U} \times \mathbb{T}') \multimap \mathbb{U}_\theta \quad \Delta_2, \rightarrow^{-1} \Delta_3 \vdash v_2 : \mathbb{U}}{\mathfrak{1}\uparrow \cdot \Delta_1, \Delta_3 \dashv E \circ \overset{\text{op}}{\text{hnames}}(\Delta_3) \langle v_2 \sim [] \rangle : \mathbb{T}' \multimap \mathbb{U}_\theta} \times \text{OP}
\end{array}$$

 $\vdash E[t] : \mathbb{T}$

(Typing judgment for commands)

$$\frac{\Delta \dashv E : \mathbb{T} \multimap \mathbb{U}_\theta \quad \Delta \vdash t : \mathbb{T}}{\vdash E[t] : \mathbb{U}_\theta} \lambda_d\text{-TY}_{\text{CMD}}$$

Figure 2.8: Typing rules of evaluation contexts ($\lambda_d\text{-TY}_E$) and commands ($\lambda_d\text{-TY}_{\text{CMD}}$)

An evaluation context has a context type $\mathbf{T} \multimap \mathbf{U}_0$. The meaning of $E : \mathbf{T} \multimap \mathbf{U}_0$ is that given $t : \mathbf{T}$, $E[t]$ returns a value of type \mathbf{U}_0 . Composing an evaluation context $E : \mathbf{T} \multimap \mathbf{U}_0$ with a new focusing component never affects the type \mathbf{U}_0 of the future command; only the type \mathbf{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 $\lambda_d\text{-TYE}/\ltimes\text{OP}$, as there is no direct term equivalent to the *open ampar* focusing component). Let's take the rule $\lambda_d\text{-TYE}/\otimes\text{E}$ as an example:

$$\lambda_d\text{-TYE}/\otimes\text{E} \quad \left| \quad \lambda_d\text{-TY}/\otimes\text{E} \right.$$

$$\frac{\begin{array}{c} \mathbf{m} \cdot \Delta_1, \Delta_2 \vdash E : \mathbf{U} \multimap \mathbf{U}_0 \\ \Delta_2, x_1 : \mathbf{m} \mathbf{T}_1, x_2 : \mathbf{m} \mathbf{T}_2 \vdash u : \mathbf{U} \end{array}}{\Delta_1 \vdash E \circ \text{case}_{\mathbf{m}} [] \text{ of } (x_1, x_2) \mapsto u : (\mathbf{T}_1 \otimes \mathbf{T}_2) \multimap \mathbf{U}_0} \quad \left| \quad \frac{\begin{array}{c} \Gamma_1 \vdash t : \mathbf{T}_1 \otimes \mathbf{T}_2 \\ \Gamma_2, x_1 : \mathbf{m} \mathbf{T}_1, x_2 : \mathbf{m} \mathbf{T}_2 \vdash u : \mathbf{U} \end{array}}{\mathbf{m} \cdot \Gamma_1 + \Gamma_2 \vdash \text{case}_{\mathbf{m}} t \text{ of } (x_1, x_2) \mapsto u : \mathbf{U}} \right.$$

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

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 2.9 and 2.10. 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 2.9 presents all the rules that don't incur any substitution on the evaluation context.

Reduction rules for function application and pattern-matching are the same as in λ_{Ldm} . Reduction rules for **to**_κ 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 **from**_κ are fairly symmetric: once we have an ampar with a value of the shape $\text{Mod}_{1\infty} v_1$ on the right, we can extract both the left and right side out of the ampar shell, into a normal pair.

Finally, **new**_κ has only a single rule that transforms it into the “identity” ampar object with just one hole on the left, and the corresponding destination on the right.

Rules of Figure 2.10 are all related to **upd**_κ and destination-filling operators, whose contraction rules modify the evaluation context deeply, instead of just pushing or popping a focusing component. For that, we need to introduce the special substitution $E(\mathbf{h} :=_{\mathbf{h}} v)$ that is used to update structures under construction, that are attached to open ampar focusing components in the stack. Such a substitution is triggered when a destination $\rightarrow \mathbf{h}$ is filled in the term under focus,

2.6. Operational semantics

$E[t] \longrightarrow E'[t']$	(Small-step evaluation of commands)
$E[t' t] \longrightarrow (E \circ t' []) [t]$	$\star \dashv\!\!\!\rightarrow \text{EF}_1$
$(E \circ t' []) [v] \longrightarrow E[t' v]$	$\dashv\!\!\!\rightarrow \text{EU}_1$
$E[t' v] \longrightarrow (E \circ [] v) [t']$	$\star \dashv\!\!\!\rightarrow \text{EF}_2$
$(E \circ [] v) [v'] \longrightarrow E[v' v]$	$\dashv\!\!\!\rightarrow \text{EU}_2$
$E[(\lambda_{\mathbf{x}} \mathbf{x}_{\mathbf{m}} \mapsto u) v] \longrightarrow E[u[\mathbf{x} := v]]$	$\dashv\!\!\!\rightarrow \text{EC}$
$E[t \mathbin{\text{\textcolor{blue}{;}}} u] \longrightarrow (E \circ [] \mathbin{\text{\textcolor{blue}{;}}} u) [t]$	$\star \text{1EF}$
$(E \circ [] \mathbin{\text{\textcolor{blue}{;}}} u) [v] \longrightarrow E[v \mathbin{\text{\textcolor{blue}{;}}} u]$	1EU
$E[() \mathbin{\text{\textcolor{blue}{;}}} u] \longrightarrow E[u]$	1EC
$E[\text{case}_{\mathbf{m}} t \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\}] \longrightarrow (E \circ \text{case}_{\mathbf{m}} [] \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\}) [t]$	$\star \oplus \text{EF}$
$(E \circ \text{case}_{\mathbf{m}} [] \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\}) [v] \longrightarrow E[\text{case}_{\mathbf{m}} v \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\}]$	$\oplus \text{EU}$
$E[\text{case}_{\mathbf{m}} (\text{Inl } v_1) \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\}] \longrightarrow E[u_1[x_1 := v_1]]$	$\oplus \text{EC}_1$
$E[\text{case}_{\mathbf{m}} (\text{Inr } v_2) \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\}] \longrightarrow E[u_2[x_2 := v_2]]$	$\oplus \text{EC}_2$
$E[\text{case}_{\mathbf{m}} t \text{ of } (x_1, x_2) \mapsto u] \longrightarrow (E \circ \text{case}_{\mathbf{m}} [] \text{ of } (x_1, x_2) \mapsto u) [t]$	$\star \otimes \text{EF}$
$(E \circ \text{case}_{\mathbf{m}} [] \text{ of } (x_1, x_2) \mapsto u) [v] \longrightarrow E[\text{case}_{\mathbf{m}} v \text{ of } (x_1, x_2) \mapsto u]$	$\otimes \text{EU}$
$E[\text{case}_{\mathbf{m}} (v_1, v_2) \text{ of } (x_1, x_2) \mapsto u] \longrightarrow E[u[x_1 := v_1][x_2 := v_2]]$	$\otimes \text{EC}$
$E[\text{case}_{\mathbf{m}} t \text{ of } \text{Mod}_{\mathbf{n}} x \mapsto u] \longrightarrow (E \circ \text{case}_{\mathbf{m}} [] \text{ of } \text{Mod}_{\mathbf{n}} x \mapsto u) [t]$	$\star !\text{EF}$
$(E \circ \text{case}_{\mathbf{m}} [] \text{ of } \text{Mod}_{\mathbf{n}} x \mapsto u) [v] \longrightarrow E[\text{case}_{\mathbf{m}} v \text{ of } \text{Mod}_{\mathbf{n}} x \mapsto u]$	$!\text{EU}$
$E[\text{case}_{\mathbf{m}} \text{Mod}_{\mathbf{n}} v' \text{ of } \text{Mod}_{\mathbf{n}} x \mapsto u] \longrightarrow E[u[x := v']]$	$!\text{EC}$
$E[\text{to}_{\times} u] \longrightarrow (E \circ \text{to}_{\times} []) [u]$	$\star \ltimes \text{TOF}$
$(E \circ \text{to}_{\times} []) [v_2] \longrightarrow E[\text{to}_{\times} v_2]$	$\ltimes \text{TOU}$
$E[\text{to}_{\times} v_2] \longrightarrow E[\{\} \langle v_2 \wedge () \rangle]$	$\ltimes \text{TOC}$
$E[\text{from}_{\times} t] \longrightarrow (E \circ \text{from}_{\times} []) [t]$	$\star \ltimes \text{FROMF}$
$(E \circ \text{from}_{\times} []) [v] \longrightarrow E[\text{from}_{\times} v]$	$\ltimes \text{FROMU}$
$E[\text{from}_{\times} \{\} \langle v_2 \wedge \text{Mod}_{1\infty} v_1 \rangle] \longrightarrow E[(v_2, \text{Mod}_{1\infty} v_1)]$	$\ltimes \text{FROMC}$
$E[\text{new}_{\times}] \longrightarrow E[\{\} \langle \boxed{1} \wedge \rightarrow 1 \rangle]$	$\ltimes \text{NEWC}$

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

Name set shift and conditional name shift:

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

Special substitution for open ampars:

$$\begin{aligned}
 (E \circ \overset{\text{op}}{\{\} \sqcup_H} \langle v_2 \wedge [] \rangle) (\langle h :=_{H'} v' \rangle) &= E \circ \overset{\text{op}}{H \sqcup_{H'}} \langle v_2 (\langle h :=_{H'} v' \rangle) \wedge [] \rangle \\
 (E \circ e) (\langle h :=_{H'} v' \rangle) &= E (\langle h :=_{H'} v' \rangle) \circ e \quad \text{if } h \notin e
 \end{aligned}$$

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

$E [\text{upd}_{\times} t \text{ with } x \mapsto t'] \longrightarrow (E \circ \text{upd}_{\times} [] \text{ with } x \mapsto t') [t]$	\star	$\times \text{UPDF}$
$(E \circ \text{upd}_{\times} [] \text{ with } x \mapsto t') [v] \longrightarrow E [\text{upd}_{\times} v \text{ with } x \mapsto t']$		$\times \text{UPDU}$
$E [\text{upd}_{\times} \langle v_2 \wedge v_1 \rangle \text{ with } x \mapsto t'] \longrightarrow (E \circ \overset{\text{op}}{H \equiv h'''} \langle v_2 [H \equiv h'''] \wedge [] \rangle) [t' [x := v_1 [H \equiv h''']]]$	\star	$\times \text{OP}$
$(E \circ \overset{\text{op}}{H} \langle v_2 \wedge [] \rangle) [v_1] \longrightarrow E [{}_{\text{H}} \langle v_2 \wedge v_1 \rangle]$		$\times \text{CL}$
$E [t \triangleleft ()] \longrightarrow (E \circ [] \triangleleft ()) [t]$	\star	$[1] \text{EF}$
$(E \circ [] \triangleleft ()) [v] \longrightarrow E [v \triangleleft ()]$		$[1] \text{EU}$
$E [\rightarrow h \triangleleft ()] \longrightarrow E (\langle h := \{\} \rangle ()) [()]$		$[1] \text{EC}$
$E [t \triangleleft \text{Inl}] \longrightarrow (E \circ [] \triangleleft \text{Inl}) [t]$	\star	$[\oplus] \text{E}_1 \text{F}$
$(E \circ [] \triangleleft \text{Inl}) [v] \longrightarrow E [v \triangleleft \text{Inl}]$		$[\oplus] \text{E}_1 \text{U}$
$E [\rightarrow h \triangleleft \text{Inl}] \longrightarrow E (\langle h := \{h'+1\} \text{Inl} [\boxed{h'+1}] \rangle) [\rightarrow h'+1]$	\star	$[\oplus] \text{E}_1 \text{C}$
$E [t \triangleleft \text{Inr}] \longrightarrow (E \circ [] \triangleleft \text{Inr}) [t]$	\star	$[\oplus] \text{E}_2 \text{F}$
$(E \circ [] \triangleleft \text{Inr}) [v] \longrightarrow E [v \triangleleft \text{Inr}]$		$[\oplus] \text{E}_2 \text{U}$
$E [\rightarrow h \triangleleft \text{Inr}] \longrightarrow E (\langle h := \{h'+1\} \text{Inr} [\boxed{h'+1}] \rangle) [\rightarrow h'+1]$	\star	$[\oplus] \text{E}_2 \text{C}$
$E [t \triangleleft \text{Mod}_m] \longrightarrow (E \circ [] \triangleleft \text{Mod}_m) [t]$	\star	$[!] \text{EF}$
$(E \circ [] \triangleleft \text{Mod}_m) [v] \longrightarrow E [v \triangleleft \text{Mod}_m]$		$[!] \text{EU}$
$E [\rightarrow h \triangleleft \text{Mod}_m] \longrightarrow E (\langle h := \{h'+1\} \text{Mod}_m [\boxed{h'+1}] \rangle) [\rightarrow h'+1]$	\star	$[!] \text{EC}$
$E [t \triangleleft (,)] \longrightarrow (E \circ [] \triangleleft (,)) [t]$	\star	$[\otimes] \text{EF}$
$(E \circ [] \triangleleft (,)) [v] \longrightarrow E [v \triangleleft (,)]$		$[\otimes] \text{EU}$
$E [\rightarrow h \triangleleft (,)] \longrightarrow E (\langle h := \{h'+1, h'+2\} (\boxed{h'+1}, \boxed{h'+2}) \rangle) [(\rightarrow h'+1, \rightarrow h'+2)]$	\star	$[\otimes] \text{EC}$
$E [t \triangleleft (\lambda x \mapsto u)] \longrightarrow (E \circ [] \triangleleft (\lambda x \mapsto u)) [t]$	\star	$[\multimap] \text{EF}$
$(E \circ [] \triangleleft (\lambda x \mapsto u)) [v] \longrightarrow E [v \triangleleft (\lambda x \mapsto u)]$		$[\multimap] \text{EU}$
$E [\rightarrow h \triangleleft (\lambda x \mapsto u)] \longrightarrow E (\langle h := \{\} \lambda x \mapsto u \rangle) [()]$		$[\multimap] \text{EC}$
$E [t \triangleleft \circ t'] \longrightarrow (E \circ [] \triangleleft \circ t') [t]$	\star	$[\] \text{E}_c \text{F}_1$
$(E \circ [] \triangleleft \circ t') [v] \longrightarrow E [v \triangleleft \circ t']$		$[\] \text{E}_c \text{U}_1$
$E [v \triangleleft \circ t'] \longrightarrow (E \circ v \triangleleft \circ []) [t']$	\star	$[\] \text{E}_c \text{F}_2$
$(E \circ v \triangleleft \circ []) [v'] \longrightarrow E [v \triangleleft \circ v']$		$[\] \text{E}_c \text{U}_2$
$E [\rightarrow h \triangleleft \circ \langle v_2 \wedge v_1 \rangle] \longrightarrow E (\langle h := \{H \equiv h''\} v_2 [H \equiv h''] \rangle) [v_1 [H \equiv h'']]$	\star	$[\] \text{E}_c \text{C}$
$E [t \triangleleft \blacktriangleleft t'] \longrightarrow (E \circ [] \triangleleft \blacktriangleleft t') [t]$	\star	$[\] \text{E}_L \text{F}_1$
$(E \circ [] \triangleleft \blacktriangleleft t') [v] \longrightarrow E [v \triangleleft \blacktriangleleft t']$		$[\] \text{E}_L \text{U}_1$
$E [v \triangleleft \blacktriangleleft t'] \longrightarrow (E \circ v \triangleleft \blacktriangleleft []) [t']$	\star	$[\] \text{E}_L \text{F}_2$
$(E \circ v \triangleleft \blacktriangleleft []) [v'] \longrightarrow E [v \triangleleft \blacktriangleleft v']$		$[\] \text{E}_L \text{U}_2$
$E [\rightarrow h \triangleleft \blacktriangleleft v] \longrightarrow E (\langle h := \{\} v \rangle) [()]$		$[\] \text{E}_L \text{C}$

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

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

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

typically in destination-filling primitives reductions, and results in the value v being written to hole $[h]$. The value v may contain holes itself (e.g. when the hollow constructor $\text{Inl}[h'+1]$ is being written to the hole $[h]$ in $\lambda_d\text{-SEM}/[\oplus]E_1C$), 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 2.9.

$\lambda_d\text{-SEM}/[1]EC$ and $\lambda_d\text{-SEM}/[\neg]EC$ of Figure 2.10 do not create any new hole; they only write a value to an existing one. On the other hand, rules $\lambda_d\text{-SEM}/[\oplus]E_1C$, $\lambda_d\text{-SEM}/[\oplus]E_2C$, $\lambda_d\text{-SEM}/[!]EC$ and $\lambda_d\text{-SEM}/[\otimes]EC$ all write a hollow constructor to the hole $[h]$ 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 new_\times is allowed to reduce to the same $\{1\}\langle 1 \wedge \rightarrow 1 \rangle$ every time), but as soon as an ampar is open (when it becomes an open ampar focusing component), it should have globally unique hole names. This restriction is enforced in rule $\lambda_d\text{-TY}_E/\ltimes OP$ by premise $\text{hnames}(E) \# \# \text{hnames}(\Delta_3)$, requiring hole name sets from E and Δ_3 to be disjoint when an open ampar focusing component is created during reduction of upd_\times . 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 $h' = \max(\text{hnames}(E) \cup \{h\}) + 1$ or similar definitions for h'' and h''' (see at the bottom of Figure 2.10) to find a new unused name¹⁶. 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 2.9 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 $h'\langle v_2 \wedge v_1 \rangle$ binds the names in H').

Rules $\lambda_d\text{-SEM}/\ltimes OP$ and $\lambda_d\text{-SEM}/\ltimes CL$ 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 $\lambda_d\text{-SEM}/\ltimes OP$, the hole names bound by the ampar gets renamed to fresh ones, and the left-hand side gets attached to the focusing component $\text{op}_{H \pm h''}\langle v_2[H \pm h''] \wedge [] \rangle$ while the right-hand side (containing destinations) is substituted in the body of the upd_\times statement (which becomes the new term under focus). The rule $\lambda_d\text{-SEM}/\ltimes CL$ triggers when the body of a upd_\times 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 $\lambda_d\text{-SEM}/[\neg]E_1C$, we write the left-hand side v_2 of an ampar value $h\langle v_2 \wedge v_1 \rangle$ to a hole $[h]$ 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 upd_\times . This renaming is carried out by the conditional shift $v_2[H \pm h'']$ and $v_1[H \pm h''']$.

¹⁶Sometimes 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

Putting into practice Let's reuse the example from Section 2.5.2 with the synthetic $(::)$ cons constructor, and see the reduction of expression $() :: \text{Inl}()$. Here, $\text{Inl}()$ 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 2.5, not the syntactic constructor defined in Listing 2.4. Finally, we consider from'_{\times} to be a primitive (for brevity) with its contraction rule $\lambda_d\text{-SEM}/\times\text{FROM}'\text{C}$: $E[\text{from}'_{\times}(\{v_2 \wedge ()\})] \rightarrow E[v_2]$ and trivial focus and unfocus rules.

$$\begin{aligned}
 & [][() :: \text{Inl}()] \\
 \rightarrow & [][\text{from}'_{\times}(\text{upd}_{\times} \text{new}_{\times} \text{ with } d \mapsto \text{case } (d \triangleleft \text{Inr} \triangleleft ()) \text{ of } (dx, dxs) \mapsto dx \triangleleft () \ ; \ dxs \triangleleft \text{Inl}())] && \text{expanding } (::) \text{ and } \triangleleft (::) \text{ definitions} \\
 \rightarrow & ([() \circ \text{from}'_{\times} [])[\text{upd}_{\times} \text{new}_{\times} \text{ with } d \mapsto \text{case } (d \triangleleft \text{Inr} \triangleleft ()) \text{ of } (dx, dxs) \mapsto dx \triangleleft () \ ; \ dxs \triangleleft \text{Inl}())] && \times\text{FROM}'\text{F} \\
 \rightarrow & ([() \circ \text{from}'_{\times} [] \circ \text{upd}_{\times} [] \text{ with } d \mapsto \text{case } (d \triangleleft \text{Inr} \triangleleft ()) \text{ of } (dx, dxs) \mapsto dx \triangleleft () \ ; \ dxs \triangleleft \text{Inl}())[\text{new}_{\times}] && \times\text{UPDF} \\
 \rightarrow & ([() \circ \text{from}'_{\times} [] \circ \text{upd}_{\times} [] \text{ with } d \mapsto \text{case } (d \triangleleft \text{Inr} \triangleleft ()) \text{ of } (dx, dxs) \mapsto dx \triangleleft () \ ; \ dxs \triangleleft \text{Inl}())[\{1\}(\overline{1} \rightarrow 1)] && \times\text{NEW}'\text{C} \\
 \rightarrow & ([() \circ \text{from}'_{\times} [])[\text{upd}_{\times} \{1\}(\overline{1} \rightarrow 1) \text{ with } d \mapsto \text{case } (d \triangleleft \text{Inr} \triangleleft ()) \text{ of } (dx, dxs) \mapsto dx \triangleleft () \ ; \ dxs \triangleleft \text{Inl}())] && \times\text{UPDU} \\
 \rightarrow & ([() \circ \text{from}'_{\times} [] \circ \text{op}_{\{2\}}(\overline{2} \wedge []))[\text{case } (\rightarrow 2 \triangleleft \text{Inr} \triangleleft ()) \text{ of } (dx, dxs) \mapsto dx \triangleleft () \ ; \ dxs \triangleleft \text{Inl}())] && \times\text{OP} \\
 \rightarrow & ([() \circ \text{from}'_{\times} [] \circ \text{op}_{\{2\}}(\overline{2} \wedge []) \circ \text{case}_{\text{Inr}} [] \text{ of } (dx, dxs) \mapsto dx \triangleleft () \ ; \ dxs \triangleleft \text{Inl}())[\rightarrow 2 \triangleleft \text{Inr} \triangleleft ()] && \otimes\text{EF} \\
 \rightarrow & ([() \circ \text{from}'_{\times} [] \circ \text{op}_{\{2\}}(\overline{2} \wedge []) \circ \text{case}_{\text{Inr}} [] \text{ of } (dx, dxs) \mapsto dx \triangleleft () \ ; \ dxs \triangleleft \text{Inl}() \circ [] \triangleleft ())[\rightarrow 2 \triangleleft \text{Inr}] && [\otimes]\text{EF} \\
 \rightarrow & ([() \circ \text{from}'_{\times} [] \circ \text{op}_{\{2\}}(\text{Inr } \overline{4} \wedge []) \circ \text{case}_{\text{Inr}} [] \text{ of } (dx, dxs) \mapsto dx \triangleleft () \ ; \ dxs \triangleleft \text{Inl}() \circ [] \triangleleft ())[\rightarrow 4] && [\oplus]\text{E}_2\text{C with sub. } E(\overline{2} := \{4\} \text{ Inr } \overline{4}) \\
 \rightarrow & ([() \circ \text{from}'_{\times} [] \circ \text{op}_{\{4\}}(\text{Inr } \overline{4} \wedge []) \circ \text{case}_{\text{Inr}} [] \text{ of } (dx, dxs) \mapsto dx \triangleleft () \ ; \ dxs \triangleleft \text{Inl}())[\rightarrow 4 \triangleleft ()] && [\otimes]\text{EU} \\
 \rightarrow & ([() \circ \text{from}'_{\times} [] \circ \text{op}_{\{6,7\}}(\text{Inr } (\overline{6}, \overline{7}) \wedge []) \circ \text{case}_{\text{Inr}} [] \text{ of } (dx, dxs) \mapsto dx \triangleleft () \ ; \ dxs \triangleleft \text{Inl}())[(\rightarrow 6, \rightarrow 7)] && [\otimes]\text{EC with sub. } E(\overline{4} := \{6,7\} (\overline{6}, \overline{7})) \\
 \rightarrow & ([() \circ \text{from}'_{\times} [] \circ \text{op}_{\{6,7\}}(\text{Inr } (\overline{6}, \overline{7}) \wedge []))[\text{case } (\rightarrow 6, \rightarrow 7) \text{ of } (dx, dxs) \mapsto dx \triangleleft () \ ; \ dxs \triangleleft \text{Inl}())] && \otimes\text{EU} \\
 \rightarrow & ([() \circ \text{from}'_{\times} [] \circ \text{op}_{\{6,7\}}(\text{Inr } (\overline{6}, \overline{7}) \wedge []))[\rightarrow 6 \triangleleft () \ ; \ \rightarrow 7 \triangleleft \text{Inl} ()] && \otimes\text{EC} \\
 \rightarrow & ([() \circ \text{from}'_{\times} [] \circ \text{op}_{\{6,7\}}(\text{Inr } (\overline{6}, \overline{7}) \wedge []) \circ [] \ ; \ \rightarrow 7 \triangleleft \text{Inl}())[\rightarrow 6 \triangleleft ()] && 1\text{EF} \\
 \rightarrow & ([() \circ \text{from}'_{\times} [] \circ \text{op}_{\{7\}}(\text{Inr } (), \overline{7}) \wedge []) \circ [] \ ; \ \rightarrow 7 \triangleleft \text{Inl}())[()] && [\]\text{E}_1\text{C with sub. } E(\overline{6} := \{ \} \ ()) \\
 \rightarrow & ([() \circ \text{from}'_{\times} [] \circ \text{op}_{\{7\}}(\text{Inr } (), \overline{7}) \wedge [])[()] \ ; \ \rightarrow 7 \triangleleft \text{Inl} ()] && 1\text{EU} \\
 \rightarrow & ([() \circ \text{from}'_{\times} [] \circ \text{op}_{\{7\}}(\text{Inr } (), \overline{7}) \wedge [])[\rightarrow 7 \triangleleft \text{Inl} ()] && 1\text{EC} \\
 \rightarrow & ([() \circ \text{from}'_{\times} [] \circ \text{op}_{\{ \}}(\text{Inr } (), \text{Inl}()) \wedge [])[()] && [\]\text{E}_1\text{C with sub. } E(\overline{7} := \{ \} \text{ Inl}()) \\
 \rightarrow & ([() \circ \text{from}'_{\times} [])[\{ \}(\text{Inr } (), \text{Inl}()) \wedge ()] && \times\text{CL} \\
 \rightarrow & [][\text{from}'_{\times}(\{ \}(\text{Inr } (), \text{Inl}()) \wedge ())] && \times\text{FROM}'\text{U} \\
 \rightarrow & [][\text{Inr} (), \text{Inl} ()] && \times\text{FROM}'\text{C}
 \end{aligned}$$

At the end of the reduction, we get the empty evaluation context with value $\text{Inr}()$, $\text{Inl}()$ 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 $E(h :=_H v)$ 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 $\vdash E[t] : \mathbf{T}$ and $E[t] \rightarrow E'[t']$ then $\vdash E'[t'] : \mathbf{T}$.*

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

As seen in the example above, a command of the form $[][v]$ cannot be reduced further, as it only contains a fully determined value, and no pending computation. This 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 [Sewell et al., 2007], 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 [Minamide, 1998] introduces two primitives, `happ` and `hcomp`, as well as the hole abstraction form, $\hat{\lambda}x \mapsto t$, 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 $\hat{\lambda}$) as far as possible, until it is almost a value except for the single occurrence 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 $\hat{\lambda}x \mapsto t$ to $\{h\}\langle t[x := \boxed{h}] \wedge \rightarrow h \rangle$. 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, $(T, S)\text{hfun}$, is translated as $S \ltimes [T]$.

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 `new×`;
- opening it with `upd×`;
- building the data constructor in which the hole variable appear by a chain of fill expressions $d \triangleleft \dots$ or $d \blacktriangleleft \dots$ (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 `upd×`, 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 `upd×`.

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:

2.9. Implementation of λ_d using in-place memory mutations

$\begin{aligned} \text{ha} &: (\mathbf{T}, \text{List } \mathbf{T})\text{hfun} \\ \text{ha} &\triangleq \hat{\lambda}x \mapsto \emptyset :: x :: (f\ 2) \end{aligned}$	$\begin{aligned} \text{ha}_{\lambda_d} &: (\text{List } \mathbf{T}) \ltimes [\mathbf{T}] \\ \text{ha}_{\lambda_d} &\triangleq \text{upd}_{\ltimes} \text{new}_{\ltimes} \text{ with } d \mapsto \text{case} \\ &\quad (d \triangleleft (::)) \text{ of } (d_0, dt) \mapsto \\ &\quad \text{case } (dt \triangleleft (::)) \text{ of } (d_1, dt') \mapsto \\ &\quad \quad d_0 \triangleleft \emptyset \ ; \ dt' \triangleleft (f\ 2) \ ; \ d_1 \end{aligned}$
--	--

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

Because destinations aren't part of [Minamide, 1998], 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:

$$\begin{aligned} \text{happ} &: (\mathbf{T}, \mathbf{S})\text{hfun} \multimap \mathbf{T} \multimap \mathbf{S} \\ \text{happ}_{\lambda_d} &: \mathbf{S} \ltimes [\mathbf{T}] \multimap \mathbf{T} \multimap \mathbf{S} \\ \text{happ}_{\lambda_d} \text{ ampar } x &\triangleq \text{from}'_{\ltimes} (\text{upd}_{\ltimes} \text{ ampar with } d \mapsto d \triangleleft x) \\ \text{hcomp} &: (\mathbf{T}, \mathbf{S})\text{hfun} \multimap (\mathbf{U}, \mathbf{T})\text{hfun} \multimap (\mathbf{U}, \mathbf{S})\text{hfun} \\ \text{hcomp}_{\lambda_d} &: \mathbf{S} \ltimes [\mathbf{T}] \multimap \mathbf{T} \ltimes [\mathbf{U}] \multimap \mathbf{S} \ltimes [\mathbf{U}] \\ \text{hcomp}_{\lambda_d} \text{ ampar}_1 \text{ ampar}_2 &\triangleq \text{upd}_{\ltimes} \text{ ampar}_1 \text{ with } d \mapsto d \triangleleft \circ \text{ ampar}_2 \end{aligned}$$

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

The Functional Essence of Imperative Binary Search Trees The portion of the work from Lorenzen, Leijen, Swierstra, and Lindley [2024] and Leijen and Lorenzen [2025] about *first-class constructor contexts* is fairly similar, in both expressiveness and presentation, to Minamide's *hole abstractions*. A hole abstraction $\hat{\lambda}x \mapsto t$ is written **ctx** $t[x := _]$, whose body is the same as t but with an underscore $_$ (or a square \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 [Lorenzen, Leijen, Swierstra, and Lindley, 2024; Leijen and Lorenzen, 2025] system to ours is the same as for Minamide's system.

For types, [Lorenzen, Leijen, Swierstra, and Lindley, 2024; Leijen and Lorenzen, 2025] only define **ctx** $\langle \mathbf{T} \rangle$, where the hole and the resulting structure (once the hole will be filled) are of the same type \mathbf{T} . This translates to $\mathbf{T} \ltimes [\mathbf{T}]$ 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. $\text{Mod}_{\omega\omega} \{h\} \langle \emptyset :: \boxed{h} \wedge \rightarrow h \rangle$ (representing a difference list $0 :: \square$ that can be used non-linearly), and then used twice, each time in a different way:

```

case ModωV {h} (θ :: h  $\wedge$   $\rightarrow$  h) of ModωV x  $\mapsto$ 
  let x1 := x append 1 in
  let x2 := x append 2 in
  toList (x1 concat x2)

```

$\longrightarrow^* \quad \emptyset :: 1 :: \emptyset :: 2 :: []$

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}_\times , 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 \mathbf{upd}_\times on it (recall that \mathbf{append} is implemented in terms of \mathbf{upd}_\times), 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 [Lorenzen, Leijen, and Swierstra, 2023], 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 [Leijen and Lorenzen, 2025]. 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 [Spiwack et al., 2022; Spiwack, 2023b], and introduce a new built-in type **Token**, together with the primitive **dup**, **drop**, and **withToken**. We also switch \mathbf{new}_\times for $\mathbf{new}_{\times\text{IP}}$:

```

dup : Token  $\multimap$  Token  $\otimes$  Token
drop : Token  $\multimap$  1
withToken : (Token  $\multimap$  !ω∞ T)  $\multimap$  !ω∞ T
new×IP : Token  $\multimap$  T  $\times$  [T]

```

Ampars produced by $\mathbf{new}_{\times\text{IP}}$ have a linear dependency on a **Token**. If the same ampar, originally created with $\mathbf{new}_{\times\text{IP}}$ tok , were to be used twice in a block t , then t would require a typing context $\{tok : \omega_\infty \mathbf{Token}\}$, 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 \mathbf{upd}_\times over an ampar and filling destinations or composing two ampars using \triangleleft 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 **Tokens** and **new_{×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 $1 \oplus (\mathbf{S} \otimes (\mathbf{T} \otimes \mathbf{U}))$. 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 \triangleleft). That's true indeed if just the address of v is written to $v'.f$. However, if v is moved into v' completely (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 flexibility 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 [Bagrel, 2024a]. Several sections have been reworked though, since the theoretical work described in Chapter 2 was still in very early stage when the article [Bagrel, 2024a] 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 [Bernardy et al., 2018]. 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, White, et al. [2024], 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 $\lambda_d\text{-TY}/\text{FROM}$, 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 @'()
$d \triangleleft \text{Inl}$	fill @'Inl d	or d &fill @'Inl
$d \triangleleft \text{Inr}$	fill @'Inr d	or d &fill @'Inr
$d \triangleleft (,)$	fill @'(,) d	or d &fill @'(,)
$d \triangleleft \text{Mod}_{\omega\infty}$	fill @'Ur d	or d &fill @'Ur
$d \triangleleft (\lambda x_m \mapsto \text{term})$	fillLeaf (\x \rightarrow term) d	or d &fillLeaf (\x \rightarrow term)
$d \triangleleft \circ \text{term}$	fillComp term d	or d &fillComp term
$d \blacktriangleleft \text{term}$	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
$[\omega\infty T]$	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 3.1. Note that Haskell uses `⋈` for typing, and `(:)` for the list *cons* constructor, which is the opposite of what these symbols mean in λ_d .

```

1  -- we recall here the definition of list type in Haskell
2  data [t] = []          -- nil constructor
3             | (:) t [t] -- cons constructor
4
5  -- we also reuse the same linear token API as in Section 1.8
6  withToken :: (Token → Ur t) → Ur t
7
8  type DList t = UAmpar [t] (UDest [t])
9
10 newDList :: Token → DList t
11 newDList = newUAmpar @[t]
12
13 append :: DList t → t → DList t
14 dlist `append` x =
15   dlist `updWith` \d → case (d &fill @'(:)) of
16     (dh, dt) → (dh &fillLeaf x) ; dt
17
18 concat :: DList t → DList t → DList t
19 dlist1 `concat` dlist2 = dlist1 `updWith` \dt1 → (dt1 &fillComp dlist2)
20
21 toList :: DList t → Ur [t]
22 toList dlist = fromUAmpar' (dlist `updWith` \dt → dt &fill @'[])
```

Listing 3.1: Implementation of difference lists in DPS Haskell

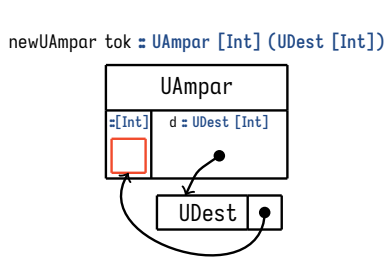


Figure 3.2: newUAmpar tok

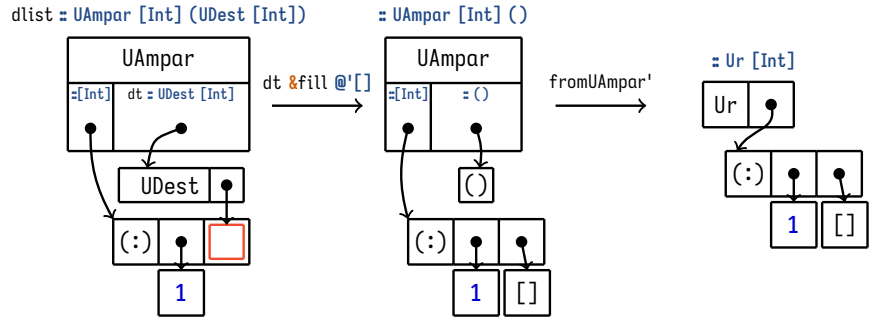


Figure 3.3: Memory behavior of toList dlist

- `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]`.

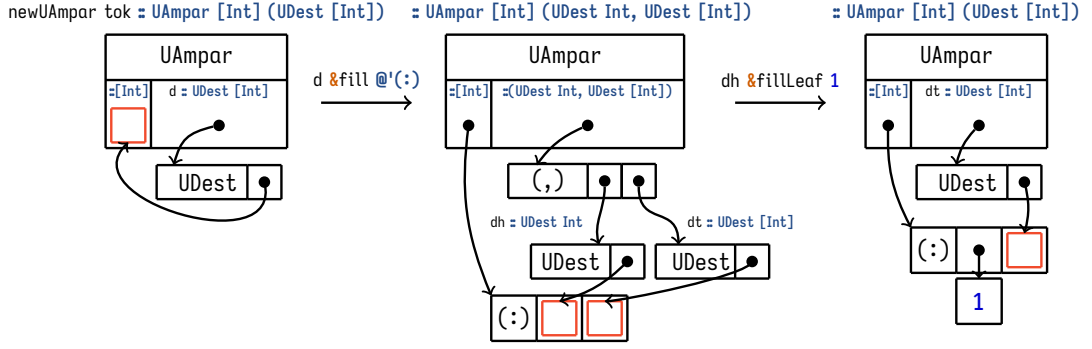


Figure 3.4: Memory behavior of append newUAmpar 1

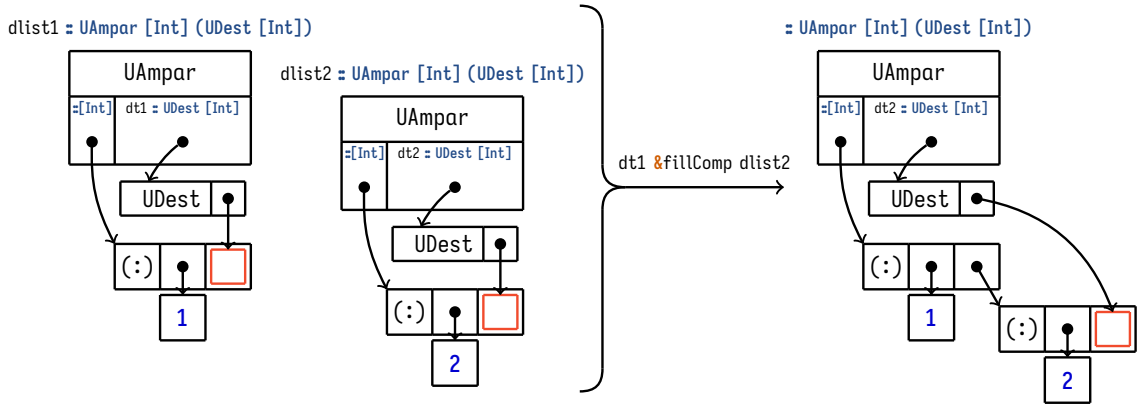


Figure 3.5: Memory behavior of concat dlist1 dlist2 (based on fillComp)

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 @[Int] (:)`, 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 essential¹⁷ 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

¹⁷In 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 $x : \square$ is represented by the function $\backslash ys \rightarrow x : ys$). 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 an 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 [Hood and Melville, 1981] queue. That's at least one thing we get from using a language that isn't fully based on destinations to build data structures¹⁸. The implementation of Hood-Melville queues is presented in Listing 3.2.

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 3.3.

Except from the choice of queue implementation, and the fact that ampars must be created from `Tokens`, 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. [2023].

¹⁸Destination-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.

```

1 data HMQueue t = HMQueue [t] [t]
2
3 newHMQueue :: HMQueue t
4 newHMQueue = HMQueue [] []
5
6 singleton :: t → HMQueue t
7 singleton x = HMQueue [x] []
8
9 toList :: HMQueue t → [t]
10 toList (HMQueue front backRev) = front ++ reverse backRev
11
12 enqueue :: HMQueue t → t → HMQueue t
13 enqueue (HMQueue front backRev) x = HMQueue front (x : backRev)
14
15 dequeue :: HMQueue t → Maybe (t, HMQueue t)
16 dequeue (HMQueue front backRev) = case front of
17   [] → case reverse backRev of
18     [] → Nothing
19     (x : front') → Just (x, HMQueue front' [])
20   (x : front') → Just (x, HMQueue front' backRev)

```

Listing 3.2: Implementation of Hood-Melville queue in Haskell

```

1 data Tree t = Nil | Node t (Tree t) (Tree t)
2
3 relabelDPS :: Token → Tree t → Tree Int
4 relabelDPS tree = fst (mapAccumBFS (\st _ → (st + 1, st)) 1 tree)
5
6 mapAccumBFS :: ∀ s t u. Token → (s → t → (s, u)) → s → Tree t → Ur (Tree u, s)
7 mapAccumBFS tok f s0 tree =
8   fromUAmper (newUAmper @(Tree u) tok `updWith` \dtree → go s0 (singleton (Ur tree, dtree)))
9   where
10     go :: s → Queue (Ur (Tree t), UDest (Tree u)) → Ur s
11     go st q = case dequeue q of
12       Nothing → Ur st
13       Just ((utree, dtree), q') → case utree of
14         Ur Nil → dtree &fill @'Nil ; go st q'
15         Ur (Node x tl tr) → case (dtree &fill @'Node) of
16           (dy, dtl, dtr) →
17             let q'' = q' `enqueue` (Ur tl, dtl) `enqueue` (Ur tr, dtr)
18             (st', y) = f st x
19             in dy &fillLeaf y ; go st' q''

```

Listing 3.3: Implementation of breadth-first tree traversal in DPS Haskell

```

1  data Token
2  dup :: Token → (Token, Token)
3  drop :: Token → ()
4  withToken :: ∀ t. (Token → Ur t) → Ur t
5
6  data UAmpar s t
7  newUAmpar :: ∀ s. Token → UAmpar s (UDest s)
8  tokenBesides :: ∀ s t. UAmpar s t → (UAmpar s t, Token)
9  toUAmpar :: ∀ s. Token → s → UAmpar s ()
10 fromUAmpar :: ∀ s t. UAmpar s (Ur t) → Ur (s, t)
11 fromUAmpar' :: ∀ s. UAmpar s () → Ur s
12 upWith :: ∀ s t u. UAmpar s t → (t → u) → UAmpar s u
13
14 data UDest t
15 type family UDestsOf lCtor t -- returns dests associated to fields of constructor
16 fill :: ∀ lCtor t. UDest t → UDestsOf lCtor t
17 fillComp :: ∀ s t. UAmpar s t → UDest s → t
18 fillLeaf :: ∀ t. t → UDest t → ()

```

Listing 3.4: Destination API for Haskell

3.4 DPS Haskell API and Design concerns

Listing 3.4 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 `newKIP` 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):

```

1  type DList t = UAmpar [t] (UDest [t])
2  data EffQueue t = EffQueue [t] (DList t)
3
4  newEffQueue :: Token → EffQueue t
5  newEffQueue tok = EffQueue [] (newUAmpar @[t] tok)
6
7  dequeue :: EffQueue t → Maybe (t, EffQueue t)
8  dequeue (EffQueue front back) = case front of
9    [] → case (toList back) of
10      Ur [] → Nothing
11      Ur (x : xs) → Just (x, (EffQueue xs (newUAmpar @[t] tok)))
12    (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`:

```

1  tokenBesides :: ∀ s t. UAmpar s t → (UAmpar s t, Token)
2
3  dequeue :: EffQueue t → Maybe (t, EffQueue t)
4  dequeue (EffQueue front back) = case front of
5    [] → case tokenBesides back of (back, tok) → case (toList back) of
6      Ur [] → drop tok ; Nothing
7      Ur (x : xs) → Just (x, (EffQueue xs (newDList tok)))
8    (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 [Spiwack et al., 2022; Spiwack, 2023a], 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 \blacktriangleleft 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 \triangleleft 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.

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

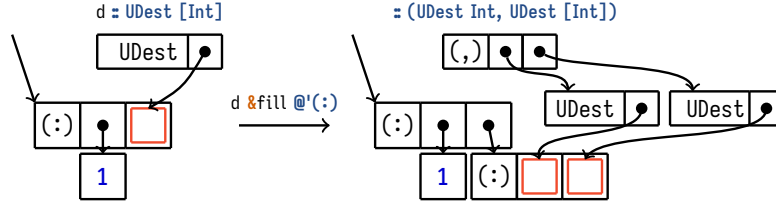


Figure 3.6: Memory behavior of `fill @'(:) :: UDest [t] → (UDest t, UDest [t])`

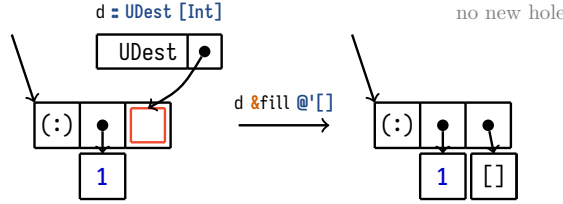


Figure 3.7: Memory behavior of `fill @[] :: UDest [t] → ()`

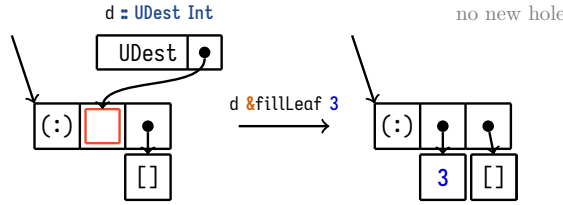


Figure 3.8: Memory behavior of `fillLeaf :: t → UDest [t] → ()`

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 UAmper 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. [2015] 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 `UAmpars` 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 3.5. 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.

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

```

1 data Token
2 dup :: Token → (Token, Token)
3 drop :: Token → ()
4 withToken :: ∀ t. (Token → Ur t) → Ur t
5
6 type Region r :: Constraint
7 inRegion :: ∀ t. (∀ r. Region r ⇒ t) → t
8
9 data UAmpar r s t
10 newUAmpar :: ∀ r s. Region r ⇒ Token → UAmpar s (UDest r s)
11 tokenBesides :: ∀ r s t. Region r ⇒ UAmpar r s t → (UAmpar r s t, Token)
12 toUAmpar :: ∀ r s. Region r ⇒ Token → s → UAmpar r s ()
13 fromUAmpar :: ∀ r s t. Region r ⇒ UAmpar r s (Ur t) → Ur (s, t)
14 fromUAmpar' :: ∀ r s. Region r ⇒ UAmpar r s () → Ur s
15 updWith :: ∀ r s t u. Region r ⇒ UAmpar r s t → (t → u) → UAmpar r s u
16
17 data UDest r t
18 type family UDestsOf lCtor r t -- returns dests associated to fields of constructor
19 fill :: ∀ lCtor r t. Region r ⇒ UDest r t → UDestsOf lCtor r t
20 fillComp :: ∀ r s t. Region r ⇒ UAmpar r s t → UDest r s → t
21 fillLeaf :: ∀ r t. Region r ⇒ t → UDest r t → ()

```

Listing 3.5: 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:

- `parseSEExpr` 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 `parseSEExpr` 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 3.6: 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 [Bagrel, 2023b].

The implementation presented in Listing 3.6 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 3.7. 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 3.7: Implementation of the S-expression parser with destinations

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

- 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

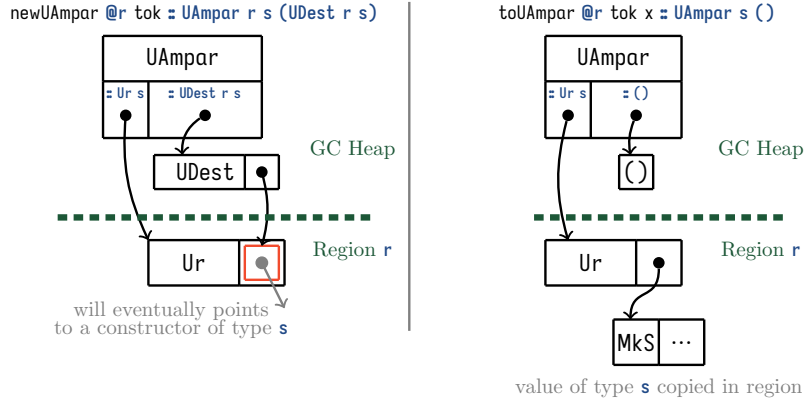


Figure 3.9: Representation of UAmpars in the compact region implementation, using `Ur` (not chosen)

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` constructor¹⁹) 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 [Bagrel, 2023b], but it could be worth comparing and benchmarking it with the one based on `Ur` indirection in a variety of real-world use-cases.

¹⁹One 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 [Spiwack, 2024].

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

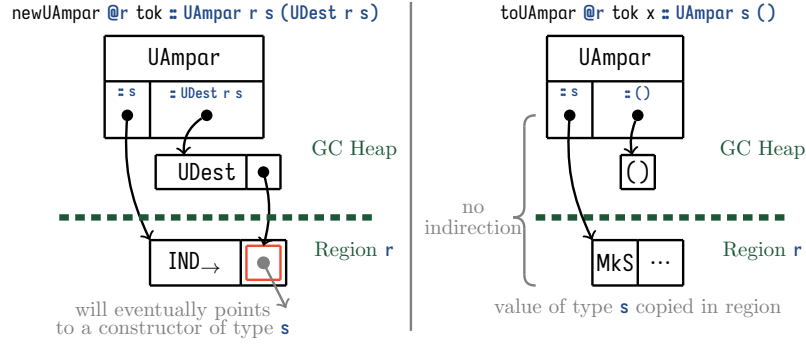


Figure 3.10: Chosen representation for UAmpars in the compact region implementation, using indirections

In the end, in [Bagrel, 2023b], we have the following private definitions²⁰ 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 type²¹ can be seen as a n-ary product constructor. So we will leverage `GHC.Generics` to find the required information.

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

²¹Dealing 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**:

```

1 | repl> :k! Rep (Maybe a) () -- display the Generic representation of Maybe a
2 | M1 D (MetaData "Maybe" "GHC.Maybe" "base" False) (
3 |   M1 C (MetaCons "Nothing" PrefixI False) U1
4 |   :+: 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 techniques²², 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 [Bagrel, 2023b], we have one specialized implementation of **fill** for every possible number of fields in **lctor**, from 0 to 7²³. The main reason is that **fill** should return a tuple containing as many new destinations as there are fields in 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

²²see `src/Compact/Pure/Internal.hs:418` in [Bagrel, 2023b]

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

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

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 *internal primop* which resolves, at compile time, a data constructor into the (static) value representing its info table pointer.
- one *external primop*, in the RTS, which allocates space inside a compact region for a hollow constructor, and which sets the info table pointer of the hollow constructor to a value received as an argument, that the developer can provide using the internal primop.

All the alterations to GHC that will be showed here are available in full form in [Bagrel, 2023a].

Internal primop: obtain the info table pointer of a constructor as a static value

Internal primops are macros that are transformed at compile time into C-- code. So for our new internal primop, in charge of obtaining the info table label (later resolved into a pointer) of a constructor, we must indicate our choice of constructor as some type-level information so that it can be inspected as compile-time (as, of course, term-level values cannot be inspected easily at compile time). For that, we have to lift the chosen constructor into a type-level value, using operator `'`, and have the internal primop accept a type parameter (representing our choice of constructor) and inspect it.

Usually, in Haskell, we use a parameter of type `Proxy t` (the unit type with a phantom type parameter `t`) as a carrier for a type-level input that we want to pass to a function. Unfortunately, primops are special functions, and due to a quirk of the compiler, they no longer have access to the type of their arguments at the stage in which they are resolved, so designing our primop with a `Proxy lctor` parameter wouldn't let it read the supplied type. However primops can —surprisingly— access their return type. Consequently, we design a custom return type `InfoPtrPlaceholder#` for our primop that carries the output value of the primop (that is to say, an info table pointer); but we also add a phantom type parameter `lctor` to `InfoPtrPlaceholder#` that has no relation to the underlying value, but just serves as a carrier for our type-level input representing the chosen constructor!

The concrete implementation is presented briefly in Listing 3.8. This implementation goes a bit further than what we just described above, as `reifyInfoPtr#` can obtain the info table pointer of data constructors (as announced), but also the info table pointer of special compiler objects like the indirection object we used in Section 3.5.3. The primop pattern-matches on the type `resTy` of its return value (which should be of the form of the form `InfoPtrPlaceholder# lctorOrSym`).

In the case it reads a lifted data constructor (which is the most common case), it resolves the primop call into the label `<ctor>_con_info` which corresponds to the info table pointer of that constructor. In the case `lCtorOrSym` is a symbol (that is, a type-level string literal), it resolves the primop call into the label `stg_<sym>` (e.g. to allocate a `stg_IND` object, as we just mentioned). The returned `InfoPtrPlaceholder# lCtorOrSym` can later be converted back to an `Addr#` using the `unsafeCoerceAddr` function (nothing is particularly *unsafe* here, despite the name, as both types share the same underlying representation).

```

1 | case primop of
2 |   [...]
3 |   ReifyInfoPtrOp → \_ → -- we don't care about the function argument (##)
4 |     opIntoRegsTy $ \[res] resTy → emitAssign (CmmLocal res) $ case resTy of
5 |       -- when the phantom type parameter is a lifted data constructor, extracts it as a DataCon
6 |       TyConApp _infoPtrPlaceholderTyCon [_typeParamKind, TyConApp tyCon _]
7 |         | Just dataCon ← isPromotedDataCon_maybe tyCon →
8 |           CmmLit (CmmLabel (
9 |             mkConInfoTableLabel (dataConName dataCon) DefinitionSite))
10 |       -- when the phantom type parameter is a Symbol, we extracts the symbol value in 'sym'
11 |       TyConApp _infoPtrPlaceholderTyCon [_typeParamKind, LitTy (StrTyLit sym)] →
12 |         CmmLit (CmmLabel (
13 |           mkCmmInfoLabel rtsUnitId (fsLit "stg_" `appendFS` sym)))
14 |       _ → [...] -- error when no pattern matches

```

Listing 3.8: `reifyInfoPtr#` implementation in `compiler/GHC/StgToCmm/Prim.hs`

External primop: allocate a hollow constructor in a region Our external primop, as part of the RTS, is directly written in C--. Its implementation is presented in Listing 3.9, under the name `stg_compactAddHollowzh`; but in Haskell code we access it through the name `compactAddHollow#`. The primop consists mostly in 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. Then, it takes the info table pointer of the constructor to allocate, received as its second parameter `W_ info` (as it cannot obtain this information itself), and write it to the first word of the heap object using `SET_HDR`. Then the pointer to the newly allocated hollow constructor is set as the return value.

Combining both primops to obtain a new hollow constructor It's time to show the two primops in action. Here is how we could allocate, for example, a hollow `Just` in a compact region:

```

1 | hollowJust :: Maybe a = compactAddHollow#
2 |   region
3 |   (unsafeCoerceAddr (reifyInfoPtr# (##) :: InfoPtrPlaceholder# 'Just ))

```

It's important to note that the expression `reifyInfoPtr# (##)` will be replaced at compile-time by a static value representing the info table pointer of `Just`; while the call to `compactAddHollow#` behaves as a normal function call, and will only be executed when the program runs.

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

```

1 // compactAddHollow#
2 // :: Compact# → Addr# → State# RealWorld → (# State# RealWorld, a #)
3 stg_compactAddHollowzh(P_ compact, W_ info) {
4     W_ pp, ptrs, nptrs, size, tag, hp;
5     P_ to, p; p = NULL; // p isn't actually used by ALLOCATE macro
6     again: MAYBE_GC(again); STK_CHK_GEN();
7
8     pp = compact + SIZEOF_StgHeader + OFFSET_StgCompactNFData_result;
9     ptrs = TO_W_(%INFO_PTRS(%STD_INFO(info)));
10    nptrs = TO_W_(%INFO_NPTRS(%STD_INFO(info)));
11    size = BYTES_TO_WDS(SIZEOF_StgHeader) + ptrs + nptrs;
12
13    ALLOCATE(compact, size, p, to, tag);
14    P_[pp] = to;
15    SET_HDR(to, info, CCS_SYSTEM);
16    #if defined(DEBUG)
17    ccall verifyCompact(compact);
18    #endif
19    return (P_[pp]);
20 }

```

Listing 3.9: compactAddHollow# implementation in rts/Compact.cmm

Built-in type family to go from a lifted constructor to the associated symbol The internal primop `reifyInfoPtr#` that we just introduced uses a constructor lifted at type level as an input. So the fill function, which uses `reifyInfoPtr#` and `compactAddHollow#` under the hood, should also use a constructor lifted at type level as a way of selecting the hollow constructor that will be added to a structure with holes.

However, fill relies on `GHC.Generics` to obtain various informations on the chosen constructor (see Section 3.5.4), and in the `Generic` representation of a type, constructors are identified by their name (as type-level strings), which are distinct type-level entities than the constructors lifted at type level like we used in `reifyInfoPtr#`.

Consequently, we need a way to translate a constructor lifted at type level into the type-level string representing the name of this constructor²⁴. This is the role of the type family `LCtorToSymbol` that we also added to `GHC`. It just 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 3.10.

Putting it all together, and further evolutions All these elements combined provide the low-level backbone needed to implement our fill function. The implementations presented above have 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.

²⁴We do the translation in this specific direction because the type-level string representing the name of the constructor is not enough to fully identify a given constructor.

```

1 | matchFamLctorToSymbol :: [Type] → Maybe (CoAxiomRule, [Type], Type)
2 | matchFamLctorToSymbol [kind, ty]
3 |   TyConApp tyCon _ ← ty, Just dataCon ← isPromotedDataCon_maybe tyCon =
4 |     let symbolLit = (mkStrLitTy . occNameFS . occName . getName $ dataCon)
5 |     in Just (axLctorToSymbolDef, [kind, ty], symbolLit)
6 | matchFamLctorToSymbol tys = Nothing
7
8 | axLctorToSymbolDef =
9 |   mkBinAxiom "LctorToSymbolDef" typeLctorToSymbolTyCon Just
10 |   (\case { TyConApp tyCon _ → isPromotedDataCon_maybe tyCon ; _ → Nothing })
11 |   (\_ dataCon → Just (mkStrLitTy . occNameFS . occName . getName $ dataCon))

```

Listing 3.10: `LctorToSymbol` implementation in `compiler/GHC/Builtin/Types/Literal.hs`

However, since this first experimentation [Bagrel, 2023a], a pull request has been opened [Bagrel, 2024b] to gather community input on the topic, with the goal to eventually merge support for hollow constructor building into the main, publicly-distributed branch of GHC. It seems that we are moving towards a slightly more complete set of primitives than the three aforementioned ones, in particular with primitive forms for `fill` being part of the GHC additions (instead of being only in library code, as it is now). Still, much work remains to be done before we can reach a mergeable state.

3.6 Evaluating the performance of DPS programming

It’s now time to see if our claims of performance are verified with our prototype destination-passing implementation [Bagrel, 2023b].

Benchmarking methodology Without surprise, we will compare programs in both naive style and DPS style, and see if we can observe differences in terms of speed but also memory efficiency. But first we must be careful in the way we measure time and space usage, especially since Haskell defaults to a lazy evaluation strategy.

In every DPS implementation that we’ll benchmark, the output produced by the program is stored in a compact region (we don’t really have the choice). As we said in Section 3.5.1, compact regions also force strictness i.e. the structure will be automatically in a fully evaluated form. It might be counterproductive in some use-cases, but at least it makes things simpler for the benchmark (in which we want to force the result to a fully evaluated form).

For the naive implementations on the other hand, we have a choice to make on the way we fully evaluate the result. We can either keep the result inside the GC heap, and use the function `Control.DeepSeq.force` to recursively force each thunk of the GC heap into an evaluated form; or we can use `Data.Compact.compact` to copy the result from the GC heap to a compact region, that is strict by default and thus will force evaluation.

That being said, having the result of the function copied into a compact region is only desirable for a handful of programs, for which the result will be held in memory for a long time and used in full (as the unused parts cannot be collected if it is in a compact region). Consequently, we

3.6. Evaluating the performance of DPS programming

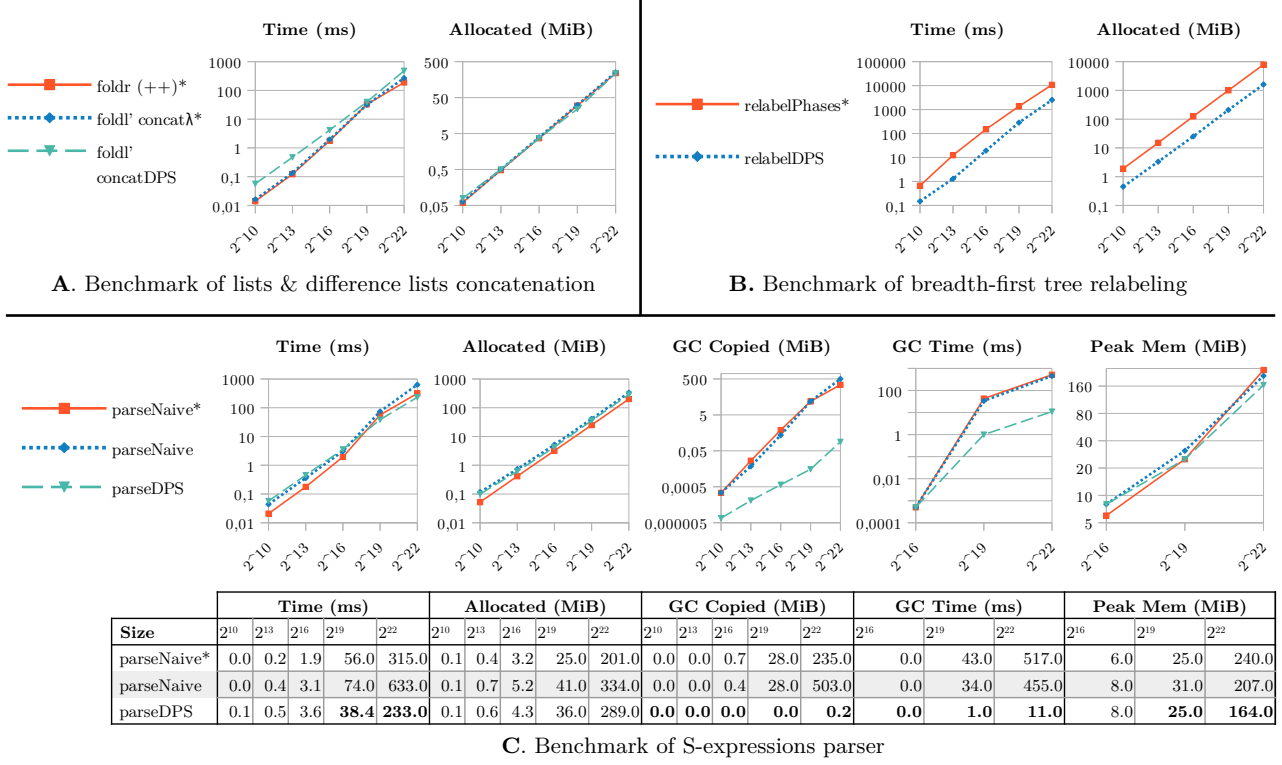


Figure 3.11: Benchmarks performed on AMD EPYC 7401P @ 2.0 GHz (single core, -N1 -O2)

often benchmark the naive implementations with `force` (the associated benchmark names are denoted with a “*” suffix), and only benchmark with both `force` and `compact` copy when it seems relevant.

3.6.1 Concatenating lists and difference lists

We compared three implementations for list concatenation.

The first one, `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 before flushing it to the disk.

Hence, for the two other implementations, the calls to `concat` nested to the left. The implementation `foldl' concatλ*` uses function-backed difference lists, and `foldl' concatDPS` uses destination-backed ones (from Section 3.3.1). They should still run in $\mathcal{O}(n)$ however, 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 implementations, while being quite slower (by a factor 2-4) on all datasets. It’s still unclear where the performance loss comes from. 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 embarrassing 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.

3.6.2 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 [Gibbons et al., 2023]:

```

1  import Control.Monad.State.Lazy (runState, state)
2
3  data Phases m a where
4    Pure :: a → Phases m a
5    Link :: (a → b → c) → m a → Phases m b → Phases m c
6
7  instance Functor (Phases m) where ...
8  instance (Applicative m) ⇒ Applicative (Phases m) ...
9
10 now :: (Applicative m) ⇒ m a → Phases m a
11 now xs = Link (curry fst) xs (Pure ())
12
13 later :: (Applicative m) ⇒ Phases m a → Phases m a
14 later xs = Link (curry snd) (pure ()) xs
15
16 runPhases :: (Applicative m) ⇒ Phases m a → m a
17 runPhases (Pure x) = pure x
18 runPhases (Link f xs ys) = pure f <*> xs <*> runPhases ys
19
20 traverseBFS :: (Applicative m) ⇒ (a → m b) → BinTree a → Phases m (BinTree b)
21 traverseBFS _ Nil = pure Nil
22 traverseBFS f (Node x tl tr) =
23   pure Node <*> now (f x) <*> later ((traverseBFS f) tl) <*> later ((traverseBFS f) tr)
24
25 relabelPhases :: BinTree () → (BinTree Int, Int)
26 relabelPhases tree = runState (runPhases $ traverseBFS (\_ → state (\s → (s, s + 1))) tree) 0

```

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. [2023]. We must admit though that we didn't particularly try to optimize the performance of the *Phases* implementation as much as we did for our DPS framework in general.

3.6.3 Parsing S-expressions

In part **C** of Figure 3.11, we compare the naive and DPS implementations of the S-expression parser presented in Section 3.5.2. For this particular program, using compact regions is interesting even without destination-passing considerations, as it will reduce the GC load of the application

as long as the data is held in memory (which is quite usual with parsed documents). Thus, we benchmark the naive version twice: once where the results stays in the GC heap and is evaluated 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 implementation 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 ($2^{22} \simeq 4$ MiB file), the DPS implementation still makes about 45% more allocations than the starred naive one, but uses 35% less memory at its peak, and more importantly, spends $47\times$ less time in garbage collection. As a result, the DPS implementation terminates in $0.65\times$ the time it takes for the best of the two naive implementation to complete, 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 `UAmpars` and `UDests` that just last one generation and thus can be discarded very early by the GC (as intended), without needing to be copied into the next generation, unlike the data nodes allocated by the naive versions that have to be persisted by GC (by copying or moving them). In a real-world scenario, where data is processed for some time after having been parsed, we would expect the naive implementation using `compact` to get more efficient in terms of garbage collection time (and thus, run time) than the starred naive version using `force`, but it would still be behind the DPS implementation which saves unnecessary copies from the GC heap to the compact region.

3.6.4 Mapping a function over a list

In a strict functional language such as OCaml, the choice of implementation for the `map` function on lists is crucial as the naive, non tail-recursive one (further referred to as `map`) makes the stack grow linearly with the size of the processed list. To alleviate the issue, a strict tail-recursive implementation is possible (`mapTR'`), but it requires an extra $\mathcal{O}(n)$ operation at the end of the processing to reverse the accumulator. In Haskell, the issue is of a lesser importance perhaps, as the lazy-by-default evaluation strategy makes the naive `map` perform quite well in most real-world programs. Still, let see what our DPS API brings to the table.

With destinations, `map` can be implemented in a tail-recursive fashion too (`mapDPS'`), as explained in Section 2.2.2, without requiring the reverse operation that `mapTR'` needs, as the list is built directly in a top-down approach. It can alternatively be implemented as a fold (`mapDPSFold'`), as we also show below. Here, the `!` in front of a variable name denotes a (weakly) strict binding, so the expression assigned to the variable will be partially evaluated before the body of the `let` is evaluated:

```

1 | append :: UDest [t] -> t -> UDest [t]
2 | d `append` x = case d &fill @'(:) of (dh, dt) -> dh &fillLeaf x ; dt
3 |
4 | mapDPS' _ [] d = d &fill @'[]
5 | mapDPS' f (x : xs) d = let !y = f x ; !d' = d `append` y in mapDPS' f xs d'
6 |
7 | mapDPSFold' f l dl = fill @'[] (foldl' (\d x -> let !y = f x in d `append` y) d l)

```

We see in part 4 of Figure 3.11 that the destination-based implementations `mapDPS'` and `mapDPSFold'` takes $1.5\text{--}4\times$ more time than the naive `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, the equivalent of `mapDPS'` is always more performant time-wise ($0.5 - 0.85\times$) than the equivalent of `mapTR'`, even for small lists, as detailed in benchmarks of [Bour, Clément, and Scherer, 2021].

3.7 Conclusion

Programming with destinations definitely has a place in practical functional programming, as shown by the recent adoption of *Tail Modulo Cons* [Bour, Clément, and Scherer, 2021] in the OCaml compiler. In this chapter, we have shown how we can take most ideas from λ_d (Chapter 2) and adapt them to allow safe destination passing in Haskell, without modifying the core principles of this existing language. We also showed that most of the examples presented in chapter Chapter 2 can be implemented in DPS Haskell with little to no adaptations required, and we also emphasized how our particular implementation of destination passing for Haskell can help to improve the performance of programs in which garbage collection is very costly. To that extend, we rely on the *compact regions* [Yang et al., 2015] memory management scheme for Haskell; but our work also extend the way in which compact regions can be used efficiently.

Finally, we also demonstrated practical gains brought by destination-passing for some of the algorithms we presented all along this thesis; even though our DPS Haskell implementation is a bit heavyweight still and thus doesn't always let the destination-based algorithms match the theoretical performance expectations.

In this chapter, we could only use DPS Haskell for structures carrying unrestricted data. This was decided to keep a (relatively) simple API while avoiding the issues of scope escape (Section 2.3). In the next chapter, we'll revisit this limitation and find ways to modify DPS Haskell so that we can safely store linear resources in structures with holes.

Chapter 4

Extending DPS support in Haskell for linear data structures

The main issue with the approach of the previous chapter is that we are not able to build data structures with a destination-based approach to linear data. This is the consequence of having a lot of flexibility in the way destinations can be used, without a system of ages to prevent scope escape in a fine-grained manner. So we had to use *unrestrictedness* as a barrier so that destinations cannot escape their scope.

Previous works however, such as [Minamide, 1998], or [Lorenzen, Leijen, Swierstra, and Lindley, 2024], 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; in these works we can only fill holes in a structure by interacting with the incomplete structure itself (the *ampar*). 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 – scope escape, updated

Starting from this point, we will say *linear destination* for an object of type `[1T]` or corresponding `Dest 1 t`, that is, a destination that can host linear data. Similarly, an *unrestricted destination* is an object of type `[ωT]` or `Dest ω t`. Regardless, a destination is always, itself, a resource that must be managed linearly. Finally, in the main part of this section, we omit compact region details, for clarity of exposition.

If we want to allow destinations to be filled with linear data, we have to be extremely careful about potential scope escape (see Section 2.3). One observation is that scope escape can only happen when we let the user fill an older linear destination in a newer scope. Because only linear destinations can be filled with another destination; and we've seen in Section 2.3 that filling a newer destination with an older one doesn't cause scope escape; only filling an older one with a newer one does.

A potential remedy would be to not allow linear destinations to be filled using `fillLeaf` or `fillComp` functions. Instead, linear destinations could only be filled by operating on the *ampar* they come from, as in [Minamide, 1998; Lorenzen, Leijen, Swierstra, and Lindley, 2024]. That way, we're sure we can only fill the freshest linear destinations (as destinations start to age when

they are made accessible in a new scope with `updWith`; so destinations still on the right side of a closed ampar are the fresher than any other destination in the context). We will use the base name `extend` for this new family of operators that operate directly on an ampar to fill its destinations. Unrestricted destinations, on the other hand, can still be used with the various fill functions. We should note that filling a destination with a hollow constructor cannot cause any scope escape issues, so it's safe to use `fill @'Ctor` on both linear and unrestricted destinations. Let's see what this updated API would look like:

```

1 | data Ampar s t
2 | data Dest m t -- m is the multiplicity, either 1 or ω
3 | newAmpar :: Token → Ampar s (Dest 1 s)
4 | updWith :: Ampar s t → (t → u) → Ampar s u
5 |
6 | -- for unrestricted destinations
7 | fillLeaf :: t → Dest ω t → ()
8 |
9 | -- for linear destinations
10 | extendLeaf :: t → Ampar s (Dest 1 t) → Ampar s ()

```

The idea is that `extendLeaf` is a way of opening a new scope, filling the linear destination belonging to that fresh scope, and closing it immediately afterwards, without letting the user interfere. So we are confident that the linear destination filled by `extendLeaf` is strictly younger than whatever ends up being 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:

```

1 | newAmpar tok0 `updWith` \(d0 :: Dest 1 (Dest 1 t)) →
2 |   newAmpar tok1 `updWith` \(d1 :: Dest 1 t) →
3 |     (newAmpar tok2 `updWith` \(d2 :: Dest ω ()) → d2 &fillLeaf () ; d0) &extendLeaf d1

```

In this particularly **cursed** but valid²⁵ excerpt, 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, but comes from a previous scope. However, it is indistinguishable from an ampar whose right side contains only its own destinations, if we just look at the types. So we are allowed to use `extendLeaf` on the cursed ampar, which causes 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.

²⁵It'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 $\lambda_d\text{-TY/ID}$.

4.2. /extend/ functions for ampars with a single hole

This is done by changing the signature from `updWith :: Ampar s t \multimap (t \multimap u) \multimap Ampar s u` to `updWith :: Ampar s t \multimap (t \multimap u) \rightarrow Ampar s u`. Notice the unrestricted function arrow \rightarrow : it means the linear function supplied by the user must not capture anything linear (but still must use its argument linearly). Adding this drastic restriction 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 (we described how convenient it is to just have one top-level linear scope in Section 1.8). Instead, we have to start a new nested linear scope (by calling `withToken`) if we want to create new ampars in the body of `updWith`, or tweak `updWith` to also provide a linear token to the inner scope, which gives `updWithToken :: Ampar s t \multimap ((t, Token) \multimap u) \rightarrow Ampar s u` (the ampar on which `updWithToken` operates is a linear resource on which we can base the chain of linearity that threads through the new token).

Also, even with this restriction on `updWith`, 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. Let's illustrate this issue through an example:

```

1  fromAmpar' (
2    newAmpar @(t, Dest 1 t) tok `updWith` \ (d :: Dest 1 (t, Dest 1 t))  $\rightarrow$ 
3      case (d &fill @(,)) of (dt :: Dest 1 t, ddt :: Dest 1 (Dest 1 t))  $\rightarrow$ 
4        ddt &fillLeaf dt
5  )

```

Here, we create a new ampar for a pair of a `t` and a `Dest 1 t`. After filling the root destination `d` 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 this danger, because the mechanisms used to prevent scope escape in Chapters 2 and 3—that is, either the age system, or the 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, it still wouldn't be safe to 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 directly, instead of destinations. In fact, that is exactly what happens in [Minamide, 1998], or [Lorenzen, Leijen, Swierstra, and Lindley, 2024]: 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).

Let's start by designing new operators for ampars with only one hole, so exactly one (linear) destination on their right-hand side. This restriction implies that 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 in the structure.

In other terms, we don't want a function working like `fill @'(:)`, that returns two new destinations; instead, we must design a function that takes a value as an input, to be filled directly into one of the fields of the pair, so that only one new destination is created²⁶. Designing such a function in a generic way is a bit cumbersome, as shown by the signature of `extend` below:

```

1 | extend :: ∀ lCtor k t m s fiTys kthFiTy othFiTys.
2 |     (OnlyLinear lCtor t,
3 |     FieldTypes lCtor t ~ fiTys,
4 |     At k fiTys ~ kthFiTy, Remove k fiTys ~ othFiTys) ⇒
5 |     othFiTys %m→ Ampar s (Dest m t) → Ampar s (Dest m kthFiTy)

```

There's a lot to decipher here. `FieldTypes` is a type family that returns the types of the fields for a given constructor (specified `lCtor` and the type `t` to which the constructor belongs). For example, `FieldTypes '(:) [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 remaining field types except the k^{th} one, in a tuple if there are more than one remaining type.

For instance, in `extend @'(:) @0 @[Int]`, `kthFiTy` is `Int`, and `othFiTys` is `[Int]` (not in tuple form because there is only one remaining type). Conversely, in `extend @'(:) @1 @[Int]`, `kthFiTy` is `[Int]`, and `othFiTys` is just `Int`.

In practice, such a signature means that `extend` takes an ampar whose right side is a destination of type `t`, with multiplicity `m` (so that we can account for both linear and unrestricted destinations, even though unrestricted destinations can be manipulated in a more flexible fashion through the fill family functions). 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} one, and then returns a destination with same multiplicity `m` for that remaining field of type `kthFiTy`.

For example, the `append :: DList t → t → DList t` function on difference lists can be expressed directly with `extend`. So far, `DList` could only store unrestricted elements, but now, we can use difference lists to carry linear elements!

```

1 | type DList t = Ampar [t] (Dest 1 [t])
2 |
3 | append :: DList t → t → DList t
4 | dlist `append` x = dlist &extend @'(:) @1 x

```

We can also use `extend` to define and use *zipper* trees (in the light of [Huet, 1997; Lorenzen, Leijen, Swierstra, and Lindley, 2024]) that can hold linear data:

²⁶It's still possible to use `updWith` to access a linear destination, then use `fill @'(:)` on it, 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` only works on unrestricted destinations.

4.2. /extend/ functions for ampars with a single hole

```

1 data Tree t = Nil | Node (Tree t) t (Tree t)
2 type ZipperTree t = Ampar (Tree t) (Dest 1 (Tree t))
3
4 nodeL :: ZipperTree t → t → Tree t → ZipperTree t
5 nodeL parent x rtree = parent &extend @'Node @0 (x, rtree)
6
7 nodeR :: ZipperTree t → Tree t → t → ZipperTree t
8 nodeR parent ltree x = parent &extend @'Node @2 (ltree, x)

```

Note that `extend` doesn't support constructors with 0 fields (it would make the signature of the function way too complex). But 0-ary constructors can be written to the structure like any other complete value anyway, with `extendLeaf`:

```

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

```

Here we've also made `extendLeaf` multiplicity-polymorphic, compared to the initial version presented in Section 4.1, but its behavior is the same.

It is a bit pointless in fact to return an `Ampar s ()` to the user from a call to `extendLeaf`; it's better to directly extract the completed structure of type `s`:

```

1 fromAmparWithLeaf :: ∀ s t m. t %m→ Ampar s (Dest m t) → s
2 fromAmparWithLeaf = fromAmpar' . extendLeaf
3
4 -- we use it to implement the remaining operator for zipper trees
5 done :: ZipperTree t → Tree t
6 done parent = parent &fromAmparWithLeaf Nil

```

Multiplicity of fields of a constructor and `extendUr` Above, in the definition of `extend`, we considered that the constructor represented by `lctor` had only linear fields, as reflected by the constraint `OnlyLinear lctor t`. Indeed, the multiplicity `m` at which we consume the values for the other fields, and at which we return the new destination, are the same as the multiplicity of the parent destination.

In λ_d , every data constructor has linear fields, except the exponential constructor `Modm` 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 needing an explicit `Ur` layer.

Ideally, we should thus refine the `extend` function further, and be polymorphic over the multiplicity of each field. However, to this date, we cannot get the information about field multiplicities through `GHC.Generics`, that we use in our practical implementation. More than that actually, we cannot derive a `Generic` instance for **types having constructors defined in GADT form.**

Hence, for now, it doesn't seem worth it to modify the `extend` function even more for arbitrary multiplicity support on fields. But there is also another consequence: **it means we have to add a dedicated version of `extend` for the `Ur` constructor, as it doesn't have a `Generic` instance:**

```

1 extendUr :: ∀ s t m. Ampar s (Dest m (Ur t)) → Ampar s (Dest w t)

```

Given that $\mathbf{m} \cdot \omega = \omega$ for any \mathbf{m} , the `extendUr` function can operate on an ampar with either a linear or unrestricted destination, but always returns an ampar with an unrestricted destination.

Using `extendUr` is the primary way, in this world, to obtain an initial unrestricted destination `Dest ω t`, as `newAmpar` initially spawns an ampar with just a linear destination on the right.

Composition of incomplete structures with `extendComp` The last function that we need, in this setting, is `extendComp`, to plug two ampars together:

```
1 | extendComp =  $\forall$  s t u. Ampar t u  $\multimap$  Ampar s (Dest 1 t)  $\multimap$  Ampar s u
```

In `extendComp`, we need the destination in the parent ampar to be a linear destination, for two reasons. First, we fill the destination from the parent ampar with the child ampar `Ampar t u`, which is itself a linear resource. Secondly, even in a world where ampars wouldn't have to be linear —like in the theoretical presentation of λ_d — the destination used for composition using `\triangleleft` / `fillComp` / `extendComp` should still be linear. Indeed, the ampar resulting from the composition inherits whatever was on the right of the child ampar (represented by `u` here), which in particular contains the remaining destination(s) from the child ampar. If we were to write the child ampar into an unrestricted destination of the parent ampar, then we would need to update the multiplicity of the remaining destinations of this child ampar, inside `u`, to be unrestricted too, and this is not easily expressible.

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

Being limited to only one destination on the right-hand side of ampars is not ideal. Indeed, we stay at the same expressivity level as existing work, and we lose part of the flexibility that made λ_d innovative in the first place.

Supporting multiple linear destinations on the right-hand side of ampars introduces significant complexity however. In the previous section, with only one destination per ampar, we avoided scope escape by requiring the user to interact with the ampar directly, rather than with the destination itself. This was straightforward, as there was no ambiguity —the single destination was the only possible target of the `extend` operation. By contrast, if we now allow arbitrary types on the right-hand side, but still aim to restrict direct user access to destinations (for the same reasons as above; `updWith` is now non-capturing, and we want to prevent scope escape and malicious interaction between sibling destinations), we must introduce a mechanism for users to indicate which specific destination they intend to operate on.

One idea for that is to ask the user for a linear lens. First, let's define what a lens 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 function to access a value of type `u1` somewhere inside a structure of type `t1`, which establishes the *focus* of the lens;
- a function 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`. In particular, when `u1 = u2`, then `t1 = t2`, and thus this function acts as an (immutable) setter for the focused value.

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

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

```
1 data LensNL t1 t2 u1 u2 = LensNL
2   { view :: t1 → u1 -- just get u1 out of t1, discard the rest
3   , update :: t1 → u2 → t2 -- discard the u1 inside t1 and replace it with a u2 to make a t2
4   }
```

Given a data structure `Triple` with 3 fields, let's create a lens to focus on the first field:

```
1 data Triple s t u = Triple { first :: s, second :: t, third :: u }
2
3 fstNL :: LensNL (Triple s t u) (Triple s' t u) s s'
4 fstNL = LensNL { view :: \ (Triple x y z) → x
5                 , update :: \ (Triple x y z) → x' → Triple x' y z }
```

With a non-linear lens `LensNL` such as above, if we use the `view` component, then we consume a value of type `t1` but discard most of it except the part of type `u1`. If we use the `update` component, it's the opposite; we discard the part of type `u1` but keep all the rest of the original `t1`.

For linear lenses, that we'll introduce now, we are not allowed to use the `view` and `update` components separately; otherwise we could drop parts of the original structure of type `t1`, as we've just seen. So instead, we group the two functions `view` and `update` into a single one:

```
1 newtype Lens t1 t2 u1 u2 = Lens (t1 → (u1, u2 → t2))
```

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`. In other terms, this second component of type `u2 → t2` is really just a functional representation of a `t2` missing a `u2` to be complete. Both of the components of the pair have to be consumed if we want to respect linearity.

It's fairly immediate to update our `Triple` example for linear lenses:

```
1 fst :: Lens (Triple s t u) (Triple s' t u) s s'
2 fst = Lens (\ (Triple x y z) → (x, \x' → Triple x' y z))
3 --           captures x ↗           ↖ captures y and z
```

Lenses to select destinations to operate on With linear lenses that we just introduced, the user can specify which destination to operate on when using the new family of `extendFocused` functions. We also get, at the same time, a way to reinsert the newly produced destinations (for fields of the new hollow constructor), or the byproduct of the filling operation, at the exact position that was previously occupied by the destination we just consumed, on the right-hand side of the ampar (thanks to the dual nature of lenses, acting both as a form of getter and setter):

```

1 | extendFocused :: ∀ lCtor u t1 t2 s m.
2 |   (OnlyLinear lCtor u) ⇒
3 |   Lens t1 t2 (Dest m u) (DestsOf lCtor m u) → Ampar s t1 → Ampar s t2
4 |
5 | extendWithNodeOnFst :: Ampar s (Triple (Dest m (Tree t1)) t2 t3)
6 |   → Ampar s (Triple (Dest m (Tree t1), Dest m t1, Dest m (Tree t1)) t2 t3)
7 | extendWithNodeOnFst ampar = ampar &(fst &extendFocused @'Node)

```

Here, `extendFocused` takes a lens of type `Lens t1 t2 (Dest m u) (DestsOf lCtor m u)`, i.e., a lens that focuses on a `Dest m u` inside the right-hand side of type `t1` of the ampar, and let us build a new right-hand side of type `t2` for the resulting ampar, where the destination of type `Dest m u` is replaced by a value of type `DestsOf lCtor m u` (that is, a tuple with the new potential destinations).

The lens is taken as a non-linear argument by `extendFocused` (as indicated by the non-linear arrow `→`), so that it cannot capture any linear resource. So the lens is forced to be just a purely structural operation that can be applied as many times as we want on various values of type `t1`.

Nonetheless, we could have implemented `extendWithNodeOnFst` in terms of `updWith` and `fill @'Node`²⁷. In fact, the added value of linear lenses becomes clear when designing the `extendFocusedLeaf` function, which for the `first time ever` (in the Haskell implementation), enables us to fill a linear destination with a value when the said destination is located within an arbitrary structure on the right-hand side of an ampar:

```

1 | extendFocusedLeaf :: ∀ u t1 t2 s m. u %m→ Lens t1 t2 (Dest m u) () → Ampar s t1 → Ampar s t2
2 |
3 | extendWithValOnFst :: t1 → Ampar s (Triple (Dest 1 t1) t2 t3) → Ampar s (Triple () t2 t3)
4 | extendWithValOnFst x ampar = ampar &(fst &extendFocusedLeaf x)

```

No scope escape can happen here. As `updWith` is non-capturing, we're sure that the destinations on the right-hand side of an ampar are the ones pointing to the holes on its left-hand side (eliminating the issue presented at the end of Section 4.1). So like `extendLeaf`, the `extendFocusedLeaf` function can only be used to fill a newer destination with existing (thus, older) values. Likewise, we cannot have sibling destinations interacting with one another. Which means, we've reached our goal for this chapter!

For completeness, we finally define `extendFocusedUr` and `extendFocusedComp`, following the same pattern as for `extendFocusedLeaf`:

```

1 | extendFocusedUr :: ∀ u t1 t2 s m.
2 |   Lens t1 t2 (Dest m (Ur u)) (Dest w u) → Ampar s t1 → Ampar s t2
3 | extendFocusedComp :: ∀ u t1 t2 s u1 u2.
4 |   Ampar u1 u2 → Lens t1 t2 (Dest 1 u1) u2 → Ampar s t1 → Ampar s t2

```

Both of these functions are very similar to their respective one-hole versions `extendUr` and `extendComp`. In `extendFocusedUr`, we consume a destination of type `Dest m (Ur u)` and replace it with a destination of type `Dest w u`. In `extendFocusedComp`, we consume a destination of type `Dest 1 u1`, to which we write the left-hand side (of type `u1`) of the child ampar, and we replace this consumed destination with whatever was on the right-hand side (of type `u2`) of the child

²⁷As said previously, `fill @'Ctor` cannot cause scope escape and thus is safe to use for both linear and unrestricted destinations

ampar. The only difference, between the `extend` and the corresponding `extendFocused` functions, is that the filling operation in the latter happens at a focused location on the right-hand side of an ampar instead of being made on the single destination at top level as in the former.

4.4 Breadth-first tree traversal, updated, and final API

The family of `extendFocused` functions, based on linear lenses, addresses the core challenge of safely filling destinations with linear data. Although the resulting interface is not entirely satisfactory —particularly from an ergonomic standpoint— it still strikes a reasonable compromise between expressiveness and complexity of the interface (given the constraint of working with the existing Haskell type system).

For instance, the DPS system supports two usable levels of nesting for destinations, allowing us to construct and use linear destinations of destinations (type `Dest 1 (Dest m t)`). This is in fact perfectly sufficient to implement non-trivial examples, such as the breadth-first traversal of a tree, but this time using the efficient, destination-based queues in Haskell! Let’s revisit these examples one last time. For the final part of this chapter, we also take compact region considerations into account (this mostly consists in adding `r` and `Region r` all over the function’s signatures). The code of efficient queues (and difference lists for linear data, that they are made of) is presented in Listing 4.1, and the code for updated breadth-first tree traversal is given in Listing 4.2.

There is no real surprise in these implementations; they derive directly from the various iterations and improvements we’ve made over the different lists, efficient queues and breadth-first traversal examples all along this document.

The only notable thing, maybe, is that we reuse the implementation of difference lists presented in Section 4.2, for which we don’t need the flexibility brought by `updWith`, and instead rely on the `extend` functions so that we can store linear data in the difference lists (very similarly to what is presented in [Minamide, 1998; Lorenzen, Leijen, Swierstra, and Lindley, 2024]).

The implementation of efficient queues displayed here, on the other hand, is very similar to the one described earlier in Sections 2.2.3 and 3.4, except that we are using `Ampar` instead of `UAmpar` to reflect the fact that we can now store linear elements inside the queue.

Full, final API for DPS Haskell with linear destination support In Listing 4.3, we sum up all the functions we’ve encountered in this chapter, that are needed to create an Haskell API as flexible as possible, while still being safe, to perform destination passing.

At the time of writing the thesis, we have not implemented this whole API yet, but we envision that there shouldn’t be any particular technical issue to do so. Indeed, the behavior of the `extend` operators derives directly from the corresponding fill functions, that already have a concrete implementation in [Bagrel, 2023b].

4.5 Conclusion

In this chapter we took time to explore what happens when we relax the drastic restriction we had imposed in Chapter 3, namely, that destinations could only be filled with non-linear data (which, consequently, meant that ampars could only be used to store non-linear data).


```

1  type DList r t = Ampar r [t] (Dest r 1 [t])
2
3  newDList :: Region r ⇒ Token → DList r t
4  newDList = newAmpar @[t]
5
6  dListToList :: Region r ⇒ DList r t → [t]
7  dListToList dlist = fromAmpar' (dlist &extendLeaf [])
8
9  append :: Region r ⇒ DList r t → t → DList r t
10 append dlist x = dlist &extend @'(:) @1 (MkSolo x)
11
12 -----
13
14 data EffQueue r t = EffQueue [t] (DList r t)
15
16 newEffQueue :: Region r ⇒ Token → EffQueue r t
17 newEffQueue tok = EffQueue [] (newDList tok)
18
19 singleton :: Region r ⇒ Token → t → EffQueue r t
20 singleton tok x = EffQueue [x] (newDList tok)
21
22 queueToList :: Region r ⇒ EffQueue r t → [t]
23 queueToList (EffQueue front back) = front ++ dListToList back
24
25 enqueue :: Region r ⇒ EffQueue r t → t → EffQueue r t
26 enqueue (EffQueue front back) x = EffQueue front (back `append` x)
27
28 dequeue :: Region r ⇒ EffQueue r t → Maybe (t, EffQueue r t)
29 dequeue (EffQueue front back) = case front of
30   [] → case tokenBesides back of (back, tok) → case (toList back) of
31     [] → drop tok ; Nothing
32     (x : xs) → Just (x, (EffQueue xs (newDList tok)))
33     (x : xs) → Just (x, (EffQueue xs back))

```

Listing 4.1: Efficient queue for linear data

In several careful steps, avoiding scope escape and similar dangers at every corner, we refined our DPS Haskell API to lift this limitation. First, by allowing the user to build ampars to store linear data, but only if they have only one hole each, in the light of the foundational works [Minamide, 1998; Lorenzen, Leijen, Swierstra, and Lindley, 2024]. This gave birth to the family of `extend` functions, that operate on ampars directly instead of destinations.

In a second time, we released yet another set of functions, denoted by `extendFocused`, that let the user fill ampars of arbitrary shapes, with no limit on the number of holes or destinations, with linear data. These functions also operate directly on an ampar, and let the user select the destination they want to focus on, inside the ampar, with a *linear lens* —that is, a convenient object that acts as both a getter and setter for immutable functional data structures.

In the end, we recover a system that is as close as possible to λ_d (Chapter 2), in terms of flexibility and expressiveness, while still conforming itself to the existing type system of Haskell. Of course, the ergonomics for filling linear destinations are not as smooth as in λ_d , but our newly gained flexibility let us implemented a revised version of breadth-first tree traversal in Haskell, that this time, uses exactly the same algorithm and underlying data structures as the initial version provided for λ_d in Section 2.4.

The final API we obtain, after all these refinements, has not been implemented yet in Haskell, though we don't think that it will be particularly challenging, as most of the hard work for low-level operations has already been done in Section 3.5.

Finally, should time be available to devote to this topic in the future, it would be worth exploring how we can leverage the theoretical framework developed in Chapter 2 to formally prove that our proposed Haskell API in Section 4.4 is indeed safe.

```

1  data Tree t = Nil | Node t (Tree t) (Tree t)
2
3  relabelDPS :: Region r => Token -> Tree t -> Tree Int
4  relabelDPS tree = fst (mapAccumBFS (\st _ -> (st + 1, st)) 1 tree)
5
6  mapAccumBFS :: ∀ r s t u. Region r => Token -> (s -> t -> (s, u)) -> s -> Tree t -> (Tree u, s)
7  mapAccumBFS tok f s0 tree =
8    case fromAmpar (newAmpar @(Ur (Tree u)) tok `updWith` \du -> go s0 (singleton (Ur tree, du &fillUr)))
9    of (Ur outTree, Ur st) -> (outTree, st)
10   where
11     go :: s -> EffQueue r (Ur (Tree t), Dest r ω (Tree u)) -> Ur s
12     go st q = case dequeue q of
13       Nothing -> Ur st
14       Just ((utree, dtree), q') -> case utree of
15         Ur Nil -> dtree &fill @'Nil ; go st q'
16         Ur (Node x tl tr) -> case (dtree &fill @'Node) of
17           (dy, dtl, dtr) ->
18             let q'' = q' `enqueue` (Ur tl, dtl) `enqueue` (Ur tr, dtr)
19             in (st', y) = f st x
20                in dy &fillLeaf y ; go st' q''

```

Listing 4.2: Breadth-first tree traversal revisited, with efficient queue for storing destinations

```

1 data Token
2 dup :: Token → (Token, Token)
3 drop :: Token → ()
4 withToken :: ∀ t. (Token → Ur t) → Ur t
5
6 type Region r :: Constraint
7 inRegion :: ∀ t. (∀ r. Region r ⇒ t) → t
8
9 data Ampar r s t
10 newAmpar :: ∀ r s. Region r ⇒ Token → Ampar s (Dest r 1 s)
11 tokenBesides :: ∀ r s t. Region r ⇒ Ampar r s t → (Ampar r s t, Token)
12 toAmpar :: ∀ r s. Region r ⇒ Token → s → Ampar r s () -- now linear in s
13 fromAmpar :: ∀ r s t. Region r ⇒ Ampar r s (Ur t) → (s, Ur t)
14 fromAmpar' :: ∀ r s. Region r ⇒ Ampar r s () → s
15 updWith :: ∀ r s t u. Region r ⇒ Ampar r s t → (t → u) → Ampar r s u -- non-capturing now
16 updWithToken :: ∀ r s t u. Region r ⇒ Ampar r s t → ((t, Token) → u) → Ampar r s u
17
18 data Dest r m t
19 type family DestsOf lCtor r m t -- now multiplicity-polymorphic
20
21 -- == API for unrestricted destinations only ==
22 fillLeaf :: ∀ r t. Region r ⇒ t → Dest r ω t → ()
23 -- we can no longer have fillComp for unrestricted destinations
24
25 -- == API for ampars with linear or unrestricted destinations ==
26 fill :: ∀ lCtor r m t. (Region r, OnlyLinear lCtor t) ⇒ Dest r m t → DestsOf lCtor r m t
27 fillUr :: ∀ r m t. Region r ⇒ Dest r m (Ur t) → Dest r ω t
28
29 -- only one destination on the right (simplified but less expressive)
30 extend :: ∀ lCtor r k t m s fiTys kthFiTy othFiTys.
31   (Region r, OnlyLinear lCtor t,
32    FieldTypes lCtor t ~ fiTys,
33    At k fiTys ~ kthFiTy, Remove k fiTys ~ othFiTys) ⇒
34   othFiTys %m→ Ampar r s (Dest r m t) → Ampar r s (Dest r m kthFiTy)
35 extendUr :: ∀ r s t m. Region r ⇒ Ampar r s (Dest r m (Ur t)) → Ampar r s (Dest r ω t)
36 extendLeaf :: ∀ r s t m. Region r ⇒ t %m→ Ampar r s (Dest r m t) → Ampar r s ()
37 extendComp :: ∀ r s t u. Region r ⇒ Ampar r t u → Ampar r s (Dest r 1 t) → Ampar r s u
38
39 -- with arbitrary type on the right; destination focused with a linear lens
40 extendFocused :: ∀ lCtor r u t1 t2 s m.
41   (Region r, OnlyLinear lCtor u) ⇒
42   Lens t1 t2 (Dest r m u) (DestsOf lCtor r m u) → Ampar r s t1 → Ampar r s t2
43 extendFocusedUr :: ∀ r u t1 t2 s m. (Region r) ⇒
44   Lens t1 t2 (Dest r m (Ur u)) (Dest r ω u) → Ampar r s t1 → Ampar r s t2
45 extendFocusedLeaf :: ∀ r u t1 t2 s m. (Region r) ⇒
46   u %m→ Lens t1 t2 (Dest r m u) () → Ampar r s t1 → Ampar r s t2
47 extendFocusedComp :: ∀ r u t1 t2 s u1 u2. (Region r) ⇒
48   Ampar r u1 u2 → Lens t1 t2 (Dest r 1 u1) u2 → Ampar r s t1 → Ampar r s t2

```

Listing 4.3: Destination-passing API for Haskell with support for linear destinations

Related Work

1 Destination-passing style for efficient memory management

Shaikhha et al. [2017] 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.

2 Programming with Permissions in Mezzo

Protzenko and Pottier [2013] introduced the Mezzo programming language, in which mutable data structures can be freed into immutable ones after having been completed. This principle is used to some extent 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 [Larus, 1989].

3 Tail modulo constructor

Another example of destinations in a compiler’s optimizer is [Bour, Clément, and Scherer, 2021]. 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

Related Work

```
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 [Shaikhha et al., 2017], 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.

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 [Minamide, 1998]. Our system is strongly inspired by theirs. In [Minamide, 1998], 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, $(\mathbf{T}, \mathbf{S})\mathbf{hfun}$ is thus a weak form of linear function type $\mathbf{T} \multimap \mathbf{S}$.

In [Minamide, 1998], 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 $\mathbf{T} \multimap \mathbf{S}$ is reinterpreted as $\mathbf{S} \wp \mathbf{T}^\perp$. We, likewise, reinterpret $(\mathbf{T}, \mathbf{S})\mathbf{hfun}$ as $\mathbf{S} \ltimes [\mathbf{T}]$ (a sort of weak "par").

A key consequence is that destinations—as first-class representations of holes—appear naturally in λ_d , while [Minamide, 1998] doesn't have them. This means that using [Minamide, 1998], or the more recent but similarly expressive system from [Lorenzen, Leijen, Swierstra, and Lindley, 2024], 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 [1998] places on hole abstractions. For instance, we can't pattern-match on \mathbf{S} in (unapplied) $(\mathbf{T}, \mathbf{S})\mathbf{hfun}$; so in λ_d , we can't act directly on the left-hand side \mathbf{S} of $\mathbf{S} \ltimes \mathbf{T}$, only on the right-hand side \mathbf{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 [1998]’s system can be embedded in λ_d , and if one stays in this fragment, we’re never confronted with ages.

5 Semi-axiomatic sequent calculus

In [DeYoung, Pfenning, and Pruiksmas, 2020] 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 $\llbracket \mathbf{T} \rrbracket$ 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.

6 Rust lifetimes

Rust uses a system of lifetimes (see e.g. [Pearce, 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 $\alpha \leq \beta$) 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.

7 Oxidizing OCaml

Lorenzen, White, et al. [2024] 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 [Lorenzen, White, et al., 2024], *once* supports weakening, whereas λ_d ’s $\mathbf{1}$ 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.

Related Work

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.

Conclusion and future work

Destination passing is not a novel technique in the history of programming languages. It has been employed very early —when proper manual memory management was required even for the most basic programs— as a way to avoid useless allocations of memory for intermediary results.

It then has been kinda forgotten during the golden era of object-oriented programming, for which automatic memory management schemes have arisen. But today, we revisit destination passing in another setting, that is, functional programming languages, with slightly different expectations.

Of course, we don't intend to go back to fully manual memory management for modern functional languages. Not having to care about memory management in most cases is a real advantage of modern programming languages. Instead, we want to be either let the compiler optimize programs automatically, at compile time, taking advantage of destination passing [Bour, Clément, and Scherer, 2021; Leijen and Lorenzen, 2023]; or, as we shown in this document, be able to manually opt-in for destination passing in specific parts of a functional program, for which we need improved performance or expressivity.

Destination passing having an imperative nature at heart, it means it need to be tamed for use in a functional programming language, especially if we want to respect immutability and purity principles which are very common in such settings.

For that, we first looked into linear type systems (Chapter 1). From their roots in (linear) logic to their modern presentations and implementations, we've seen how linear types can be leveraged to ensure that a resource will only be used exactly once.

With linear types in hand, we embarked in a journey in which we tried to capture the essence of destination passing in a formal, functional model, denoted λ_d . (Chapter 2). The idea was that a destination —a pointer to a yet unused memory cell— must be used exactly once, and that after it has been used, we are allowed to read the memory cell it pointed to, and the value contained within is now guaranteed not to change (to abide by immutability). Consequently, the core principle is that a destination should always point to an existing hole (a yet unused memory cell), and that every remaining hole in a structure must be referenced by exactly one destination that hasn't been used yet. Destinations acts as both witnesses and a way of filling those holes, and when no destination remains, a structure (should) no longer have holes.

Early on this journey however, we discovered the dreaded *scope escape* issue, which asked for more than a year of research to be spent on the topic before we could finally devise a solution. Scope escape (Section 2.3) is what can happen when a destination is used to store another destination. Even with a linear discipline imposed on destinations, it can become undistinguishable at type level whether a destination has been used —which means the associated

Conclusion and future work

memory cell now contains a valid value— or if the destination has been stored away for later use, which means we shouldn’t read the content of the corresponding memory cell yet otherwise we risk a segmentation fault or a similar memory issue.

Because of scope escape being possible, we had to reinforce the type system of λ_d , and add a *system of ages* in addition to the linear discipline, to make it memory safe. Fortunately, in Section 1.4, we had developed a *modal* presentation of a linear type system, based on an underlying semiring algebraic structure, which is well suited for extension. So we were able to combine the linear system and the new age system in the same semiring structure. This led to a type system for λ_d (Section 2.5) which is very close to earlier work.

In λ_d , the novel objects, compared to an usual λ -calculus, are ampars—which encapsulate data structures which may be yet incomplete, so which may contain holes—and destinations, which are linear-controlled pointers to these holes. Holes (in structures) are not first-class objects of the language themselves, unlike destinations, but they still play a very important role in the typing and semantics of λ_d . Notably, we developed a new typing strategy for structure with holes in Section 2.6.1. Both the holes of an ampar, and destinations attached to the ampar, appear as named bindings inside the typing context, and a hole binding and destination binding of the same name then cancel each other out. Thus a well-balanced ampar types in an empty typing context; while a non-empty context indicates that the ampar has either a non-referenced hole, or an extra destination pointing to a hole that has been filled already! This is how we ensure, within the type system, that no destination can be reused or forgotten.

The operational semantics of λ_d (Section 2.6.3) relies on a rather abstract memory model, where the state of the program is purely described by the evaluation context and the term under focus, and where memory writes resulting from destination-filling operations are conducted as global substitutions on named holes inside the evaluation context. This model for semantics appear to be simpler than store semantics techniques (as the typing rules for evaluation contexts can be directly derived from the term ones, while typing rules for a store are much more complex) while still capturing satisfyingly the essence of destination-induced memory mutations.

A main contribution of this work is that we were able to formalize λ_d in the Coq proof assistant (Section 2.7) and then produce a mechanically-verified proof of the usual safety theorems for λ_d , namely *progress* and *preservation*.

With λ_d , we thus obtain a formal functional language with destination passing so rooted at its core that we have maximal flexibility to use destinations, and for instance don’t even need normal data constructors to be part of the language (they can be recovered from destination filling operators). But this comes at the cost of needing a custom type system to make the system safe.

Thus, the second, and equally important part of the PhD—especially given the fact that the PhD is funded by a company with industrial functional programming in its DNA— was focused on bringing most of the ideas of λ_d , and destination passing in general, to a practical, industrial functional programming language. Haskell was a language of choice for this, given its existing support for linear types. All the more so since my industrial supervisor is one of the authors of Linear Haskell, and a major proponent for linear type support and improving their ergonomics inside GHC. The main challenge of implementation destination passing in Haskell, is that of course, we cannot use the custom type system based on linearity and ages of λ_d to ensure safety. We have to do with linearity alone.

Initial experiments for destination-passing style programming in Haskell actually predates the theoretical work on λ_d . Indeed, at that time, scope escape was still very much an issue, for which we had not yet found a valid solution (so the age system had not been invented either), so we decided, in the meantime, to start writing down an initial API for destination passing in Haskell (Chapter 3), in which scope escape cannot happen because we decided, as a safeguard, to forbid storing destinations inside other destinations. A major concern was finding a way to not make the Haskell memory model blow up by doing unsafe mutations. For that, we used *compact regions*, that is, memory areas free from the surveillance of the garbage collector, in which we could temporarily hold unfinished data structures. This work ultimately ranged from designing a cohesive and ergonomic high-level API, down to the low-level implementation of compiler’s primitives into which destination-filling operations are translated. At the end, we obtain encouraging results and benchmarks (Section 3.6) which show that destination passing is a tool of choice, in specific situations, to improve the performance of functional programs, but also to make some of them easier to express, in a more direct, imperative-like style, without sacrificing memory guarantees. Yet, there is still work to do to improve both the ergonomics and performance of the DPS implementation, if we hope for DPS to become a more widespread technique in the functional programming community.

One inconvenience with DPS Haskell, in the form of this first implementation, is that the restriction we imposed initially —that destinations cannot store any linear resource to avoid scope escape— impedes the flexibility and expressivity gains that we claimed true for λ_d . In particular, one of our preferred examples, the traversal of a tree in breadth-first fashion, cannot be implemented with just our shiny destination toolbox, and we have to revert to normal Haskell data structures at some point, instead of using only our new destination-based data structures.

In Chapter 4, we try to reclaim much of this lost expressivity. But relaxing the safety constraint that destinations cannot be filled with linear resources means we must be particularly careful to avoid scope escape, and a range of other issues that can manifest themselves when we don’t have ages (and age control) to help. Still, at the end of this journey, we get a way to write our dear breadth-first tree traversal with just destination-enabled data structures, in Haskell, with an API that we think is safe.

Further work would consist in making sure the initial and upgraded DPS Haskell APIs, devised in Chapters 3 and 4 are safe. We claimed that λ_d can be used as a framework to reason about the safety and correctness of other destination-passing systems, and it would be time to see if it stands up to the test.

The second axis of improvement would consist in implementing DPS support for Haskell inside the garbage-collected heap (the “normal” memory area in which most Haskell objects live), outside of the protective shell of compact regions. The difficulty being that the safety of such a system is very reliant on the knowledge of the garbage collection system, and in particular, when precisely the assumptions of immutability are used.

I hope that at this point, the reader has a got a good and enjoyable overview of this exotic journey, trying to reconcile imperative techniques and functional principles, from high-level and formal considerations down to low-level implementation and the associated performance concerns. Finally, we can celebrate: safe, functional, destination passing is not an oxymoron; it exists!

Bibliography

- Howard, W H (1969). *The formulae-as-types notion of construction*. en. Transcribed in 2017 to LATEX by Armando B. Matos. URL: <https://www.dcc.fc.up.pt/~acm/howard2.pdf> (visited on 05/05/2025).
- Hood, Robert and Robert Melville (1981). “Real-time queue operations in pure LISP.” In: *Information Processing Letters* 13.2, pp. 50–54. ISSN: 0020-0190. DOI: [https://doi.org/10.1016/0020-0190\(81\)90030-2](https://doi.org/10.1016/0020-0190(81)90030-2). URL: <https://www.sciencedirect.com/science/article/pii/0020019081900302>.
- Hughes, John (Jan. 1986). “A Novel Representation of Lists and its Application to the Function ”reverse”.” In: *Inf. Process. Lett.* 22, pp. 141–144.
- Felleisen, Matthias (1987). “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. URL: <https://www2.ccs.neu.edu/racket/pubs/dissertation-felleisen.pdf>.
- Girard, Jean-Yves (1987). “Linear logic.” In: *Theoretical Computer Science* 50.1, pp. 1–101. ISSN: 0304-3975. DOI: [https://doi.org/10.1016/0304-3975\(87\)90045-4](https://doi.org/10.1016/0304-3975(87)90045-4). URL: <https://www.sciencedirect.com/science/article/pii/0304397587900454>.
- Larus, James Richard (1989). “Restructuring symbolic programs for concurrent execution on multiprocessors.” AAI9006407. phd. University of California, Berkeley.
- Andreoli, Jean-Marc (June 1992). “Logic Programming with Focusing Proofs in Linear Logic.” In: *Journal of Logic and Computation* 2.3, pp. 297–347. ISSN: 0955-792X. DOI: [10.1093/logcom/2.3.297](https://doi.org/10.1093/logcom/2.3.297). URL: <https://doi.org/10.1093/logcom/2.3.297> (visited on 12/05/2024).
- Girard, Jean-Yves, Andre Scedrov, and Philip J. Scott (Apr. 1992). “Bounded linear logic: a modular approach to polynomial-time computability.” In: *Theoretical Computer Science* 97.1, pp. 1–66. ISSN: 0304-3975. DOI: [10.1016/0304-3975\(92\)90386-T](https://doi.org/10.1016/0304-3975(92)90386-T). URL: <https://www.sciencedirect.com/science/article/pii/030439759290386T> (visited on 12/05/2024).
- Gibbons, Jeremy (1993). “Linear-time Breadth-first Tree Algorithms: An Exercise in the Arithmetic of Folds and Zips.” en-gb. In: No. 71. Number: No. 71. URL: <https://www.cs.ox.ac.uk/publications/publication2363-abstract.html> (visited on 10/18/2023).
- Bierman, G.M. (Aug. 1994). *On intuitionistic linear logic*. Tech. rep. UCAM-CL-TR-346. University of Cambridge, Computer Laboratory. DOI: [10.48456/tr-346](https://doi.org/10.48456/tr-346). URL: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-346.pdf>.
- Girard, J.-Y. (1995). “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, pp. 1–42. ISBN: 978-0-511-62915-0. DOI: [10.1017/CB09780511629150.002](https://doi.org/10.1017/CB09780511629150.002). URL: https://www.cambridge.org/core/product/identifier/CB09780511629150A008/type/book_part (visited on 03/21/2022).

- Huet, Gérard (Sept. 1997). “The Zipper.” en. In: *Journal of Functional Programming* 7.5, pp. 549–554. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796897002864. URL: <https://www.cambridge.org/core/journals/journal-of-functional-programming/article/zipper/0C058890B8A9B588F26E6D68CF0CE204> (visited on 05/20/2025).
- Minamide, Yasuhiko (Jan. 1998). “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, 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).
- Curien, Pierre-Louis and Hugo Herbelin (2000). “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, pp. 233–243. ISBN: 1581132026. DOI: 10.1145/351240.351262. URL: <https://doi.org/10.1145/351240.351262>.
- Okasaki, Chris (Sept. 2000). “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, 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).
- Pierce, Benjamin C. (2002). *Types and Programming Languages*. 1st. The MIT Press. ISBN: 0262162091.
- Danvy, Olivier and Lasse R. Nielsen (Nov. 2004). “Refocusing in Reduction Semantics.” en. In: *BRICS Report Series* 11.26. ISSN: 1601-5355, 0909-0878. DOI: 10.7146/brics.v11i26.21851. URL: <https://tidsskrift.dk/brics/article/view/21851> (visited on 12/17/2024).
- Harrington, Dana (Mar. 2006). “Uniqueness logic.” en. In: *Theoretical Computer Science* 354.1, 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).
- Sørensen, Morten Heine B. and Paweł Urzyczyn (July 2006). *Lectures on the Curry-Howard Isomorphism*. en-US. 1st. Vol. 149. Studies in Logic and the Foundations of Mathematics. Elsevier Science. 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).
- Biernacka, Małgorzata and Olivier Danvy (May 2007). “A syntactic correspondence between context-sensitive calculi and abstract machines.” In: *Theoretical Computer Science*. Festschrift for John C. Reynolds’s 70th birthday 375.1, 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).
- Sewell, Peter et al. (2007). “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, pp. 1–12. ISBN: 9781595938152. DOI: 10.1145/1291151.1291155. URL: <https://doi.org/10.1145/1291151.1291155>.
- Protzenko, Jonathan and François Pottier (Sept. 2013). “Programming with Permissions in Mezzo.” In: *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. arXiv:1311.7242 [cs], pp. 173–184. DOI: 10.1145/2500365.2500598. URL: <http://arxiv.org/abs/1311.7242> (visited on 10/16/2023).
- Ghica, Dan R. and Alex I. Smith (2014). “Bounded Linear Types in a Resource Semiring.” en. In: *Programming Languages and Systems*. Ed. by Zhong Shao. Berlin, Heidelberg: Springer, pp. 331–350. ISBN: 978-3-642-54833-8. DOI: 10.1007/978-3-642-54833-8_18.

- Petricek, Tomas, Dominic Orchard, and Alan Mycroft (Aug. 2014). “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, 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).
- Yang, Edward Z. et al. (Aug. 2015). “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, 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).
- Shaikhha, Amir et al. (Sept. 2017). “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, 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).
- Atkey, Robert (2018). “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, pp. 56–65. ISBN: 9781450355834. DOI: 10.1145/3209108.3209189. URL: <https://doi.org/10.1145/3209108.3209189>.
- Bernardy, Jean-Philippe et al. (Jan. 2018). “Linear Haskell: practical linearity in a higher-order polymorphic language.” In: *Proceedings of the ACM on Programming Languages* 2.POPL. 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).
- Abel, Andreas and Jean-Philippe Bernardy (Aug. 2020). “A unified view of modalities in type systems.” In: *Proc. ACM Program. Lang.* 4.ICFP. DOI: 10.1145/3408972. URL: <https://doi.org/10.1145/3408972>.
- DeYoung, Henry, Frank Pfenning, and Klaas Pruiksma (2020). “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, 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>.
- Bour, Frédéric, Basile Clément, and Gabriel Scherer (Feb. 2021). “Tail Modulo Cons.” In: *arXiv:2102.09823 [cs]*. arXiv: 2102.09823. URL: <http://arxiv.org/abs/2102.09823> (visited on 03/22/2022).
- Pearce, David J. (Apr. 2021). “A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust.” In: *ACM Trans. Program. Lang. Syst.* 43.1, 3:1–3:73. ISSN: 0164-0925. DOI: 10.1145/3443420. URL: <https://dl.acm.org/doi/10.1145/3443420> (visited on 01/15/2025).
- Marshall, Danielle, Michael Vollmer, and Dominic Orchard (2022). “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, 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).
- Spiwack, Arnaud et al. (Aug. 2022). “Linearly qualified types: generic inference for capabilities and uniqueness.” In: *Proceedings of the ACM on Programming Languages* 6.ICFP, 95:137–95:164. DOI: 10.1145/3547626. URL: <https://dl.acm.org/doi/10.1145/3547626> (visited on 10/16/2023).

- Bagrel, Thomas (2023a). *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. (Visited on 10/19/2023).
- (2023b). *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. (Visited on 10/19/2023).
- Gibbons, Jeremy et al. (Aug. 2023). “Phases in Software Architecture.” en. In: *Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Software Architecture*. Seattle WA USA: ACM, 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).
- Leijen, Daan and Anton Lorenzen (Jan. 2023). “Tail Recursion Modulo Context: An Equational Approach.” en. In: *Proceedings of the ACM on Programming Languages* 7.POPL, pp. 1152–1181. ISSN: 2475-1421. DOI: 10.1145/3571233. URL: https://dl.acm.org/doi/10.1145/3571233 (visited on 01/25/2024).
- Lorenzen, Anton, Daan Leijen, and Wouter Swierstra (Aug. 2023). “FP²: Fully in-Place Functional Programming.” en. In: *Proceedings of the ACM on Programming Languages* 7.ICFP, pp. 275–304. ISSN: 2475-1421. DOI: 10.1145/3607840. URL: https://dl.acm.org/doi/10.1145/3607840 (visited on 12/11/2023).
- Spiwack, Arnaud (Nov. 2023a). *Linear constraints proposal by aspiwack · Pull Request #621 · ghc-proposals/ghc-proposals*. en. URL: https://github.com/ghc-proposals/ghc-proposals/pull/621 (visited on 03/25/2025).
- (Mar. 2023b). *Linear Constraints: the problem with scopes*. en. Blog. URL: https://tweag.io/blog/2023-03-23-linear-constraints-linearly/ (visited on 03/25/2025).
- Bagrel, Thomas (Jan. 2024a). “Destination-passing style programming: a Haskell implementation.” In: *35es Journées Francophones des Langages Applicatifs (JFLA 2024)*. Saint-Jacut-de-la-Mer, France. URL: https://inria.hal.science/hal-04406360.
- (Nov. 2024b). *Primitives for zero-copy compact regions by tbagrel1 · Pull Request #683 · ghc-proposals/ghc-proposals*. en. URL: https://github.com/ghc-proposals/ghc-proposals/pull/683 (visited on 04/01/2025).
- Lorenzen, Anton, Daan Leijen, Wouter Swierstra, and Sam Lindley (June 2024). “The Functional Essence of Imperative Binary Search Trees.” In: *Proc. ACM Program. Lang.* 8.PLDI. DOI: 10.1145/3656398. URL: https://doi.org/10.1145/3656398.
- Lorenzen, Anton, Leo White, et al. (Aug. 2024). “Oxidizing OCaml with Modal Memory Management.” In: *Proc. ACM Program. Lang.* 8.ICFP, 253:485–253:514. DOI: 10.1145/3674642. URL: https://dl.acm.org/doi/10.1145/3674642 (visited on 01/06/2025).
- Spiwack, Arnaud (Jan. 2024). *Ur can’t be a newtype*. en. Blog. URL: https://www.tweag.io/blog/2024-01-18-linear-desugaring/#ur-cant-be-a-newtype (visited on 04/01/2025).
- Bagrel, Thomas and Arnaud Spiwack (Apr. 2025). “Destination Calculus: A Linear λ -Calculus for Purely Functional Memory Writes.” In: *Proc. ACM Program. Lang.* 9.OOPSLA1. DOI: 10.1145/3720423. URL: https://doi.org/10.1145/3720423.
- Leijen, Daan and Anton Lorenzen (2025). “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).